Université de Lille, Faculté des sciences et technologies

Département de formation doctorale en informatique **École doctorale MADIS Lille**
UFR IEEA

DRUID

Metacompilation of Baseline JIT Compilers

Métacompilation des compilateurs JIT de base

THÈSE

présentée et soutenue publiquement le 28 November 2025

pour l'obtention du

Doctorat de l'Université de Lille

(spécialité informatique et application)

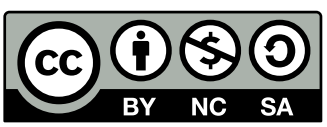
par

Nahuel Palumbo

Composition du jury

Présidents :	Fabrice RASTELLO	Directeur de Recherche Université de Grenoble Alpes, Inria
Rapporteurs :	Gaël THOMAS	Directeur de Recherche Telecom SudParis, Inria
	Wolfgang DE MEUTER	Professeur Vrije Universiteit Brussel
Examinateurs :	Emmanuelle SAILLARD-VIROULEAU	Chargée de Recherche Université de Bordeaux, Inria
Directeurs de thèse :	Guillermo POLITO	Chargé de Recherche Université de Lille, Inria
	Stéphane DUCASSE	Directeur de Recherche Université de Lille, Inria
Invité :	Björn FRANKE	Professeur Université de Edinburgh

Centre de Recherche en Informatique, Signal et Automatique de Lille — UMR 9189
INRIA Lille - Nord Europe



Agradecimientos

Gracias a todos los miembros del jurado por tomarse el laburo de leer nuestro trabajo y escucharme presentarlo. Esperamos que lo hayan encontrado interesante. Nada de este trabajo hubiera podido ser posible sin mis supervisores ni la gente que me ha enseñado tanto durante estos años. Gracias a todos los que formaron parte de esta experiencia. Siempre me sentí cómodo y bien recibido en el equipo, desde el primer día, lo que fue un gran impulso para llegar hasta acá.

Tuve la suerte de hacerme varios amigos por acá. Distintas personas que, viniendo de lugares tan distantes y con culturas distintas, nos encontramos los unos en los otros. Con ellos convivimos, viajamos, escalamos, trabajamos y reímos mucho, siempre con placer. Hemos hecho fiestas y tomado birra, mucha birra. No tengo palabras de agradecimiento que alcancen a expresar lo bien que me hicieron durante estos años. Por más que estuviera lejos de mi casa y de mi gente, incluso en momentos difíciles, nunca me sentí solo porque siempre los tuve a ustedes ahí. Todos esos momentos vividos juntos son de las cosas más lindas que me llevo conmigo.

Por otro lado, quiero aprovechar para agradecer a mis viejos amigos que están en Argentina, los cuales extraño continuamente. Ya sea en el ámbito personal, sentimental o laboral, contar con su apoyo es algo que no tiene precio para mí. Gran parte de lo que soy ahora es gracias a ustedes, que de infinitas maneras me ayudaron a construir el camino que me hizo llegar acá. Como siempre, espero el momento para volvernos a encontrar.

Por último y no menos importante, agradecerle a mi familia, la mejor del mundo. Ellos forjaron lo mejor de mi persona desde el día cero y representan una parte muy importante de mi vida. Siempre estuvieron, están y estarán ahí para lo que necesite. Estoy muy orgulloso de la familia que me tocó. Ojalá nos volvamos a juntar pronto, y que tanto los momentos buenos como los difíciles nos encuentren siempre unidos.

Finalmente, brindo por mi querida Argentina, a la que estoy ansioso por volver. Para que la fraternidad nos encuentre en el futuro próximo, y que podamos convivir entre nosotros y con el resto de las naciones de este bello mundo.

Muchas gracias.

Acknowledgments

Thanks to all the members of the jury for taking the time to read our work and listen to my presentation. We hope you found it interesting. None of this would have been possible without my supervisors and the people who have taught me so much over these years. Thanks to everyone who was part of this experience. I always felt comfortable and warmly welcomed in the team since the very first day, which was a great motivation to make it all the way here.

I was lucky to make several friends here. Different people who, coming from distant places and cultures, somehow found each other. With them I lived, traveled, climbed, worked, and laughed a lot, always with joy. We partied and drank beer, lots of beer. I have no words of gratitude strong enough to express how much you all meant to me during these years. Even though I was far from home and from my people, even in difficult moments, I never felt alone because you were always there. All those moments we shared together are among the most beautiful things I take with me.

I also want to thank my old friends back in Argentina, whom I constantly miss. Whether in personal, emotional, or professional matters, having your support is priceless to me. A large part of who I am today is thanks to you, who helped me in countless ways to build the path that brought me here. As always, I look forward to the moment we meet again.

Lastly, and by no means least, I want to thank my family, the best in the world. They have shaped the best parts of who I am since day one and represent an essential part of my life. They have always been, are, and will always be there whenever I need them. I am truly proud of the family I have. I hope we can get together again soon, and that both the good and hard times always find us united.

Finally, I raise a toast to my beloved Argentina, which I'm eager to return to. May fraternity bring us together in the near future, and may we live in harmony among ourselves and with the rest of the nations of this beautiful world.

Thank you very much.

Abstract

Virtual Machines (VMs) combine interpreters and just-in-time (JIT) compilers to achieve good performance. However, maintaining multiple execution engines significantly increases development and maintenance costs. While prior work has explored automatically generating optimizing JIT compilers, the question of whether metacompilation can be applied effectively to baseline JIT compilers, which favor fast compilation and short warmup times, has remained open.

In this thesis, we present **Druid, a source-to-source ahead-of-time metacompiler that generates a baseline JIT compiler** directly from interpreter definitions. Language developers guide the metacompilation process through annotations and intrinsics, while Druid targets an existing JIT backend to reuse mature optimizations and runtime infrastructure, delivering competitive warmup performance.

We applied Druid to the production Pharo VM and evaluated the performance of the automatically generated JIT compiler. The resulting JIT frontend runs $1.8\times$ faster than the interpreter and achieves 70% of the performance of a handwritten JIT compiler that has been refined for over a decade. This required changes at only 60 call sites in the interpreter, demonstrating that Druid greatly reduces the engineering effort needed to maintain and evolve language VMs.

Baseline JIT compilers must compile quickly while still applying essential optimizations. A widely used strategy is compile-time abstract interpretation, which performs register allocation, constant propagation, and instruction scheduling in a single pass. However, isolating and quantifying the benefits of this technique is difficult when comparing different, independently engineered compilers.

Using Druid, we automatically generated several variants of the Pharo JIT compiler to experimentally evaluate the impact of compile-time abstract interpretation under controlled conditions. We use metacompilation as a means to (a) reduce the experimentation effort and (b) to produce comparable compiler variants, reducing implementation noise. Our results show that abstract interpretation reduces machine code size by 12% and improves execution speed by 10% on average (up to 30%), with no measurable increase in JIT compilation time compared to a direct translation approach.

Keywords: JIT compilation, Virtual machines, Compiler generation, Metacompilation, Language implementation, Optimizations

Résumé

Les machines virtuelles (VM) combinent des interprètes et des compilateurs à la volée (JIT) pour obtenir de bonnes performances. Cependant, maintenir plusieurs moteurs d'exécution augmente considérablement les coûts de développement et de maintenance. Alors que des travaux antérieurs se sont intéressés à la génération automatique de compilateurs JIT optimisants, la question de savoir si la métacompilation peut être appliquée efficacement aux compilateurs JIT de base, qui privilégient une compilation rapide et un temps de préchauffage réduit, restait ouverte.

Dans cette thèse, nous présentons **Druid, un métacompilateur source-à-source avant d'exécution qui génère un compilateur JIT de base** directement à partir des définitions de l'interpréteur. Les développeurs du langage guident le processus de métacompilation à l'aide d'annotations et d'intrinsèques, pendant que Druid cible une infrastructure JIT existante afin de réutiliser des optimisations et des composants d'exécution éprouvés, offrant ainsi des performances de préchauffage compétitives.

Nous avons appliqué Druid à la machine virtuelle Pharo utilisée en production, et évalué les performances du compilateur JIT généré automatiquement. Le JIT de base obtenu est $1.8\times$ plus rapide que l'interpréteur et atteint 70% des performances d'un compilateur JIT écrit manuellement et amélioré depuis plus de dix ans. Cela n'a nécessité que 60 modifications ponctuelles dans l'interpréteur, démontrant que Druid réduit considérablement l'effort d'ingénierie requis pour maintenir et faire évoluer les VM de langage.

Les compilateurs JIT de base doivent compiler rapidement tout en appliquant des optimisations essentielles. Une stratégie couramment utilisée est l'interprétation abstraite à la compilation, qui réalise en un seul passage l'allocation de registres, la propagation de constantes et l'ordonnancement des instructions. Cependant, il est difficile d'isoler et de quantifier les bénéfices de cette technique en comparant des compilateurs développés indépendamment.

Avec Druid, nous avons généré automatiquement plusieurs variantes du compilateur JIT de Pharo afin d'évaluer expérimentalement l'impact de l'interprétation abstraite dans des conditions contrôlées. Nous utilisons la métacompilation pour: (a) réduire l'effort d'expérimentation et (b) produire des variantes de compilateurs comparables, réduisant le bruit lié aux différences implémentations. Nos résultats montrent que l'interprétation abstraite réduit la taille du code machine de 12% et améliore la vitesse d'exécution de 10% en moyenne (jusqu'à 30%), sans augmentation mesurable du temps de compilation JIT par rapport à une approche de traduction directe.

Mots-clés: Compilation JIT, Machines Virtuelles, Génération du compilateur, Métacompilation, Implémentation du langage, Optimisations

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Contents

1.1 JIT Compilers Generation

Language Virtual Machines (VMs) use multiple execution tiers to optimize programs at run time [Alpern 2000, Arnold 2005]. These tiers are often structured around an interpreter and one or more just-in-time (JIT) compilers to balance runtime responsiveness, program analysis depth, and compilation effort [Deutsch 1984, Würthinger 2013, Bolz 2015, Miranda 2018].

In such architectures, programming language features must often be implemented multiple times, across different execution tiers and levels of abstraction. This redundancy increases the complexity of building, understanding, and extending VMs [Ertl 2003, Groß 2024]. Re-implementing language semantics across interpreters and compilers introduces risks of semantic mismatches and possible bugs [Polito 2022b]. Furthermore, it raises the cost of language evolution and disrupts the division of responsibilities between language developers and VM experts.

To address these challenges, several generative approaches have been proposed, aiming to reduce duplication and complexity by automatically generating compiled code from interpreter definitions [Bolz 2009, Yermolovich 2009, Würthinger 2013, Thakur 2019]. These solutions are typically based on the first Futamura projection [Futamura 1999], where partial evaluation is applied to an interpreter specialized with a given program, resulting in optimized machine code. In these models,

language developers focus on writing interpreters, which are generally easier to implement and debug than compilers [Ertl 2003]. However, most *metacompilation* systems perform these transformations at execution time, making them sensitive to increased compilation and warm-up times.

In this thesis, we present **Druid, a source-to-source ahead-of-time metacompiler** designed to generate baseline JIT compiler frontends from interpreter definitions. Unlike partial evaluation approaches, Druid follows the handwritten compiler generator (*Cogen*) methodology [Birkedal 1994]. Specifically, we propose to target existing JIT compiler backends, generating only the compiler frontend that describes the language semantics [Torczon 2007]. Our solution introduces *metadata and intrinsic* functions for the interpreter that developers use to guide the metacompilation process. Furthermore, Druid automatically performs *staging* of computations, deciding whether expressions should be executed ahead-of-time, at JIT compile time, or at run time in the generated machine code.

This architecture enables a clear separation of concerns between VM components: language developers define interpreters using the interpreter framework, while Druid automatically generates the corresponding JIT compiler frontends, removing the need for manual duplication. As we demonstrate later in this thesis, our approach also enables the generation of different versions of baseline JIT compilers with varying optimization levels, reducing the noise of manual implementation details and facilitating the evaluation of optimization impacts.

We validate our approach by applying Druid to the Pharo VM, a production VM for the Pharo programming language [Ingalls 1997, Miranda 2018]. Pharo is a fully dynamic, object-oriented language that executes bytecode-based and primitive methods [Ducasse 2022]. The Pharo VM is architected around an interpreter and a baseline JIT compiler, known as Cogit [Miranda 2011].

1.2 Metacompilation using Druid

To explain how our solution works, we must first define the relevant components within a virtual machine's (VM) execution engine. A VM executes programs by processing a set of VM instructions. Each execution tier (*i.e.* interpreter, JIT compilers) must implement the semantics of these instructions through what we call *instruction handlers*. In an interpreter, an instruction handler directly performs the instruction's semantics, while in a compiler, the handler generates code that will reproduce the same behavior at run time.

Definition 1 (Instruction handler) *An instruction handler is a function that implements the semantics of a VM instruction within a specific execution tier.*

We extended the Pharo VM, which defines instruction handlers for both the interpreter and the baseline JIT compiler [Ingalls 1997,Miranda 2011,Miranda 2018].

A key characteristic of this architecture is that each instruction handler must be implemented twice: once for the interpreter and once for the JIT compiler. Beyond the inherent differences between writing an interpreter (a one-stage execution engine) and a compiler (a two-stage execution engine), maintaining semantic consistency across tiers increases the complexity and risk of divergence [Polito 2022b].

We propose to generate the instruction handlers for the baseline JIT compiler automatically from the interpreter definitions, ensuring semantic consistency while significantly reducing implementation effort.

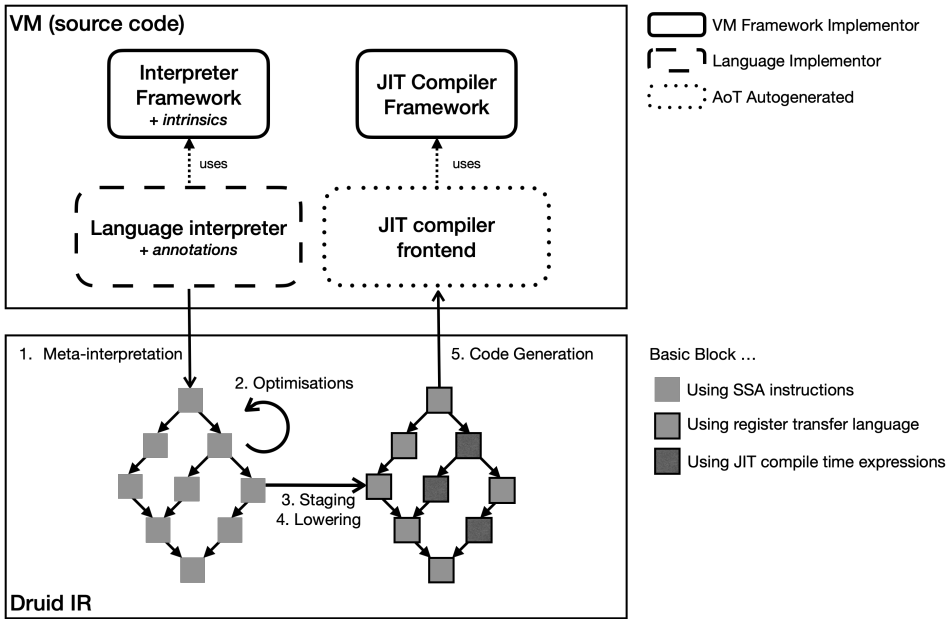


Figure 1.1: Overview of the solution presented in this work. Instruction handlers in the interpreter are metacompiled into equivalent handlers in the baseline JIT compiler.

To this end, we developed **Druid, an ahead-of-time source-to-source metacompiler** that generates baseline JIT compiler frontend code from interpreter instruction handlers. Our prototype targets an existing JIT compiler backend, leveraging its mature code generation infrastructure and runtime optimizations. Consequently, Druid focuses on generating the compiler frontend, which encodes the language's semantics.

The metacompilation process is performed entirely ahead of time, ensuring that it does not add runtime overhead during program execution. Moreover, the generated JIT compiler source code is seamlessly integrated into the VM alongside manually written components, making our solution compatible with other approaches and suitable for production environments.

Example of metacompilation. We illustrate a concrete example of metacompilation: the pushNewArray bytecode instruction from the Pharo VM. This bytecode pops the top N elements from the stack, stores them in a newly allocated array, and pushes the resulting array back onto the stack. The input (interpreter) and output (JIT compiler) are presented in Figure 1.2.

On the left side of the figure, the interpreter implementation is straightforward: lines 3-6 allocate the array, lines 8-10 perform the loop to populate it by popping stack elements, and line 12 pushes the resulting array.

On the right side of the figure, we show a simplified version of the corresponding baseline JIT compiler frontend. Although it implements the same semantics, the code is more intricate due to staging and the need to interact with runtime services and abstract interpreter optimizations. The methods prefixed with gen_ emit instructions to be executed at runtime.

To efficiently metacompile this bytecode handler, Druid must:

1. **Stage the fetch of the array size** (line 6), it is constant at compile time.
2. **Spill the abstract stack** (line 9) before the runtime call (lines 17-18). Importantly, spilling must occur *before* the branching point (line 11).
3. **Stage the loop** (lines 24-30) at compile time because it does not depend on runtime information. Thus, the compiled machine code is *unrolled*.

1.3 Challenges of Baseline JIT Generation from Interpreter Definition

Our metacompilation approach translates interpreter code into JIT compiler code that preserves the same semantics. However, JIT compilation can take advantage of this translation process to *weave* optimization decisions into the generated code [Hölzle 1992,Holzle 1994]. For example, not all interpreter features are worth JIT-compiling: compiling slow paths within instruction handlers might increase JITted code size unnecessarily, as these paths are executed infrequently. In such cases, the JIT compiler can emit a call back to the interpreter instead. Furthermore, JIT compilers can exploit compile-time invariants, such as values that are constant during JIT compilation, enabling optimizations at JIT compile time.

The challenge lies in embedding these decisions into the generated code, without having runtime information available during metacompilation. This introduces the following key challenges:

Ahead-of-time compilation scoping. *The baseline JIT metacompiler must statically determine which paths within instruction handlers should be compiled.* In a VM with multiple execution engines, at least one engine (typically the interpreter) must support the full language specification. JIT compilers, however, are allowed to compile only a subset of all possible execution paths,

```
Interpreter >> pushNewArrayBytecode
  | size array |
  size := self fetchByte.
  array := (self fitsInNewSpace: size)
    ifTrue: [ self fastInstantiateArray: size ]
    ifFalse: [ self slowInstantiateArray: size ].

  0 to: size – 1 do: [ :i | | element |
    element := self popStack.
    array at: size – i – 1 put: element ].

  self push: array
```

```
JITCompiler >> gen_pushNewArrayBytecode
  | size r1 jump jumpFalse |
  "Initialization code"
  ...
  "Fetch is staged"
  size := self fetchByte.

  "Spilling"
  self absStackSpill.

  "Runtime branch"
  jumpFalse := self gen_fitsInNewSpace: size.
  self gen_fastInstantiateArray: size.
  jump := self gen_jump.
  jumpFalse target: self Label.
  "Runtime call"
  self
    gen_runtimeCall: #slowArray: with: size.

  "Runtime merge point"
  jump target: self Label.

  "Pops are staged"
  0 to: size – 1 do: [ :i | | abstractValue |
    abstractValue := self absPop.
    self
      gen_store: ResultReg
      at: size – i – 1
      put: abstractValue
  ].

  "Push the array"
  self absPush: ResultReg
```

Figure 1.2: Bytecode for creating a new array with N elements popped from the stack. Implementation in the interpreter (left) and in the baseline JIT compiler (right).

falling back to the interpreter for less common or complex cases. While traditional optimizing JIT compilers guide these decisions using runtime profiling information, an ahead-of-time metacompiler must provide mechanisms to statically identify and exclude slow paths during metacompilation to avoid generating unnecessary code.

Runtime calls from machine code. *The metacompiler must statically decide when the JIT generated code should interact with the runtime system.* Interpreters

are typically implemented in the same language as the rest of the runtime components (*e.g.* memory managers, JIT compilers), allowing seamless integration. In contrast, JIT-compiled code requires explicit management of runtime interactions. For example, when JIT-compiled code triggers a garbage collection (GC), the compiler must ensure that registers are spilled onto the stack (*deoptimization*) so that the GC can properly traverse and identify live objects. The metacompiler must therefore insert appropriate runtime calls and ensure correct calling conventions and state synchronization between compiled code and the runtime system.

Different staging levels. *The metacompiler must statically decide at which stage (ahead-of-time, JIT-compile-time, or run-time) each operation should be executed, producing a residual program with staged computations.* Interpreters operate as one-stage machines, directly executing code; meanwhile, compilers are two-stage machines, where decisions are made statically and encoded into the generated code. JIT compilers introduce an additional layer, where certain properties or values are known constants at JIT compile time, enabling further optimizations. The metacompiler must analyze the interpreter definitions and determine which computations can be staged to JIT-compile time, while leaving others to be executed at run time in the residual machine code.

1.4 Contributions

This section presents the research and technical contributions of our work.

1.4.1 Research Contributions

1. **Metacompilation of baseline JITs.** We designed and developed Druid, a source-to-source ahead-of-time metacompiler that generates a baseline JIT compiler directly from interpreter definitions. Druid performs metacompilation entirely ahead-of-time, introducing no runtime overhead, which aligns with the goals of baseline JIT compilers that prioritize fast compilation. Our metacompiler automatically *stages* computations that are independent of runtime information so they can be evaluated once at JIT compile time. We also describe the intrinsics and annotations that guide the metacompilation process, allowing language developers to influence code generation without requiring compiler expertise.

2. **Targeting fast abstract-interpreter baseline JITs.** Druid leverages an existing dynamic machine code generator to benefit from a production-grade JIT infrastructure, enabling us to focus on frontend generation rather than

low-level backend concerns. In particular, Druid targets a compile-time abstract interpreter, a well-established approach to building fast baseline JIT compilers. This abstract interpreter applies, in a single pass, optimizations such as register allocation, constant propagation, and instruction scheduling. We describe the additional metacompilation passes required to satisfy the invariants and constraints imposed by abstract interpreters, enabling Druid to generate frontend code fully compatible with these optimizations.

1.4.2 Technical Contributions

1. **Evaluation on a real-world virtual machine.** We implemented and evaluated our approach in the production Pharo VM. We generated a baseline JIT compiler frontend from the Pharo interpreter and measured its performance in terms of runtime speed, JIT compilation overhead, and development effort. To assess the quality of our generated JIT compiler, we compared it against a handwritten baseline JIT maintained by VM experts for over a decade.

2. **Evaluation of abstract-interpreter baseline JITs.** We studied the benefits and trade-offs of abstract-interpreter-based baseline JIT compilers by comparing two equivalent implementations: one using abstract interpretation and the other using just *direct translation*. Both were obtained through metacompilation, using the same interpreter definitions and infrastructure, which reduced implementation noise and experimentation effort. This controlled setting allowed us to fairly measure the impact of JIT-time abstract interpretation on code size, performance, and compilation overhead.

1.5 Thesis Outline

The thesis is organized into three main parts.

The first part introduces the state of the art on building efficient virtual machines and introduces the Pharo VM, used as the experimental context of this work.

The second part presents our metacompiler prototype, called Druid, and evaluates it in terms of runtime performance, development effort, and JIT compilation overhead.

The third part describes the extensions made to Druid to target abstract interpreter-based baseline JIT compilers and evaluates the impact of abstract interpretation at JIT compile time in a fair and controlled way.

The chapters are organized as follows:

- **Chapter 1** introduces the goals of multi-tier virtual machines and the challenges of generating baseline JIT compilers from interpreter definitions.

Part 1: **Building Efficient Virtual Machines**

- **Chapter 2** reviews the state of the art in JIT compiler generation, highlighting existing metacompilation strategies and identifying the gaps regarding baseline JIT compilers.

- **Chapter 3** presents the Pharo Virtual Machine and the Cogit baseline JIT compiler framework. We describe the VM's architecture, based on an interpreter and a JIT compiler, and explain how our metacompilation approach integrates into this structure.

Part 2: **Metacompilation of Baseline JIT Compilers**

- **Chapter 4** details the design of Druid, our metacompiler for generating baseline JIT compiler frontends from interpreter definitions. We describe its architecture, implemented optimizations, and the intrinsic functions and annotations used to guide the metacompilation process within the Pharo VM.

- **Chapter 5** evaluates the performance of the generated baseline JIT compiler, comparing it against both the interpreter-only execution and the existing handwritten baseline JIT compiler. We assess the performance in terms of runtime execution, development effort, and JIT compilation overhead. We also analyze the gap between the generated and handwritten JIT compilers and discuss the impact of missing optimizations.

Part 3: **Metacompilation targeting Abstract Interpretation**

- **Chapter 6** introduces the abstract interpreter implemented in the Cogit framework. We explain how it is used by the baseline JIT compiler to perform optimizations at JIT compile time.

- **Chapter 7** presents the extensions made to Druid to target Cogit's abstract interpreter. We describe the additional compilation passes required to respect the invariants imposed by the abstract interpreter infrastructure.

- **Chapter 8** evaluates the performance of the baseline JIT compiler generated by Druid when targeting the Cogit abstract interpreter. We perform a fair comparison by generating different versions of the JIT compiler from the same interpreter, each applying different levels of optimizations.

- **Chapter 9** summarizes the contributions of this thesis and outlines possible directions for future work.

Appendixes

- **Appendix A** list all the intrinsic functions and metadata supported by our prototype.

- **Appendix B** list all the traditional optimizations implemented in our multi-pass compiler infrastructure.
- **Appendix C** presents several examples about the metacompilation of the Pharo VM instructions produced by our prototype.
- **Appendix D** list all the programs used as benchmarks in our evaluations.

1.6 List of Publications

The list of papers published in the context of the thesis is listed below in chronological order:

Interpreter Register Autolocalisation: Improving the performance of efficient interpreters.
[Polito 2022a] - Workshop Paper
Guillermo Polito, Nahuel Palumbo, Pablo Tesone, Soufyane Labsari, Stéphane Ducasse.
More VM International Workshop, Apr 2022, Porto, Portugal.
https://hal.science/hal-03594766.

Ordering Optimisations in Meta-Compilation of Primitive Methods.
[Palumbo 2022a] - Workshop Paper
Nahuel Palumbo, Guillermo Polito, Pablo Tesone, Stéphane Ducasse.
FAST Workshop 2022 on Smalltalk Related Technologies, Nov 2022, Buenos Aires, Argentina.
https://hal.science/hal-04237932.

Metacompilation of Baseline JIT Compilers with Druid.
[Palumbo 2025b] - Journal Paper
Nahuel Palumbo, Guillermo Polito, Stéphane Ducasse, Pablo Tesone.
The Art, Science, and Engineering of Programming, Feb 2025, Prague, Czech Republic.
https://doi.org/10.48550/arXiv.2502.20543.

Are Abstract-interpreter Baseline JITs Worth it? – An Empirical Evaluation through Metacompilation.
In submission - Conference Paper (Core A)
Nahuel Palumbo, Guillermo Polito, Stéphane Ducasse, Pablo Tesone.
International Symposium on Code Generation and Optimization, Jan 2026, Sydney, Australia

I also contributed to researching other components of virtual machines and programming language implementations:

Selecting Semi-permanent Object Candidates in Dynamically-Typed Reflective Languages.
[Palumbo 2022b] - Poster
Nahuel Palumbo, Pablo Tesone, Guillermo Polito, Stéphane Ducasse.
Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes, Sep 2022, Brussels, Belgium.
https://hal.science/hal-03785536.

Heap Fuzzing: Automatic Garbage Collection Testing with Expert-Guided Random Events.
[Polito 2023] - Conference Paper (Core A)
Guillermo Polito, Pablo Tesone, Jean Privat, Nahuel Palumbo, Stéphane Ducasse.
International Conference on Software Testing, Apr 2023, Dublin, Ireland.
https://hal.science/hal-03962007.

Garbage Collector Tuning in Pathological Allocation Pattern Applications.
[Palumbo 2023] - Workshop Paper
Nahuel Palumbo, Sebastian Jordan Montaño, Guillermo Polito, Pablo Tesone, Stéphane Ducasse.
International Workshop on Smalltalk Technologies, Aug 2023, Lyon, France.
https://hal.science/hal-04225588.

Clean Blocks at the Opal Compiler.
[Palumbo 2025a] - Workshop Paper
Nahuel Palumbo, Marcus Denker.
International Workshop on Smalltalk Technologies, Jul 2025, Gdansk, Poland.
https://hal.science/hal-05299729.

Acknowledgment

This thesis was funded by Inria's Action Exploratoire AlaMVic.

Part I

Building Efficient Virtual Machines

CHAPTER 2

State of the Art

Contents

The development of efficient virtual machines (VMs) involves complex engineering tasks, particularly when incorporating just-in-time (JIT) compilation. While traditional VM development requires significant manual effort to implement interpreters and JIT compilers separately, recent research has explored techniques to reduce this burden through automated or generative approaches. This thesis explores the generation of a baseline JIT compiler via ahead-of-time metacompilation from an interpreter. This chapter surveys a range of such approaches, focusing on those most relevant to our work.

We organize this chapter by first describing and categorizing the main components of execution engines in efficient VMs. We then review frameworks and tools designed to reduce the development effort by promoting the reuse of components across VM implementations. Finally, we present generative techniques, particularly for JIT compilers, which align with the metacompilation strategy adopted in this thesis.

2.1 Multi-tier Virtual Machines

A virtual machine is commonly organized as a multi-tier engine because it usually contains more than one execution engine with different properties [Holzle 1994, Glück 1997, Geoffray 2010, WebKit Community 2025]. On these machines, program execution usually switches between being interpreted, pre-compiled, or compiled just-in-time.

In this section, we catalog and describe the various characteristics of each execution engine relevant to this thesis.

2.1.1 Interpreters, Compilers and JIT Compilers

We start by defining an interpreter, a compiler, and a JIT compiler [Klint 1981, Torczon 2007, Titzer 2024].

Interpreter. An interpreter is a one-stage engine: it directly executes source code to get the result. It easily interacts with other runtime components, such as the memory manager and the garbage collector, because they are defined at the same level of abstraction. Thus, by living in the same execution model, they are easier to develop than a compiler. For execution, an interpreter receives the program source and program input and retrieves the result, as described in the next equation.

$$result = interpreter(source, input) \tag{2.1}$$

Compiler. The goal of a compiler is to translate a program written in a source language to an equivalent program written in a target language. The goal of such translation is to reduce execution time, assuming that a program will be compiled once and executed many times. In other words, the goal is to satisfy the following equation.

$$target = compiler(source) \tag{2.2}$$

$$result = target(input) \tag{2.3}$$

$$t_{interpreter}(source, input) > t_{target}(input) \tag{2.4}$$

Where $t_{interpreter}(source, input)$ represents the time interpreting $source$ with the input $input$, and $t_{target}(input)$ represents the time executing the compiled $source$ in $target$ language with the same input $input$.

A compiler is a two-stage engine: instead of executing source code, it generates residual code with the same behavior as the source code. Thus, they are harder to develop than interpreters, due to such translation requires, for example, understanding the transformation of code between two different representations/abstraction levels (*e.g.* stack bytecode to register machine code).

JIT compiler. JIT compilers take such constraints to the extreme because compilation happens at run time. The following equation introduces $t_{compiler}(source)$, which represents the time compiling $source$ to a $target$ language. When compilation time is considered at run time, the sum of the compilation time and the execution time should not be greater than the interpreter time for compilation to be worth it.

$$t_{interpreter}(source, input) > t_{compiler}(source) + t_{target}(input) \tag{2.5}$$

Developing a JIT compiler increases the complexity of ahead-of-time compilers due to optimizations based on runtime information and code generation during program execution. For example, these engines differentiate between which parts of the source program could be executed at each stage (compile time or run time), and decide when the compiler should stop and emit a runtime function call (*e.g.* for running the GC or deoptimization).

In multi-tier machines, only the first tier (typically the interpreter) must support the entire language implementation. Other tiers usually trade off space for execution time, compiling only common paths and falling back to previous tiers when rare paths are hit.

2.1.2 Examples of Multi-tier Virtual Machines

Today, several industrial-grade virtual machines implement multiple execution engines with varying levels of optimization. These systems are complex and require deep expertise as well as significant maintenance effort. In this section, we present the most relevant examples for this thesis.

SELF. The SELF programming language VM was a pioneer in implementing multiple optimization levels during execution. It includes both a non-inlining compiler and an optimizing compiler, orchestrated through profiling information and type feedback [Chambers 1989, Chambers 1991, Ungar 1992, Holzle 1994].

JVM. The HotSpot Java Virtual Machine is a sophisticated execution engine with more than 20 years of maturity. It includes an interpreter and two JIT compilers, C1 and C2, which apply different levels of profiling and perform both low- and high-level optimizations [Lindholm 2014].

V8. V8 is the JavaScript VM used in Chrome and Node.js. It implements a complex multi-tier pipeline composed of an interpreter (Ignition), a baseline JIT (Sparkplug), a fast optimizing compiler (Maglev), and a heavyweight optimizing compiler (TurboFan) [V8 2017, V8 2021, V8 2022].

SpiderMonkey. SpiderMonkey, developed by Mozilla and used in the Firefox web browser, features a multi-tier execution engine that includes a bytecode interpreter, a baseline interpreter (with inline caches), a baseline compiler, and an optimizing compiler (WarpMonkey) [SpiderMonkey nd].

JavaScriptCore. JavaScriptCore (JSC), developed by Apple and used in Safari, implements a four-tier architecture consisting of a low-level interpreter, a baseline JIT, a data flow graph (DFG) compiler, and a "Faster Than Light" (FTL) optimizing compiler [WebKit 2014].

2.1.3 JIT Compilers by Optimization Strategy

Equation 2.5 expresses the fundamental trade-off in JIT compilation: the balance between minimizing $t_{compiler}(source)$ and $t_{target}(input)$. JIT compilers can be classified according to their strategy in this trade-off: baseline JITs prioritize fast compilation at the expense of missed optimization opportunities, while the optimizing ones invest more time in aggressive optimizations to reduce execution time.

Baseline JIT Compilers. Baseline JIT compilers aim to minimize $t_{compiler}$, often at the cost of reduced optimization. They typically eliminate interpretation overhead and apply lightweight, one-pass optimizations. This makes them suitable for code that is executed infrequently or where quick startup is important [Titzer 2024].

Optimizing JIT Compilers. In contrast, optimizing JIT compilers focus on minimizing t_{target} by spending more time during compilation to apply advanced optimizations. This results in highly optimized code and improved execution performance, but incurs higher compilation overhead [Cooper 2002].

Optimizing and baseline JIT compilers are combined in practice to achieve the best of both worlds: a lot of code that is rarely executed is compiled with baseline JIT compilers, and key parts of the code that are commonly executed are compiled with optimizing JIT compilers [Holzle 1994, Glück 1997, WebKit Community 2025].

2.1.4 JIT Compilation Scopes

As discussed in previous sections, JIT compilers are triggered during execution when a *hot path* is detected. Consequently, a key design decision for JIT implementors is selecting the compilation scope, *i.e.* which portions of the source program are visible and available for compilation at any given moment.

Although various strategies exist in practice, this section categorizes the most common and relevant approaches for this thesis.

Method-based This approach compiles code at the granularity of methods or functions. It enables optimizations within a method but does not consider interprocedural optimizations across method boundaries.

This technique is often combined with inlining and multiple recompilation strategies, allowing the system to generate separate versions of a method depending on its calling context for splitting fast and slow paths [Chambers 1989, Lindholm 2014, Miranda 2011].

Trace-based. Trace-based compilation follows the actual execution path through the program, starting and ending at arbitrary points. A compiled trace includes executions from many methods. The resulting compiled trace is then optimized for the specific input values encountered during tracing [Bala 2000, Bebenita 2010, Gal 2006].

Since only one path of each conditional branch is compiled, runtime guards are inserted to validate assumptions. If any assumption fails, execution must be deoptimized and resumed in another tier or trace.

Instruction-based. In metacompilation approaches, the compilation scope shifts to the level of individual VM instructions. Here, the interpreter's semantic definitions are metacompiled into equivalent definitions for a JIT compiler. This technique requires the metacompiler to analyze and understand the semantics of each instruction at the host language in which the interpreter is implemented [Bolz 2009, Würthinger 2013, Futamura 1999, Birkedal 1994].

This fine-grained approach enables systematic generation of baseline JIT compilers and allows applying optimizations at the instruction level, independent of higher-level program structure.

2.2 VM Frameworks

The growing complexity of VMs has motivated the development of frameworks and tools that assist in building and maintaining them, particularly their runtimes [Marr 2008]. Prior work has explored ways to reuse components and optimizations across language implementations, enabling features developed for one VM to be adapted or integrated into another. In this section, we describe several of these approaches and frameworks.

2.2.1 Reusable Components

We present relevant approaches to reuse VM components in different language implementations.

VMKit. VMKit [Geoffray 2010] proposes a reusable, language-independent substrate for the construction of managed runtime environments (MREs), leveraging the LLVM compiler infrastructure for JIT compilation and the MMTk toolkit for garbage collection [Blackburn 2004]. Its design enables the development of VMs for multiple high-level languages with low implementation effort, providing abstractions for key runtime components such as memory management, threading, and code generation. The system includes implementations of both a Java Virtual Machine (JVM) and a Common Language Infrastructure (CLI) runtime, demonstrating its generality and applicability.

In contrast to earlier systems that embed language-specific assumptions or demand substantial low-level engineering, VMKit isolates language-independent runtime services behind a consistent and reusable API. VMKit consolidates these principles into a practical framework that enables rapid prototyping of complete managed runtimes. Its integration with LLVM facilitates portable and efficient JIT compilation across architectures, avoiding the need to implement separate backend infrastructures.

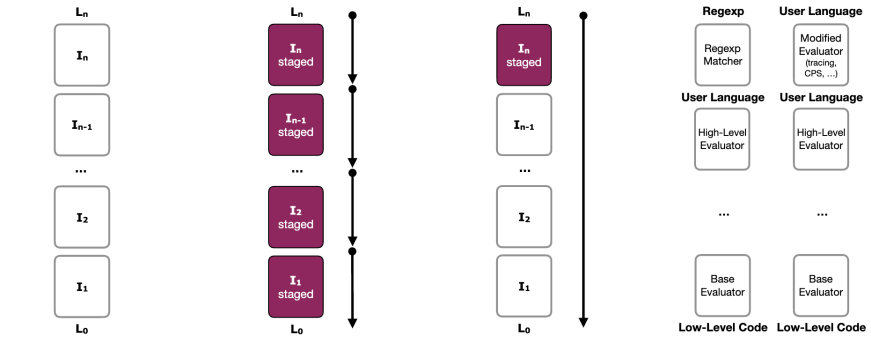
Performance evaluation on standard benchmarks shows that VMs built with VMKit achieve decent execution times. This confirms that the abstraction provided by VMKit does not incur prohibitive overhead, and supports its claim as a viable foundation for both experimental and production-grade language runtimes.

Tower of Interpreters. The tower of interpreters, as defined by Smith [Smith 1982], is based on metalevel interpretation. A program is interpreted by an interpreter, which is interpreted by a meta-interpreter and so on, possibly ad infinitum. This leads to a tower of interpreters, where each level defines the semantics of the program (interpreter of application) it interprets.

The tower of interpreters presents a model where one can define and redefine the semantics of the program it is executing. We can modify the behavior of our program (in the floor zero) by jumping one level above it in the tower and modifying the interpreter running on that floor. In the same sense, we can jump one level above this interpreter to change also its behavior and so on. This allows the indirect modification of a program's behavior *i.e.* a change in an interpreter in a level n changes the behavior of the interpreter in the level n - 1, which impacts the interpreter below it and so on, up to the base level. Smith's tower of interpreters model is flexible and coherent regarding the manipulation of the reflective behavior of a program.

The tower does not present a limit on the number of interpreters we can stack, presenting a problematic infinite potential. This idea collides with the non-infinite resources in the real world. On one hand, limited memory prevents us from having a tower of infinite interpreters running at the same time. On the other hand, above certain limit of interpreters, this approach becomes too slow and impractical, as discussed in [Malenfant 1996, Malenfant 1992].

Recent work by Amin and Rompf [Amin 2017] introduces a technique for collapsing towers of interpreters by transforming an interpretive stack into a compiler, as presented in Figure 2.1, *i.e.* a single-pass executable that removes interpretive overhead altogether. They formalize this via a multi-level λ-calculus enriched with staging constructs and stage polymorphism, which enable an evaluator to dynamically choose between interpreting or generating code depending on runtime parameters. In this system, multiple self-interpreting layers, including those with semantic modifications, can be mechanically collapsed into direct base-level code, eliminating layers of indirection while preserving reflective semantics.



(a) Tower L_n to L_0 (b) n-Pass Compiler (c) One-Pass Compiler (d) Examples

Figure 2.1: Collapsing tower of interpreters via compilation and possible scenarios to be applied.

Hierarchical VM. Yermolovich *et al.,*. [Yermolovich 2009] have built a VM for the LUA language based on a hierarchical layering of virtual machines, where a dynamic language interpreter (*guest VM*) runs atop an already optimizing trace-based virtual machine (*host VM*). This multi-layer architecture steps on existing host VM services, like JIT compilation and memory management, without requiring new low-level implementations for each guest language.

The key insight is that by embedding the guest interpreter inside a performant host VM, one obtains a meta-virtual machine: the guest benefits from the host's tracing JIT, which identifies hot paths across interpreter functions and compiles them into optimized machine code. This achieves substantial performance gains with minimal engineering effort.

However, layered designs can introduce overhead due to abstraction mismatches between the guest and host runtimes. To mitigate this, the authors propose the use of explicit hints embedded in the guest interpreter to guide the host VM's trace optimizer. These hints provide metadata about guest-level control flow, such as loop boundaries or type information, helping the host VM produce better-optimized traces despite the semantic gap. Empirical results show that while some hand-

optimized VMs may still outperform in specific cases, the layering approach—when combined with hints—offers respectable performance, achieving a balance between ease of implementation and execution efficiency.

2.2.2 Metacircular VMs

Metacircular VMs are VMs programmed in the same language they support, *e.g.* a Java VM written in Java or a Smalltalk VM written in Smalltalk. This approach is based on the principles of high-level low-level programming *i.e.* expressing low-level concerns using high-level languages [Frampton 2009]. Metacircular VMs benefit from the abstraction power and tooling of a high-level language to manipulate their own VMs. This also means that during the build-time of a metacircular VM, we can express the manipulations of its application runtime in terms of the high-level language.

Pharo and Squeak VMs The Squeak VM is an open-source, metacircular virtual machine for the Smalltalk language [Goldberg 1983, Deutsch 1989, Ingalls 1997, Miranda 2018]. It includes a generational garbage collector, an efficient interpreter, and a baseline JIT compiler called *Cogit* [Ungar 1984,Miranda 2011]. The VM was later extended with an adaptive optimization system known as *Sista* [Béra 2017].

The Squeak VM represents a high-performance Smalltalk implementation. Its bytecode set enables bytecode compilers to inline message-based control flow operations such as ifTrue:ifFalse:[1]. The interpreter achieves fast bytecode execution by avoiding the traditional "spaghetti stack" of reified contexts, opting instead for a linear and more efficient representation. It further incorporates optimizations such as *Static Type Prediction*, indirect threading, and a global method lookup cache. The JIT compiler performs fast translation to machine code and employs Polymorphic Inline Caches (PICs) [Hölzle 1991a].

The Squeak VM is implemented in *Slang*, a restricted subset of Smalltalk that is exported to C for compilation into the final VM binary. Slang is limited to constructs that can be expressed using standard C code. However, it preserves the structural advantages of Smalltalk through class-based organization, and it can be executed within a simulator environment that allows developers to interact dynamically with a running VM instance. This setup enables immediate feedback: VM developers can modify the source code of the interpreter or runtime and observe the effects directly within the simulator.

The **Pharo VM** is a fork of the Squeak VM developed for the Pharo programming language [Ducasse 2022]. It inherits Squeak's core framework and optimizations while introducing several extensions, including automatic *autolocalization* of interpreter registers, a permanent object space, and an improved infrastructure for testing [Polito 2021, Polito 2022b, Polito 2022a, Palumbo 2022b, Polito 2023].

[1] In Smalltalk, control flow operations are expressed as messages to boolean objects.

In this thesis, *we use the Pharo VM as the experimental context for our prototype*. Further implementation details are discussed in the following chapters.

Jikes. Jikes (*a.k.a.* Jalapeño) is a metacircular research VM for Java written in Java [Alpern 2000]. The Jikes VM features several different garbage collectors and does not execute bytecodes but directly compiles to native code. With metacircularity in mind, Jikes implements its VM components in Java itself, as well as using a high-level, low-level programming framework. The Jikes VM had performance as a major goal; direct unobstructed interaction with the low-level world is necessary using a specialized framework, where a complex adaptive optimization system is implemented.

Frampton *et al.,* present a high-level low-level framework packaged as *org.vmmagic*, which is used as a system interface for Jikes. This framework introduces highly controlled low-level interaction in a statically typed context. This framework provides a memory-oriented API to manipulate runtime entities at VM generation time, which is used to implement VM concerns. Once the VM is compiled to native code, the interface exposed by the framework is also compiled and not accessible from Java programs executed on top of the Jikes VM.

Maxime. Maxine is a metacircular Java VM [Wimmer 2013] focused on an efficient developer experience. Typically, VM frameworks focus on abstraction at the code level, which should yield simpler code and thus help reduce development efforts. However, in most situations, the programmer is still forced to use existing general-purpose tools, for instance, to debug the VM. In contrast, the Maxine VM provides dedicated tools to interact with the VM in development. Maxine uses abstract and high-level representations of VM-level concepts and consistently exposes them throughout the development process. The Maxine project follows an approach where reflection is used at compile-time, *i.e.* once the VM is generated, the metacircular property of the VM is lost. However, during development, Maxine tools provide a live interaction with the complete state of the running VM artifact while debugging it.

Klein. Klein is a metacircular VM for the Self programming language that has no separation into VM and language [Ungar 2005]. A main difference between Klein and the already seen metacircular VM projects is that the reification of the VM-level elements survives the code generation and compilation time. The VM structures are exposed to the language as Self objects, thus allowing their manipulation from the application runtime. Klein also supports advanced mirror-based [Bracha 2004] debugging tools to inspect and modify a remote VM.

In addition to the advances in reflection and metacircularity, Klein focuses on fast compilation turnarounds and incremental modifications for a responsive development process. This project proved that it is possible and build a language runtime

without the classical separation of the language side and the VM. From the literature presented about the Klein project, we see a strong focus on the improvements of the development tools.

2.3 Generative Techniques

In previous sections, we discussed various approaches aimed at reducing the complexity involved in developing and maintaining VM components. In this section, we present techniques based on the generation of VM components, which avoid the responsibility of writing these components from developers to tools that generate them automatically. These generative approaches share a common idea of decoupling the definition of the language semantics from low-level VM optimizations, thus increasing the maintainability of language implementations by a division of concerns.

Interpreter Generation. Ertl *et al.,* are responsible for the Vmgen and Tiger projects [Ertl 2002, Casey 2005]. They propose a framework to generate efficient interpreters in C from a declarative definition of the language semantics expressed in a DSL. The tool enables developers to describe the semantics of instructions and define how those instructions are translated into low-level C code, allowing the generator to produce efficient threaded interpreters based on instruction-level profiling (*e.g.* instruction-transition graph in Figure 2.2).

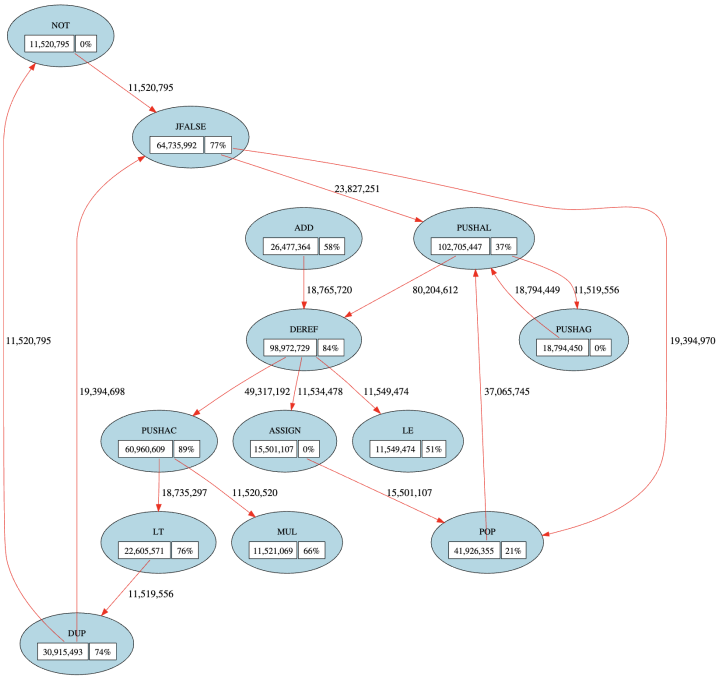


Figure 2.2: Instruction-transition graph generated by Tiger [Casey 2005].

Users can also declare optimizations such as instruction specialization, stack caching, and super-instructions, which are staged into the interpreter during generation. This approach enables a clear separation of concerns between language designers, who define the instruction semantics, and interpreter implementors, who are responsible for developing low-level execution and optimization strategies.

Partial Evaluation. In 1971, Futamura proposed to use partial evaluation to generate compilers via self-application, which are now known as the *Futamura projections* [Futamura 1999]. Since then, many authors have studied this approach for various programming languages [Beckman 1976, Jones 1993].

These projections describe how a general-purpose partial evaluator can be applied to an interpreter to automatically derive a compiler. The first projection specializes a program with respect to some of its input; the second specializes an interpreter with respect to a source program, effectively compiling it; and the third specializes the partial evaluator itself with respect to the interpreter, generating a compiler. There were also studies around the fourth Futamura Projection [Glück 2010], where compiler generators are derived from different specializers, generating compilers with different characteristics for the same language.

Figure 2.3 illustrates a specializer and two compiler generators (left). The figure shows the subject language N, the target language N, and the implementation language S of the N/S-specializer S_{new}. The two compiler generators have an additional language, written in the center of their T-diagrams, namely the target language of the generating extensions that they produce (S in the case of cog_{old}, N in the case of cog_{new}). Once cog_{new} is derived, it is used to turn an L-interpreter int written in N into an L-to-N-compiler $comp$ written in S (right).

Partial evaluation has inspired a wide range of research on semantics-directed compilation techniques and language implementations. A recent example is LuaAOT, a compiler for Lua derived from the interpreter [Musso Gualandi 2021].

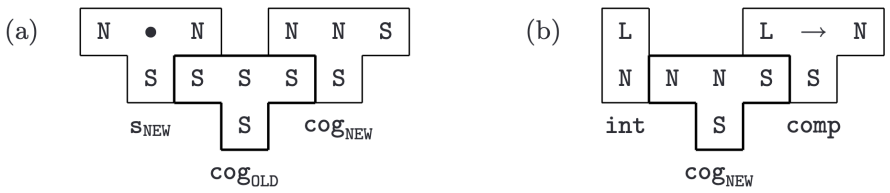


Figure 2.3: (a) Transformation of a specializer S_{NEW} into a compiler generator Cog_{NEW}. (b) Transformation of an interpreter `int` into a specialized compiler `comp`.

Cogen. The handwritten Cogen approach proposes a technique for producing efficient offline partial evaluators by manually writing compiler generators based on a binding-time annotated interpreter [Leone 1993, Birkedal 1994]. Unlike automatic generation methods, which often suffer from poor performance or limited control over specialization, this method relies on human-guided transformation of an interpreter into a generating extension tailored for a specific source language. The resulting Cogen produces residual programs through specialization, guided by binding-time analysis. An architecture of this approach is presented in Figure 2.4, where a compiler generator $cogen$ receives a general program p (as an interpreter) and generates a compiler $p - gen$. Then, it is used to compile programs written in p into a specialized form $Pin1$. Finally, the compiled program receives the input (data) to produce an output (result).

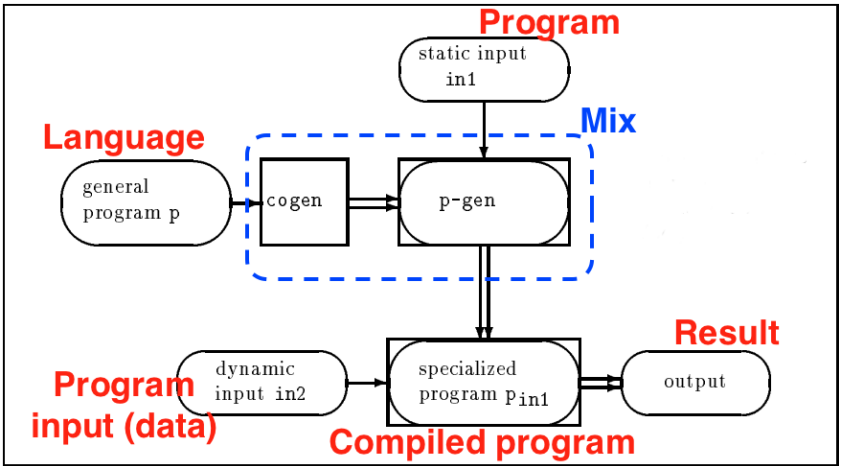


Figure 2.4: Abstract architecture of a Generator of Program Generators.

This approach differs from earlier self-applicable partial evaluators, such as those found in the Futamura projections or systems like Mix, which emphasize automation but often at the cost of transparency and efficiency. Handwritten Cogens offer finer control over code generation, enabling more aggressive optimizations and cleaner residual code, particularly when targeting realistic languages or complex semantics. By explicitly encoding specialization behavior into the Cogen, developers can avoid common pitfalls such as code explosion, inefficient residualization, and redundant computations.

The effectiveness of this method is demonstrated through practical case studies, including logic and functional languages, where the handwritten Cogen yields residual programs with significantly better performance than those produced by fully automatic partial evaluators. Additionally, the framework supports modularity, allowing Cogen developers to systematically incorporate domain-specific optimizations, making it a robust and flexible approach for generating efficient specialized programs.

Targetting compiler backends. VCODE [Engler 1996] provides a fast, portable, and retargetable interface for emitting machine code at runtime using a low-level API; it achieves dynamic code generation speeds competitive with hand-tuned generators without relying on an intermediate representation. In another way, Dynamo [Bala 2000] dynamically optimizes existing native binaries during execution by detecting *hot traces* and re-optimizing them in a software code cache, with no need for source modification or specialized compiler support. SPUR [Bebenita 2010] also offers a trace-based JIT compiler specifically for CIL (Common Language Infrastructure), applying tracing techniques to generate native code from bytecode execution traces.

These systems provide efficient runtime code generation and optimization without requiring the compiler developer of high-level languages to implement low-level backend infrastructure from scratch. By emitting code directly to such engines, one can take advantage of mature support for instruction selection, register allocation, and machine code emission. Targeting these backends allows compiler developers to focus on language semantics and high-level transformations, while reusing sophisticated execution and optimization mechanisms already present in the host environment. This separation of concerns enables rapid development of high-performance compilers with reduced implementation complexity.

2.4 JIT Compiler Generators

The main contribution of this thesis is a method for generating a baseline JIT compiler from an interpreter definition. In this section, we review prior work related to JIT compiler generation, including frameworks, techniques, and tools that are closely aligned with our approach. These contributions explore various strategies for deriving JIT compilers, with the common goal of reducing the engineering effort required to build efficient execution engines.

GraalVM. GraalVM is a polyglot virtual machine written in Java, built around a method-based optimizing JIT engine capable of aggressive inlining and speculative optimizations [Würthinger 2013, Duboscq 2013]. The project also includes the Truffle framework [Wimmer 2012], a language implementation framework based on self-optimizing AST interpreters. Truffle applies partial evaluation, an instance of the first Futamura projection, at run time to specialize AST nodes and trigger JIT compilation for hot paths. As illustrated in Figure 2.5, this technique combines runtime specialization with aggressive JIT optimization, enabling high-performance execution of dynamic languages such as Smalltalk, JavaScript, Python, and Ruby [Wöss 2014, Seaton 2014, Marr 2016, Niephaus 2019].

GraalVM also provides Native Image, a tool that compiles Java applications into standalone native executables using ahead-of-time (AOT) compilation under closed-world assumptions [Wimmer 2019, Kozak 2023]. This approach achieves

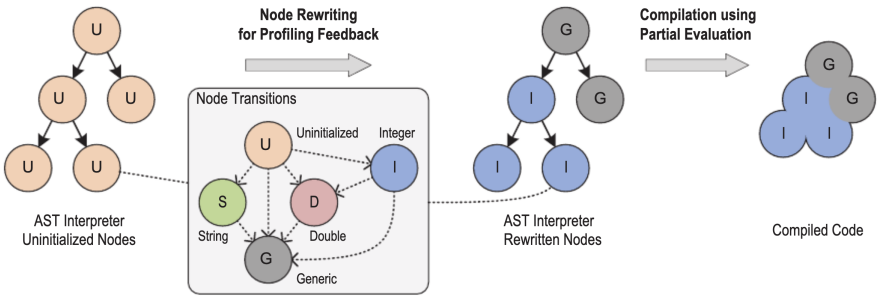


Figure 2.5: Node rewriting for profiling feedback and partial evaluation performed in the Graal VM.

near-instant startup times and reduced memory footprints by precomputing bytecode compilation, while still supporting dynamic features. It is particularly useful in contexts requiring predictable performance and low-latency startup, such as cloud functions.

A recent contribution by Latifi *et al.,* introduced CompGen [Latifi 2021], an ahead-of-time approach that precomputes the IR generator from a Truffle interpreter using the second Futamura projection. This technique enables the generation of a JIT compiler before execution, without relying on live runtime information.

Pypy. PyPy is a high-performance implementation of the Python programming language featuring a meta-tracing JIT compiler automatically derived from interpreters written in RPython [Rigo 2006, Bolz 2009]. RPython is a statically analyzable subset of Python that is translated to C and primarily used to implement language interpreters for performance reasons [Ancona 2007].

The PyPy framework generates two interpreters from the RPython source. The first is a regular interpreter used for standard execution; the second is a profiled interpreter that enables the meta-tracing infrastructure to observe and record the execution of interpreted programs. When a hot path is detected, control switches from the regular to the profiled interpreter, which traces execution at the meta-level. It records the operations performed by the interpreter while executing a specific user program. The resulting trace is then optimized and compiled into machine code, producing a specialized version of the interpreter path and effectively functioning as an optimizing tracing JIT compiler.

Because tracing operates over the interpreter itself, language implementers do not need to write a separate JIT compiler, it is automatically derived from the interpreter's behavior. This separation between the interpreter and the trace-based compilation layer enables rapid development of high-performance virtual machines, not only for Python, but also for other dynamic languages [Bolz 2008, Bruni 2009, Marr 2015].

LuaJIT Remake. LuaJIT Remake (LJR) is an experimental version of LuaJIT implemented using Deegen, a metacompiler that automatically generates both an interpreter and a baseline JIT from a single bytecode semantic description in C++ [Xu 2023, Xu 2024]. The generated JIT tier compiles fast and emits machine code employing a copy-and-patch technique [Xu 2021]. Key optimizations include automatic inline caching, hot-cold code splitting, and self-modifying inline cache stubs.

LJR is a promising approach to automatically produce interpreters and JIT tiers that rival expert hand-written implementations. This demonstrates that modern metacompilation frameworks can yield state-of-the-art VM components from declarative semantics, reducing engineering effort and maintenance costs while delivering production-grade performance.

2.5 Conclusion

In recent decades, significant research has emerged to reduce the engineering effort required to implement high-performance virtual machines. In this chapter, we reviewed the main techniques and approaches that have driven the field toward increasing automation and component reuse. Understanding these designs and their underlying principles not only clarifies the landscape of existing solutions but also motivates the approach taken in this thesis: to leverage instruction-level interpreter definitions and automatically generate baseline JIT compilers.

As presented in the following chapters, our work builds on the idea of ahead-of-time metacompilation through a handwritten **metacompiler called Druid**. We evaluate this approach using the Pharo VM, a metacircular virtual machine written in Slang, a subset of Pharo translatable to C. Since the VM already includes a mature JIT infrastructure, called Cogit, we reuse its backend components by targeting our metacompilation process to it.

We propose a source-to-source transformation, applied during VM development, that automatically generates the baseline JIT compiler frontend. Importantly, Druid runs ahead-of-time and does not contribute to runtime overhead. During the VM distribution process, both the handwritten and generated code are compiled together, ensuring seamless integration.

CHAPTER 3

The Pharo Virtual Machine

Contents

In this chapter, we present the architecture of the Pharo VM execution engine, composed of a stack-based interpreter and the Cogit baseline JIT compiler, which are the source and target of our metacompilation process, respectively. This serves as a technical background and provides a common vocabulary for a better understanding of our metacompilation approach and prototype implementation, explained in later chapters.

In this thesis, we use the Pharo Virtual Machine (VM) as the experimental context to evaluate our metacompilation approach. Pharo is an open-source, fully dynamic, object-oriented language, based on the Smalltalk technology, implemented in terms of bytecode and primitive methods [Goldberg 1983,Ingalls 1997,Ducasse 2022].

The Pharo VM is a fork of the OpenSmalltalk-VM, with more than 20 years in production [Miranda 2018]. It implements optimizations including a generational

garbage collector, an efficient interpreter, and JIT compilation [Ungar 1984, Hölzle 1991b, Miranda 2011, Polito 2022a, Palumbo 2022b]. The Pharo VM is competitive at the industrial level, with more than 30 companies and universities using the language.

3.1 A Stack Machine

Pharo is an object-oriented language based on late-binding messages that execute methods. Methods contain bytecode and/or primitive instructions designed to operate on a stack machine. An example is shown in Figure 3.1 for the addition method + in SmallInteger.

For clarity of presentation, this section introduces some vocabulary points.

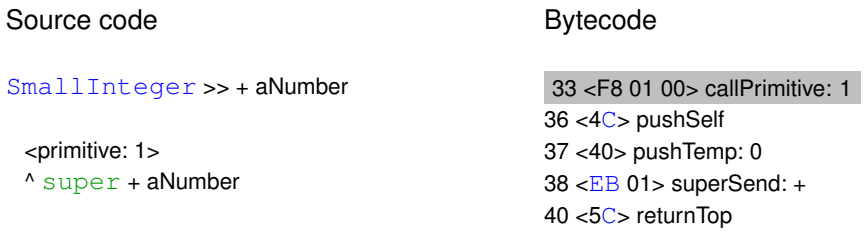


Figure 3.1: Primitive method for small integer addition and its fall-back bytecode.

Primitive Methods. Primitive methods are standard Pharo methods annotated to call a primitive instruction. For example, the method for integer addition, annotated with <primitive 1>, is shown on the left side of Figure 3.1.

Primitive Instructions. Primitive instructions are native operations exposed by the virtual machine, similar to native methods in other language implementations [Alpern 2000]. They implement fundamental functionality such as arithmetic operations and object allocation. By design, primitive instructions can *fail*, in which case execution continues with a fall-back bytecode method. Otherwise, when the primitive succeeds, the execution engine returns control to the caller with the result. The primitive instruction for integer addition is presented later in Figure 3.5.

Bytecode Instructions. Bytecode instructions serve as the VM intermediate language. Pharo source code is compiled to a sequence of stack-based bytecodes, *e.g.* push instance variable, duplicate the top of the stack [Béra 2014]. An example of a bytecode stream inside a method is shown on the right side of Figure 3.1.

From the perspective of this thesis, we consider both bytecodes and primitives as *VM instructions*.

Some functionality in the VM is implemented both as bytecode *and* as primitive instructions, duplicated for performance reasons, *e.g.* integer addition and comparisons (see Section 3.3.4). Moreover, primitive instructions are typically larger and more complex than bytecodes [Polito 2022b].

Definition 2 (Instruction handler) *An instruction handler is a function that implements the semantics of a VM instruction within a specific execution tier.*

In the Pharo VM, both the interpreter and the JIT compiler frontend are implemented using a table-dispatch approach. Bytecode and primitive opcodes are mapped to functions called *instruction handlers* (see Section 3.3.2 and Section 3.4.2).

3.2 Slang: the VM Framework

The Pharo VM is implemented in *Slang*, a restricted subset of Pharo that can be exported to C and compiled into the production VM binary. In practice, Slang works as a high-level C preprocessor, limited to constructs that can be expressed in standard C.

Although Slang shares the syntax of Pharo, it is semantically constrained to expressions that can be *resolved statically* at compilation time. For instance, Slang supports and propagates type annotations that are not present in ordinary Pharo code. Semantically, Slang is much closer to C than to Pharo. However, its class-based structure helps organize the source code, and VM tests can be executed directly in the Pharo environment by running the Slang methods in simulation before exporting them to C.

Additionally, Slang applies optimizations such as inlining and code transformations during C generation. One major transformation is the inlining of dispatch tables into the interpreter loop, as we show in the next section.

Figure 3.2 illustrates the generation of C code from Slang. On the left, we show the Slang definition of the primitive addition along with helper methods annotated with types. On the right, we show the generated C code, after inlining and type propagation have been applied.

All the transformations and optimizations presented in this thesis are written in Slang. The corresponding C code of the VM is just generated to compile a production-ready binary.

3.3 The Interpreter

The interpreter is a stack-based machine with handlers for all 240 bytecode and 263 primitive instructions that define the Pharo language. It interacts with other

```
Interpreter >> primitiveAdd
  | maybeSmallInteger maybeSmallInteger2 result |

  maybeSmallInteger := self stackValue: 0.
  maybeSmallInteger2 := self stackValue: 1.

  (objectMemory isIntegerObject:
      maybeSmallInteger)
    ifFalse: [ ^ self primitiveFail ].
  (objectMemory isIntegerObject:
      maybeSmallInteger2)
    ifFalse: [ ^ self primitiveFail ].

  result := self
    sumSmallInteger: maybeSmallInteger
    withSmallInteger: maybeSmallInteger2
    ifOverflow: [ ^ self primitiveFail ].

  self pop: 2 thenPush: result

Interpreter >> stackValue: offset
  ^ stackPages unsignedLongAt: stackPointer + (
      offset * objectMemory wordSize)

Interpreter >> pop: nItems thenPush: oop
  "stacks grow down."
  | sp |
  <inline: true>
  <returnTypeC: #void>
  <var: #sp type: #'char *'>
  stackPages
    unsignedLongAt: (sp := stackPointer + ((
      nItems - 1) * objectMemory wordSize))
    put: oop.
  stackPointer := sp
```

```
/* InterpreterPrimitives>>#
   primitiveAdd */
static void primitiveAdd(void)
{
  DECL_MAYBE_SQ_GLOBAL_STRUCT;
  sqInt a;
  sqInt b;
  sqInt maybeSmallInteger;
  sqInt maybeSmallInteger2;
  sqInt result;
  sqInt result2;
  char *sp;

  maybeSmallInteger =
    unsignedLongAt(GIV(stackPointer) + (0 *
      BytesPerWord));
  maybeSmallInteger2 =
    unsignedLongAt(GIV(stackPointer) + (1 *
      BytesPerWord));

  if ((((maybeSmallInteger & 7) == 1))) { ... }
  if ((((maybeSmallInteger2 & 7) == 1))) { ... }

  a = maybeSmallInteger >> 3;
  b = maybeSmallInteger2 >> 3;
  result2 = a + b;
  { ... }

  result = ((((usqInt) result2 ) << 3) | 1);
  unsignedLongAtput(
    (sp = GIV(stackPointer) + ((2 - 1) *
      BytesPerWord)),
    result);
  GIV(stackPointer) = sp;
}
```

Figure 3.2: Code written in Slang (left) and the corresponding generated C code (right) after inlining and type propagation.

components of the runtime system (*i.e.* garbage collector), manages green threads (user-level scheduling), and applies machine-dependent optimizations such as indirect threading and the *autolocalization* of global variables inside the interpreter loop [Ertl 2003, Stallings 2011, Polito 2019, Polito 2022a].

In this section, we focus on how the language semantics are implemented in the interpreter, *i.e.* through the instruction handlers. These handler definitions are the input to our metacompilation process, which generates semantically equivalent

definitions for the JIT compiler.

At the same time, we introduce the interpreter-level optimizations most relevant to this thesis. These optimizations make metacompilation more challenging: not only because they increase the complexity of the interpreter code, but also because they make it harder for the automatically generated JIT compiler to match, or surpass, the performance of such a carefully optimized interpreter.

3.3.1 The Interpreter Loop

Slang generates the interpreter loop based on the instruction handlers defined in two dispatch tables: one for bytecodes and another for primitives.

Figure 3.3 shows parts of the dispatch tables. In the bytecode table, the size of the bytecode and the range of opcode values are defined for each handler. In contrast, the primitive table simply maps primitive numbers to their handlers.

Bytecodes

```
  "(size from to handler)"
#(
  ( 1 0 15 pushRcvrVar )
  ...
  ( 2 226 226 extPushRcvrVar )
  ...
  ( 3 252 252 storeRemTemp  )
  ...
)
```

Primitives

```
  "(number handler)"
#(
  ( 1 primitiveAdd )
  ( 2 primitiveSubtract )
  ...
  ( 62 primitiveSize )
  ...
)
```

Figure 3.3: Bytecode and primitive dispatch tables defined in the interpreter.

The generated main interpreter loop fetches the next instruction, evaluates a switch, and calls the corresponding handler in the case statement, as illustrated in Figure 3.4.

3.3.2 Interpreter Handlers

Each handler implements the language semantics for its corresponding instruction. For example, Figure 3.5 shows the definition of the primitiveAdd instruction, corresponding to the primitive method in Figure 3.1 that adds two tagged small integers.

The handler proceeds as follows:

1. It first fetches the two arguments from the top of the stack.
2. It checks whether both arguments are tagged small integer objects. If not, the primitive fails.
3. If both are integers, it sums their values. In case of overflow (when the result cannot be represented as a tagged small integer), the primitive fails.

```
void interpret() {
  while(true) {
    int instruction = fetchNextInstruction();
    switch(instruction) {
      case INSTRUCTION_1:
        ... handler1() ...
        break;
      case INSTRUCTION_2:
        ... handler2() ...
        break;
      ... }}}
```

Figure 3.4: Main interpreter loop.

4. Otherwise, the two arguments are popped from the stack and the result is pushed to it.

As described in Section 3.1, if the primitive fails, execution falls back to the bytecode implementation presented on the right side of Figure 3.1.

```
1  Interpreter >> primitiveAdd
2   | maybeSmallInteger maybeSmallInteger2 result |
3
4   maybeSmallInteger := self stackValue: 0.
5   maybeSmallInteger2 := self stackValue: 1.
6
7   (objectMemory isIntegerObject: maybeSmallInteger)
8     ifFalse: [ ^ self primitiveFail ].
9   (objectMemory isIntegerObject: maybeSmallInteger2)
10    ifFalse: [ ^ self primitiveFail ].
11
12  "Check for overflow"
13  result := self
14    sumSmallInteger: maybeSmallInteger
15    withSmallInteger: maybeSmallInteger2
16    ifOverflow: [ ^ self primitiveFail ].
17  self pop: 2 thenPush: result
```

Figure 3.5: Primitive instruction handler for integer addition.

3.3.3 Method Execution

The key operation in object-oriented programs is the message send. Before executing a message send bytecode, the receiver and all arguments are pushed onto the stack.

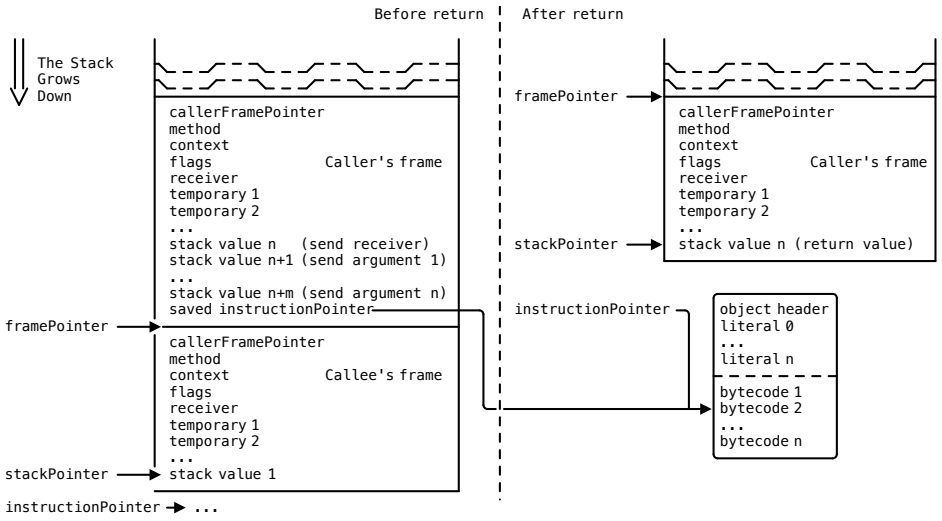


Figure 3.6: Transition of the interpreter stack between a method activation by a message send and the return instruction.

The message send handler performs the *lookup* of the method to execute and then activates it. Activating a method implies creating a new stack frame for the callee and preparing the interpreter to execute its instructions. Execution of the method continues until a return instruction is encountered.

When a return instruction is interpreted, the interpreter restores the three key variables: framePointer, stackPointer, and instructionPointer from the caller's frame, and then destroys the callee frame. In the caller's frame, the receiver and arguments are popped, and the return value is pushed onto the stack.

Figure 3.6 illustrates the mutation of the interpreter stack across this process: from message send, through method activation, and finally method return.

3.3.4 Interpreter Optimizations

The Pharo VM interpreter has accumulated numerous optimizations over more than 20 years of development. In this section, we focus on the optimizations most relevant to this thesis.

The primary source of overhead in interpreted execution is the *fetch & dispatch* of bytecode instructions. Several optimizations are therefore aimed at making this cycle faster. Another significant cost in object-oriented virtual machines arises from *method lookup*, that is, locating the method to execute for a given message send.

Indirect Threading. It is well known that a major source of interpretation overhead comes from the way VM instructions are dispatched. Indirect threaded code

is the most efficient dispatch strategy, relying on hardware branch predictors and avoiding the cost of a single `switch` with indirect branches [Ertl 2003, Brunthaler 2010].

The Pharo VM implements indirect threading by fetching the next instruction and jumping directly to the corresponding handler using goto. Thus, the code for each case in the interpreter loop from Figure 3.4 is transformed into the threaded style, as exemplified in Figure 3.7.

```
int instruction = fetchNextInstruction();
switch(instruction){
  case INSTRUCTION_1:
  INSTRUCTION_1:
    ... handler1() ...
    goto fetchNextInstruction();
    break;
  case INSTRUCTION_2:
  INSTRUCTION_2:
  ...
}
```

Figure 3.7: Indirect threading of the interpreter loop.

Global Lookup Cache. The most frequent operation in object-oriented languages is the message send. This involves *looking up* the method to execute based on the receiver type, a costly process that is the main target of optimizations in such VMs. A common strategy is a global lookup cache that avoids traversing the class hierarchy on every send [Deutsch 1984].

The Pharo VM implements a global cache indexed by the receiver class identifier and the method selector. Thus, the first time that an object of class A receives a message m, the system looks up the method to execute by traversing the class hierarchy and fills the cache. Then, following message sends of m to objects of the class A will find the entry in the cache, providing fast access to the invoked method.

Static Type Prediction. *Static Type Prediction* is a message-send optimization that predicts the types of the receiver and arguments based on the message selector. The key idea of this optimization is to statically inline common and predictable sends into a fast path guarded by type checks. If the guard fails, execution falls back to the slow path, which performs a normal message send [Holzle 1994].

For example, the Pharo VM applies this optimization to common integer operations such as arithmetic, comparisons, and bitwise manipulations. For instance, although developers can redefine the addition operator + in their own classes, most call sites will only observe integers at run time. Thus, the bytecode compiler emits

a special *Static Type Prediction*-optimized bytecode for addition, illustrated with a simplified version of the interpreter code in Figure 3.8.

The instruction handler implements two execution paths: a fast path guarded by type checks on both operands, and a slow path that performs a regular message send. If an overflow occurs, the semantics require delegating to user code, so the slow path is taken in that case as well.

```
1  Interpreter >> bytecodeAdd
2    | rcvr arg result |
3    rcvr := self stackValue: 1.
4    arg := self stackValue: 0.
5    "Type check both operands"
6    (objectMemory areIntegers: rcvr and: arg) ifTrue: [
7      result := (objectMemory integerValueOf: rcvr) + (objectMemory
8      integerValueOf: arg).
9      "Type check the result.
10         If overflow, roll back to the slow path"
11     (objectMemory isIntegerValue: result) ifTrue: [
12     self pop: 2 thenPush: (objectMemory integerObjectOf: result).
13     ^ self fetchNextBytecode ]].
14   "Slow path: send"
15   messageSelector := self specialSelector: 0.
16   argumentCount := 1.
17   self normalSend
```

Figure 3.8: Bytecode handler for addition with *Static Type Prediction* for small integers.

Super-Instructions. In addition to *Static Type Prediction*, some interpreter handlers also implement *Super-Instructions*.

Source code	Bytecode

`SomeClass >> someMethod`
`  | a b |`
`  a < b ifTrue: [ ... ]`
	33 <40> pushTemp: 0
	34 <41> pushTemp: 1
	35 <62> send: <
	36 <C1> jumpFalse: 39
	...
	39 <58> returnSelf

Figure 3.9: Optimized bytecode for a control flow expression ifTrue: preceded by a comparison. The predictable bytecode sequence is interpreted as one super-instruction.

This optimization inlines the execution of the next predictable bytecode directly into the fast path of the current instruction. As a result, both the dispatch and execution of the next instruction are avoided if the sequence matches a known pattern.

For example, in Figure 3.9 we show the compiled bytecode for a comparison expression < followed by a control flow expression ifTrue:. As typically conditional branches are preceded by comparisons, all comparison operators are implemented as special bytecodes optimized with *Static Type Prediction*. Their handlers perform a *look-ahead* to check the following bytecode and, if the next instruction is a conditional jump, the interpreter directly performs the jump, bypassing the intermediate push.

Figure 3.10 illustrates the handler code for the comparison operator <. The handler implements, combining *Static Type Prediction* and *Super-Instructions*, two levels of optimization:

- When both operands are integers and the next instruction is a conditional jump, the handler performs the jump directly (line 12).
- When both operands are integers but the next instruction is not a jump, the result of the comparison is pushed to the stack (line 16).
- When operands are not integers, the slow path executes a regular message send (line 20).

```
1  Interpreter >> bytecodeLessThan
2    | rcvr arg aBool |
3
4    rcvr := self stackValue: 1.
5    arg := self stackValue: 0.
6    "Type check both operands"
7    (objectMemory areIntegers: rcvr and: arg) ifTrue: [ | next |
8      "Assume next bytecode is jumpIfFalse (99%)"
9      next := self fetchByte.
10     (next == JUMP_FALSE and: [ (rcvr < arg) not ]) ifTrue: [ | offset |
11       offset := ...
12       ^ self jump: offset ].
13     "Cases for jump if true and other boolean values"
14     ...
15     "Not followed by jump, but integers nevertheless"
16     ^ self pushBoolean: rcvr < arg ].
17   "Slow path: send"
18   messageSelector := self specialSelector: 2.
19   argumentCount := 1.
20   self normalSend
```

Figure 3.10: Bytecode handler for comparisons with *Static Type Prediction* for integers and *Super-Instructions* for jump instructions. The *look-ahead* to the next bytecode is highlighted.

3.4 Cogit: The Baseline JIT Compiler

The Pharo VM includes a baseline JIT compiler, in production for more than 10 years, called *Cogit* [Miranda 2011]. Cogit is a method-based dynamic machine-code generator that takes as input a bytecode/primitive Pharo method and outputs executable machine code for a specific architecture.

The baseline JIT supports the compilation of all 240 bytecodes that define the language, as well as the most frequently used 130 primitive instructions. If a method contains unsupported instructions, Cogit does not compile it. Instead, execution falls back to the interpreter.

Cogit implements specific JIT optimizations such as *Polymorphic Inline Caches* (PICs), a register-based calling convention, and peephole optimizations. It also reimplements the interpreter's optimizations, such as *Static Type Prediction* and *Super-Instructions*.

3.4.1 Architecture Overview

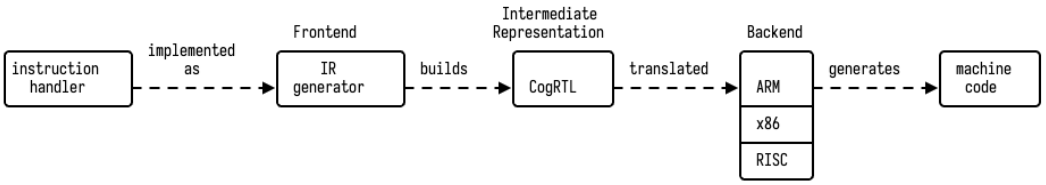


Figure 3.11: Architecture overview of the Cogit framework implementing an instruction handler.

The Cogit JIT compiler follows a classical compiler architecture:

Frontend. Defines the instruction handlers. Each handler encodes the language semantics of the corresponding bytecode or primitive and emits intermediate representation (IR) instructions for the method.

Intermediate Representation. Cogit uses a custom IR called *CogRTL*, which is register-transfer based. During IR generation, Cogit performs lightweight optimizations, including peephole optimizations and local transformations.

Backend. Translates the CogRTL instructions into machine code for a concrete target architecture. Cogit currently provides backends for x86, x86-64, ARM, ARM64, and RISC-V architectures.

In Figure 3.11, we illustrate the Cogit framework architecture in the context of implementing an instruction handler. The frontend emits CogRTL code, which can be optimized before being lowered to the backend. Finally, the backend maps CogRTL instructions using virtual registers to concrete machine instructions for the running architecture.

3.4.2 Compiler Handlers: IR Generators

Analogous to the interpreter, the JIT compiler defines its handlers in two dispatch tables, one for bytecode instructions and another for primitive instructions. While interpreter handlers directly execute the instruction, JIT compiler frontend handlers generate nodes in the intermediate representation (IR). We refer to these compiler handlers as *IR generators*.

The Cogit dispatch tables are partially presented in Figure 3.12.

Bytecodes

```
 "(size from to handler extra*)"
#(
  ( 1 0 15 genPushRcvrVar isInstVarRef )
  ...
  ( 1 128 143 genSendSelector0Args isMapped )
  ...
  ( 1 176 183 genShortUncondJump branch short )
  ...
  ( 2 226 226 genExtPushRcvrVar isInstVarRef )
  ...
  ( 3 252 252 genStoreRemTemp needsFrame 0 )
  ...
)
```

Primitives

```
  "(number handler numArgs)"
#(
  ( 1 genPrimitiveAdd  1 )
  ( 2 genPrimitiveSubtract  1 )
  ( 3 genPrimitiveLessThan  1 )
  ( 4 genPrimitiveGreaterThan  1 )
  ( 5 genPrimitiveLessOrEqual  1 )
  ...
  ( 62 genPrimitiveSize 0 )
  ( 63 genPrimitiveStringAt 1 )
  ( 64 genPrimitiveStringAtPut 2 )
  ...
)
```

Figure 3.12: Bytecode and primitive dispatch tables defined in the JIT compiler.

They include the same information as the interpreter, presented in Section 3.3.2, plus additional information required by the compiler backend. For bytecode instructions, annotations such as isInstVarRef or isMapped are included and later used during compilation to specialize or optimize the generated code. For primitive instructions, the table explicitly specifies the number of arguments expected by each handler.

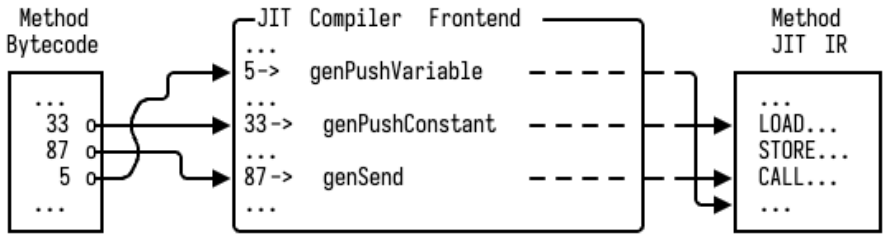


Figure 3.13: The JIT compiler frontend uses the bytecode dispatch table to map opcodes to IR generators and build the IR for the method.

The IR for a bytecode method is produced by iterating over its bytecode sequence and invoking the corresponding IR generators, as illustrated in Figure 3.13. The IR generators emit CogRTL instructions that encode the semantics of the bytecode or primitive while enabling subsequent optimizations and lowering to backend-specific code.

3.4.3 The CogRTL Intermediate Language

IR generators are defined using a register-transfer language, namely *CogRTL*. The Cogit Register Transfer Language (CogRTL) is designed as an Intel-inspired two-address code IR (2AC). Cogit uses a fixed set of registers with virtual names (*e.g.* TempReg, ReceiverResultReg, ClassReg), which are statically mapped ahead of time to machine registers by JIT compiler developers.

Figure 6.1 presents a simplified version of the CogRTL syntax. The language models a stack machine and manipulates two kinds of values: registers (r) and numeric constants (c). Input programs are composed of *concrete*, assembly-like instructions (*e.g.* Move, Push, Load). Extending it to other instructions is trivial.

$$
\begin{array}{rcl}
r & = & \text{a virtual register} \\
c & = & \text{a constant} \\
v & = & r \mid c \\
Input & = & inst^* \\
inst & = & \mathsf{Push}(v) \mid \mathsf{Pop}(r) \mid \mathsf{Move}(v, r) \mid \mathsf{Load}(c, r, r) \mid \mathsf{Ret}
\end{array}
$$

Figure 3.14: CogRTL syntax.

Figure 3.15 shows the IR generator, written in CogRTL, for the bytecode instruction that duplicates the top of the stack. Line 2 should be interpreted as Load(offset, baseRegister, destinationRegister), which loads the value at the top of the stack (SPReg refers to the StackPointer in Figure 3.6) into TempReg, a virtual register reserved for temporary values. Line 3 pushes the value in TempReg back onto the stack, duplicating the top element. Finally, line 4 returns 0, indicating that IR generation finished successfully.

```
1 JITCompiler >> genDuplicateTopBytecode
2   self gen_Load: 0 r: SPReg r: TempReg.
3   self gen_Push: TempReg.
4   ^ 0
```

Figure 3.15: CogRTL for the instruction that duplicates the top of the stack.

At JIT compile time (*i.e.* during program execution), if the *dup* bytecode is found in the bytecode stream, Cogit invokes genDuplicateTopBytecode to generate the IR for the method under compilation. Later, the IR is lowered to backend-specific machine code.

Virtual Register Allocation. IR generators are written as code templates in CogRTL. In practice, Cogit behaves mostly like an assembler: JIT compiler frontend developers implement the instruction entirely in CogRTL and perform manual register allocation. This design favors simplicity and predictability, but it places the burden of register usage and correctness directly on the VM developers.

3.4.4 Runtime Interactions

JIT compilation techniques focus on compiling the most frequently used features of a language, *i.e.* the *fast paths*.

One example, mentioned earlier in Section 3.4, is that the baseline JIT in the Pharo VM does not support all primitives defined in the language, but only the most common ones. If a method contains a primitive that is not supported by the JIT compiler, the system detects this during JIT compilation and falls back to executing the method directly in the interpreter.

Another case arises when an instruction handler contains both fast and slow paths. In such situations, only the fast path is compiled, protected by guard checks. If a slow path is taken during execution, control is transferred back to the runtime system. This transfer is implemented as a runtime call from JIT-compiled code via a *deoptimization instruction*.

Definition 3 (Deoptimization Instruction) *A deoptimization instruction performs a runtime call from JIT-compiled code to resume execution in the interpreter.*

Deoptimization Points. We call *deoptimization points* the locations where a non-compiled slow path is reached in the JIT-compiled machine code of a method. At such points, a runtime call is generated to continue execution in the interpreter. Before transferring control, values in registers are spilled to the stack, the current method frame is transformed into an interpreter frame, and the program counter variable is remapped from machine code back to the corresponding bytecode in the method.

Figure 3.16 shows the IR generator for the jumpFalse bytecode (see Figure 3.9). This bytecode must jump if the receiver is falseObject, or continue with the next bytecode if it is trueObject. But, if the receiver is not a Boolean object, it must raise a MustBeBoolean exception.

Since exception throwing is not part of the fast path, the JIT frontend only compiles the boolean cases inside guard checks: line 9 jumps to the target if the

receiver is falseObject, and line 12 continues execution if it is trueObject. For non-boolean receivers, a runtime call is generated at line 14 to execute the slow path in the interpreter.

```
1  JITCompiler >> genJumpIfFalse
2    | targetPC isTrueObject boolean |
3    targetPC := ...
4    boolean := objectMemory falseObject.
5    "Receiver is falseObject?"
6    self gen_PopR: TempReg.
7    self gen_SubConstant: boolean R: TempReg.
8    "Jump to target"
9    self gen_JumpZero: targetPC.
10   "Else, is trueObject?"
11   self gen_CmpCq: (objectMemory trueObject – objectMemory falseObject) R: TempReg.
12   isTrueObject := self gen_JumpZero: 0.
13   "Throw MustBeBoolean exception"
14   self gen_CallMustBeBooleanFor: boolean.
15   "Continue execution of next bytecode"
16   isTrueObject jmpTarget: (self annotateBytecode: self Label).
17   ^ 0
```

Figure 3.16: IR generator for the bytecode jumpFalse. Only *fast paths* are compiled, for cases where the receiver is a boolean object. Otherwise, a runtime call is generated (line 14) to throw an exception, which is considered a *slow path*.

In the Cogit framework, runtime calls are implemented using *trampolines*: small machine-code routines that follow the system calling convention when transferring control between generated machine code and the runtime. Before entering a runtime call, register values must be spilled to the stack, since runtime functions assume that all data resides on the stack. For instance, the garbage collector traverses the stack to locate object roots. This stack spilling is not performed automatically by trampolines, it must be handled explicitly by the JIT compiler frontend code.

Transferring execution from JIT-compiled code back to the interpreter also requires knowing which bytecode within the method corresponds to the deoptimization point in the machine code. To support this, Cogit maintains a mapping between deoptimization points and bytecode positions. The annotateBytecode: function (line 16 in Figure 3.16), together with the isMapped annotation in the dispatch table (Figure 3.12), is used by the Cogit framework to build this mapping.

3.4.5 Compiler Optimizations

The Cogit framework implements a series of optimizations well known in the literature. The metacompilation approach proposed in this thesis targets the JIT compiler generation within the Cogit framework, aiming to reuse these existing optimizations.

In this section, we introduce the most relevant Cogit optimizations for our work. Additional optimizations performed at JIT compile time, based on abstract interpretation, are detailed in Chapter 6.

Polymorphic Inline Caches. The Cogit JIT compiler infrastructure has built-in implementation for dynamically bound message sends using (polymorphic) inline caches (ICs) [Hölzle 1991a]. ICs are implemented by patching machine code at runtime. They support unbound call sites linked to a *miss* trampoline, monomorphic call sites linked directly to methods, and polymorphic or megamorphic call sites linked to machine code stubs.

Calling Convention. As explained in Section 3.3.3, the interpreter uses a stack-based calling convention to invoke a method from a message send. Cogit, instead, uses a register-based calling convention for JIT-compiled methods, where the receiver and arguments are passed in well-known virtual registers (*i.e.* ReceiverResultReg, Arg0Reg, Arg1Reg). This optimization avoids memory accesses by keeping values in registers, thus speeding up method invocations.

In cases where the compiler cannot allocate enough registers to hold all arguments, either because the method has many arguments or the target architecture provides a limited number of registers, the JIT compiler spills the values to the stack, falling back to the interpreter-style calling convention.

Peephole Optimizations. The Cogit framework includes a *compile-time abstract interpreter* to perform peephole optimizations across bytecodes inside a method. This optimization eliminates unnecessary stack operations (Push and Pop) by keeping intermediate values in registers. Details of this optimization are presented in Chapter 6.

3.5 Conclusion

In this chapter, we introduced the Pharo VM, used as the experimental context for the prototype presented in this thesis. We described the execution engine of the Pharo Virtual Machine, composed of an efficient interpreter and a baseline JIT compiler, both refined over more than 20 years of development.

The **interpreter** is a stack machine defined by bytecode and primitive instruction handlers. It implements several optimizations, including indirect threading, a global lookup cache, *Static Type Prediction*, and *Super-Instructions*.

The **baseline JIT compiler** is implemented on top of the Cogit framework. Its instruction handlers are IR generators written in CogRTL as part of the compiler frontend. The backend implements JIT compiler optimizations such as polymorphic inline caches, peephole optimizations, and a register-based calling convention, together with architecture-specific translations.

These descriptions provide the necessary background for the next chapters of this thesis. On one side, metacompiling an optimized interpreter is a challenge due to the increased complexity of its instruction handlers. On the other side, automatically generating a baseline JIT compiler that is as performant as the fine-tuned, handwritten version developed by VM experts over more than a decade is an ambitious long-term goal.

It is also important to stress that the Pharo VM is an industrial-grade implementation, used by many companies and universities worldwide. Pharo is far from being a *toy* or research-only language. We expect that the ideas presented in this thesis can contribute to addressing the real-world needs of language users.

In the next chapter, we introduce *Druid*, our metacompiler prototype. We describe the process of automatically generating a Cogit frontend (IR generators) from the interpreter's instruction handlers, together with the changes required in the Pharo VM to make metacompilation feasible.

Part II

Metacompilation of Baseline JIT Compilers

CHAPTER 4

Metacompilation with Druid

Contents

This thesis explores the automatic generation of baseline JIT compilers through *metacompilation*.

In this chapter, we present our approach and explain how a baseline JIT compiler for the Pharo VM is automatically generated. We introduce the architecture of the metacompiler and describe the main components involved in the transformation from **one-stage** interpreter definitions into **two-stage** JIT compiler code.

The chapter begins by describing our proposal for baseline JIT generation, building on the design of the Pharo VM introduced in the previous chapter. It then presents the architecture of our prototype, called *Druid*, and concludes by detailing the key aspects of the metacompilation process that enable the automatic generation of a JIT compiler.

4.1 Metacompilation Process Overview

In Chapter 3, we presented the design of the Pharo VM execution engine. In particular, we show how the instruction handlers are implemented in both the interpreter and the baseline JIT compiler, to drive execution and compilation. A key characteristic of this architecture is that each instruction handler must be written twice: once for the interpreter and once for the JIT compiler.

Beyond the inherent differences between writing an interpreter (a one-stage execution engine) and a compiler (a two-stage execution engine), the language semantics must remain consistent across both implementations. This duplication not only increases development effort but also introduces the risk of semantic divergence between the interpreter and the JIT compiler [Polito 2022b].

Our solution is to **generate the instruction handlers for the baseline JIT compiler frontend from those defined in the interpreter**, ensuring semantic consistency while reducing implementation effort and maintenance cost.

To illustrate this process, Figure 4.1 shows the implementation of the bytecode that duplicates the top of the stack, as defined in the Pharo VM. The bytecode is implemented both in the interpreter (left) and in the JIT compiler frontend (right) using CogitRTL (see Section 3.4.3).

```
Interpreter >> bytecodeDup
  self push: self stackTop
```

```
JITCompiler >> gen_bytecodeDup
  self MoveMw: 0 r: SPReg R: TempReg.
  self PushR: TempReg
```

Figure 4.1: Interpreter and baseline JIT handlers for the bytecode *duplicate top*.

Our solution is a source-to-source metacompiler, called *Druid*. Druid takes as input the handler defined in the Pharo interpreter (left of Figure 4.1) and produces as output the corresponding handler for the baseline JIT compiler frontend, written in CogitRTL (right of Figure 4.1).

4.2 Druid in a Nutshell

To drive the experiments of this thesis, we designed and developed *Druid*[1], an **ahead-of-time source-to-source metacompiler** that generates JIT compiler frontend code from the Pharo VM interpreter. Conceptually, Druid follows the design of a compiler generator (Cogen) [Birkedal 1994]. For simplicity and control, we implemented Druid as a hand-written compiler rather than a partial evaluator [Glück 1997, Futamura 1999]. To prototype these ideas, we extended the Pharo VM interpreter framework [Ingalls 1997, Miranda 2011, Miranda 2018].

[1] https://github.com/Alamvic/druid visited on 2025-09-06.

Figure 4.2 shows an overview of our solution. On the left, we illustrate the Pharo VM runtime, where instruction handlers are executed based on bytecode instructions in a method (see Chapter 3). Methods are either interpreted (top) or JIT compiled (bottom) depending on *hot-path* detection. On the right, Druid performs ahead-of-time metacompilation of the interpreter handlers into semantically equivalent JIT compiler handlers.

In summary, Druid takes as input an interpreter instruction handler and produces the corresponding handler for the baseline JIT compiler frontend. This metacompilation process is performed entirely ahead of time, making it transparent to a production-ready virtual machine.

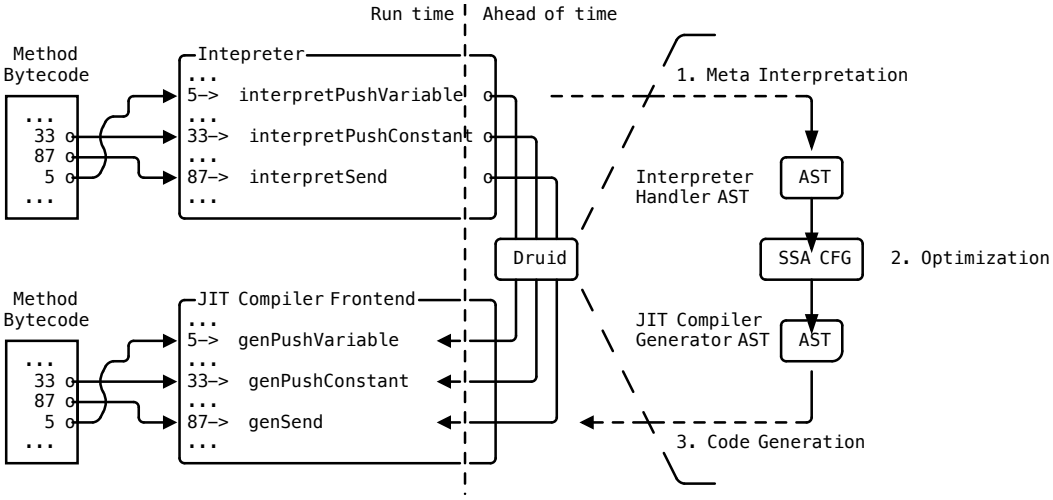


Figure 4.2: Overview of the solution presented in this thesis. On the left, the Pharo VM runtime executes bytecode instructions inside a method, either by interpretation (top) or JIT compilation (bottom). On the right, Druid performs ahead-of-time metacompilation of interpreter handlers into JIT compiler handlers.

Druid is internally structured as a traditional multi-pass compiler, extended with transformations specific to metacompilation. Its architecture can be divided into a *frontend*, a *midend*, and a *backend* [Aho 2006, Torczon 2007].

Druid's Frontend: Metainterpretation. (Figure 4.2, step ①) The frontend takes as input the Pharo methods implementing an interpreter instruction handler. It builds an intermediate representation (IR) by performing an *abstract interpretation* of the method's abstract syntax tree (AST). This process is guided by *intrinsics* and aggressive inlining. Because this step effectively interprets the interpreter, we refer to it as *metainterpretation*.

Druid's Midend: Optimizations. (Figure 4.2, step ②) At the core of Druid's pipeline is its IR, represented as a control flow graph (CFG) in static single-assignment (SSA) form [Cytron 1991, Rastello 2022]. Standard compiler optimizations such as constant folding, dead code elimination, and global value numbering are applied [Appel 2002, Torczon 2007, Palumbo 2022a]. Additionally, Druid introduces metacompilation-specific transformations, including *staging* (see Section 4.3.3), which separates compile-time computations from runtime computations.

Druid's Backend: Code generation. (Figure 4.2, step ③) The backend targets Cogit, the baseline JIT compiler framework described in Section 3.4. Druid lowers the IR by leaving the SSA form, assigning virtual registers defined by the Cogit framework, and finally emitting the compiler handler in CogitRTL form. The resulting method is integrated into the JIT compiler frontend and deployed as part of the VM.

4.3 The Magic behind the Druid

In the previous section, we introduced our prototype *Druid*, engineered as a traditional multi-pass compiler with extensions dedicated to the metacompilation of instruction handlers. In this section, we present those extensions in detail, explaining how Druid makes the automatic generation of JIT compiler code possible.

Figure 4.3 illustrates the metacompilation process using Druid. At the top, we show the structure of a VM's source code, divided into framework components, language semantics, and automatically generated code. At the bottom, we depict the transformations performed by Druid to translate interpreter definitions into JIT compiler definitions.

In this schema, VM developers implement the frameworks to write the interpreter and the JIT compiler, including optimizations and profiling mechanisms. Language implementors use these frameworks to define the interpreter instruction handlers, effectively specifying the language semantics. They use *intrinsic functions* and annotations to guide the metacompilation process. Druid takes each interpreter instruction handler and metacompiles it into a JIT compiler instruction handler, targeting the existing JIT compiler framework. Finally, a production-ready VM source code is produced, including both handwritten and metacompiled code.

A key consequence of this architecture is that language implementers no longer need expertise in the internals of the JIT compiler, which is typically more complex than the interpreter [Ertl 2003]. Furthermore, Figure 4.3 highlights that Druid is *not* part of the deployed VM: it operates ahead-of-time, taking interpreter code as input and producing JIT compiler code as output. Druid is therefore not tied to the host language runtime, introduces no metacompilation overhead at runtime, and can evolve independently from the VM implementation.

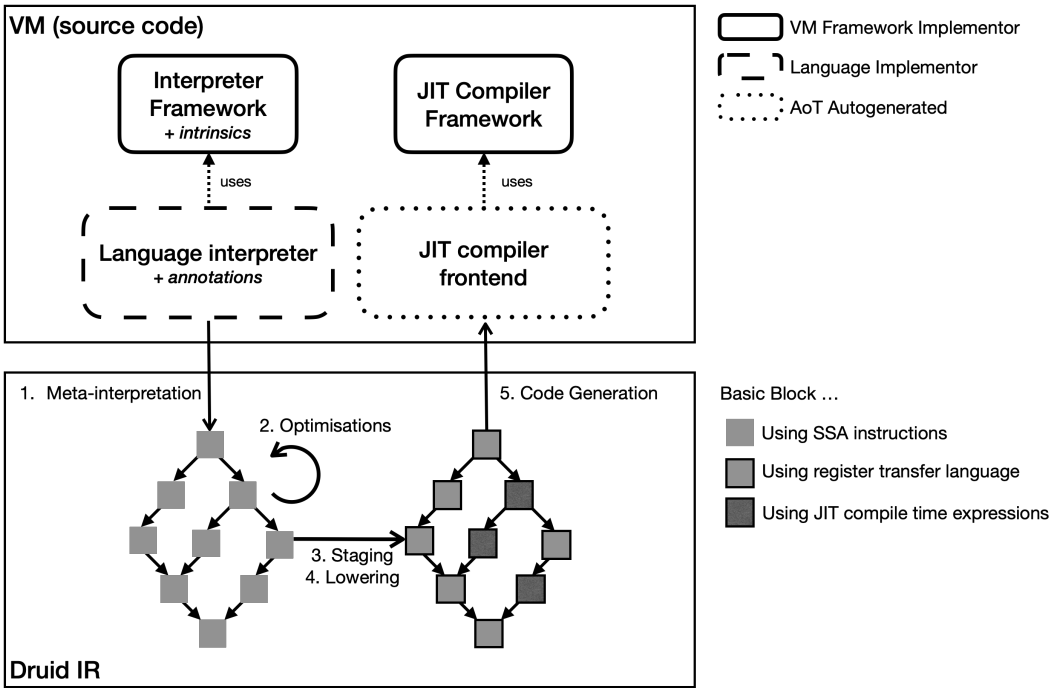


Figure 4.3: Metacompilation process by Druid. At the top, the structure of the source code of a virtual machine, divided into VM framework implementation, language implementation, and automatically generated code. At the bottom, a representation of the transformations performed by Druid to metacompile interpreter definitions into JIT compiler ones.

In the following sections, we expand the metacompilation pipeline into five main phases: 1. Metainterpretation of the interpreter handler to generate the Druid IR. 2. Application of a series of standard optimizations on the IR. 3. Staging analysis, which marks the instructions to be executed at JIT-compile time based on data dependencies. 4. Lowering of the IR into a Cogit-compatible representation. 5. Emission of the JIT compiler instruction handler in CogitRTL form.

4.3.1 Metainterpretation

The input to Druid is an interpreter instruction handler method. To generate its intermediate representation (IR), we perform an *abstract interpretation* of the method's abstract syntax tree (AST). The Druid IR is constructed as a control flow graph (CFG) in static single-assignment (SSA) form, using basic blocks and three-address code (3AC) instructions [Cousot 1977a, Cytron 1991, Aho 2006, Torczon 2007, Rastello 2022]. We call this process *metainterpretation*, since it effectively interprets the interpreter's code.

Metainterpretation differs from ordinary interpretation because it must reason

about the semantics of the host language in which the VM is implemented. Our prototype targets the Pharo VM, which is written in *Slang*, a restricted subset of the Pharo language specifically designed for VM development and translatable to C (see Chapter 3).

Because the interpreter is written in Pharo, it follows the Smalltalk tradition that "all computations are done through late-bound messages" [Goldberg 1983]. Consequently, our metainterpreter acts as a **message-directed translator**, guided by a system of *intrinsics* and metadata annotations that provide information to the metacompiler. Depending on the message being sent, the metainterpreter applies:

- **Inlinings:** Most of the message sends inside an interpreter instruction handler are calls to helper methods. These methods are inlined, *i.e.* metainterpreted in the same IR.
- **Intrinsics:** Special cases are managed by intrinsics in the metacompiler to build instructions in the IR, or ignored.

In the remainder of this chapter, we describe the main extension points and mechanisms that make metainterpretation effective for baseline JIT compiler generation.

4.3.1.1 Inlinings

The Pharo VM is written in Slang, a restricted subset of the Pharo language that is later translated to C for production [Ingalls 1997,Miranda 2011,Miranda 2018]. As a result, programs written in Slang exhibit semantics that are much closer to C than to fully dynamic Pharo code. An important consequence of this is that, although Slang uses Pharo syntax and object model, *all message sends are statically bound to their target methods* by construction.

We exploit this property to **aggressively inline all method invocations** during metainterpretation. In practice, this means that when a message is sent to a helper method inside an interpreter handler, the callee is evaluated in the same IR context as the caller, instead of emitting a function call node.

A notable implication of this design is the treatment of block closures and non-local returns [Goldberg 1983]. As a high-level language, Pharo relies heavily on closures, including for implementing control-flow structures such as conditionals and loops. In our approach, all block closures used in interpreter code are inlined at the end of the metacompilation process. Because the Druid IR is built in static single-assignment (SSA) form, the metainterpreter must track abstract states across branches and insert *phi functions* at control-flow merge points to preserve data flow consistency [Cytron 1991].

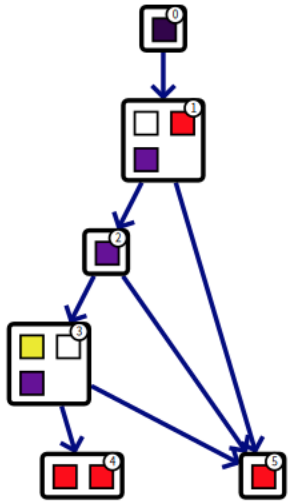
As a result of this process, the generated IR contains no remaining message sends or function call instructions. All computation and control flow of the handler is explicitly represented in the control flow graph (CFG) and use-def chains. The resulting IR operates only on low-level values and instructions, reflecting a level of abstraction much closer to machine code than to a high-level language.

Figure 4.4 illustrates an excerpt of primitiveAdd (see Figure 3.5) where part of the work is delegated to the helper class Memory. At the bottom, we show the corresponding IR after metainterpretation. The expressions ... ifFalse: [^ self primitiveFail] are lowered into conditional jumps that branch to a failure block. The methods isIntegerObject: and smallIntegerTag are fully inlined and translated into t2 Test 1 and t1 Test 1 conditional tests, respectively.

```
Interpreter >> primitiveAdd
  ...
  (memory isIntegerObject: int1)
    ifFalse: [ ^ self primitiveFail ].
  (memory isIntegerObject: int2)
    ifFalse: [ ^ self primitiveFail ].
  ...
```

```
Memory >> isIntegerObject: oop
  ^ oop anyMask: self smallIntegerTag

Memory >> smallIntegerTag
  ^ 1
```



```
Block 1:
  t1 := LoadReceiver
  t2 := LoadArg 0
  Jump if t2 Test 1 -> 2
    if false -> 5

Block 2:
  Jump if t1 Test 1 -> 3
    if false -> 5

...

Block 5:
  PrimitiveFail false
```

Figure 4.4: Excerpt of primitiveAdd using helper methods (top) and the generated IR with basic blocks (bottom). After aggressive inlining, only low-level and control-flow operations remain.

The introduction of low-level Druid instructions and the creation of explicit basic blocks are performed via *intrinsics*, which we describe in the next section.

4.3.1.2 Intrinsics

Previously in this chapter, we said that our metainterpreter acts as a *message-directed translator*. That is, translation rules are based on messages and their selectors. Druid specifies special translation rules for a subset of selectors called *intrinsics*.

Definition 4 (Intrinsic) *An intrinsic is a message send that has a specific IR translation defined by the compiler.*

```
1  DRAbstractInterpreter >> interpretAddWith: aMessageNode
2    | operand1 operand2 instruction |
3    "Interpret the operands to generate IR instructions"
4    operand1 := self interpretOperand: aMessageNode receiver.
5    operand2 := self interpretOperand: aMessageNode arguments first.
6    "Create an instruction for the addition"
7    instruction := self instantiate: DRAdd operands: { operand1. operand2 }.
8    "Add instruction to the current basic block in the CFG"
9    self addInstruction: instruction from: aMessageNode.
10   "Push the instruction to the abstract stack"
11   self pushOperand: instruction.
12   "Return"
13   ^ instruction
```

Figure 4.5: Intrinsic definition for the addition operation (+). It receives the message node as a parameter and builds the related IR instructions in the CFG. A specific instruction (of type DRAdd) is created and pushed to the abstract stack.

Thus, intrinsic generation rules are defined for a specific message selector and are written based on the AST node representing the message send at the call site. A basic example is the arithmetic operation for addition (*i.e.* messages with the selector +), presented in Figure 4.5.

This intrinsic creates a Druid instruction DRAdd (line 7), adds it to the current IR basic block (line 9), and pushes it onto the abstract stack for metainterpretation (line 11). The operands of the new instruction are obtained by recursively metainterpreting the operands of the AST node (lines 4-5).

Our metacompiler design further classifies intrinsics. Appendix A lists all intrinsics by category.

Essential Intrinsics. These implement the semantics of standard Pharo code, including arithmetic operations (as in Figure 4.5) and control-flow operations. For example, Figure 4.6 shows the metainpretation of ifTrue: messages. It inserts a conditional branch instruction in the current basic block (lines 3-8) and interprets the block closure for the *true branch* (lines 9-14). The abstract execution states of both

branches are merged at the merge point (lines 15-19), taking into account cases with non-local returns (lines 23-28).

```
1  DRAbstractInterpreter >> interpretIfTrueWith: aMessageNode
2    | conditionalJump trueBranchBasicBlockOut startingBasicBlock executionStateBeforeBranch |
3    "Condition"
4    aMessageNode receiver acceptVisitor: self.
5    conditionalJump := self
6      instantiateNoResultInstruction: DRBranchIfCondition
7      operands: { DREqualsThanComparison new. self popOperand. true asDRValue }.
8    self currentBasicBlock endInstruction: conditionalJump.
9    "Branch"
10   executionStateBeforeBranch := executionState copy.
11   trueBranchBasicBlockOut := self
12     branchFrom: self currentBasicBlock
13     onEdge: [ :branchEntryBlock | conditionalJump trueBranch: branchEntryBlock ]
14     doing: [ self interpretBlockBody: aMessageNode arguments first ].
15   "Merge point"
16   startingBasicBlock := self currentBasicBlock.
17   self newBasicBlock.
18   self currentBasicBlock addPredecessor: startingBasicBlock.
19   conditionalJump falseBranch: self currentBasicBlock.
20   "Could happen that the evaluated block had a non-local return.
21     In that case, this block should not arrive at this merge point
22       => do not add the jump"
23   trueBranchBasicBlockOut hasFinalInstruction
24     ifTrue: [ executionState := executionStateBeforeBranch ]
25     ifFalse: [ executionState := self
26         onBlock: currentBasicBlock
27         mergeAll: { executionStateBeforeBranch. executionState copy }.
28       trueBranchBasicBlockOut jumpTo: self currentBasicBlock ]
```

Figure 4.6: Definition of the intrinsic for the message ifTrue: in our prototype. This intrinsic inserts branches in the CFG.

VM-specific Intrinsics. These intrinsics model VM-specific operations, such as stack access (*e.g.* push: pop:), memory access, and type coercions. They are consistent with the semantics of the Pharo-to-C translation performed by Slang (see Chapter 3).

For example, Figure 4.7 shows an excerpt using sumSmallInteger:withSmallInteger:ifOverflow:, a VM-specific intrinsic that performs checked arithmetic with overflow detection. Such intrinsics are inlined when translating the interpreter to C, incurring no runtime overhead, and are translated to *jump if overflow* instructions when generating JIT compiler code.

```
Interpreter >> primitiveAdd
  ...
  result := self
    sumSmallInteger: smi1
    withSmallInteger: smi2
    ifOverflow: [ ^ self primitiveFail ].
  ...
```

```
JITFrontend >> gen_primitiveAdd
  ...
  self AddCq: -1 R: SendNumArgsReg.
  self AddR: ClassReg R:
     SendNumArgsReg.
  jump2 := self JumpOverflow: target.
  ...
```

Figure 4.7: Metacompilation of a primitive using a VM-specific intrinsic that performs SmallInteger addition with overflow checking.

Guiding Intrinsics. These intrinsics work as interpreter no-ops that only guide metacompilation decisions. They allow fine-grained control over whether code should be considered during JIT compilation or interpretation.

As illustrated in Figure 4.8, druidIgnore: is used to avoid compiling code that is redundant or expensive to JIT (*e.g. Static Type Prediction* during closure activation). *Static Type Prediction* is highly beneficial in the interpreter (avoiding expensive lookups, see Section 3.3.4), but is redundant in JIT-compiled code where *Polymorphic Inline Caches* [Hölzle 1991a] already perform the optimization. Similarly, interpreterIgnore: is used to rewrite code sections that must be handled differently in the JIT compiler, for instance, to delay stack mutation until after deoptimization checks.

```
Interpreter >> bytecodePrimValue
  ...
  self druidIgnore: [
    isBlock := ...
    isBlock ifTrue: [ ^ self primitiveFullClosureValue ] ].
  self normalSendSpecialSelector: 25 argumentCount: 0
```

```
Interpreter >> storeAndPop: obj receiverVariable: fieldIndex withValue: anOop
  ...
  "Interpreter needs to do the pop before check immutability"
  self druidIgnore: [ self pop: 1 ].
  objectMemory storePointerImmutabilityCheck: fieldIndex ofObject: obj withValue: anOop.
  "Druid resolves it with a deoptimization to the interpreter,
       we cannot modify the stack until be sure."
  self interpreterIgnore: [ self pop: 1 ].
```

Figure 4.8: Excerpts of code using druidIgnore: and interpreterIgnore: intrinsics.

Other guiding intrinsics are used to mark *exit points*, which we explain in the next section.

4.3.1.3 Exit Points

Druid *exit points* define the scope of metacompilation, allowing us to selectively exclude certain interpreter features from JIT compilation. From a JIT compiler design perspective, exit points are metacompiled as *deoptimization points* (see Section 3.4.4) for avoiding the generation of machine code for *slow paths*, rarely executed. During the iterative development of Druid, exit points also enabled incremental progress: they allowed us to produce early, working versions of the JIT compiler even when support for all features was not yet complete.

Exit points are triggered using the druidExitPoint intrinsic. Unlike druidIgnore:, which completely removes a code fragment from metacompilation, druidExitPoint is translated into an explicit *deoptimization point* (see Section 3.4.4). When this point is reached at runtime, execution is transferred from JIT-compiled machine code back to the interpreter.

The exact semantics of exit points depend on the type of instruction:

- **Bytecode instructions:** The current JIT frame is *deoptimized* by spilling all registers to the stack and resuming execution in the interpreter through a runtime trampoline call.
- **Primitive instructions:** Compilation skips to the end of the primitive, generating a fall-through to the bytecode.

Figure 4.9 illustrates the use of an exit point in the bitAnd primitive. When either operand is not an integer, the computation requires switching to a more general integer representation (lines 7-10). Instead of generating complex JIT code for this rare case, Druid inserts a deoptimization point so that execution returns to the interpreter when the branch is taken.

```
1  Interpreter >> primitiveBitAnd
2    | rcvr arg |
3    rcvr := self stackValue: 1.
4    arg := self stackTop.
5    (objectMemory areIntegers: arg and: rcvr)
6      ifTrue: [ self pop: 2 thenPush: (arg bitAnd: rcvr) ]
7      ifFalse: [
8        self druidExitPoint.
9        "Resolve in another integer representation"
10       ... ]
```

Figure 4.9: Excerpt of the interpreter primitive for bitAnd. The slow path (line 8) is explicitly marked with a druidExitPoint, which causes the JIT compiler to generate a deoptimization point for this branch.

Exit points can also be declared through *annotations* (called pragmas in Pharo parlance [Ducasse 2016]) placed on interpreter methods that should be excluded from JIT compilation entirely.

In Figure 4.10, the contextSize method is annotated with druidExitPoint. Accessing context information involves reifying execution state from the stack, which is a rare execution path and complex for JIT-compilation. Druid therefore excludes this method from JIT compilation, deferring it to the interpreter.

```
Interpreter >> contextSize
 "Special version of primitiveSize for accessing contexts."
 <druidExitPoint>
 ....
```

Figure 4.10: The druidExitPoint annotation on contextSize method skips metacompilation of complex operations such as context reification, which are then handled by the interpreter through deoptimization.

In addition to excluding methods from compilation, annotations also attach metadata required by the JIT compiler framework, as we describe in the next section.

4.3.1.4 Metacompilation Metadata

The metacompilation of the JIT compiler relies on *metadata annotations* attached to each instruction method in the interpreter. This metadata is primarily consumed by the target JIT backend to make decisions such as whether to use frameful or frameless compilation, and how to generate bytecode-to-machine-code mappings for deoptimization (see Section 3.4).

For **primitive instructions**, metadata specifies:

- the number of arguments;
- whether a runtime type check should be staged when the receiver is of a known type, *e.g.* integer, float, or character (see Section 4.3.3).

For **bytecode instructions**, metadata specifies:

- the *stack delta*, that is, the change in stack height after execution [Casey 2005, Ertl 2002];
- whether the instruction defines a suspension point;
- whether a stack frame is required for its execution.

In addition, the metacompiler uses this metadata to transform the interpreter's *stack-based* calling convention into the *register-based* convention required by the Cogit backend.

Appendix A lists all metadata annotations defined in our prototype. As future work, we plan to automatically infer this metadata from the interpreter code itself, using Druid's metainterpretation infrastructure.

Examples

Figure 4.11 illustrates two interpreter methods annotated with metadata: one for a primitive and one for a bytecode instruction handler.

```
Interpreter >> primitiveAdd
  <numberOfArguments: 1>
  <customisedReceiverFor: #smallInteger>
  ...
```

```
Interpreter >> bytecodePrimAdd
  <compilationInfo: #isMapped>
  <druidInfo: #hasSend>
  ... "Check for integers"
  ... "If fail, send message +"
```

Figure 4.11: Examples of metadata annotations in interpreter methods, respectively, defining a primitive and a bytecode instruction handler.

On the left, the method defining the primitive for adding two small integers declares both the number of arguments (one argument plus the receiver) and that the metacompiled version should be specialized for smallInteger receivers.

On the right, the bytecode handler implements a fast-path for addition using *Static Type Prediction*. Because it may fall back to a message-send in the slow path, it is annotated with #hasSend. Additionally, it is marked with #isMapped to indicate that its program counter must be recorded in the bytecode-to-machine-code mapping, as it contains a deoptimization point (see Section 3.4.4).

4.3.2 Optimizations

Druid follows the design of a traditional **multi-pass compiler**. A key advantage of this architecture is that it supports incremental development, allowing us to progressively integrate new optimizations without affecting the rest of the pipeline.

For our prototype, we explicitly selected an effective set of optimizations to mitigate the well-known *phase-ordering problem* [Cooper 2002, Palumbo 2022a]. Each optimization operates *in place* by mutating the IR directly. They are implemented as standalone classes, each providing an applyTo: cfg method that takes a control flow graph (CFG) as input and rewrites it according to the optimization logic.

Figure 4.12 shows the implementation of the *Copy Propagation* optimization. The pass iterates over all instructions in the CFG, looking for Copy instructions whose operand is itself an instruction. For each match, it propagates the operand instruction to all users of the Copy, effectively bypassing the redundant indirection. As a result, the original Copy instruction may end up with no users, in which case it becomes *dead code* and will be removed by a subsequent *Dead Code Elimination* pass [Rastello 2022].

```
CopyPropagation >> applyTo: cfg
 "Replace
      user -> copy -> instruction
    by
      user -> instruction
   This optimization does not remove the copy"
 cfg instructions copy do: [ :i |
   (i isCopy and: [ i operand isInstruction ] )
    ifTrue: [ self propagateCopyFrom: i ] ]

CopyPropagation >> propagateCopyFrom: copy
 copy users copy do: [ :user |
   user replaceDependency: copy by: copy operand ]
```

Figure 4.12: Implementation of the *Copy Propagation* optimization for the Druid IR.

At the time of writing, most of the optimizations implemented in Druid, apart from those explicitly described in this chapter, are well-known techniques drawn from classical compiler literature [Aho 2006, Torczon 2007, Rastello 2022].

A full list of the optimizations applied by our prototype, along with a short description of each, is provided in Appendix B.

4.3.3 Staging

Druid implements a *staging pass* that runs at the end of the pipeline, after all optimizations described in the previous section, but just before register allocation and code generation. This pass is designed as a forward, sparse data-flow analysis operating over SSA use-def chains.

Conceptually, staging is similar to *Constant Propagation* [Aho 2006,Rastello 2022], but with a different focus: rather than identifying values that are constant at program runtime, it identifies expressions that are constant at JIT compile time. After the analysis, **any expression determined to be constant during JIT compilation is marked for staging**, meaning it will be evaluated once at compile time and will not generate runtime instructions.

Figure 4.13 illustrates staging in action. The interpreter handler (left) decodes a character from the bytecode stream and pushes it onto the stack. Because the character value is constant for the lifetime of the compiled method, all decoding and object-tagging operations are staged and computed by the compiler at JIT compile time. The generated JIT handler (right), therefore, consists only of a single absPushConstant: instruction and state resets.

The global variables extB and numExtB (shared between interpreter and compiler) are reset as usual after being consumed [Béra 2014]. The absPushConstant: form is specific to Cogit's abstract interpreter, which performs peephole optimiza-

```
Interpreter >> pushCharacterBytecode
    "#iiiiiiiii (+ Extend B * 256)"
    | value char |
    <needsFrameNever: 1>
    value := self fetchByte + (extB << 8).
    char := self characterObjectOf: value
    self push: char.
    numExtB := extB := 0
```

```
JITCompiler >> gen_PushCharacterBytecode
    "AutoGenerated by Druid"
    | s5 s6 s4 s2 s7 s3 |
    s2 := byte1.
    s3 := extB.
    s4 := s3 << 8.
    s5 := s2 + s4.
    s6 := s5 << 3.
    s7 := s6 + 2.
    self absPushConstant: s7.
    extB := 0.
    numExtB := 0.
    ^ 0
```

Figure 4.13: On the left, the interpreter handler for pushing a character onto the stack. On the right, the JIT handler generated by Druid. Since the character value is embedded in the bytecode stream, it is constant at JIT compile time, and all computations are staged.

tions between bytecodes. As a result, this bytecode may leave no residual machine code at all if the constant is consumed by the next instruction at compile time. Further details on how Cogit performs such optimizations are presented in Chapter 6.

Staging via intrinsics. While the automatic staging pass covers most cases, some scenarios require explicit guidance. For example, method literal accesses are constant during JIT compilation and can be directly embedded into the generated machine code. To support such cases, we extend the metainterpreter with the intrinsic druidStageable:, which explicitly marks a block of code to be staged, conceptually similar to a *lift* operator in other metacompilers [Glück 1997, Rompf 2014].

Figure 4.14 shows an example using the intrinsic to access a literal of the method. This allows literal access to be fully resolved during JIT compilation, avoiding any runtime memory fetch.

Staging via annotations. Another mechanism for staging is the customisedReceiverFor: annotation, introduced earlier. This annotation is primarily used in primitive handlers to perform type checks at JIT compile time, *i.e.* whether the receiver is a small integer, float, or character. If the check fails, JIT compilation is aborted, and the method continues to be interpreted.

Based on the annotation, the metainterpreter introduces a staged conditional branch in the CFG. Optimizations such as *Dead Branch Elimination* then remove the redundant receiver type checks at execution time from the IR. As seen in Figure 4.15, the interpreter version performs three checks (two type checks and one

```
Interpreter >> pushLiteralConst
  <needsFrameNever: 1>
  | literalIndex literal |
  self fetchNextBytecode.
  literalIndex := currentBytecode bitAnd: 16r1F.
  literal := self
            literal: literalIndex
            ofMethod: (self iframeMethod:
      framePointer).
  self push: literal

Interpreter >> literal: offset ofMethod:
      method
  ^ self druidStageable: [
      objectMemory
        fetchPointer: offset + LiteralStart
        ofObject: method ]
```

```
JITCompiler >> gen_PushLiteralConst0
  "AutoGenerated by Druid"
  | s5 s1 s4 s2 s6 |
  s5 := 0 + LiteralStart.
  s1 := s5 << 3.
  s4 := methodObj + 8.
  s2 := s1 + s4.
  s6 := self int64AtPointer: s2.
  self absPushConstant: s7.
  ^ 0
```

Figure 4.14: The interpreter uses druidStageable: to mark literal load operations as staged. In the generated JIT handler, the literal value is embedded directly as a constant.

overflow check), whereas the JIT version retains only the argument check (jump1) and overflow check (jump2), avoiding the generation of the redundant receiver check at execution time.

Although this seems like a minor reduction, it occurs in extremely hot paths: every arithmetic operation in the system benefits from this elimination. This optimization is particularly important given that our JIT compiler does not support *Static Type Prediction* for message sends in bytecode handlers (see Section 4.3.1.2).

The function mclassIsSmallInteger is provided by the Cogit backend. Its use highlights one of the key benefits of targeting a mature JIT framework: we can rely on existing infrastructure for efficient compile-time decisions.

```
Interpreter >> primitiveAdd
  <numberOfArguments: 1>
  <customisedReceiverFor: #smallInteger>
  | arg1 arg2 rawArg1 rawArg2 rawResult result |
  "Fetch values"
  arg1 := self stackValue: 0.
  arg2 := self stackValue: 1.
  "Check for integers"
  (memory isIntegerObject: arg1)
      ifFalse: [ ^ self primitiveFail ].
  (memory isIntegerObject: arg2)
      ifFalse: [ ^ self primitiveFail ].
  "Perform the addition"
  rawArg1 := memory rawValueOf: arg1.
  rawArg2 := memory rawValueOf: arg2.
  rawResult := rawArg1 + rawArg2.
  "Check overflow"
  (memory isIntegerValue: rawResult)
      ifFalse: [ ^ self primitiveFail ].
  "Push result"
  result := memory integerObjectOf: rawResult.
  self pop: 2 thenPush: result
```

```
JITCompiler >> gen_PrimitiveAdd
  "AutoGenerated by Druid"
  | jump1 jump2 currentBlock |
  self mclassIsSmallInteger
      ifFalse: [ ^ Unimplemented ].
  self MoveR: ReceiverResultReg
      R: ClassReg.
  self MoveR: Arg0Reg
      R: SendNumArgsReg.
  self TstCq: 1 R: SendNumArgsReg.
  jump1 := self JumpZero: 0.
  self AddCq: -1 R: SendNumArgsReg.
  self AddR: ClassReg
      R: SendNumArgsReg.
  jump2 := self JumpOverflow: 0.
  self MoveR: SendNumArgsReg
      R: ReceiverResultReg.
  self genPrimReturn.
  currentBlock := self Label.
  jump1 jmpTarget: currentBlock.
  jump2 jmpTarget: currentBlock.
  ^ 0
```

Figure 4.15: Metacompilation of the primitiveAdd handler. On the left, the interpreter version uses the customisedReceiverFor: annotation for smallInteger. On the right, the generated JIT handler performs the check at JIT compile time via mclassIsSmallInteger, aborting compilation if it fails.

4.3.4 Lowering

As discussed at the beginning of this chapter, our metacompilation process targets the Cogit backend. Therefore, in the final stage of the pipeline, the Druid IR is *lowered* to comply with the constraints of the Cogit framework (see Section 3.4). During this phase, the Druid IR exits SSA form and is transformed into a representation suitable for direct code generation by Cogit.

In this section, we describe the main transformations performed during this lowering pass.

Accommodating Cogit register allocation. Druid includes a register allocation pass that converts IR instructions from a three-address code (3AC) representation to the two-address code (2AC) format expected by CogitRTL. The allocation algorithm targets Cogit's virtual registers and uses a constraint-based approach inspired by [Mössenböck 2002].

When metacompiling **primitive instructions**, Druid directly assigns Cogit's fixed virtual registers (such as ReceiverResultReg, Arg0Reg, etc.). These virtual

registers are later mapped to physical machine registers by the Cogit backend.

For example, Figure 4.16 presents the metacompiled version of the primitive bitAnd:, used for bit operation. It uses explicit virtual registers for the operations. In this case, values are copied to other registers before operate them, because in case of failure the original values must keep in the expected registers: ReceiverResultReg, Arg0Reg.

```
JITCompiler >> gen_PrimitiveBitAnd
  "AutoGenerated by Druid"
  | jump1 currentBlock |
  self gen_MoveR: ReceiverResultReg r: ClassReg.
  self gen_MoveR: Arg0Reg r: SendNumArgsReg.
  self gen_MoveR: SendNumArgsReg r: Extra0Reg.
  self gen_AndR: ClassReg r: Extra0Reg.
  self gen_TstCq: 1 r: Extra0Reg.
  jump1 := self gen_JumpZero: 0.
  self gen_AndR: ClassReg r: SendNumArgsReg.
  self gen_MoveR: SendNumArgsReg r: ReceiverResultReg.
  self genPrimReturn.
  currentBlock := self gen_Label.
  jump1 jmpTarget: currentBlock.
  ^ 0
```

Figure 4.16: Metacompiled version of the primitive bitAnd:, using Cogit's fixed virtual registers.

When metacompiling **bytecode instructions**, Druid emits instructions that allocate registers dynamically at JIT compile time. This is essential for Cogit's abstract interpreter, which uses an abstract stack model to apply peephole optimizations across bytecodes (see Chapter 6).

Figure 4.17 shows a metacompiled handler that pushes the second instance variable of the receiver to the stack. Here, Cogit's allocateReg: function dynamically selects a free register, avoiding conflicts by updating the live register mask.

```
JITCompiler >> gen_BytecodePushReceiverVariableIndex1
  "AutoGenerated by Druid"
  | live currentBlock t0 |
  live := 0.
  self ensureReceiverResultRegContainsSelf.
  t0 := self allocateReg: live.
  live := live bitOr: (self registerMaskFor: t0).
  self gen_MoveR: ReceiverResultReg r: t0.
  self ssPushBase: t0 offset: 16.
  ^ 0
```

Figure 4.17: Metacompiled version of a push receiver variable bytecode handler with `index = 1`. Register t0 is allocated dynamically at JIT compile time.

Handling Cogit runtime interactions. Whenever generated machine code must interact with runtime functions (via *trampolines*), Cogit requires registers to be spilled to the stack and that the calling convention be respected (see Section 3.4.4). Druid explicitly emits this boilerplate during metacompilation.

Figure 4.18 shows a metacompiled conditional jump bytecode for short offsets. The jump offset is computed from the currentBytecode index. Druid generates a specialized handler for each offset value (the example shows `offset = 1`).

```
1  Interpreter >> shortConditionalJumpTrue
2    <compilationInfo: #isMapped>
3    <compilationInfo: #branch>
4    <compilationInfo: #isBranchTrue>
5    | boolean offset |
6    offset := (currentBytecode bitAnd: 7) + 1.
7    boolean := self stackTop.
8    self pop: 1.
9    boolean = objectMemory trueObject
10     ifTrue: [ self jump: offset ]
11     ifFalse: [
12       boolean = objectMemory falseObject ifFalse:
         [
13         self internalMustBeBoolean: boolean ].
14       self fetchNextBytecode ]
```

```
JITCompiler >>
    gen_ShortConditionalJumpTrue0
 "AutoGenerated by Druid"
 | jump1 s5 s2 currentBlock s12 t0 jump2 s9 |
 self annotateBytecode: self Label.
 t0 := ... "Allocate register"
 (self ssValue: 0) copyToReg: t0.
 self ssPop: 1.
 s5 := objectMemory trueObject.
 self ssFlushStack.
 self CmpCq: s5 R: t0.
 jump1 := self JumpNonZero: 0.
 s9 := bytecodePC + 2.
 self Jump: (self ensureFixupAt: s9).
 jump2 := self Jump: 0.
 currentBlock := self Label.
 jump1 jmpTarget: currentBlock.
 s12 := objectMemory falseObject.
 self CmpCq: s12 R: t0.
 jump1 := self JumpZero: 0.
 self MoveR: t0 R: TempReg.
 self CallRT:
    ceSendMustBeBooleanTrampoline.
 currentBlock := self Label.
 jump2 jmpTarget: currentBlock.
 jump1 jmpTarget: currentBlock.
 ^ 0
```

Figure 4.18: Conditional jump bytecode metacompilation for `offset = 1`, including runtime call to a trampoline in the exceptional path.

The isMapped compilation annotation in the interpreter signals that this bytecode needs mapping between its bytecode index and the corresponding machine-code address for correct deoptimization. Druid inserts Cogit's annotateBytecode: function to establish this mapping.

Calls to internalMustBeBoolean: in the interpreter (line 13) are metacompiled as deoptimization points. At runtime, when this case is encountered, the JIT-compiled code invokes a trampoline (*e.g.* ceSendMustBeBooleanTrampoline) to return control

to the interpreter, which throws the corresponding exception[2].

Handling polymorphic inline caches. For *message sends*, Druid reuses Cogit's built-in support for polymorphic inline caches (PICs) and call-site linking (see Section 3.4). Message sends are expressed in the interpreter through intrinsic functions that produce DRSend instructions in the Druid IR. During lowering, these are translated into patchable call-site sequences in CogitRTL.

Figure C.4 shows the metacompilation of a bytecode handler for the equality message (=). The interpreter version uses *Static Type Prediction* and *Super-Instructions* optimizations for integers (via maybePushBoolean:), but these are ignored in the JIT version using the druidIgnore: intrinsic.

For the actual message send, Druid emits a sequence invoking marshallSendArguments: and genMarshalledSend:numArgs:sendTable:. These handle PIC setup, runtime lookup, and code patching. Since they also generate the bytecode-to-machine-code mapping, no explicit annotateBytecode: calls are required in this case.

```
Interpreter >> bytecodeSendEquals
  <compilationInfo: #isMapped>
  <druidInfo: #hasSend>

  self druidIgnore: [
    | rcvr arg |
    rcvr := self stackValue: 1.
    arg := self stackValue: 0.
    (objectMemory areIntegers: rcvr and: arg)
      ifTrue: [
       ^ self maybePushBoolean: rcvr = arg ] ].

  self normalSendSpecialSelector: 6
      argumentCount: 1
```

```
JITCompiler >> gen_BytecodeSendEquals
  "AutoGenerated by Druid"

  self marshallSendArguments: 1.
  self
    genMarshalledSend: –7
    numArgs: 1
    sendTable: ordinarySendTrampolines.
  ^ 0
```

Figure 4.19: Metacompilation of the bytecode handler for the equality (=) message send. The JIT version skips *Static Type Prediction* optimizations and directly generates a PIC-enabled call-site.

4.3.5 Code Generation

The final step of the metacompilation pipeline is *code generation*. At this stage, the lowered IR is linearized in *reverse post-order*, ensuring a topologically sorted traversal of the control flow graph [Kennedy 2001]. Each instruction is then visited

[2]In Pharo, conditionals are written as messages to boolean objects (*e.g.* boolean ifTrue: [...]). The message is optimized by the bytecode compiler as conditional jump, thus the error case of a non-boolean object as the receiver must be handled.

sequentially to generate the corresponding abstract syntax tree (AST) for the JIT compiler handler, expressed in CogRTL form. Once the AST is built, the resulting method is compiled and integrated with the rest of the VM codebase. Several examples of metacompiled code generated by Druid are presented in Appendix C.

Thanks to Pharo's AST-based code formatter, the generated handlers remain clean and human-readable. Importantly, all auto-generated code is just regular Pharo code: developers can browse, inspect, and even modify these methods interactively within the VM development environment. This keeps the metacompilation process transparent and approachable for VM engineers.

4.4 Changes to the Compiler Architecture

Beyond adding intrinsic functions and annotations to the interpreter, we performed several extensions and improvements to the Cogit compiler architecture to make it a suitable target for metacompilation.

The original Cogit frontend was hand-tailored for compiling Pharo bytecode and implemented several ad-hoc workarounds to support advanced features, rather than exposing well-defined, reusable APIs. Moreover, its design assumed that the compiler should always be capable of compiling the entire language, which made incremental development and experimentation difficult.

Supporting incremental compiler development. We extended the runtime to prevent repeatedly re-compiling methods that had previously failed JIT compilation. Originally, Cogit operated as a *compile-all-or-nothing* compiler: when a compilation failed, either because register allocation constraints were violated or a bytecode lacked a defined IR generator, the failure was not recorded. As a result, every subsequent execution of the same method triggered another compilation attempt, repeatedly failing and introducing unnecessary overhead. While this behavior was acceptable for the production-quality handwritten frontend, which covered the entire language, it severely hindered our ability to iteratively develop and test partial translations. We modified the process so that when a JIT compilation fails, the method is marked accordingly, avoiding further compilation attempts and ensuring it is permanently executed by the interpreter instead.

Partial compilation support. We also extended the compiler with support for arbitrary *deoptimization points*, enabling partial compilation of methods. Deoptimization points allow us to focus on the hot paths and fall back to interpretation for less common or complex cases. When a deoptimization point is triggered at runtime, the JIT infrastructure spills all live registers to the stack, reconstructs the interpreter stack frame, and transfers control to the interpreter, ensuring correct execution continues seamlessly.

Improving the target compiler API. Finally, we simplified and standardized the frontend-backend interface of Cogit. In its original design, the frontend relied on undocumented backend details and used inconsistent calling conventions when activating methods, block closures, and trampolines. Rather than pushing this complexity into Druid, we refactored the Cogit API and unified the calling conventions. These changes made Cogit a cleaner target for metacompilation and reduced the complexity of the generated code.

4.5 Features

In this chapter, we presented our approach to **automatically generating a baseline JIT compiler from interpreter definitions via metacompilation**. This section highlights the key advantages of our solution.

High-level development model. Language developers do not need to interact with low-level abstractions (*i.e.* machine code semantics) used by the JIT compiler. Instead, they define language semantics entirely at the interpreter level, while Druid translates them into register-based code automatically.

Guaranteed semantic consistency. Because the JIT compiler frontend is generated directly from interpreter definitions, the semantics of the language are guaranteed to remain consistent across both execution tiers. Any change to the interpreter is automatically propagated to the JIT compiler.

Declarative and incremental optimizations. The multi-pass architecture of Druid allows optimizations to be developed and integrated incrementally. Each optimization pass can be enabled or disabled independently, making it straightforward to generate and experiment with multiple JIT compiler variants, as we demonstrate in the second part of this thesis.

Compatibility with other approaches. The output of our metacompilation process is human-readable code for the JIT compiler frontend. The generated methods coexist with the rest of the handwritten VM code and can be combined with other components generated using alternative techniques.

Improved maintainability. By eliminating the need to manually duplicate instruction semantics between the interpreter and the JIT compiler, we reduce the risk of inconsistencies and human-introduced bugs. Together with the features above, this results in a more maintainable and robust VM codebase.

No runtime metacompilation overhead. Since Druid performs metacompilation entirely ahead of time, it introduces no runtime overhead. This is consistent with the design goals of a baseline JIT compiler, where compilation speed is critical. Furthermore, our prototype is implemented entirely in Pharo, without relying on the restricted Slang subset used by the VM itself.

4.6 Conclusion

In this chapter, we presented our approach to *automatically generating a baseline JIT compiler from interpreter definitions*.

We introduced **Druid**, a **source-to-source, ahead-of-time metacompiler prototype** designed as a multi-pass SSA-based compiler, extended with specialized mechanisms for metacompilation. Druid bridges the gap between interpreter definitions and JIT compiler frontends by transforming interpreter semantics into register-based code that targets an existing JIT backend. We validate our approach by using Druid to generate the baseline JIT compiler for the production Pharo VM.

Intrinsics drive metacompilation. Interpreter code is annotated and extended with intrinsics that guide the metacompilation process. They define the JIT compilation scope (*e.g.* deoptimization points), provide specialized IR generation rules, and selectively disable interpreter-only optimizations that are unnecessary or unsuitable in JIT-compiled code.

Staging expressions at JIT compile time. A dedicated staging pass determines which IR expressions are constant during JIT compilation. These expressions are evaluated at JIT compile time, simplifying the generated machine code and reducing runtime overhead.

Reusing a mature backend infrastructure. By targeting the Cogit framework and emitting code in its RTL form, Druid benefits from an existing, production-grade JIT infrastructure. This includes reusing hot-spot detection, runtime trampolines, polymorphic inline caches, and Cogit's peephole optimizations, allowing us to focus on frontend generation rather than low-level backend concerns.

Overall, this chapter demonstrated that **metacompilation is a practical and effective approach** for automatically generating a baseline JIT compiler while ensuring semantic consistency, maintainability, and low development overhead. In the next chapter, we evaluate the results of applying Druid to the Pharo VM, comparing the generated baseline JIT compiler with the existing handwritten implementation in terms of performance and development effort.

CHAPTER 5

Metacompiled vs. Handwritten JITs

Contents

In the previous chapter, we presented the details of our approach and introduced our metacompiler prototype called *Druid*. We applied this approach to generate the baseline JIT compiler frontend for the Pharo VM.

In this chapter, we evaluate our solution by measuring the performance and development impact of the metacompiled JIT compiler. Specifically, we compare our prototype against two baselines: the interpreter and the handwritten baseline JIT compiler of the Pharo VM, previously introduced in Chapter 3.

To guide this evaluation, we formulate the following **research questions**:

- RQ1: Does our solution generate JIT compilers that improve the performance of the interpreter?
- RQ2: Is the JIT compiler generated by Druid competitive in performance against the handwritten version?
- RQ3: Does our generated approach increase the overhead at compile time?
- RQ4: How much development effort is affected by our automatic approach?

The results show that our approach increases the speed-up of the interpreter by $1.8\times$, up to $5\times$, without adding significant compilation overhead in a production VM. Our generated JIT compiler achieves $0.7\times$ of the speed of a handwritten version matured over 10 years, making the VM code easier to maintain and evolve in the long run.

5.1 Setup and Methodology

For our evaluation, we built four different variations of the Pharo VM. Each variation exhibits different behavior regarding JIT compilation scope, the amount of compiled code, and supported optimizations. They are illustrated in Figure 5.1.

All JIT compilers (the one generated by Druid and both handwritten versions) share the same Cogit backend. Therefore, they differ only in their frontend IR generators (see Chapter 3).

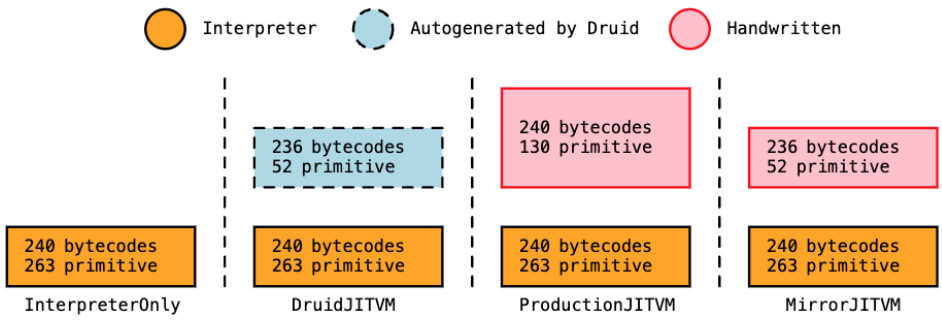


Figure 5.1: Variations of the Pharo VM used in the evaluation. The boxes show the amount of bytecode and primitive instructions implemented in each component.

InterpreterOnly. A Pharo VM without a JIT compiler, implementing only the interpreter described in Section 3.3, including optimizations such as *Static Type Prediction* and *Super-Instructions*. It fully supports the language semantics with 240 bytecodes and 263 primitive instructions. This VM serves as the baseline for speed-up measurements.

ProductionJITVM. A Pharo VM extending *InterpreterOnly* with a handwritten JIT compiler frontend described in Section 3.4. It supports JIT compilation of 240 bytecodes and 130 primitive instructions. This VM includes optimizations such as *Static Type Prediction* and *Super-Instructions*.

DruidJITVM. A Pharo VM extending *InterpreterOnly* with a JIT compiler frontend automatically generated by Druid as described in Chapter 4. It supports compilation of 236 bytecodes and 52 primitive instructions. The four missing bytecodes correspond to unused or experimental instructions from the Sista bytecode set [Béra 2014] and the stack-frame reification instruction (pushThisContext), which we left outside of our prototype because we consider them as part of the program's slow paths (*e.g.* exceptions, debugging).

MirrorJITVM. A Pharo VM derived from *ProductionJITVM*, configured to compile only the same subset of bytecodes and primitive instructions as *DruidJITVM*. We use this variant to analyze the differences between our generated approach and the handwritten version under equivalent compilation scopes.

We run a set of classical benchmark programs written in Pharo on each VM

variation. We use the Rebench framework [Marr 2016, Marr 2018] to execute the benchmarks. The complete list of benchmarks and descriptions is presented in Appendix D.

All benchmarks are run with 30 VM iterations. Micro-benchmarks are executed with 10 in-process iterations, out of which the first two iterations are ignored as a warm-up. This number of iterations has proven sufficient to capture the behavior of non-optimizing baseline JIT compilers [Georges 2007]. For macro-benchmarks, which consist of running the test suites of Pharo packages, we perform a single in-process iteration per VM iteration, as these are long-running workloads with negligible warm-up requirements.

5.2 Results

In this section, we present the results obtained from running our benchmark suite and provide answers to the research questions. All benchmarks were executed on a machine running Linux Debian 5.10.140-1, equipped with an AMD64 CPU E5620 @ 2.40GHz processor and 16 GB DDR3 RAM.

5.2.1 RQ1: Does our solution generate JIT compilers that improve the performance of the interpreter?

This research question evaluates whether a baseline JIT compiler frontend generated by Druid provides performance improvements over a pure interpreter.

Figure 5.2 shows, for each benchmark, the execution time relative to *InterpreterOnly*, which is fixed at $1\times$ as it serves as the baseline. The same results are summarized in the first column of Table 5.1.

Our results show that the automatically generated JIT compiler frontend yields a geometric mean speed-up of $1.8\times$ compared to interpretation. In the best case, it achieves a performance up to $5.09\times$ in the *Richards* benchmark. For macro benchmarks, we observed speed-ups ranging from $1.09\times$ (*Microdown*) to $1.52\times$ (*Compiler*).

There are two benchmarks with negative results, *Fasta* is 8 % slower, and *ThreadRing* 5 %. Additionally, *PiDigits* shows a performance similar to *InterpreterOnly*, with a speed-up of just 8 %. As we will discuss in Section 5.2.2, *PiDigits* and *ThreadRing* show negligible sensitivity to JIT compilation, having similar results across all VM variations (their respective lines in Table 5.1, comparing all the VMs, are always within the $\pm$ 10 %). For the case of *Fasta*, this is probably due to low-quality generated machine code in a common execution path, which is not well-optimized in our current prototype.

Table 5.1: Relative speed-up between the VMs per benchmark. Each column presents the ratios between two VMs. Comparison between JIT compilers was executed with and without optimized bytecodes, where *Static Type Prediction* and *Super-Instructions* are implemented in the handlers. Higher is better.
`D=DruidJITVM; I=InterpreterOnly; P=ProductionJITVM; M=MirrorJITVM`

Speed-up							
Bytecode Set		Optimized			Unoptimized		
Benchmark	D/I	D/P	D/M	M/P	D/P	D/M	M/P
BinaryTrees	3.08	0.66	0.66	1.00	0.69	0.69	1.00
Chameleons	1.50	0.95	0.97	0.98	0.96	0.97	0.99
ChameneosRedux	2.96	0.80	0.78	1.03	0.82	0.84	0.98
DeltaBlue	2.90	0.64	0.65	0.97	0.44	0.87	0.51
Fasta	0.92	0.39	0.65	0.60	0.45	0.69	0.65
KNucleotide	2.20	0.89	0.93	0.95	0.79	0.96	0.82
Mandelbrot	2.08	0.56	0.56	0.99	0.60	0.59	1.01
Meteor	2.32	0.51	0.52	0.98	0.67	0.69	0.97
NBody	2.28	0.61	0.63	0.97	0.62	0.69	0.89
PiDigits	1.08	0.97	0.98	0.99	0.99	0.98	1.01
RegexDNA	2.86	0.59	0.81	0.72	0.45	0.86	0.52
ReverseComplement	1.68	0.88	0.70	1.26	0.75	0.72	1.04
Richards	5.09	0.69	0.72	0.96	0.52	0.78	0.67
Slopstone	1.27	0.42	0.78	0.54	0.54	0.89	0.61
SpectralNorm	1.42	0.61	0.60	1.01	0.64	0.64	1.00
ThreadRing	0.95	0.91	0.98	0.93	0.99	1.05	0.94
Compiler	1.52	0.89	0.89	1.00	0.89	0.93	0.96
Microdown	1.09	0.59	0.61	0.98	0.68	0.72	0.95
Network	1.29	0.80	0.81	0.99	0.84	0.94	0.90
Startup	1.40	0.74	0.76	0.98	0.81	0.89	0.91
Best	5.09	0.97	0.98	1.26	0.99	1.05	1.04
Worst	0.92	0.39	0.52	0.54	0.44	0.59	0.51
Geomean	1.79	0.68	0.74	0.93	0.68	0.81	0.85

RQ1 Conclusion. Our solution generates JIT compilers that improve performance over purely interpreted VMs, achieving a mean **speed-up of** $1.8\times$**, up to** $5\times$ in the best case.

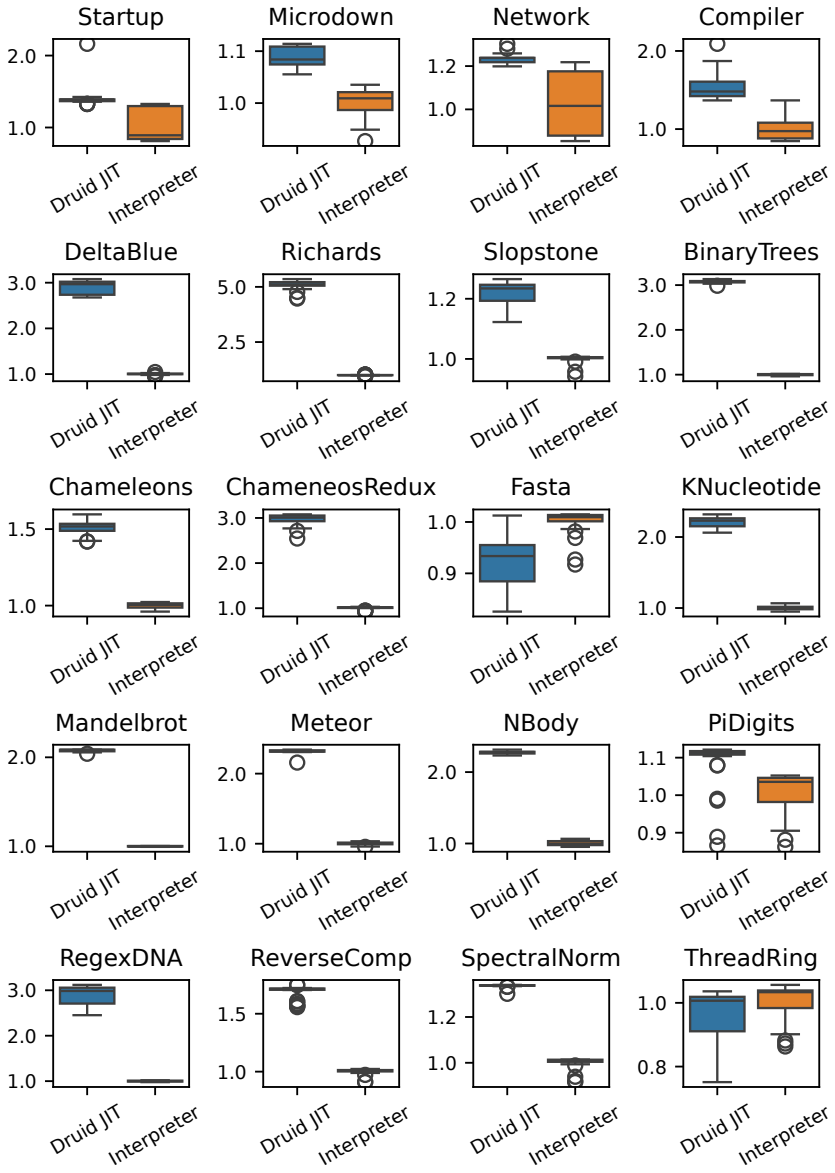


Figure 5.2: Speed-up of the Pharo VM using the JIT compiler generated automatically by our approach (*DruidJITVM*) compared to *InterpreterOnly* (baseline). Higher is better.

5.2.2 RQ2: Is the JIT compiler generated by Druid competitive in performance against the handwritten version?

This research question evaluates how well Druid performs compared to a 10-year-old matured handwritten JIT compiler frontend. For this purpose, we compare *DruidJITVM* against *ProductionJITVM*, which serves as an upper bound. The main

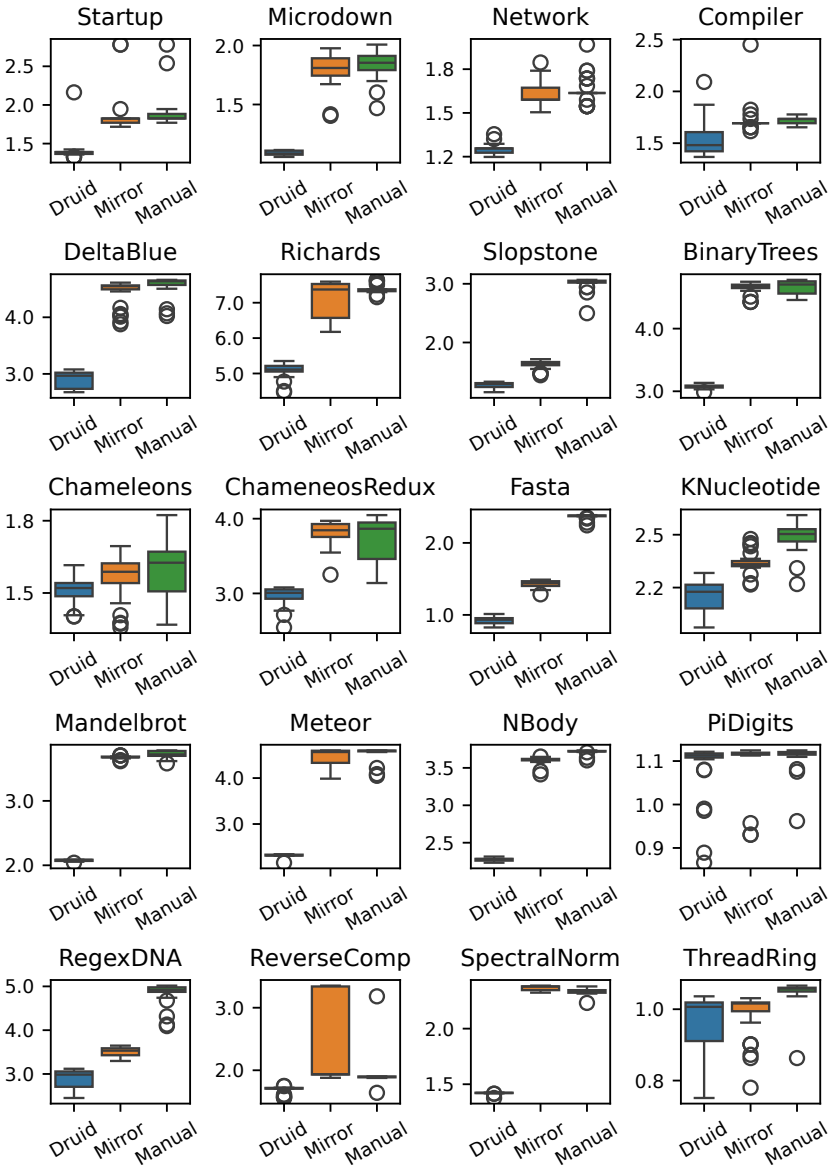


Figure 5.3: Speed-up of the VM using different versions of the JIT compiler: *DruidJITVM*, *MirrorJITVM*, and *ProductionJITVM*. *InterpreterOnly* is used as the baseline. Higher is better.

objective for *DruidJITVM* is to reach *ProductionJITVM*'s performance. However, a direct comparison is not entirely fair, since ProductionJITVM *compiles more VM instructions than* DruidJITVM ($1.3\times$ more instructions).

To perform a better comparison regarding code quality, we introduce *MirrorJITVM*, a variant that compiles the same set of bytecodes and primitive instructions as *DruidJITVM*. The only difference is that *MirrorJITVM* uses the handwritten

handler implementations extracted from *ProductionJITVM*.

Figure 5.3 shows the benchmark results for these three VM variants, relative to *InterpreterOnly*. Additionally, Table 5.1 (Optimized) presents the numerical ratios of improvement for all configurations.

The results should be interpreted as follows:

- When *DruidJITVM* and *MirrorJITVM* are close (*e.g. Slopstone*), it indicates that the quality of our generated JIT compiler is similar to the handwritten one. Conversely, significant discrepancies reveal opportunities for further optimization in our generated JIT.
- When *MirrorJITVM* and *ProductionJITVM* are close (*e.g. Meteor*), it suggests that the subset of compiled instructions sufficiently covers the benchmark's working set. Large gaps, on the other hand, indicate missing translations in our prototype that affect performance.

From our results, ***DruidJITVM* achieves $0.7\times$ of the performance of *ProductionJITVM***. For macro-benchmarks, we observe $0.6\times$ in *Microdown* and $0.9\times$ in *Compiler*. *MirrorJITVM*, on the other hand, performs very close to *ProductionJITVM* with a geometric mean of $0.9\times$, except for benchmarks such as *Slopstone*, *Fasta*, and *RegexDNA*, where it is slightly slower. Overall, *DruidJITVM* achieves $0.7\times$ of the performance of *MirrorJITVM*.

These results indicate that *DruidJITVM* successfully supports a representative set of language instructions (matching with *MirrorJITVM*), but *the quality of the generated JIT compiler is not yet on par with the handwritten version matured over a decade*.

Analysis. As we said, one source of this gap is the instruction coverage: *DruidJITVM* translates 77 % of the instructions that *ProductionJITVM* supports. This is mitigated in our experiment by the construction of *MirrorJITVM*. Another source is the presence of optimizations in *ProductionJITVM* that are currently missing in our metacompiled version (see Section 3.4.5):

1. *Static Type Prediction* (STP) for arithmetic and bitwise operations [Holzle 1994].
2. *Super-Instructions* (SI) for comparisons and jumps [Casey 2003].

To measure the impact of these optimizations, we recompiled the benchmark programs using unoptimized bytecode, effectively disabling cases where the VM can apply STP and SI optimizations. The results of these experiments are presented in Table 5.1 (Unoptimized).

Figure 5.4 summarizes the estimated contributions of each factor to the performance gap between *DruidJITVM* and *ProductionJITVM*. The experiments suggest that approximately 30 % of the performance gap is composed of:

- 6 % due to unimplemented VM instructions in the translation set.
- 7 % from missing optimizations, such as *Static Type Prediction* and *Super-Instructions*.
- The remaining 16 % is attributed to the quality of the generated JIT-compiled code.

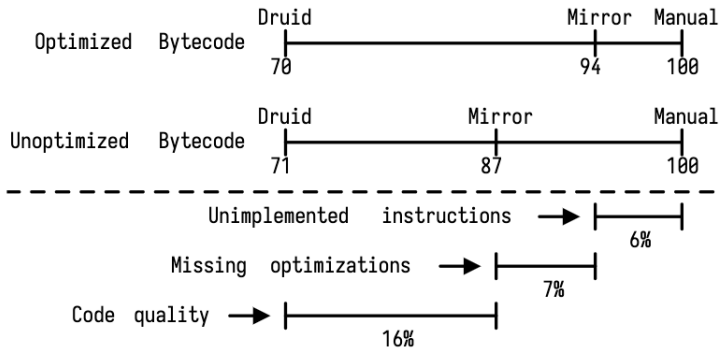


Figure 5.4: Estimating the impact of generated code quality, missing optimizations, and unimplemented instructions on the performance gap between *DruidJITVM* and *ProductionJITVM*.

RQ2 Conclusion. Our solution generates a JIT compiler that **achieves** $0.7\times$ **of the speed of a handwritten version**, matured over 10 years. By improving instruction coverage, an estimated 6 % performance gain is possible. Another 7 % could be achieved by implementing optimizations such as *Static Type Prediction* and *Super-Instructions*. The remaining performance gap comes from the quality of the generated code produced by the current prototype.

5.2.3 RQ3: Does our generated approach increase the overhead at compile time?

To ensure that our approach does not generate JIT compiler frontends with significant compilation overhead, we measured the compilation time per benchmark and calculated its percentage relative to the total execution time. We then compared these values against those obtained for *ProductionJITVM*. The results are presented in Table 5.2.

JIT compilation time represents, on geometric mean, $\sim$4 % of the total execution time for both *DruidJITVM* and *ProductionJITVM*. No significant differences were observed between the two VMs.

The largest deviations occur in micro-benchmarks such as *PiDigits* and *ThreadRing*, which were already discussed in RQ1 (Section 5.2.1) as benchmarks not particularly sensitive to JIT compilation. Additionally, the *Microdown* macro-benchmark shows a compilation overhead that is $2.3\times$ higher in *DruidJITVM* compared to *ProductionJITVM*. We plan to investigate these *pathological compilation cases* in the future.

Table 5.2: Compile time relative to total execution time. The last column shows the difference between VMs. The last three rows show the best, worst, and geometric mean (average for the difference) of each column, respectively. Lower is better.

Compile Time relative to total run time			
Benchmark	*DruidJITVM*	*ProductionJITVM*	**difference**
BinaryTrees	1.92 %	2.50 %	-0.58
Chameleons	5.22 %	4.07 %	+1.15
ChameneosRedux	8.96 %	7.06 %	+1.90
DeltaBlue	4.72 %	5.41 %	-0.69
Fasta	1.59 %	2.67 %	-1.08
KNucleotide	1.99 %	1.99 %	0.00
Mandelbrot	0.75 %	0.94 %	-0.19
Meteor	2.97 %	5.08 %	-2.11
NBody	1.46 %	1.57 %	-0.11
PiDigits	13.33 %	8.78 %	+4.55
RegexDNA	6.75 %	9.56 %	-2.81
ReverseComplement	4.73 %	7.17 %	-2.44
Richards	6.24 %	5.77 %	+0.47
Slopstone	4.64 %	8.56 %	-3.92
SpectralNorm	0.52 %	0.60 %	-0.08
ThreadRing	29.44 %	25.34 %	+4.10
Compiler	3.83 %	4.48 %	-0.65
Microdown	12.77 %	5.57 %	+7.20
Network	4.35 %	4.57 %	-0.22
Startup	5.86 %	6.63 %	-0.77
Best	0.52 %	0.60 %	-3.92
Worst	29.44 %	25.34 %	+7.20
Geomean/Average	3.98 %	4.33 %	+0.18

Figure 5.5 presents the absolute execution and compilation times for all benchmarks when running on *DruidJITVM*. Despite these outliers, the average overhead difference between *DruidJITVM* and *ProductionJITVM* is a small 0.18 *pp*, which indicates that our metacompilation process does not introduce noticeable compilation overhead.

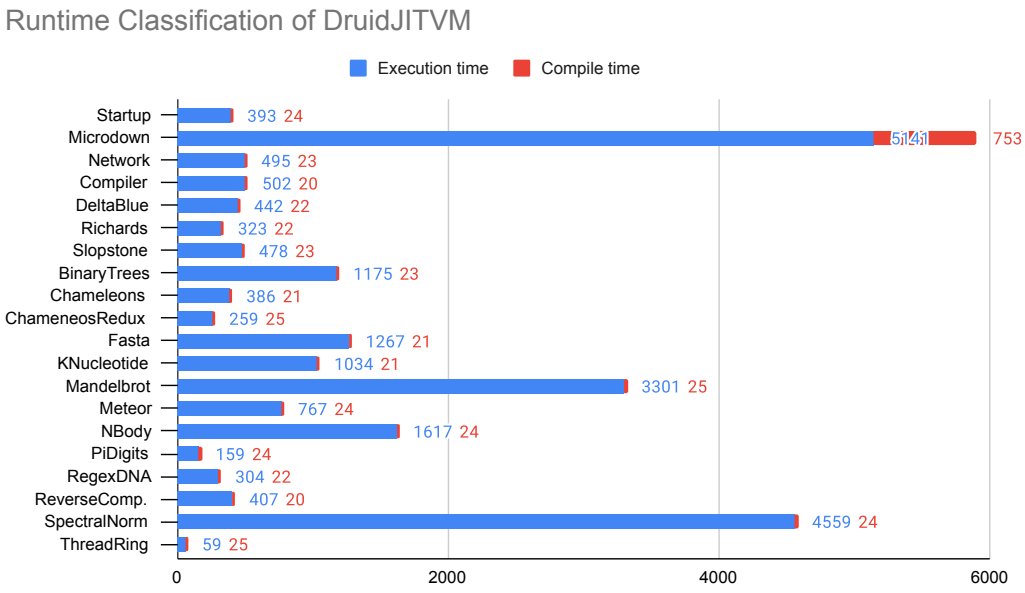


Figure 5.5: Compile and execution times for *DruidJITVM*.

RQ3 Conclusion. Druid's generated JIT compiler adds negligible 0.18 *pp* additional JIT compilation overhead, on average, compared to the production version. For most benchmarks, both solutions consume an equivalent fraction of the total execution time. This demonstrates that **our approach does not introduce significant additional compilation overhead in a production VM**.

5.2.4 RQ4: How much development effort is affected by our automatic approach?

In addition to the performance results and the relative speed-up compared to a handwritten JIT compiler, we are also interested in evaluating the impact of our solution on development effort. Traditionally, language developers needed in-depth compiler knowledge to extend the Pharo Virtual Machine, but using our approach, they can extend the interpreter and metacompile the JIT compiler versions automatically.

In this section, we estimate the development effort by measuring the amount of generated code (in KLOCs), the amount of handwritten code (in KLOCs) in the existing compiler frontend, and the number of modified interpreter call sites. Lines of code were measured by selecting the relevant Pharo methods and counting the source lines, including comments and whitespace.

Figure 5.6 presents the measurements for the relevant components, comparing the scenario with and without using Druid. We considered the following components:

Handwritten compiler frontend. All IR generator methods, identified by the gen prefix in the StackToRegisterMappingCogit hierarchy for bytecodes and in the CogObjectRepresentationFor64BitSpur hierarchy for primitives. We exclude methods in the Cogit superclass, as they belong to the backend.

Generated frontend. All methods automatically generated by Druid, identified by the gen_ prefix in the DruidJIT class.

Druid compiler infrastructure. All methods comprising Druid's compiler infrastructure, located in the Druid package. Tests, experimental features, and Cogit extensions are excluded from this count.

Druid Cogit extensions. All methods related to extending the Druid's compiler infrastructure for metacompilation targeting the Cogit backend, from the Druid-Cogit and Druid-Staging subpackages.

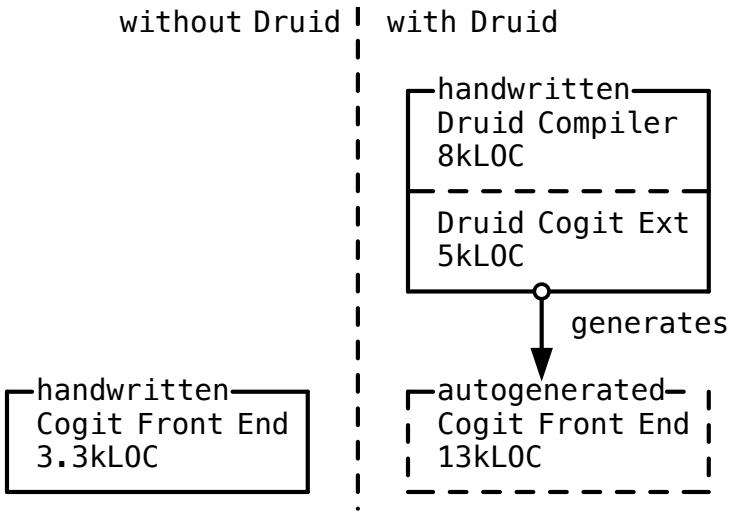


Figure 5.6: Lines of code for the different parts of our solution. Druid is a handwritten compiler comprising 8KLOC of infrastructure plus 5KLOC of Cogit-specific extensions. It generates 13KLOC for the JIT compiler frontend, equivalent to the 3.3KLOC handwritten in the production VM.

In the following discussion, we contextualize these measurements. While manually written code tends to leverage abstractions and avoid code duplication for maintainability, Druid performs aggressive inlining during code generation, resulting in larger generated methods.

Separation of concerns. The JIT compiler is one of the most complex components to maintain within a virtual machine [Kennedy 2001]. By eliminating the need to handwrite the compiler frontend, we lower the entry barrier for many developers who are not VM or compiler experts but wish to contribute to the language. In the context of an open-source project like Pharo, this *democratization* of language development is an important goal. Thanks to Druid's architecture, language developers can focus on annotating the interpreter without requiring compiler expertise to obtain a JIT compiler. Meanwhile, compiler developers maintain Druid and the VM components, significantly reducing the maintenance of language-specific implementations.

In our experiments, the JIT compiler frontend generated by **Druid comprises 13.4KLOC, compared to 3.3KLOC handwritten and maintained** by VM compiler developers over the last decade. This approximate 4x difference arises from Druid's aggressive inlining strategy, which trades manual abstractions for automatically expanded code.

Interpreter and runtime maintenance. Our solution minimizes the impact on language developers, who only need to use intrinsics and annotations to guide the metacompilation process. Throughout our experiments, we ensured that interpreter modifications remained backward compatible. To estimate the cost of adapting the interpreter, we collected all call sites where intrinsics were applied (see Section 4.3.1.2). These modifications amounted to only **60 call sites**. Nevertheless, while the barrier for producing JIT compilers is reduced, achieving good performance still requires knowledge in language implementation, profiling, and optimization.

Trading-off metacompiler maintenance. The primary trade-off of our approach is shifting maintenance effort from handwritten compiler frontends to maintaining a metacompiler infrastructure. Druid consists of a modular compiler architecture totaling **13KLOC of Pharo code**. Out of this, 8KLOC implements standard SSA-based compiler infrastructure, including IR instructions and optimization passes, that is reused in broader compilation contexts [1]. The remaining 5KLOC implements metacompilation-specific passes, which handle the interpretation of VM semantics, intrinsic processing, and code generation targeting Cogit.

This trade-off is mitigated for several reasons. First, maintaining Druid and the Pharo language can be performed by developers with distinct skill sets. While the Pharo VM code, written in Slang [Miranda 2018], restricts the use of advanced libraries, Druid itself is written in full Pharo, allowing developers to use rich abstractions and idiomatic code. Second, compiler generation approaches as Druid, are adaptable for evolving existing JIT compilers. VM developers can evolve the

[1] We are now experimenting with building an optimizing bytecode compiler based on Druid infrastructure.

metacompiler, and the frontends will be automatically regenerated to match the target backend. Finally, Druid is implemented as a traditional multi-pass compiler infrastructure, which makes it applicable in other metacompilation or code-generation scenarios beyond its current scope.

RQ4 Conclusion.
Druid is an optimizing metacompiler composed of 13K lines of pure Pharo code. It generates a total of 13.4KLOC of JIT compiler frontend, equivalent to 3.3KLOC handwritten in the production VM. Maintaining and evolving these lines of code requires a different skill set than language interpreter maintenance, which only involved modifications in 60 call sites. The size of custom Druid extensions remains in the same order of magnitude as the manually written frontend, while successfully separating language semantics from compilation concerns. Moreover, Druid's traditional multi-pass compiler design allows it to be leveraged in other contexts beyond this work. All in all, we believe that such an architecture makes language VMs **easier to maintain and evolve in the long run**.

5.3 Conclusion

In this chapter, we evaluated the viability of our metacompilation approach for baseline JIT compilers, as presented in Chapter 4. We used Druid to generate the baseline JIT compiler frontend for the Pharo VM and compared its performance against the interpreter and the handwritten JIT compiler developed and maintained by VM-compiler experts over the last 10 years, as discussed in Chapter 3. From our evaluation, we derived the following conclusions:

Improving interpreter performance. The generated baseline JIT compiler improves the interpreter's execution performance by $1.8\times$, achieving up to $5\times$ faster.

No increased compilation overhead. Our solution does not introduce additional JIT compilation overhead when compared to the handwritten JIT compiler. This is largely due to the metacompilation process being performed ahead-of-time, without introducing runtime overhead during execution.

Trade-off in compiler maintainability. Regarding development effort and software maintainability, our approach trades 3.3KLOC of handwritten JIT compiler frontend code for 5KLOC of extensions to the Druid architecture to support metacompilation targeting the Cogit backend. Although this represents an increase in code size, the design enables a clear separation of concerns, allowing language developers to focus on the interpreter semantics without requiring knowledge of

low-level, register-based compiler implementations. It is essential, however, to annotate the interpreter code with intrinsics and metadata to guide the metacompilation process effectively, as is common in similar generative approaches.

Towards metacompilation of JIT compiler optimizations. While the generated JIT compiler achieves significant improvements over interpretation, the comparison with the handwritten JIT highlights room for further enhancements. Implementing additional optimizations in the metacompiler architecture, and supporting the metacompilation of optimizations already present in the interpreter (*i.e. Static Type Prediction* and *Super-Instructions*), are key next steps to closing this performance gap.

In the last part of this thesis, we extend Druid to target the *Cogit abstract interpreter* at compile time. The abstract interpreter performs peephole optimizations between bytecodes, as well as we will be able to metacompile the *Static Type Prediction* optimizations implemented in the interpreter. We will describe the implementation details and evaluate the impact of abstract interpretation on JIT compilation within the context of our metacompilation approach.

Part III

Metacompilation targeting Abstract Interpretation

CHAPTER 6

Baseline JIT Compilation with Abstract Interpretation

Contents

In Chapter 4, we presented our metacompilation approach and the design of Druid, our prototype, which automatically generates baseline JIT compilers from interpreters. We used Druid to automatically generate the baseline JIT compiler for the Pharo VM and compare it with the handwritten version. We analyzed that part of the performance gap is due to optimizations implemented in the handwritten JIT compiler, but absent in our generated version (see Section 5.2.2).

In this part, we extend our metacompilation approach to target the Cogit abstract interpreter at JIT compile time. Abstract interpretation is a powerful technique to write fast baseline JIT compilers that need to compile early and as fast as possible, by implementing in a single pass optimizations such as register allocation, constant propagation, and instruction scheduling.

This chapter complements the Cogit description introduced in Chapter 3, focusing specifically on its abstract interpretation-based baseline JIT compilation. We present how abstract interpretation is applied to enable optimizations, such as

peephole optimization between bytecodes and *Static Type Prediction*, via dynamic register allocation during code generation, and we identify the challenges this imposes on our metacompilation approach.

Later, Chapter 7 describes the adaptations required in our metacompiler to target both *abstract interpreters* and *direct translators*, in combination with *Static Type Prediction* optimizations. Chapter 8 evaluate the benefits of this technique by generating several JIT compiler variants for the PharoVM, by making fairly comparable VMs, and reducing implementation noise.

6.1 Stack-based Abstract Interpreter CogRTL

In this chapter, we extend the CogitRTL syntax from Chapter 3 to include abstract interpretation operations. The complete syntax is shown in Figure 6.1. Input programs consist of two kinds of instructions: *concrete* instructions (such as Move, Ret, previously introduced) and *abstract* stack instructions (such as absPush and absPop), which manipulate an abstract stack.

$$
\begin{array}{rcl}
r & = & \text{a virtual register} \\
c & = & \text{a constant} \\
v & = & r \mid c \\
Input & = & inst^* \\
inst & = & conc\_inst \mid abs\_inst \\
conc\_inst & = & \mathsf{Push}(v) \mid \mathsf{Pop}(r) \mid \mathsf{Move}(v, r) \mid \mathsf{Load}(c, r, r) \mid \mathsf{Ret} \\
abs\_inst & = & \mathsf{absPush}(v) \mid \mathsf{absPop}(r)
\end{array}
$$

Figure 6.1: Extended CogRTL syntax including abstract interpretation operations.

The abstract machine is represented as a three-tuple consisting of: (1) the list of remaining input instructions, (2) an abstract stack, and (3) a list of generated concrete instructions.

Figure 6.2 shows the operational semantics (*reductions*) for each input instruction, based on the syntax in Figure 6.1. We use tuple notation and pattern matching for list manipulation on the instruction sequence and the stack.

The first three rules in Figure 6.2 show the reduction of concrete instructions by producing the same instruction in the output instruction list. Extending the model for other kinds of concrete instructions is trivial and follows the same pattern.

The last two rules show the code behavior of the abstract interpreter: absPush does not modify the instruction list. Instead, it pushes the value to the stack of the abstract machine. Conversely, the absPop instruction takes the value at the top of the stack and generates a Move instruction.

$$P \vdash \langle\langle Ret, \mathcal{I}s\rangle, \mathcal{S}tack, \mathcal{O}s\rangle \hookrightarrow \langle \mathcal{I}s, \mathcal{S}tack, \langle \mathcal{O}s, Ret\rangle\rangle \quad [\textit{genRet}]$$

$$P \vdash \langle\langle Push(v), \mathcal{I}s\rangle, \mathcal{S}tack, \mathcal{O}s\rangle \hookrightarrow \langle \mathcal{I}s, \mathcal{S}tack, \langle \mathcal{O}s, Push(v)\rangle\rangle \quad [\textit{genPush}]$$

$$P \vdash \langle\langle Pop(r), \mathcal{I}s\rangle, \mathcal{S}tack, \mathcal{O}s\rangle \hookrightarrow \langle \mathcal{I}s, \mathcal{S}tack, \langle \mathcal{O}s, Pop(r)\rangle\rangle \quad [\textit{genPop}]$$

$$P \vdash \langle\langle absPush(v), \mathcal{I}s\rangle, \mathcal{S}tack, \mathcal{O}s\rangle \hookrightarrow \langle \mathcal{I}s, \langle v, \mathcal{S}tack\rangle, \mathcal{O}s\rangle \quad [\textit{abs\_push}]$$

$$P \vdash \langle\langle absPop(r), \mathcal{I}s\rangle, \langle top, Tail\rangle, \mathcal{O}s\rangle \hookrightarrow \langle \mathcal{I}s, Tail, \langle \mathcal{O}s, Mov(top, r)\rangle\rangle \quad [\textit{abs\_pop}]$$

Figure 6.2: Reduction rules for input programs using abstract interpretation.

This machine halts if an absPop happens on an empty stack. It is outside of the scope of this thesis to guarantee or verify that input programs have the correct/intended semantics. However, given these machine semantics, we can define whether an input program is well-formed or not. For the rest of the thesis, all input programs must be well-formed, and the metacompiler generating such input programs should ensure this property.

Well-formedness. We say an input program is *well-formed* if its execution does not halt and the final machine's stack is empty.

6.2 Compile with Abstract Interpreters by Example

Compilation can be formulated as an abstract interpretation problem [Cousot 1977b, Cousot 2002]. This section introduces such a compilation approach using a stack-based abstract machine. The machine takes as input programs written in CogitRTL, represented as a list of stack instructions. Its goal is to perform optimizations during the code generation step, thereby avoiding the need for a separate optimization pass (*e.g.* through pattern matching). This *single-pass strategy* is particularly suitable for baseline JIT compilers, which prioritize fast compilation over long optimization phases.

To illustrate how such an abstract interpreter is used in practice, we consider an example of translating stack bytecode into machine code. The compiler operates as an abstract interpreter over the bytecode stream, where each opcode is associated with a handler function (*IR generators*) responsible for generating intermediate representation (IR) instructions, as presented in Chapter 3.

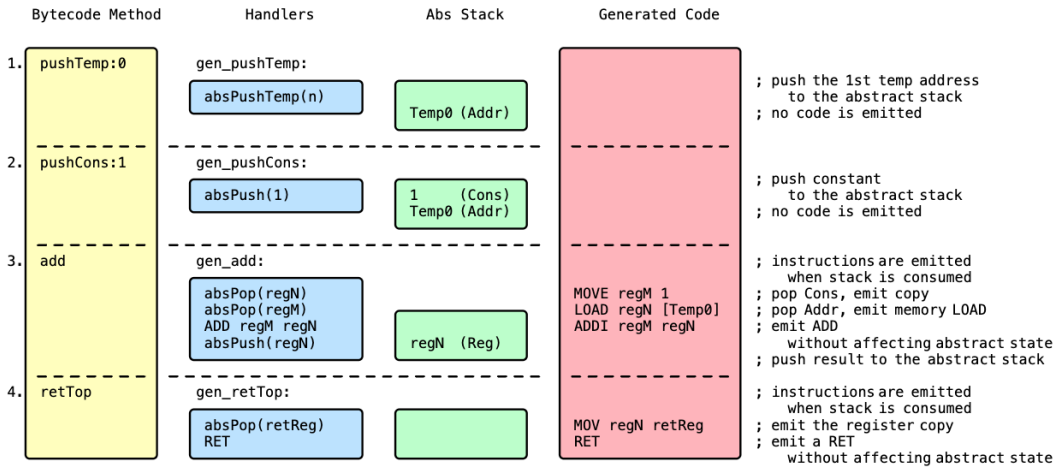


Figure 6.3: JIT compilation of a bytecode method using an abstract interpreter for the code return temp0 + 1. The abstract interpreter resolves stack operations at compile time and generates an optimized IR. The emitted machine code does not manipulate the stack at run time.

In this abstract machine, IR instructions are explicitly appended to an output stream, and stack accesses are performed over the abstract stack, as described in Figure 6.2. Instructions are emitted only when the abstract state is actually used, allowing the compiler to eliminate unnecessary stack manipulations.

Figure 6.3 shows a detailed example of compiling the expression return temp0 + 1. The left column displays the corresponding bytecode instruction sequence. Each instruction is linked, in the second column, to a compiler handler that generates the IR implementing its semantics. The third column shows the state of the abstract stack after each handler is applied. Finally, the rightmost column presents the code generated up to that point. A small comment on each step is appended at the right of the figure.

Each entry in the abstract stack is annotated with a type (in parentheses in column three of Figure 6.3) that indicates where the concrete value is located at run time. There are three possible types:

- **Register (Reg).** The value is in a register. The register must be preserved until the value is popped or spilled.
- **Memory (Addr).** The value is stored in memory. Its address is saved in the abstract stack.
- **Constant (Cons).** The value is a known constant. Its value is saved.

The example in Figure 6.3 compiles as follows:

1. pushTemp:0 - the handler pushes the address of the temporary variable to the abstract stack. No instructions are generated in the IR.
2. pushConst:1 - the handler pushes a constant to the abstract stack. No instruc-

tions are generated in the IR.

3. add - the handler pops the two entries in the abstract stack. For each of them, instructions are generated in the IR to save the concrete values in registers. The addition instruction is also generated, saving the result in a register. Then, the register is pushed to the abstract stack.
4. retTop - the handler pops the result from the abstract stack into the register reserved for the return value. It generates instructions in the IR to move the result to the expected register and, finally, the return instruction.

This example illustrates how abstract-interpreter-based JIT compilers can eliminate most of the runtime stack operations by maintaining an abstract stack at compile time. As a result, they perform *peephole optimizations* across bytecode instruction boundaries in a single forward pass, without separate optimization phases.

6.3 Dynamic Register Allocation

Cogit's abstract interpreter performs peephole optimizations by maintaining an abstract stack at compile time [Titzer 2024]. To support this, it relies on *dynamic register allocation*: registers are tracked through the abstract stack, assigned on demand, and spilled when necessary. Register allocation is scoped to the value stack, meaning that the registers referenced in the abstract stack are considered *live*.

6.3.1 Live and Free Registers

When registers are pushed onto the abstract stack, they must be preserved *i.e.* not overwritten by subsequent handlers, until they are either popped or spilled to memory.

Definition 5 (Live register) *A register is* live *if it is present in the abstract stack or is currently being used by the handler.*

The IR generator handlers targeting the abstract interpreter dynamically allocate registers by requesting a free one.

Definition 6 (Free register) *A register is* free *if it is not live,* i.e. *not present in the abstract stack and not currently used by the handler.*

The following code excerpt illustrates how generator handlers allocate registers dynamically. If all registers are currently live, the allocReg function triggers a spilling of the abstract stack until a register becomes available or aborts the compilation.

```
... "Request a free register dynamically"
freeReg := self allocReg.
... "Use the register"
self absPush: freeReg.
```

6.3.2 Spilling

In the Cogit framework, entries in the abstract stack are spilled to the runtime stack implicitly, by allocating a new free register, or explicitly, via the abstract interpreter API.

Implicit Spilling. When a register is requested using allocReg, the Cogit's backend performs:

- If a free register is available (*i.e.* not live), it is returned;
- If not, the stack is spilled starting from its base, pushing entries to memory, updating the state in the abstract stack, until a register becomes free;
- If no registers are free after spilling all stack entries, the compilation process halts with an error, and execution will proceed on the interpreter.

Explicit Spilling. Handlers in the frontend spill the abstract stack by explicit calls to the abstract interpreter API. For example, before a *deoptimization point* (see Section 3.4.4), all the entries are spilled to the runtime stack because the execution continues interpreted.

```
...
self absStackSpill. "Spill all the entries"
self deoptimize. "Deoptimize to the interpreter"
...
```

6.3.3 Example with Spilling

From the point of view of the JIT compiler, spilling means to generate the PUSH instructions to the runtime stack, and later compensate by generating the expected POP instructions. To better understand how the abstract interpreter works, we present a case where *spilling* the abstract stack is required in Figure 6.4.

The example shows a simplified case where only one register is available to use (Reg1) and it was pushed to the abstract stack (live register). Then, the bytecode pushLit:1 requires a new free register, causing an implicit spilling of the stack.

When the gen_pushLit handler is invoked, the abstract stack already contains two entries: one in memory and another in register regN. This handler needs to allocate a new register to load the literal value. If no free registers are available, spilling is triggered starting from the base of the stack.

The first entry, Temp0, points to a memory address, so spilling emits a LOAD followed by a PUSH, using SReg, a reserved internal-use register reserved for this situation. Since no register was freed by spilling Temp0, the next entry, Val (stored in Reg0), is also spilled. A PUSH is emitted, and the abstract entry updates its location to reflect that it now resides in memory (on the runtime stack). Now that Reg0 is free, it can be reused to allocate the literal reference.

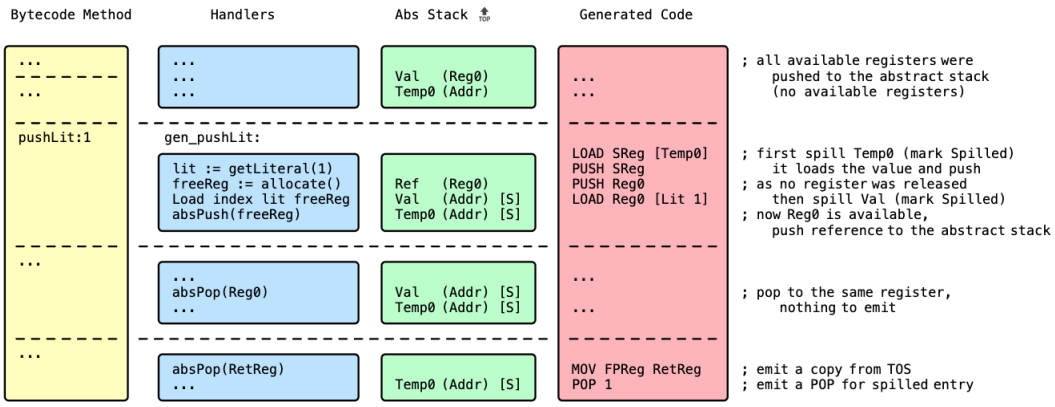


Figure 6.4: Requesting a new register causes the abstract stack to be spilled. Entries marked with `[S]` have been moved to the runtime stack.

Later, two handlers perform pops from the abstract stack. The first pops a non-spilled value that is already in the required register, so no instructions need to be emitted. The second pops a spilled value, so a MOV and a POP are emitted to synchronize the runtime stack and the abstract state.

6.4 Optimizations using JIT Abstract Interpreters

Cogit's abstract interpreter performs peephole optimizations across bytecodes in the JIT compiler backend, as explained in Section 6.2. In addition, handlers implement optimizations in the frontend, such as *Static Type Prediction* and *Super-Instructions* introduced in Chapter 3.

By using the information on the abstract stack at JIT compile time, *optimizations are staged* and decisions are made at compile time, while others are deferred to run time. As production-ready implementations could be so complex, in this section, we describe simplified versions.

All these decisions executed at different stages make the metacompilation of optimizations harder, as we describe in the next section.

6.4.1 *Static Type Prediction*

Implementing *Static Type Prediction*, compiler implementors statically speculate at compile time the likelihood of types from the used operator (*e.g.* the operators of an addition operation are likely to be integers), and statically inline common operations and type guards based on those assumptions.

For example, Figure 6.5 presents an excerpt of a bytecode handler for the addition, analogous to Figure 3.8 in the interpreter, that also implements *Static Type Prediction* optimization for integers. We observe:

- Checks on the pushed abstract values are performed at compile time to know if they are constants and integers (lines 4 - 11);
- The case when both values are constants and integers, the addition is performed at compile time, including overflow checks (lines 13 - 27);
- The case when some operand is constant but not an integer, the optimization skipped (lines 29 - 31);

```
1  JITCompiler >> genBytecodeAdd
2    | argIsConst rcvrIsConst argConst rcvrConst result |
3    "Get compile-time information about the operands"
4    argIsConst := (self absValue: 0) type = Constant.
5    rcvrIsConst := (self absValue: 1) type = Constant.
6    argIsInt := argIsConst and: [
7      argConst := (self absValue: 0) constant.
8      memory isIntegerObject: argConst ].
9    rcvrIsInt := rcvrIsConst and: [
10     rcvrConst := (self absValue: 1) constant.
11     memory isIntegerObject: rcvrConst ].
12
13   "If both objects are constants and SmallIntegers,
14        resolve them at compile time"
15   (argIsInt and: argIsConst and: rcvrIsInt and: rcvrIsConst) ifTrue:
16     [
17       rcvrConst := memory integerValueOf: rcvrConst.
18       argConst := memory integerValueOf: argConst.
19       result := rcvrConst + argConst.
20       "If not overflow, pop operands and push the result"
21       (memory isIntegerValue: result) ifTrue:
22         [
23           self absPop: 2.
24           self absPush: (memory integerObjectOf: result)
25           ^ 0 ].
26       "Overflow, send normal message"
27       ^ self genSpecialSelectorSend ].
28
29   "If we know that any object is not SmallInteger, send a normal
       message"
30   ((rcvrIsConst and: rcvrIsInt not) or: [argIsConst and: argIsInt not]) ifTrue:
31     [ ^ self genSpecialSelectorSend ].
32
33   "Other cases where the optimization is compiled,
34        checks are executed at run time"
35   ...
```

Figure 6.5: Excerpt of bytecode handler for the addition with *staged Static Type Prediction* for integers.

6.4.2 *Super-Instructions*

Bytecode handlers implement *Super-Instructions* in the JIT compiler via the abstract stack through the abstract stack at compile time. For example, if a Boolean object was pushed as a constant by a previous handler, then the subsequent conditional jump is folded at JIT-compile time.

Figure 6.6 shows an excerpt of the handler for a conditional jump bytecode, analogous to Figure 3.10 in the interpreter. We observe:

- Checks on the pushed abstract values are performed at compile time to know if the top of the stack is a boolean object (lines 6 - 9);
- The case when the object is a boolean, a jump to the target, or No-OPeration is compiled based on the value (lines 11 - 19);

```
1  JITCompiler >> genJumpIfFalse
2    | distance target objIsBoolean boolean |
3    distance := ...
4    target := distance + 1 + bytecodePC.
5
6    "Get compile time information about the object"
7    objIsBoolean := self absTop type = Constant and: [
8      self absTop constant = objectMemory trueObject or: [
9        self absTop constant = objectMemory falseObject ] ].
10
11   "If object is a boolean,
12        generate jump to correct target"
13   objIsBoolean ifTrue: [
14     boolean := self absTop constant.
15     self absPop: 1. "Pop the boolean"
16     boolean = objectMemory falseObject
17       ifTrue: [ self genJump: target ]
18       ifFalse: [ self genNop ].
19     ^ 0 ].
20
21   "Slow path, check at run time"
22   ...
```

Figure 6.6: Excerpt of bytecode handler for the conditional jump with *staged Super-Instructions*.

6.5 Challenges of Metacompilation targeting Abstract Interpreters

We illustrate the challenges of metacompiling baseline JITs with a concrete example: the bytecode pushNewArray instruction from the Pharo VM. This bytecode pops the top N elements from the stack, stores them in a newly allocated array, and pushes the resulting array back onto the stack.

```
Interpreter >> pushNewArrayBytecode
  | size array |
  size := self fetchByte.
  array := (self fitsInNewSpace: size)
    ifTrue: [ self fastInstantiateArray: size ]
    ifFalse: [ self slowInstantiateArray: size ].

  0 to: size – 1 do: [ :i | | element |
    element := self popStack.
    array at: size – i – 1 put: element ].

  self push: array
```

```
JITCompiler >> gen_pushNewArrayBytecode
  | size r1 jump jumpFalse |
  "Initialization code"
  ...
  "Fetch is staged"
  size := self fetchByte.

  "Possible runtime call"
  self absStackSpill.

  "Runtime branch"
  jumpFalse := self gen_fitsInNewSpace: size.
  self gen_fastInstantiateArray: size.
  jump := self gen_jump.

  "Runtime branch"
  jumpFalse target: self Label.
  "Runtime call"
  self
    gen_runtimeCall: #slowArray: with: size.

  "Runtime merge point"
  jump target: self Label.

  "Pops are staged"
  0 to: size – 1 do: [ :i | | abstractValue |
    abstractValue := self absPop.
    self
      gen_store: ResultReg
      at: size – i – 1
      put: abstractValue
  ].

  "Push the array"
  self absPush: ResultReg
```

Figure 6.7: Bytecode for creating a new array with N elements popped from the stack. Implementation in the interpreter (left) and in the baseline JIT compiler frontend (right).

On the left side of Figure 6.7, the interpreter implementation is straightforward: lines 3-6 allocate the array, lines 8-10 perform the loop to populate it by popping stack elements, and line 12 pushes the resulting array.

On the right side of the figure, we show a simplified version of the corresponding baseline JIT compiler frontend. Although it implements the same semantics, the code is more intricate due to staging and the need to interact with runtime services and abstract stack optimizations.

The methods prefixed with gen_ emit instructions to be executed at runtime. Because the Pharo VM uses a *generational scavenger* [Ungar 1984], it distinguishes between fast and slow object allocation paths. The fast path uses a bump-pointer allocator, while the slow path manages a larger memory space using a free tree. In the JIT, only the fast path is compiled; the slow allocation path is delegated to a runtime call.

To efficiently compile this bytecode using the abstract interpreter optimizations, the JIT compiler must:

1. **Stage the fetch of the array size** (line 6), so it becomes a known constant at compile time.
2. **Spill the abstract stack** (line 9) before the conditional allocation, because the slow path introduces a runtime call (lines 19-20). Importantly, spilling must occur *before* the branching point to ensure that the abstract state is consistent at the control-flow merge point (line 23).
3. **Stage the loop** (lines 26-32) to unroll its body at compile time (by being interpreted N times) and the resulting machine code is fully linearized. This ensures that the absPop operations are resolved during JIT compilation. In this case, this is mandatory for preserving the abstract and runtime stacks synchronized, as we explain in the next chapter.

In our approach, generating an equivalent JIT compiler frontend from the interpreter requires the metacompiler to embed these decisions into the generated code. In particular, these opportunities come at the cost of introducing constraints on the JIT compiler frontends that our metacompiler generates. To ensure correct translation of instruction handlers, **our metacompiler must preserve the invariants imposed by the abstract interpreter**, which we summarize as follows:

- **Synchronized abstract states at merge points.** *The abstract states from different branches must not conflict at control-flow merge points.* This implies that the abstract stack must not be modified (pushed, popped, or spilled) within runtime-generated branches.

- **Spilling before runtime interactions.** *The abstract stack must be spilled before any call to the runtime system.* This ensures that, at execution time, all values are stored in memory and accessible to runtime services.

- **Synchronization between abstract and runtime stacks.** *Operations on the abstract stack must preserve the same order as those on the runtime stack.*

After linearization of the control-flow graph at the end of the metacompilation pipeline, the order in which the stack access instructions (read/pop/push) are executed must remain consistent with the runtime behavior.

6.6 Conclusion

In this chapter, we presented the Cogit abstract interpreter for JIT compilation. We illustrated, through an example, how it improves the quality of JIT-compiled code by applying *peephole optimizations* across bytecode instructions. The abstract interpreter is also used to implement optimizations such as *Static Type Prediction* and to stage decisions at JIT compile time.

In the next chapter, we explain how Druid was extended with custom passes to target the Cogit abstract interpreter. We also describe the metacompilation of a first version of handlers with *Static Type Prediction*, as implemented in the interpreter.

Then, in Chapter 8, we compare different versions of JIT compilers generated by Druid, targeting either *abstract interpreters or direct translators*, showing that abstract interpretation techniques are indeed beneficial for baseline JIT compilers.

CHAPTER 7

Metacompilation Targeting Abstract Interpreter

Contents

In the previous chapter, we presented the JIT compilation strategy based on abstract interpretation. The Cogit abstract interpreter performs optimizations at JIT compile time (*e.g.* peephole optimizations across bytecodes), but it also imposes structural constraints on how the JIT compiler frontend must be written.

In this chapter, we detail the extensions implemented in Druid, introduced in Chapter 4, to target the Cogit abstract interpreter. We explain how our metacompiler generates JIT compiler frontends that conform to the coding style and structural invariants required by abstract interpretation.

7.1 Invariants of Cogit's Abstract Interpreter

This section summarizes the key design decisions of the Cogit JIT abstract interpreter (discussed in Chapter 6) that directly influence our ahead-of-time metacompilation approach for baseline JIT compilers.

The goal of our metacompiler is to generate, from interpreter implementations, semantically equivalent JIT compiler frontends. When targeting a JIT-compile-time abstract interpreter, the metacompiler must preserve the invariants imposed by it to ensure correct and optimizable output.

Sync abstract states at merge points. *The abstract stacks from different branches must not diverge at control-flow merge points.*

The Cogit JIT abstract interpreter does not perform automatic merging of abstract states at merge points. Instead, it relies on the JIT compiler frontend (therefore on our metacompiler) to ensure that the abstract state is consistent across all control paths.

This implies that the abstract stack must not be modified (*e.g.* via push, pop, or spill operations) inside a runtime branch without compensating instructions in the other branch.

Spilling at runtime interactions. *The abstract stack must be spilled to the runtime stack before invoking any runtime call.*

Runtime services, such as deoptimization or garbage collection, expect stack values to be in memory. It is the responsibility of the JIT compiler frontend to ensure this by spilling the abstract stack before any such call.

This becomes non-trivial because runtime calls are often conditionally executed within branches, and due to the previous invariant, stack spilling must occur *before* the branch to maintain a consistent abstract state at the merge point.

Sync abstract and runtime stacks. *The abstract stack must be accessed at JIT compile time in the same order as the runtime stack.*

This synchronization becomes critical during the linearization of the control flow graph at the end of the metacompilation pipeline. Since the compiler's control flow may differ from the runtime's, ensuring that abstract stack operations (read, pop, push) follow the same ordering as at runtime is essential to preserve correctness.

In the following section, we present the concrete transformations implemented in our metacompiler to enforce these invariants and generate correct JIT compiler frontends for abstract interpretation.

7.2 Transformations Preserving Cogit Invariants

We leverage the multi-pass optimizing compiler architecture of Druid (see Chapter 4) to implement transformation passes to preserve the invariants required by Cogit's abstract interpreter. We extended and added dedicated passes executed at the end of the metacompilation pipeline, illustrated in Figure 7.1, after the staging and before the lowering. As the rest of the transformations, these operate on the Druid IR, a SSA-based control flow graph.

In this section, we first describe some considerations about how abstract interpretation is affected by *staged branches*. Later, we introduce the following trans-

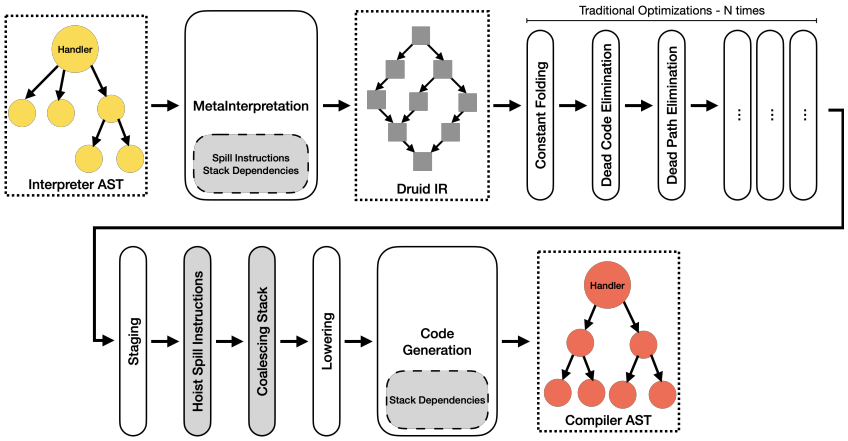


Figure 7.1: Pipeline of the metacompiler. Marked passes preserve the Cogit's abstract interpreter invariants.

formations to preserve Cogit invariants in the presence of the abstract interpreter at compile time:

- Hoisting spill instructions;
- Coalescing stack instructions; and
- Scheduling stack dependencies.

7.2.1 Staged Branches and Linearization

As discussed in Section 4.3.3, Druid stages all non-runtime-dependent code to JIT compile time. Particularly, when conditional branches are staged, they are executed at JIT compile time, becoming actual conditionals (if, while) in the JIT compiler code [Chambers 2002]. As a result, *the generated machine code inside the branch is linearized*.

Figure 7.2 illustrates the cases, for the same IR, with and without staging a conditional branch instruction.

On top, the condition is not staged; it is evaluated at execution time in the JIT compiled code. Both branches must be generated, *i.e.* interpreted by the abstract interpreter. Thus, the abstract stack has to be synchronized with the runtime at the merge point.

At the bottom, the condition is staged for compile time, evaluated by the abstract interpreter. Thus, only one branch is compiled (*i.e.* interpreted). As a consequence, the compiled code does not have branches, so the merge point does not exist, and collision between abstract states does not occur.

The fact that the generated code depends on *staged conditional branches*, *i.e.* what the abstract interpreter effectively interprets at JIT compile time, invariants regarding runtime branches do not apply because they do not exist in the compiled code.

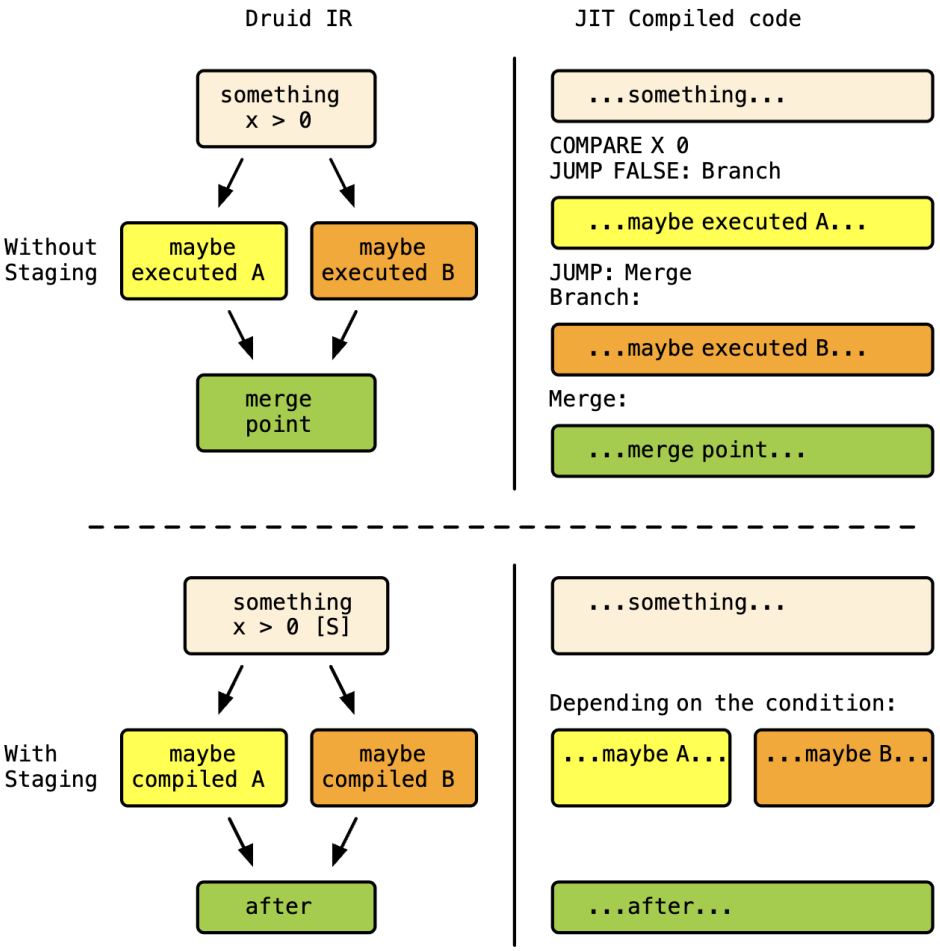


Figure 7.2: Druid IR with a conditional branch, with and without staging, and its compiled code. Compiled code is linearized by staging the conditional jump `[S]`.

Loops. In the case that the condition instruction belongs to a loop header, its body is interpreted N times at JIT compile time and *unrolled* into linear code. Thus, abstract stack operations are synchronized with the runtime stack at the loop's exit point.

As a general example, Figure 7.3 shows a simplified version of the metacompiled pushArray bytecode handler. This handler presents a runtime branch to allocate the array because the Pharo VM implements a *generational scavenger* [Ungar 1984] with two ways to allocate objects (lines 3-13). In this case, one branch is performing a runtime call (line 10), thus the abstract stack needs to be spilled before, as we explain in the following section. Later, a loop is staged to fill the array with the pushed elements (lines 15-20). As the abstract interpreter evaluates the loop's body N times, it is unrolled, and the compiled code is linear.

```
JITCompiler >> gen_pushNewArrayBytecode
  ...
  "Runtime branch (true)"
  jumpFalse := self gen_fitsInNewSpace: size.
  self gen_fastInstantiateArray: size.
  jump := self gen_jump.

  "Runtime branch (false)"
  jumpFalse target: self gen_label.
  self gen_runtimeCall: #slowArray: with: size.

  "Runtime merge point"
  jump target: self gen_label.

  "Staged loop"
  0 to: size – 1 do: [ :i | | value |
    value := self absPop.
    "Code generation is interpreted N times"
    self gen_store: ResultReg at: size – i – 1 put: value ].
  "NO runtime merge point"
  ...
```

Figure 7.3: Simplified bytecode handler for creating an array from stack elements. The code contains a *runtime branch* for the array creation, and a *staged loop* for filling it.

As a side note, if the loop depends on runtime values and the body depends on the runtime stack, staging becomes impossible, and the abstract interpreter cannot synchronize abstract and runtime states. In that case, Druid raises a compile-time error and aborts metacompilation.

7.2.2 Hoisting Spill Instructions

Before invoking a runtime call (*e.g.* garbage collection, message send), all the entries in the abstract stack must be spilled into the runtime stack. To implement this behavior, we introduce a new IR instruction, SpillStack that is metacompiled as an *explicit spilling* call to the abstract interpreter. We extended the metainterpreter to insert a SpillStack before each runtime call instruction during IR construction.

However, runtime calls often occur within conditional branches, where spilling breaks the synchronization between the abstract and runtime stacks at the merge point. To solve this problem, we *hoist* all SpillStack instructions to their *closest critical dominator basic block*, as shown in Figure 7.4.

If multiple branches contain spill instructions, they are merged into a single one inserted before the control flow diverges. This hoisting ensures abstract state consistency at merge points due to their entries are spilled to the same memory

addresses, eliminating conflicts before branching.

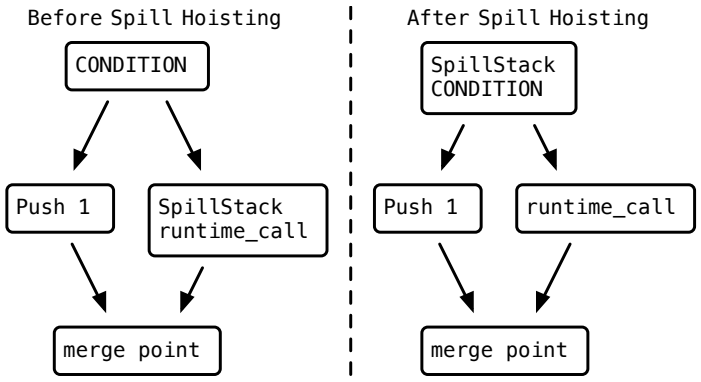


Figure 7.4: Spilling the abstract stack to the runtime stack. The hoisting pass moves SpillStack instructions outside branches to synchronize abstract states at merge points.

The cost of this transformation is that it may spill values unnecessarily in some branches, slightly increasing runtime pressure. Nevertheless, it significantly simplifies the synchronization of the stacks at the merge point.

Message sends. As introduced in Chapter 3, message sends are compiled using Polymorphic Inline Caches (PICs) [Hölzle 1991a] supported by the Cogit framework. These are implemented as *trampolines*, which act as runtime calls and are metacompiled accordingly.

However, message sends in machine code follow a *calling convention* different from the interpreter (see Section 3.4.5). Particularly, all entries in the abstract stack must be spilled to runtime, except the receiver and arguments, which could travel by registers if they fit. We refer to this process of taking care of the calling convention as *marshalling*.

Moreover, receiver and arguments are not present in the runtime stack after the message send (due to the marshalling or because they were popped in the trampoline). Thus, we synchronize the abstract stack by popping the entries, *without affecting the runtime stack*. We extended the abstract interpreter with a method absPop: n spill: false to perform pops only in the abstract stack, and then synchronize it with the runtime stack.

Thus, we implemented a specialized metacompilation for these instructions. The example in Figure 7.5 shows the metacompiled handler for a message send with one argument. The key aspects are:

- All entries in the abstract stack are spilled to memory, except the receiver and arguments at the top (whose number is known at metacompilation time), before calling the trampoline (line 3).

- After the trampoline (lines 5-8), abstract pops are performed without affecting the runtime stack (spill: false at line 9) because values were already popped from the runtime stack by the trampoline.
- The trampoline places the result of the message send into the ReceiverResultReg, which the Druid's register allocator accounts for during lowering (line 10).

```
1  JITCompiler >> gen_SendLiteralSelector1ArgBytecode
2    "AutoGenerated by Druid"
3    self absSpillStackExceptTop: 2.
4    self marshallSendArguments: 1.
5    self
6      genMarshalledSend: 0
7      numArgs: 1
8      sendTable: ordinarySendTrampolines.
9    self absPop: 2 spill: false.
10   self absPushRegister: ReceiverResultReg.
11   ^ 0
```

Figure 7.5: JIT compiler generator handler for a message send. Receiver and arguments in the abstract stack are particularly managed due to polymorphic inline caches. Pops after the call affect only the abstract stack for synchronization with the runtime.

The particular pops that only affect the abstract stack (spill: false) introduce issues for the *coalescing pass* of stack instructions, described next.

7.2.3 Coalescing Stack Instructions

To prevent divergence of abstract states across branches, we coalesce all instructions affecting the stack (Push, Pop) to their *closest critical post-dominator basic block*. The transformation identifies stack instructions inside branches and moves them to a merge point using a post-dominator tree. This guarantees that abstract stack operations occur outside of conditional branches.

Coalescing Push instructions. Figure 7.6 illustrates coalescing for Push instructions. Values to be pushed are first assigned to SSA temporaries. A phi function merges these values, and the resulting value is pushed after the merge [Cytron 1991].

This solution increases register pressure since phi functions require values to be copied into the same register, possibly consuming more registers.

Coalescing Pop instructions. Pop instructions are similarly moved to merge points. Unlike Push, no value merging is needed, as illustrated in Figure 7.7.

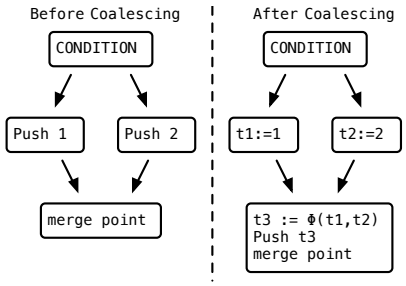


Figure 7.6: Coalescing Push instructions inside branches using phi functions.

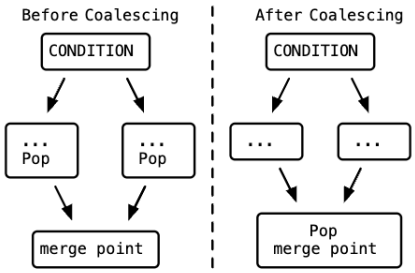


Figure 7.7: Coalescing Pop instructions inside branches.

Unspilling. A complex case arrives on coalescing Pop instructions if one of them has (spill: false). For example, when only one branch performs a message send.

We created *unspill instructions* to unspill entries from the stack. That means to pop entries in the runtime stack, and mark them as unpilled without popping the abstract stack.

Figure 7.8 illustrates the transformation to synchronize the runtime and abstract stack at the merge point by *unspilling* one branch. This ensures consistency: all branches perform the same runtime stack operations, enabling a clean Pop without spilling at the merge point.

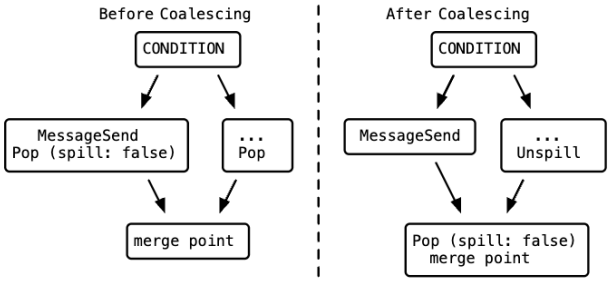


Figure 7.8: Coalescing Pop instructions using *unspilling*.

7.2.4 Scheduling Stack Dependencies

To ensure that the order of abstract stack operations matches runtime behavior, Druid tracks stack-instruction dependencies during metainterpretation. Each abstract stack instruction (read, push, pop) is linked to its predecessor, forming an explicit dependency chain as presented in Figure 7.9. This dependency information is used during the linearization phase, at the end of the pipeline, to emit abstract stack operations in the correct order.

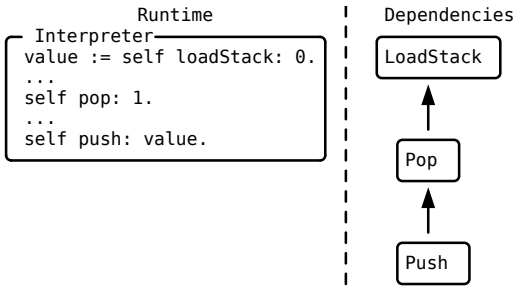


Figure 7.9: Dependencies between stack instructions based on interpreter code.

7.3 Metacompiling *Static Type Prediction*

In this section, we present an integral example involving the transformations presented in this chapter. Thanks to these extensions, we use Druid to metacompile, for example, the interpreter bytecode handler for the addition, including *Static Type Prediction* for integers, presented in Section 3.3.4. The metacompiled JIT compiler frontend code is available, at the end of this chapter, in Figure 7.11.

The metacompilation of this handler, including the optimization, is complex because it contains two main branches that affect the stack:

- *Fast path*: in case that the operands are integers, the result is computed for checking overflow. If ok, the operands are popped and the result pushed.
- *Slow path*: it is resolved as a message send.

Figure 7.10 illustrates the transformations applied, at the end of the pipeline, to metacompile the addition bytecode. After the traditional optimization passes, Druid performs the following transformations to target Cogit's abstract interpreter.

1. Hosting the SpillStack instruction from message send;
2. Coalescing the Push instructions;
3. Coalescing the Pop instructions with unspilling.

We aim to metacompile the *Static Type Prediction* optimizations implemented in the interpreter to close the gap between the baseline JIT compiler frontend generated by Druid and the handwritten by experts, evaluated in Chapter 5 (see Sec-

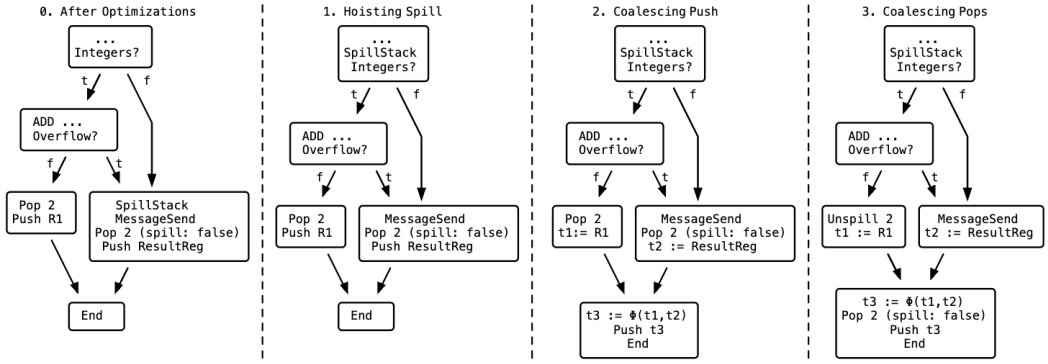


Figure 7.10: Druid transformations to target Cogit's abstract interpreter for the metacompilation of the addition bytecode.

tion 5.2.2). However, it is important to notice that our metacompiled versions of handlers implementing *Static Type Prediction* are not as sophisticated as the hand-written implementations, presented in Figure 6.5.

In our version, the optimization checks are always compiled at run time. Druid does not yet support decisions based on the type of entries in the abstract stack to check for constants. How to express constant checks using our approach is part of the future work of this thesis.

7.4 Conclusion

In this section, we presented the transformation passes implemented in our metacompiler to guarantee the invariants imposed by the Cogit abstract interpreter introduced in the previous chapter.

The transformation passes are:

- **Coalescing stack instructions.** To merge push instructions outside branches.
- **Hoisting spill instructions.** To spill, outside branches, the abstract stack to memory before runtime calls.
- **Scheduling stack dependencies.** To perform the abstract-stack-based operations in the same order as at run time.

All those transformations manage the cases where conditional branches are staged to compile time, thus the generated code is linearized and the invariants are saved. *e.g.* by staging a loop, the code generated inside the body is unrolled at run time.

In the following chapter, we evaluate the benefits and drawbacks of abstract interpreter baseline JIT compilers by metacompiling several compilers with different features from the same interpreter. Particularly, Druid generates JIT compiler frontends in either *direct or abstract style* to adapt the code to the API of the direct translator or the abstract interpreter, respectively.

```
JITCompiler >> gen_BytecodeAdd
   "AutoGenerated by Druid"

   | t0 t1 t2 jump1 jump2 jump3 currentBlock live |
   live := 0.
   t0 := self allocateReg: live.
   live := live bitOr: (self registerMaskFor: t0).
   t1 := self allocateReg: live.
   live := live bitOr: (self registerMaskFor: t1).
   t2 := self allocateReg: live.
   live := live bitOr: (self registerMaskFor: t2).

   "Load values"
   (self absValue: 1) copyToReg: t0.
   (self absValue: 0) copyToReg: t1.

   self MoveR: t0 R: t2.
   self AndR: t1 R: t2.
   self absSpillStackExceptTop: 2. "Spill for message send"
   self TstCq: 1 R: t2. "Test for both integers"
   jump1 := self JumpZero: 0.

   "Perform the integer addition"
   self AddCq: -1 R: t0.
   self AddR: t1 R: t0.
   jump2 := self JumpOverflow: 0.

   "No overflow, success"
   self absUnspillStackAt: 0. "Unspill argument"
   self absUnspillStackAt: 1. "Unspill receiver"
   self MoveR: t0 R: t2.
   jump3 := self Jump: 0.

   "Message send"
   currentBlock := self Label.
   jump1 jmpTarget: currentBlock.
   jump2 jmpTarget: currentBlock.
   self marshallSendArguments: 1.
   self genMarshalledSend: -1 numArgs: 1 sendTable: ordinarySendTrampolines.
   self MoveR: ReceiverResultReg R: t2.

   "Pop values and push result"
   currentBlock := self Label.
   jump3 jmpTarget: currentBlock.
   self absPop: 2 spilled: false.
   self absPushRegister: t2.
   ^ 0
```

Figure 7.11: JIT compiler generator handler performing *Static Type Prediction* for integer values metacompiled from Figure 3.8.

CHAPTER 8

Experiment with Abstract Interpreter

Contents

In previous chapters, we study the impact of abstract interpretation for metacompiled baseline JIT compilers.

We presented, in chapter Chapter 6, how the abstract interpreter included in the Cogit framework performs peephole optimizations to reduce the accesses to memory during exection.

However, targeting the metacompilation to the Cogit abstract interpreter imposes a restricted style to generate the JIT compiler frontend. This *abstract style* of writing JIT compilers imposes more constraints than just the abstract interpreter API. We presented the transformation passes added to the metacompilation pipeline to preserve the Cogit abstract interpreter invariants in Chapter 7.

In this chapter, we investigate the benefits and drawbacks of abstract interpreter by metacompiling, from the same interpreter, different versions of baseline JIT compilers with different features.

The evaluation of our experiments answers the next **research questions**:

- RQ1: Does abstract interpretation reduce code size?
- RQ2: Does abstract interpretation improve performance?
- RQ3: Does abstract interpretation introduce JIT compile time overhead?

Precisely, we use our metacompiler to generate the different baseline JIT compilers for the Pharo VM presented in Chapter 3. We build **comparable versions of**

the same VM using our metacompilation approach to generate baseline JIT compilers targeting both *direct translators* and *abstract interpreters*. Druid generates JIT compiler code in either *direct or abstract style* to adapt the code to the API of the direct translator or the abstract interpreter, respectively.

8.1 Experimental Design Overview

To answer our research questions, we propose to metacompile different JIT compiler versions of the Pharo VM and compare them in (a) generated code size, (b) performance and (c) JIT compile-time. We considered the following variations.

Direct translation Handlers are generated in a *direct style*, using only concrete instructions from the CogitRTL. Data is shared between different bytecode instructions by emitting push/pop instructions that manipulate the runtime stack.

Abstract stack translation Handlers are generated in an *abstract style*, using the extended CogitRTL instructions with abstract stack manipulations. Data is shared between different bytecode instructions by pushing/popping values to the abstract stack. This version includes all the transformations discussed in Chapter 7.

Moreover, we study how the effects of these translators compound with metacompiled *Static Type Prediction*s [Holzle 1994].

8.2 Methodology

Benchmark Configurations We configured Druid to generate JIT compiler targetting direct translators and abstract interpreters, with and without *Static Type Prediction*. This leaves us with 5 VM variants:

- ONLY INTERPRETER. A Pharo VM without JIT compilation. It is used as a baseline for execution speed.
- JIT ABSTRACT. Pharo VM using a baseline JIT with Abstract Interpretation, performing abstract stack optimizations to avoid accessors to the stack at runtime.
- JIT DIRECT. Pharo VM using a baseline JIT with Direct Translators, emitting all instructions declared by the JIT compiler.
- JIT ABSTRACT WITH *Static Type Prediction*. Same as JIT ABSTRACT with *Static Type Prediction* optimization for special messages.
- JIT DIRECT WITH *Static Type Prediction*. Same as JIT DIRECT with *Static Type Prediction* optimization for special messages.

Benchmark Programs We use each VM to run a set of well-known benchmark programs and a set of macro benchmarks, described in Appendix D, using the Rebench framework [Marr 2016, Marr 2018]. We run each benchmark using 30 VM iterations. Micro benchmarks use 10 in-process with two (ignored) warmup iterations. This number of iterations showed enough to measure the behavior of the baseline JIT compilers. We use the test suites of several Pharo packages as macro benchmarks. Macro benchmarks use a single in-process iteration.

Hardware Configuration We build each VM and run the benchmarks using a Linux Debian 5.10.140-1 distribution on a Intel(R) Xeon(R) CPU E5620 @ 2.40GHz processor with 16 GB DDR3 of RAM.

8.3 Results

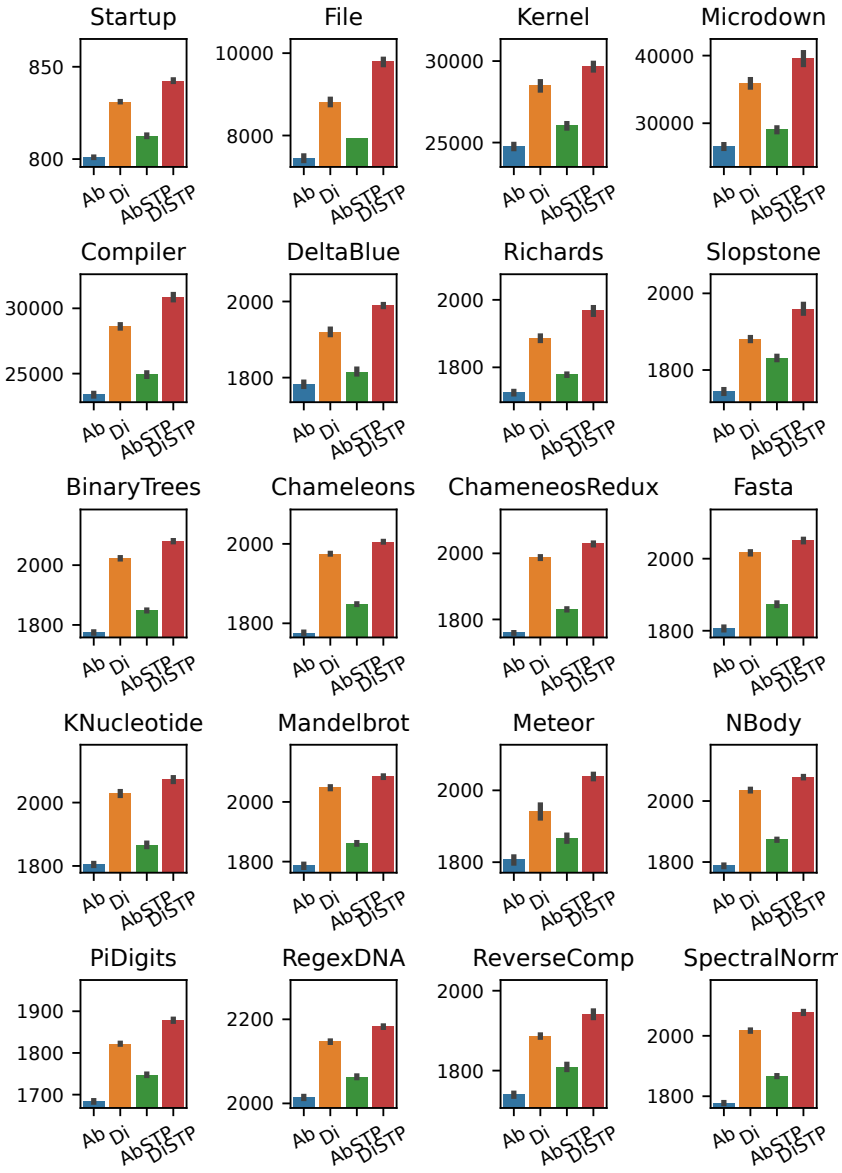
In this section, we show the results of comparing the VMs using different metacompiled JIT compilers. We answer the research questions by measuring the emitted instructions size (jitted code), execution time, and JIT compiler overhead.

8.3.1 RQ1: Does abstract interpretation reduce code size?

Figure 8.1 presents the emitted instruction size by JIT compilation for each configuration per benchmark. Values are in MB. Lower is better. The first two bars describe JIT ABSTRACT and JIT DIRECT, respectively. The next two are the same, but including *Static Type Prediction*.

Abstract interpretation reduces the size of emitted instructions for every benchmark compared to direct translations. There are 12% fewer emitted instructions, and 11% with *Static Type Prediction*, using a geometric mean. MICRODOWN has the best result with 26% on emitted instruction reduction, meanwhile the worst case is BENCHREGEXDNA with ~5% of reduction.

RQ1 Conclusion. Targetting JIT compilation to abstract interpreters reduces the emitted machine code size by 12%, using a geometric mean, up to 26%. All our benchmarks showed a reduction in the emitted machine code size, being ~5% the worst case. Using *Static Type Prediction* on top of abstract interpretation further increases the emitted instruction size by 1%.

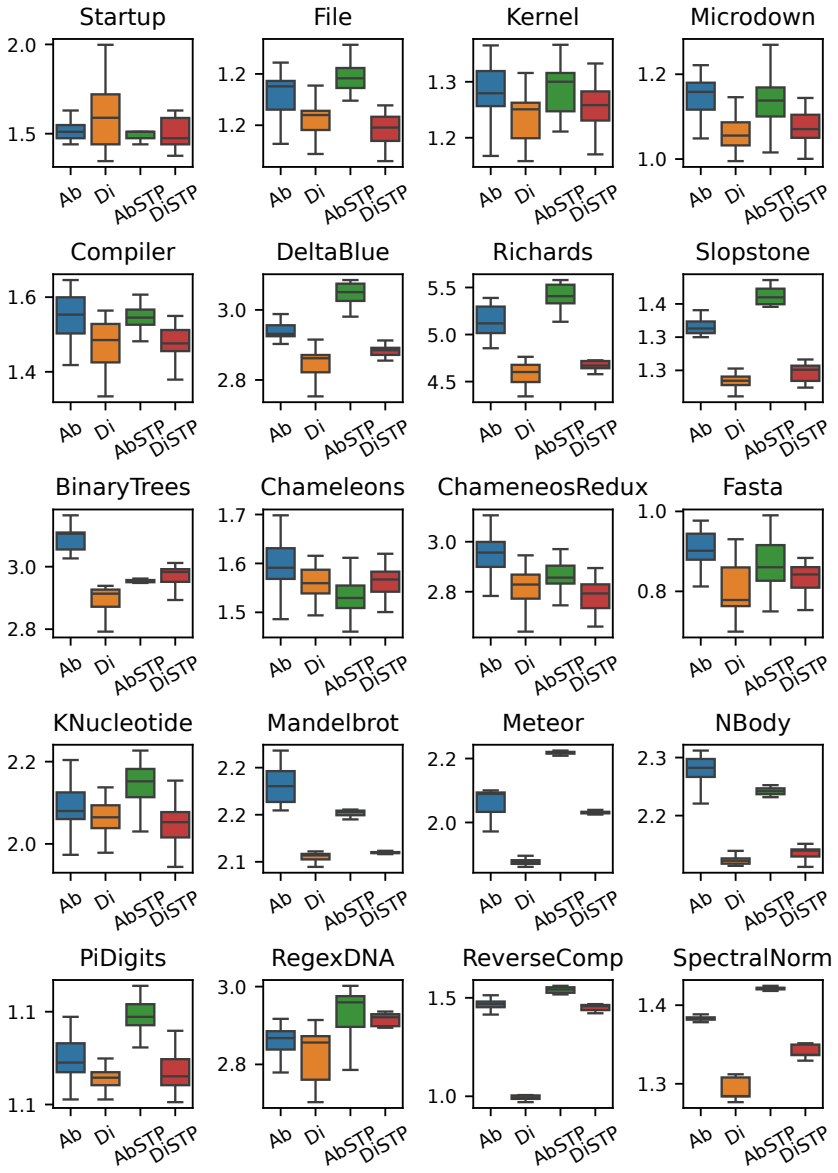


`Ab = JIT ABSTRACT; Di = JIT DIRECT; *STP = with Static Type Prediction`

Figure 8.1: Emitted code size by JIT compilation (MB). Lower is better.

8.3.2 RQ2: Does abstract interpretation improve performance?

Figure 8.2 presents the speed-up of each configuration relative to ONLY INTERPRETER runtime, per benchmark. Higher is better. Benchmark results reveal that JIT DIRECT executes, on geometric mean, 1.7x faster, up to 4.7x. JIT ABSTRACT is 1.8x faster, up to 5.1x. Thus, JIT ABSTRACT increases by 10%, up to 30%, the speed of JIT DIRECT.



`Ab = JIT ABSTRACT; Di = JIT DIRECT; *STP = with Static Type Prediction`

Figure 8.2: Speed-up relative to ONLY INTERPRETER. Higher is better.

We observe negative results for the same pathological cases not sensitive to JIT compiation presented in Chapter 5. For the *Fasta* benchmark, using JIT compilation is slower than ONLY INTERPRETER. Moreover, JIT ABSTRACT is 5% slower than JIT DIRECT for *PiDigits*.

Applying *Static Type Prediction* optimizations, the execution speed increases 2%, up to 7%, for JIT ABSTRACT. The optimization has more effect in JIT DI-

RECT, with a speed-up of 4%, up to 30%. In both cases, the worst case is 5% slower. This small improvement of the optimization is evidence of why the hand-written JIT compiler takes more decisions at JIT compile time about when to apply this optimizations (see Figure 6.5). It is one line of research for the future work of this project.

RQ2 Conclusion. Using abstract interpreters for JIT compilation increases the execution speed by 10%, up to 30%. JIT compilers performing *Static Type Prediction* optimizations further increases the execution speed by 2%, up to 7%. The performance of some benchmarks is not sensitive to JIT compilation, presenting a slowdown between 5% and 20% compared with ONLY INTERPRETER.

8.3.3 RQ3: Does abstract interpretation introduce JIT compile time overhead?

We measure the time that the JIT compiler is running to compute the impact on the overhead of our solution. Figure 8.3 shows the ratio of JIT compile time over the total execution time, per benchmark. Lower is better.

Most of the benchmarks present a JIT compilation overhead less or equal to 1%. The overhead on MICRODOWN is ~10%, being the case with the most overhead, probably due to a *pathological case*. STARTUP and COMPILER present an overhead of 5%.

Our results do not show significant differences between the configurations. The biggest differences appeared in the MICRODOWN benchmark, where JIT DIRECT increases the overhead by 1% of the total execution time over JIT ABSTRACT, and using *Static Type Prediction* increases another 1%.

RQ3 Conclusion. Abstract interpreters do not present a significant impact on JIT compilation overhead. The geometric mean of all compilation overheads is below 1%. One benchmark represents the worst case of 10% overhead due to a *compilation pathological case*. Targetting JIT compilation to abstract interpreters does not show significant differences of this overhead, neither performing *Static Type Prediction*.

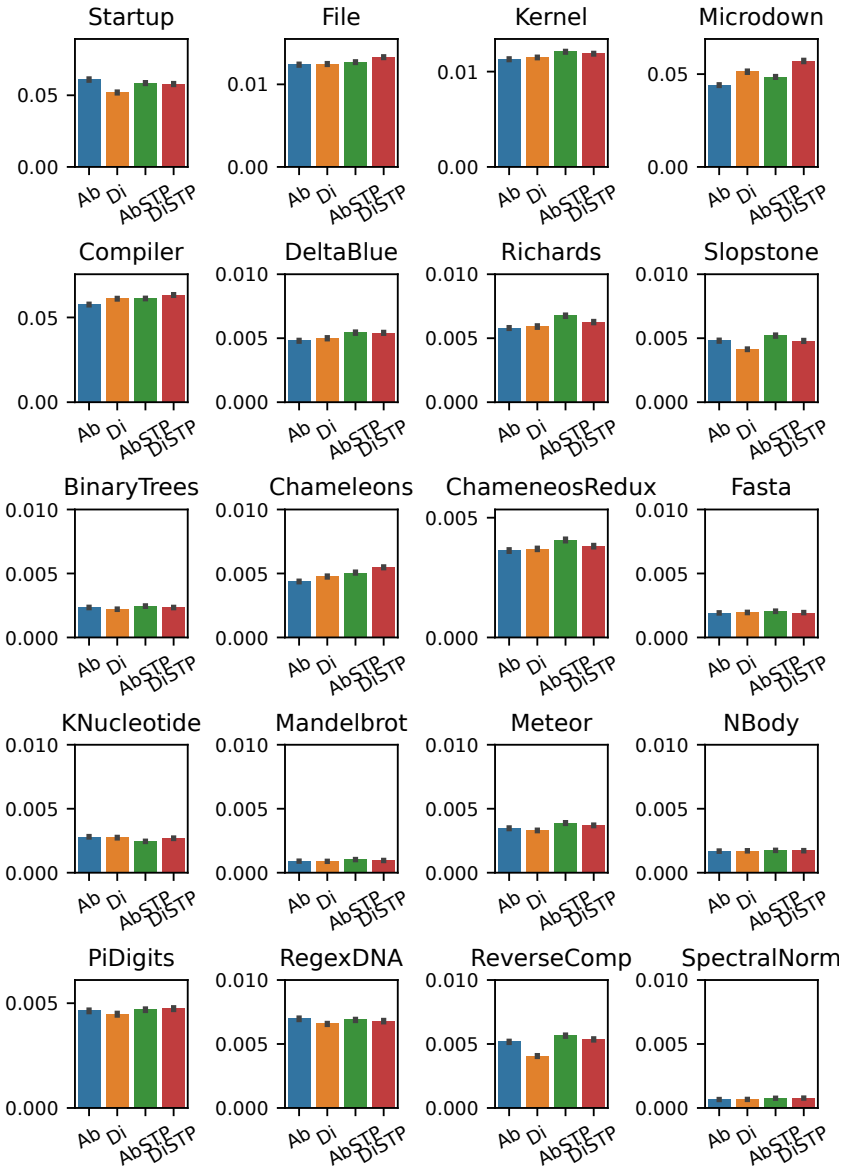


Figure 8.3: Ratio of JIT compilation time relative to total execution time. Lower is better.

`Ab = JIT ABSTRACT; Di = JIT DIRECT; *STP = with Static Type Prediction`

8.4 Conclusion

In this chapter, we explored the benefits of abstract interpretation in baseline JIT compilers. We use a metacompilation technique for targeting existing compile-time abstract interpreters and compare its performance over direct translators. Thus,

using the same metacompiler and same interpreter definitions, we aim to reduce implementation noise and experientation efforts.

To evaluate our work, we generated many versions of the Pharo VM using different baseline JIT compilers with different configurations. Then we measured the emitted instructions size (jitted code), execution time, and JIT compiler overhead to compare the implementations.

Our benchmarks reveal that baseline JIT compilers using compile-time abstract interpreters reduce the emitted machine code size by 12% on average (geomean), and increase the execution speed up to 30%, without harming JIT compilation overhead, over direct translators. On top of this, *Static Type Prediction* optimizations increase the emitted instruction size by an extra 1% while improving execution speed by an additional 2% on average (geomean), up to 7%.

CHAPTER 9

Conclusion

Contents

In this chapter, we summarize the findings of our research and present the future work following these findings.

9.1 Summary

In this thesis, we presented Druid, a metacompiler generating baseline JITs from interpreters. Druid introduces interpreter annotations and intrinsics to guide metacompilation. Our experiments target Cogit, an existing JIT compiler backend used in the Pharo VM. Our solution stages computations to be executed at JIT compile time, and finally lowers code to CogRTL, generating a baseline JIT compiler frontend.

Our generated baseline JIT compiler improves interpreter performance by $1.8\times$, up to $5\times$. We show that our metacompilation approach is feasible and reduces the development effort: *e.g.* the Pharo interpreter required only 60 semantic-preserving changes in the interpreter, mostly refactorings and code annotations. With this schema, VM maintenance is decoupled between the implementors of a particular language and the developers of the compiler infrastructure.

Later, we explored the metacompilation of baseline JIT compilers targeting compile-time abstract interpreters. We used Druid to generate, from the same interpreter, variations of the baseline JIT compilers based on abstract interpretation and direct translators. We evaluated the impact of different JIT-compile-time optimizations, reducing implementation efforts and noise. We target metacompilation to the existing Cogit's abstract interpreter, preserving its imposed invariants. We described the adaptations required in our metacompilation to target abstract interpreters or direct translators, and how interpreter optimizations, like *Static Type Prediction*, are metacompiled to the baseline JIT compiler.

Our benchmarks reveal that baseline JIT compilers using compile-time abstract interpreters reduce the emitted machine code size by a geometric mean of 12%, and increase the execution speed up to 30%, without harming JIT compilation overhead, over direct translators. On top of this, *Static Type Prediction* optimizations increase the emitted instruction size by an extra 1% while improving execution speed by an additional 2%, up to 7%.

Druid demonstrates that it is possible to automatically generate a production-quality baseline JIT compiler frontend directly from interpreter definitions, preserving language semantics while reducing maintenance effort and leveraging an existing JIT backend.

In the following lines, we summarize the research problems discussed in this thesis and our contributions per chapter.

Chapter 2

- We presented the multi-tier virtual machine architecture based on an interpreter and a JIT compiler;
- We describe different approaches for building this kind of virtual machine, visiting previous works using frameworks and generative techniques.

Chapter 3

- We presented the architecture of the Pharo VM, used for our experiments;
- We described the interpreter and the baseline JIT compiler based on instruction handlers;
- We presented the Cogit framework and its register transfer language used to write the instruction handlers in the JIT compiler.

Chapter 4

- We presented our metacompilation approach for generating baseline JIT compilers from interpreters;
- We described the architecture of Druid, our metacompiler prototype, used to generate a baseline JIT compiler for the Pharo VM;
- We listed the implemented transformations, intrinsic functions, and hints used to guide the metacompilation process;
- We presented real metacompilation examples for handlers in the Pharo VM.

Chapter 5

- We evaluated the performance of the baseline JIT compiler generated using our metacompilation approach;
- We compared our JIT compiler against just interpreted code and the existing JIT compiler handwritten by experts and matured over a decade;
- We evaluated the performance in terms of runtime performance, development effort, and JIT compilation;
- We analyzed the performance gap between our generated baseline JIT compiler and the handwritten one, discussing the impact of the missing optimizations.

Chapter 6

- We presented the existing compile-time Cogit's abstract interpreter in the Pharo VM;
- We described the peephole optimizations implemented by the abstract interpreter at JIT compile time;
- We illustrated how it is used to take optimization decisions inside instruction handlers at JIT compile time;
- We identified the metacompilation challenges to target JIT compilation to the abstract interpreter.

Chapter 7

- We presented the invariants imposed by Cogit's abstract interpreter to our metacompiler;
- We described the extensions implemented in the metacompiler to target JIT compilation using the existing abstract interpreter;
- We presented a real example about the metacompilation of *Static Type Prediction* optimizations implemented in the interpreter.

Chapter 8

- We evaluated the benefits of abstract interpreters at JIT compile time through our metacompilation approach;
- We generated versions of the Pharo VM using baseline JIT compilers with different features from the same interpreter, reducing implementation noise;
- We compared the performance of the baseline JIT compiler using Cogit's abstract interpreter, against direct translators, in terms of the amount of jitted code, runtime performance, and JIT compilation overhead.

9.2 Future work

In this section, we present possible lines of work to continue the research on metacompilation of baseline JIT compilers.

9.2.1 Metacompiler Features

In Chapter 5, we evaluated that Druid generates a baseline JIT compiler that achieves $0.7\times$ the performance of the existing JIT compiler that has been manually hand-tuned for more than 10 years. We present possible ways to reduce the gap by improving the code of the metacompiled baseline JIT compiler.

Include compile-time abstract types. Currently, our prototype targets metacompilation to the compile-time abstract interpreter, but it does not support decisions based on the type of entries in the abstract stack (*e.g.* checks for constants). We could avoid the compilation of runtime checks by staging them to the abstract stack and improving the quality of the jitted code.

Register allocation. Our metacompiler implements a linear scan register allocation algorithm that is not optimal. We plan to implement a more advanced register allocation algorithm, like graph coloring, to improve the quality of the generated JIT compiler frontend.

Traditional optimizations. We have many well-known optimizations implemented in the metacompiler architecture (*i.e.* dead code elimination, constant folding, redundant branch elimination), but there is still room for improvements. We plan to implement more optimizations around code motion, *e.g.* partial common subexpression elimination, loop invariant code motion.

9.2.2 Virtual Machine Infrestructure

Next to the improvements on the metacompiler itself, we want to improve the development of the virtual machine. We plan to improve the JIT compiler backend, targeted by our metacompilation approach, to include more optimizations. We also aim to develop profiling tools or techniques for VM developers to ease the discovery of optimization opportunities.

Compile-time abstract interpreter. The Cogit's abstract interpreter tracks values in the stack to allocate them in registers and perform peephole optimizations between bytecodes. We plan to extend this feature to also perform local variable allocation on registers and include type information to avoid redundant checks. We also would like to reduce the constraints imposed on the frontend of the JIT

compiler (*e.g.* merge states at merge points), and then evaluate if it is better to implement those transformations in the metacompiler or be supported by the JIT compiler framework.

Profiling. We want to improve the profiling of the JITted code. For the development of our prototype, we created a tool to classify the execution time on the JIT compiled code (*e.g.* primitives, bytecodes, trampolines) and find current bottlenecks. We could use our metacompilation approach to generate a JIT compiler for profiling and get more accurate information. With the profiled information, we could experiment with different instruction sets, validate the current assumptions, and even find new opportunities for *Static Type Prediction*.

Part IV

Appendixes

APPENDIX A

Intrinsics and Annotations

Contents

This appendix presents the **intrinsic functions and annotations** supported by Druid for the baseline JIT compiler generation in the Pharo VM.

A.1 Instrinsics

We perform inlinings of invoked functions during the abstract interpretation of the VM instructions defined in the interpreter (*i.e.* metainterpretation). However, there are functions in the interpreter in which our metacompiler has defined functions for their interpretation, these are *instrinsic*.

Tables A.1 - A.5 describe all the intrinsics used in Druid for the baseline JIT compiler generation in the Pharo VM. Intrinsic's implementations are defined in the metacompiler to directly manipulate the Druid IR (control flow graph).

Table A.1: Guiding Intrinsics used during metacompilation.

Function	Description
	Druid
druidIgnore:	Ignore code during meta-interpretation.
interpreterIgnore:	Ignore code by the interpreter but meta-interpreted.
druidStageable:	Mark code as stageable for metacompilation.
druidForceInterpretation	Jump the execution to the interpreter (deoptimization).

Table A.2: Essential Intrinsics used during metacompilation.

Function(s)	Description
	Control flow
value value: cull:	Block closure activation.
ifTrue: caseOf:otherwise:	Conditional control flow.
whileTrue whileTrue:	Condition control flow for loops.
	Comparisons
= ~= < <= > >= | &	Operators for equality, inequality, less, less or equal, greater, greater or equals, or and and.
	Arithmetics
+ - * / // \ negated	Operators for addition, subtraction, multiplication, division, integer division, mod and negation.
	Bit Manipulation
<< >> >>>	Operators and functions for left shift, right shift,
bitAnd: bitOr: bitXor:	arithmetic shift, and, or, xor, mask comparison and
anyMask: rotateLeft:by:	rotation.

Table A.3: VM-specific Intrinsics used during metacompilation.

Function(s)	Description
	SmallIntegers
sumSmallInteger:with:	SmallInteger object operations for addition, subtrac-
ifOverflow:	tion and multiplication with custom overflow check.
subSmallInteger:with:	
ifOverflow:	
multiplySmallInteger:with:	
ifOverflow:	
	Stack access
stackTop stackValue:	Read values from the stack.
pop: push: pop:thenPush:	Pop or push values from the stack.
	Memory access
byteAt:put:	Load and write signed integers in 8, 32 and 64 bits.
longAt: longAt:put:	
uint8At: uint8At:put:	Load and write unsigned integers in 8, 16, 32 and 64
uint16At: uint16At:put:	bits.
uint32At: uint32At:put:	
uint64At: uint64At:put:	

Table A.4: VM-specific Intrinsics used during metacompilation (continuation).

Function	Description
	Message send
normalLiteralSelectorAt: argumentCount:	Message send of a literal selector.
normalSendSplSelector: argumentCount:	Message send of a special selector.
sendSuper:numArgs: sendDirectSuper:numArgs:	Super message sends.
	Method invocation
iframeNumArgs:	Access to the number of arguments.
iframeMethod:	Access to the method object.
itemporary:in: itemporary:in:put:	Load or store a value from a temporary variable.
fetchNextBytecode	Load the next bytecode to execute.
fetchByte	Load the next byte in the bytecode sequence.
commonReturn: commonCallerReturn:	Return from a method or closure activation.
internalMustBeBoolean:	Throws a *MustBeBoolean* exception.
checkEventsCnxtSwitch:	Execution interruptions (GC and Threads management).
	Pharo features
error: assert: deny:	VM errors and assertions.
createFullClosureInIndex: numCopied:ignoreContext:	Instantiate a Block closure object.
executeFullCogBlock: closure:mayContextSwitch:	Activate a Block closure compiled to machine code.
remember:	Adds an object in the Remembered Set [Ungar 1987].
newHashBitsOf:	Calls a function to create a hash number for an object.

Table A.5: Instrinsic variables in the metacompiler.

Variable	Description
	Execution
instructionPointer	Stores the current instruction pointer.
framePointer	Stores the current frame pointer.
stackPointer	Stores the current stack pointer.
stackLimit	Stores the limit of the stack.
argumentCount	Stores the argument count for the current method.
primFailCode	Stores the current fail code of a primitive execution.
classTableFirstPage	Pointer to the first page of the class table.
	Well known objects
nilObj trueObj falseObj	Pointers to nil, true and false objects.
hiddenRootsObj	Pointer the root objects table.
	Garbage collection
scavengeThreshold	Pointer to the threshold for Scavenger.
freeStart	Pointer to the beginning of a free bunch of memory.
newSpaceStart	Pointer to the beginning of the *New Space*.
spaceMaskToUse	Stores a mask to know if an object is in the *New*, *Old*
newSpaceMask	or *Permanent Space*.
oldSpaceMask	
permSpaceMask	
	Extension bytecodes
extA extB	Stores values for extended bytecodes.
numExtA numExtB	Stores the number of extended bytecodes executed consecutively.

A.2 Annotations

Druid uses *annotations* to declare meta-data for the VM instructions in the interpreter. The meta-data are analyzed during metacompilation to guide the meta-interpretation and to have extra information required by the JIT compiler framework.

Table A.6 describes all the annotations supported by Druid. Figure A.1 presents an extract of the IR generators bytecode table for the JIT compiler generated by Druid using extra meta-data from annotations. The JIT compiler framework uses this information during the JIT compilation of methods on runtime.

Table A.6: Annotations with metacompilation options.

Annotation	Description
	For primitives
numberOfArguments:	One or many expected numbers of arguments for the primitive method, failing JIT compilation in other cases.
customisedReceiverFor:	Conditional JIT compilation based on primitive method class.
druidExitPoint	Primitive fails at this point on runtime.
	For bytecodes
compilationInfo:	Adds extra information on bytecode declaration for the JIT compiler framework (*i.e.* mappings between bytecodes and JITted code, jumps, etc).
druidInfo:	Computes extra information during bytecode metacompilation (*i.e.* message send invocations).
needsFrameNever:	Says if the bytecode has access to the frame (if none, the frame is not created in JITted code).
needsFrameIfInBlock:	
needsFrameIfImmutability:	

```
JITFrontend >> bytecodeTable
  ^ {
    ...
    { 1. 64. 64. #gen_PushTemporaryVariableBytecode0 }.
    ...
    { 1. 88. 88. #gen_ReturnReceiver. #return. #isMappedInBlock.
     #needsFrameIfInBlock:. 0 }.
    ...
    { 1. 92. 92. #gen_ReturnTopFromMethod. #return. #isMappedInBlock.
     #needsFrameIfInBlock:. -1 }.
    ...
    { 1. 95. 95. #gen_ExtNopBytecode. #needsFrameNever:. 0 }.
    { 1. 96. 96. #gen_BytecodePrimAdd. #isMapped }.
    ...
    { 1. 128. 128. #gen_SendLiteralSelector0ArgsBytecode0. #isMapped }.
    ...
    { 1. 176. 176. #gen_ShortUnconditionalJump0. #branch. #v3:ShortForward:
     Branch:Distance: }.
    ...
    { 1. 184. 184. #gen_ShortConditionalJumpTrue0. #branch. #isBranchTrue.
     #isMapped. #v3:ShortForward:Branch:Distance: }.
    ...
    { 1. 200. 200. #gen_StoreAndPopReceiverVariableBytecode0.
     #isInstVarRef. #isMappedIfImmutability.
     #needsFrameIfImmutability:. -1 }.
    ...
    { 3. 255. 255. #unknownBytecode } }
```

Figure A.1: Extract of the bytecode table generated by Druid using meta-data.

APPENDIX B

Metacompiler Optimizations

In this appendix, we present the optimizations implemented in our metacompiler.
Table B.1 shows all implemented optimisations in our prototype.

Optimisations	
Name	**Description**
BRANCH COLLAPSE	Inline conditions in conditional jumps
CLEAN CONTROL FLOW	Merge instructions in consecutive blocks to avoid unnecessary jumps
COPY PROPAGATION	Replace copy instructions in operands by real value
DEAD BLOCK ELIMINATION	Remove inaccessible blocks
DEAD BRANCH ELIMINATION	Remove branches with only dead paths
DEAD CODE ELIMINATION	Remove unused instructions
DEAD EDGE SPLITTING	Duplicate blocks with dead and not-dead paths
FAILURE CODE TAIL DUPLICATION	Tail duplicate block with resulted code
PHI SIMPLICATION	Replace one operand phis with copy instruction
REDUNDANT COPY ELIMINATION	Remove copies of form x := x
SPARCE CONDITIONAL CONSTANT PROPAGATION	Performs constants folding and propagation

Table B.1: Optimisations implemented in metacompiler. For each optimisation, we present a brief description about the tranformation.

APPENDIX C

Examples of Metacompilation

Contents

In this chapter, we present some examples of VM instructions in the Interpreter and the version for a baseline JIT compiler metacompiled by Druid. Since the VM instructions in the PharoVM are too complex, we present simplified versions of them to show how Druid metacompiles the main language features.

C.1 Metacompilation of Primitives

Figure C.1 presents the primitive to make the addition between two SmallInteger objects (receiver and argument). At left is the definition in the interpreter and at right is the metacompiled version of that definition for the JIT compiler frontend.

The annotation numberOfArguments: defines which values are accessed by stackValue: intrinsic. It is required because, by the calling convention in the machine code, arguments (and the receiver) are passed by registers instead of the stack. Thus, values accessed by the stack are replaced by expected registers (*i.e.* ReceiverResultReg, Arg0Reg) during metacompilation.

The annotation customisedReceiverFor: indicates that this primitive only works if the receiver is an instance of a special class, SmallInteger in this case. The metacompiler optimizes it by assuming that the primitive method must be installed in the expected class and makes that check at JIT compile time (*i.e.* mclassIsSmallInteger) to avoid compilation in other cases. Several compiler optimizations remove the check of the receiver type at runtime, only the check for the argument is performed in JIT-compiled code (by inlining the isIntegerObject: function).

Finally, the intrinsic sumSmallInteger:with:ifOverflow: is metacompiled to deal with tag, perform the addition and check overflows. Metacompilation of the intrinsic pop:thenPush: moves the result to the expected register (*i.e.* ReceiverResultReg),

```
Interpreter >> primitiveAdd
  <numberOfArguments: 1>
  <customisedReceiverFor: #smallInteger>

  | rcvr arg rawInt1 rawInt2 rawResult result  |
  "Get the values from the stack
     "
  rcvr := self stackValue: 0.
  arg := self stackValue: 1.
  "Check small integer objects"
  (memory isIntegerObject: rcvr)
    ifFalse: [ ^ self primitiveFail ].
  (memory isIntegerObject: arg)
    ifFalse: [ ^ self primitiveFail ].
  "Perform the addition,
    checking overflow"
  result := self
    sumSmallInteger: rcvr
    with: arg
    ifOverflow: [ ^ self primitiveFail ].
  "Pop values and push the
    result"
  self pop: 2 thenPush: result
```

```
JITFrontend >> gen_PrimitiveAdd
  "AutoGenerated by Druid"

  | jump1 jump2 currentBlock |
  self mclassIsSmallInteger ifFalse: [ ^
     UnimplementedPrimitive ].
  self TstCq: 1 R: Arg0Reg.
  jump1 := self JumpZero: 0.
  self MoveR: Arg0Reg R: TempReg.
  self AddCq: -1 R: TempReg.
  self MoveR: ReceiverResultReg R:
     ClassReg.
  self AddR: ClassReg R: TempReg.
  jump2 := self JumpOverflow: 0.
  self MoveR: TempReg R:
     ReceiverResultReg.
  self genPrimReturn.
  currentBlock := self Label.
  jump1 jmpTarget: currentBlock.
  jump2 jmpTarget: currentBlock.
  ^ CompletePrimitive
```

Figure C.1: Metacompilation result of addition primitive for SmallInteger objects.

following the register-based calling convention. In cases where the primitive fails (*i.e.* primitiveFail), the flow jumps to the end of the generated machine code, where a fallback method will be executed.

C.2 Metacompilation of Bytecodes

Figure C.2 shows the input (left) and output (right) of the metacompilation of a bytecode responsive to push a receiver instance variable value to the stack.

The interpreter's definition uses the currentBytecode variable to compute the instance variable index. This variable contains the executed bytecode index in the bytecode table, thus the same definition is used in many entries for accessing different indexes. We metacompile different definitions for each entry by staging the currentBytecode to the current number and constant-fold it, if possible. The example in Figure C.2 was metacompiled for index = 1 in the bytecode table, so it accesses the second instance variable (0-based). This value was constant folding, after the inline of fetchPointer:ofObject: function, to offset = 16 (bytes) in the metacompiled code (8 bytes of the object header + 8 bytes of the first instance variable).

```
Interpreter >>
    bytecodePushReceiverVariable

 | fieldIndex receiver |
 fieldIndex := currentBytecode bitAnd: 16rF.
 receiver := stackPages longAt: framePointer +
    FoxIFReceiver.
 self push: (objectMemory fetchPointer:
    fieldIndex ofObject: receiver).
 self fetchNextBytecode
```

```
JITFrontend >>
    gen_bytecodePushReceiverVariable_1
 "AutoGenerated by Druid"

 | live currentBlock t0 |
 live := 0.
 self ensureReceiverResultRegContainsSelf.
 t0 := self
       allocateRegNotConflictingWith: live
       ifNone: [ ^ self unknownBytecode ].
 live := live bitOr: (self registerMaskFor: t0).
 self MoveR: ReceiverResultReg R:
    t0.
 self ssPushBase: t0 offset: 16.
 ^ 0
```

Figure C.2: Push receiver variable bytecode metacompilation with index = 1.

The way of accessing the receiver object from the frame in the interpreter is metacompiled by using the register-based calling convention (*i.e.* using ReceiverResultReg). Since previous bytecodes in the compiling method could override the register value, the definition requires calling ensureReceiverResultRegContainsSelf compiler function.

Bytecode metacompilation implies dynamic register allocation at JIT-compile time by using allocateRegNotConflictingWith:ifNone: compiler function. Each metacompiled bytecode definition keeps a variable (live) with the status of already used registers to avoid conflicts.

To push the value, Druid uses the ssPushBase:offset: function exposed by the compiler backend to track the values in the stack and perform peephole optimizations during JIT compilation.

In the end, the fetchNextBytecode intrinsic, present in every bytecode definition in the interpreter, is ignored during metacompilation due to the JIT compiler framework resolves the compilation of a sequence of bytecodes inside the method.

C.3 Using the JIT Compiler Framework

Figure C.3 introduces a conditional jump bytecode for short distances. The offset on the interpreter is computed based on the currentBytecode (the bytecode index in the table). Similar to the previously presented case, we metacompile a version for each entry in the table (with offset = 1 in the given example).

The annotation with the compilation info isMapped in the interpreter definition indicates that this bytecode needs to save the link between the bytecode index and

```
Interpreter >> shortConditionalJumpTrue
  <compilationInfo: #isMapped>
  <compilationInfo: #branch>
  <compilationInfo: #isBranchTrue>

  | boolean offset |
  offset := (currentBytecode bitAnd: 7) + 1.
  boolean := self stackTop.
  self pop: 1.
  boolean = objectMemory trueObject
    ifTrue: [ self jump: offset ]
    ifFalse: [
      boolean = objectMemory falseObject ifFalse:
      [
        self internalMustBeBoolean: boolean ].
      self fetchNextBytecode ]
```

```
JITFrontend >>
    gen_ShortConditionalJumpTrue0
  "AutoGenerated by Druid"

  | jump1 s5 s2 currentBlock s12 t0 jump2 s9 |
  self annotateBytecode: self Label.
  t0 := ... "Allocate register"
  (self ssValue: 0) copyToReg: t0.
  self ssPop: 1.
  s5 := objectMemory trueObject.
  self ssFlushStack.
  self CmpCq: s5 R: t0.
  jump1 := self JumpNonZero: 0.
  s9 := bytecodePC + 2.
  self Jump: (self ensureFixupAt: s9).
  jump2 := self Jump: 0.
  currentBlock := self Label.
  jump1 jmpTarget: currentBlock.
  s12 := objectMemory falseObject.
  self CmpCq: s12 R: t0.
  jump1 := self JumpZero: 0.
  self MoveR: t0 R: TempReg.
  self CallRT:
      ceSendMustBeBooleanTrampoline.
  currentBlock := self Label.
  jump2 jmpTarget: currentBlock.
  jump1 jmpTarget: currentBlock.
  ^ 0
```

Figure C.3: Conditional true jump bytecode metacompilation with offset = 1.

the address of the JIT-compiled code because of deoptimization possibilities. This link is created using the annotateBytecode: function, exposed by the JIT compiler framework, at the beginning of the JITFrontend definition. The deoptimization case occurs when the internalMustBeBoolean: intrinsic is executed[1]. As this case produces an exception and escapes from the fast path that we want to JIT compile, this intrinsic is metacompiled as a call to the trampoline ceSendMustBeBooleanTrampoline to manage the case by the runtime system. The trampolines are generated by the JIT compiler framework at the beginning of execution and deal with the required transformation to continue the execution in the runtime (*e.g.* deoptimize the stack frame).

The metacompiled generator handler definition in the JIT frontend uses func-

[1]In Pharo, conditionals are written as messages to boolean objects (*e.g.* boolean ifTrue: [...]). The message is compiled to bytecode as a conditional jump. The bytecode must support the case of a non-boolean object as the receiver (error case).

tions exposed by the JIT backend to perform optimizations at JIT compile time. On one side, the functions to access and modify the stack, ssValue: and ssPop:, use an abstract stack to track operations and perform peephole optimizations. As a result, we must call ssFlushStack before any potential deoptimization to ensure that values in registers are spilled to the stack in case of continuing the execution in the runtime. Conversely, we use the ensureFixupAt: function to track jumps between bytecodes and perform dead code elimination.

The JITFrontend version in Figure C.3 also presents staged operations at JIT compile time. In the expression where the bytecode to jump is computed, s9 := bytecodePC + 2, our metacompiler identifies which values can be *constant folded* at metacompile time—the final value 2 is produced by 1 (offset) + 1 (next bytecode is always avoided) —and which values must be staged for JIT compile time—the addition bytecodePC + 2 is evaluated by the JIT compiler since it does not rely on runtime information.

A similar case occurs on comparing boolean objects—the expressions objectMemory trueObject and objectMemory falseObject return a pointer to each object. Since they are special objects, they are fixed in memory and are not moved during Garbage Collection. Thus, we cannot get the value at metacompile time but at JIT compile time and inline the value in the JIT-compiled code.

C.4 Message Send

Figure C.4 shows both definitions for a *message send* bytecode. The presented bytecode is a special case for = message, used for equality comparison in Pharo.

```
Interpreter >> bytecodeSendEquals
  <compilationInfo: #isMapped>
  <druidInfo: #hasSend>

  self druidIgnore: [
    | rcvr arg |
    rcvr := self stackValue: 1.
    arg := self stackValue: 0.
    (objectMemory areIntegers: rcvr and: arg)
      ifTrue: [
       ^ self booleanCheat: rcvr = arg ] ].

  self normalSendSpecialSelector: 6
      argumentCount: 1
```

```
JITFrontend >> gen_bytecodeSendEquals
  "AutoGenerated by Druid"

  self marshallSendArguments: 1.
  self
    genMarshalledSend: -7
    numArgs: 1
    sendTable: ordinarySendTrampolines.
  ^ 0
```

Figure C.4: Special message send bytecode metacompilation for = message.

The interpreter definition of the bytecode includes the two optimizations commented in Section 5.2.2. A *static type prediction* is implemented for the case of

SmallInteger objects, performing the comparison of the tagged values. Also, *dynamic super-instructions* is implemented inside the booleanCheat: function, where a look-ahead of the next bytecode is performed to check if it is a conditional jump and directly execute it with the computed boolean value. Our metacompiler does not support these optimizations yet, so they are ignored using the intrinsic druidIgnore:. As we say in Section 9.1, we are still working to support them.

The targeting JIT compiler backend Cogit already has support for message sends. We metacompile the bytecodes with message sends by using marshallSendArguments: and genMarshalledSend:numArgs:sendTable: JIT compiler backend functions. These functions generate calls to trampolines to perform the lookup and patch the code with PICs. They also create the mapping between bytecode and compiled code, so the annotateBytecode: function is not invoked by the frontend.

C.5 Staging Conditional Branches

Figure C.5 presents a version of the push new array bytecode, *i.e.* it instantiates a new array object and fills all slots with references to nil object. The Interpreter version was simplified to focus on the staging features of the metacompiler. The JITFrontend version only shows the important parts of this section.

In previous examples, we presented that our metacompiler stages expressions to be executed at JIT compile time. For the cases where these expressions are the condition of a conditional branch, the metacompiler stages the branches too. We can see the staged branches by using ifTrue: and whileTrue: functions in the JITFrontend version in Figure C.5.

The first case is where the condition size < 0 ifTrue: in the Interpreter is metacompiled as s2 < 0 ifTrue: in the JITFrontend. Since the value of size is encoded in the 2 bytes instruction, the condition does not depend on runtime information and can be resolved at JIT compile time. In this case, the JIT compiler backend already performs the fetch of the next bytes storing them in variables for the frontend, so the fetchByte intrinsic is metacompiled as an access to the expected variable byte1. Inside the branch, the druidForceInterpretation intrinsic is metacompiled by calling deoptimize JIT compiler backend function. In that case, a trampoline is called to continue the execution on the Interpreter (slow path).

The second case of staged branches occurs during the metacompilation of the to:do: function, defined based on whileTrue: intrinsic. Here also, the condition depends only on the size variable, thus the loop is staged. In this case, the metacompiler must determine which expressions belong to the pre-header, the conditional header, and the loop body.

```
Interpreter >> bytecodePushNewArray

  | size array |
  size := self fetchByte.
  size < 0 ifTrue: [
    self druidForceInterpretation.
    "..."
  ].
  size := size bitAnd: 127.
  self fetchNextBytecode.
  array := ... "Instantiate new array"
  0 to: size – 1 do: [ :i |
    objectMemory
      storePointerUnchecked: i
      ofObject: array
      withValue: objectMemory nilObject ].
  self push: array
```

```
JITFrontend >> gen_bytecodePushNewArray
  "AutoGenerated by Druid"

  | ... |
  s2 := byte1.
  self ssFlushStack.
  s2 < 0 ifTrue: [
    self deoptimize.
    ^ 0 ].
  s7 := s2 bitAnd: 127.
  t0 := ... "Allocate register"
  t1 := ... "Allocate register"
  ... "Store new array in memory,
     saved in t1"
  s28 := s7 – 1.
  s29 := 0.
  t2 := ... "Allocate register"
  [ ((s29<=s28)) ] whileTrue: [
    self genMoveConstant: objectMemory
      nilObject R: t0.
    s34 := s29 << 3.
    self MoveR: t1 R: t2.
    self AddCq: s34 R: t2.
    self MoveR: t0 M64: 8 r: t2.
    s38 := s29 + 1.
    s29 := s38 ].
  self ssPushRegister: t1.
  ^ 0
```

Figure C.5: Push new array bytecode metacompilation (some parts of the code).

APPENDIX D
Benchmarks

Our set of classical **micro benchmarks**[1] is as follows:
BinaryTrees. An adaptation of *Hans Boehm's GCBench* for Garbage Collection.
Chameleons. Creates many threads to wait for mutex semaphores.
ChameneosRedux. Small problem size for *Chameleons* benchmark.
DeltaBlue. Classic object-oriented *constraint solver*.
Fasta. DNA sequence generation algorithm based on weighted random selection.
KNucleotide. Map the DNA letters into a hash table to accumulate count values.
Mandelbrot. Plot the Mandelbrot set on an N-by-N bitmap.
Meteor. Uses *ByteStrings* to solve a puzzle.
NBody. Classic n-body simulation of the solar system.
PiDigits. Generates and prints the first N digits of Pi.
RegexDNA. A simple regex pattern and actions to manipulate FASTA format data.
ReverseComplement. Computes the reverse complement of a DNA sequence.
Richards. Simulates a *task dispatcher* on a multitasking operating system.
Slopstones. *Smalltalk Low-level OPeration Stones*, a series of low-level operations.
SpectralNorm. Calculate the spectral norm of a N-by-N matrix.
ThreadRing. A token ring algorithm using threads and mutex semaphores.

In addition, we run some applications as **macro benchmarks**:
Compiler. Tests for source-to-bytecode Pharo compiler. (10166 tests)
File Tests for file system access library. (514 tests)
Kernel Tests for basic language features. (1537 tests)
Microdown. Tests for markup documentation library. (634 tests)
Network. Tests for network communication library. (655 tests)
Startup. Just evaluate `1 + 1`, measure the time for starting up the system.

[1] Implementations are in https://github.com/guillep/SMark visited on 2024-02-04.

Bibliography

[Aho 2006] Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman. Compilers: Principles, techniques, and tools (2nd edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[Alpern 2000] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J. D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan and J. Whaley. *The Jalapeño virtual machine*. IBM Systems Journal, vol. 39, no. 1, pages 211–238, 2000.

[Amin 2017] Nada Amin and Tiark Rompf. *Collapsing towers of interpreters*. Proc. ACM Program. Lang., vol. 2, no. POPL, December 2017.

[Ancona 2007] D. Ancona, M. Ancona, A Cuni and N. Matsakis. *RPython: a Step Towards Reconciling Dynamically and Statically Typed OO Languages*. In Proceedings of the 2007 Symposium on Dynamic Languages (DSL 07), pages 53–64. ACM, 2007.

[Appel 2002] Andrew W. Appel. Modern compiler implementation in Java. Cambridge University Press, New York, NY, USA, second édition, 2002. with Jens Palsberg.

[Arnold 2005] Matthew Arnold, Stephen J Fink, David Grove, Michael Hind and Peter F Sweeney. *A survey of adaptive optimization in virtual machines*. Proceedings of the IEEE, vol. 93, no. 2, pages 449–466, 2005.

[Bala 2000] Vasanth Bala, Evelyn Duesterwald and Sanjeev Banerjia. *Dynamo: A Transparent Dynamic Optimization System*. In Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, PLDI '00, pages 1–12, New York, NY, USA, 2000. Association for Computing Machinery.

[Bebenita 2010] Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann and Herman Venter. *SPUR: a trace-based JIT compiler for CIL*. In Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '10, pages 708–725, New York, NY, USA, 2010. ACM.

[Beckman 1976] Lennart Beckman, Anders Haraldson, Östen Oskarsson and Erik Sandewall. *A partial evaluator, and its use as a programming tool*. Artificial Intelligence, vol. 7, no. 4, pages 319–357, 1976.

[Béra 2014] Clément Béra and Eliot Miranda. *A bytecode set for adaptive optimizations*. In International Workshop on Smalltalk Technologies (IWST), August 2014.

[Béra 2017] Clément Béra. *Sista: a Metacircular Architecture for Runtime Optimisation Persistence*. PhD thesis, Université de Lille, 2017.

[Birkedal 1994] Lars Birkedal and Morten Welinder. *Hand-writing program generator generators*. In Manuel Hermenegildo and Jaan Penjam, editeurs, Programming Language Implementation and Logic Programming, pages 198–214, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[Blackburn 2004] S. M. Blackburn, P. Cheng and K. S. McKinley. *Oil and water? High performance garbage collection in Java with MMTk*. In Proceedings. 26th International Conference on Software Engineering, pages 137–146, 2004.

[Bolz 2008] Carl Friedrich Bolz, Adrian Lienhard, Nicholas D. Matsakis, Oscar Nierstrasz, Lukas Renggli, Armin Rigo and Toon Verwaest. *Back to the future in one week — implementing a Smalltalk VM in PyPy*. In Self-Sustaining Systems, volume 5142 of *LNCS*, pages 123–139. Springer, 2008.

[Bolz 2009] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski and Armin Rigo. *Tracing the meta-level: PyPy's tracing JIT compiler*. In ICOOOLPS '09: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, pages 18–25, New York, NY, USA, 2009. ACM.

[Bolz 2015] Carl Friedrich Bolz and Laurence Tratt. *The impact of meta-tracing on VM design and implementation*. Science of Computer Programming, pages 408–421, February 2015.

[Bracha 2004] Gilad Bracha and David Ungar. *Mirrors: design principles for meta-level facilities of object-oriented programming languages*. In Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM SIGPLAN Notices, pages 331–344, New York, NY, USA, 2004. ACM Press.

[Bruni 2009] Camillo Bruni and Toon Verwaest. *PyGirl: Generating Whole-System VMs from High-Level Prototypes using PyPy*. In Objects, Components, Models and Patterns, Proceedings of TOOLS Europe 2009, volume 33 of *LNBIP*, pages 328–347. Springer-Verlag, 2009.

[Brunthaler 2010] Stefan Brunthaler. *Inline Caching Meets Quickening*. In Proceedings of the 24th European Conference on Object-Oriented Programming, ECOOP'10, pages 429–451, Berlin, Heidelberg, 2010. Springer-Verlag.

[Casey 2003] Kevin Casey, David Gregg, M. Ertl and Andy Nisbet. *Towards superinstructions for Java interpreters*. Lecture Notes in Computer Science, jan 2003.

[Casey 2005] Kevin Casey, David Gregg and M. Anton Ertl. *Tiger – An Interpreter Generation Tool*. In Rastislav Bodik, editeur, Compiler Construction, pages 246–249, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[Chambers 1989] C. Chambers and D. Ungar. *Customization: Optimizing Compiler Technology for SELF, a Dynamically-typed Object-oriented Programming Language*. In Programming Language Design and Implementation (PLDI), PLDI '89, pages 146–160, New York, NY, USA, 1989.

[Chambers 1991] Craig Chambers and David Ungar. *Making Pure Object-Oriented Languages Practical*. In Proceedings OOPSLA '91, ACM SIGPLAN Notices, volume 26, pages 1–15, November 1991.

[Chambers 2002] Craig Chambers. *Staged compilation*. SIGPLAN Not., vol. 37, no. 3, page 1–8, January 2002.

[Cooper 2002] Keith D. Cooper, Devika Subramanian and Linda Torczon. *Adaptive Optimizing Compilers for the 21st Century*. The Journal of Supercomputing, vol. 23, no. 1, pages 7–22, 2002.

[Cousot 1977a] Patrick Cousot and Radhia Cousot. *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. In Principles of Programming Languages (POPL '77), pages 238–252, New York, NY, USA, 1977.

[Cousot 1977b] Patrick Cousot and Radhia Cousot. *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*. In Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery.

[Cousot 2002] Patrick Cousot and Radhia Cousot. *Systematic design of program transformation frameworks by abstract interpretation*. SIGPLAN Not., vol. 37, no. 1, page 178–190, January 2002.

[Cytron 1991] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman and F. Kenneth Zadeck. *Efficiently computing static single assignment form and the control dependence graph*. ACM Trans. Program. Lang. Syst., vol. 13, no. 4, pages 451–490, 1991.

[Deutsch 1984] L. Peter Deutsch and Allan M. Schiffman. *Efficient Implementation of the Smalltalk-80 system*. In Proceedings POPL '84, Salt Lake City, Utah, January 1984.

[Deutsch 1989] L. Peter Deutsch. *The Past, Present and Future of Smalltalk*. In S. Cook, editeur, Proceedings ECOOP '89, pages 73–87, Nottingham, July 1989. Cambridge University Press.

[Duboscq 2013] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer and Hanspeter Mössenböck. *Graal IR: An Extensible Declarative Intermediate Representation*. In Asia-Pacific Programming Languages and Compilers Workshop, 2013.

[Ducasse 2016] Stéphane Ducasse, Eliot Miranda and Alain Plantec. *Pragmas: Literal Messages as Powerful Method Annotations*. In International Workshop on Smalltalk Technologies IWST'16, Prague, Czech Republic, August 2016.

[Ducasse 2022] Stéphane Ducasse, Gordana Rakic, Sebastijan Kaplar and Quentin Ducasse. Pharo 9 by example. Book on Demand – Keepers of the lighthouse, 2022. Originally written by A. Black and S. Ducasse and O. Nierstrasz and D. Pollet with D. Cassou and M. Denker.

[Engler 1996] Dawson R. Engler. *VCODE: a retargetable, extensible, very fast dynamic code generation system*. SIGPLAN Not., vol. 31, no. 5, page 160–170, May 1996.

[Ertl 2002] M. Anton Ertl, David Gregg, Andreas Krall and Bernd Paysan. *Vmgen-a generator of efficient virtual machine interpreters*. Software: Practice and Experience, vol. 32, no. 3, pages 265–294, 2002.

[Ertl 2003] M. Ertl and David Gregg. *The Structure and Performance of Efficient Interpreters.* J. Instruction-Level Parallelism, vol. 5, November 2003.

[Frampton 2009] Daniel Frampton, Stephen M. Blackburn, Perry Cheng, Robin J. Garner, David Grove, Eliot and Sergey I. Salishev. *Demystifying magic: high-level low-level programming*. In Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, VEE '09, pages 81–90, New York, NY, USA, 2009. ACM.

[Futamura 1999] Yoshihiko Futamura. *Partial Evaluation of Computation Process: An Approach to a Compiler-Compiler*. Higher Order Symbol. Comput., vol. 12, no. 4, pages 381–391, 1999.

[Gal 2006] Andreas Gal, Christian W. Probst and Michael Franz. *HotpathVM: An Effective JIT Compiler for Resource-constrained Devices*. In Virtual Execution Environments, VEE '06, 2006.

[Geoffray 2010] Nicolas Geoffray, Gaël Thomas, Julia Lawall, Gilles Muller and Bertil Folliot. *VMKit: a substrate for managed runtime environments*. In Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, VEE '10, pages 51–62, New York, NY, USA, 2010. ACM.

[Georges 2007] Andy Georges, Dries Buytaert and Lieven Eeckhout. *Statistically Rigorous Java Performance Evaluation*. In Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. Association for Computing Machinery.

[Glück 1997] Robert Glück and Jesper Jørgensen. *An automatic program generator for multi-level specialization*. Lisp and Symbolic Computation, vol. 10, pages 113–158, 1997.

[Glück 2010] Robert Glück. *An Experiment with the Fourth Futamura Projection*. In Amir Pnueli, Irina Virbitskaite and Andrei Voronkov, editeurs, Perspectives of Systems Informatics, pages 135–150, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[Goldberg 1983] Adele Goldberg and David Robson. Smalltalk 80: the language and its implementation. Addison Wesley, Reading, Mass., May 1983.

[Groß 2024] Samuel Groß. *The V8 Sandbox*, April 2024. Accessed: 2025-07-01, https://v8.dev/blog/sandbox.

[Hölzle 1991a] Urs Hölzle, Craig Chambers and David Ungar. *Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches*. In European Conference on Object-Oriented Programming (ECOOP'91), 1991.

[Hölzle 1991b] Urs Hölzle, Craig Chambers and David Ungar. *Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches*. In P. America, editeur, Proceedings ECOOP '91, volume 512 of *LNCS*, pages 21–38, Geneva, Switzerland, July 1991. Springer-Verlag.

[Hölzle 1992] Urs Hölzle, Craig Chambers and David Ungar. *Debugging Optimized Code with Dynamic Deoptimization*. In Programming Language Design and Implementation (PLDI' 92), pages 32–43, New York, NY, USA, 1992. ACM.

[Holzle 1994] Urs Holzle. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming*. PhD thesis, Stanford University, Stanford, CA, USA, 1994.

[Ingalls 1997] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace and Alan Kay. *Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself*. In Proceedings of Object-Oriented Programming, Systems, Languages, and Applications conference (OOPSLA'97), pages 318–326. ACM Press, November 1997.

[Jones 1993] Neil J. Jones, Carsten K. Gomard and Peter Sestoft. Partial evaluation and automatic program generation. Prentice-Hall, 1993.

[Kennedy 2001] Ken Kennedy and John R. Allen. Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[Klint 1981] Paul Klint. *Interpretation Techniques*. Software: Practice and Experience, vol. 11, no. 9, pages 963–973, 1981.

[Kozak 2023] David Kozak, Vojin Jovanovic, Codrut Stancu, Tomáš Vojnar and Christian Wimmer. *Comparing Rapid Type Analysis with Points-To Analysis in GraalVM Native Image*. In Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, MPLR 2023, page 129–142, New York, NY, USA, 2023. Association for Computing Machinery.

[Latifi 2021] Florian Latifi, David Leopoldseder, Christian Wimmer and Hanspeter Mössenböck. *CompGen: generation of fast JIT compilers in a multi-language VM*. In Proceedings of the 17th ACM SIGPLAN International Symposium on Dynamic Languages, pages 35–47, 2021.

[Leone 1993] Mark Leone and Peter Lee. *Deferred Compilation: The Automation of Run-Time Code Generation*. Rapport technique CMU-CS-93-225, Carnegie Mellon University, USA, 1993.

[Lindholm 2014] Tim Lindholm, Frank Yellin, Gilad Bracha and Alex Buckley. The java virtual machine specification. Pearson Education, 2014.

[Malenfant 1992] Jacques Malenfant, Christophe Dony and Pierre Cointe. *Behavioral Reflection in a prototype-based language*. In A. Yonezawa and

B. Smith, editeurs, Proceedings of Int'l Workshop on Reflection and Meta-Level Architectures, pages 143–153, Tokyo, November 1992. RISE and IPA(Japan) + ACM SIGPLAN.

[Malenfant 1996] J. Malenfant, M. Jacques and F.-N. Demers. *A tutorial on behavioral reflection and its implementation*. In Proceedings of Reflection, pages 1–20, 1996.

[Marr 2008] Stefan Marr. Modularisierung virtueller maschinen. Master's thesis, Hasso Plattner Institute, Germany, 2008.

[Marr 2015] Stefan Marr and Stéphane Ducasse. *Tracing vs. Partial Evaluation: Comparing Meta-Compilation Approaches for Self-Optimizing Interpreters*. In International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'2015), OOPSLA'15, pages 821–839. ACM, jun 2015.

[Marr 2016] Stefan Marr, Benoit Daloze and Hanspeter Mössenböck. *Cross-Language Compiler Benchmarking: Are We Fast Yet?* In Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, pages 120–131, New York, NY, USA, 2016. Association for Computing Machinery.

[Marr 2018] Stefan Marr. *ReBench: Execute and Document Benchmarks Reproducibly*, aug 2018. Version 1.0.

[Miranda 2011] Eliot Miranda. *The Cog Smalltalk Virtual Machine*. In Proceedings of VMIL 2011, 2011.

[Miranda 2018] Eliot Miranda, Clément Béra, Elisa Gonzalez Boix and Dan Ingalls. *Two decades of Smalltalk VM development: live VM development through simulation tools*. In Proceedings of International Workshop on Virtual Machines and Intermediate Languages (VMIL'18), pages 57–66. ACM, 2018.

[Mössenböck 2002] Hanspeter Mössenböck and Michael Pfeiffer. *Linear Scan Register Allocation in the Context of SSA Form and Register Constraints*. In R. Nigel Horspool, editeur, Compiler Construction, pages 229–246, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[Musso Gualandi 2021] Hugo Musso Gualandi and Roberto Ierusalimschy. *A Surprisingly Simple Lua Compiler*. In Proceedings of the 25th Brazilian Symposium on Programming Languages, pages 1–8, 2021.

[Niephaus 2019] Fabio Niephaus, Tim Felgentreff and Robert Hirschfeld. *GraalSqueak: Toward a Smalltalk-Based Tooling Platform for Polyglot Programming*. In Proceedings of the 16th ACM SIGPLAN International Con-

ference on Managed Programming Languages and Runtimes (MPLR'19), pages 14–26, New York, NY, USA, 2019. ACM.

[Palumbo 2022a] Nahuel Palumbo, Guillermo Polito, Pablo Tesone and Stéphane Ducasse. *Ordering Optimisations in Meta-Compilation of Primitive Methods*. In FAST Workshop 2022 on Smalltalk Related Technologies, 2022.

[Palumbo 2022b] Nahuel Palumbo, Pablo Tesone, Guillermo Polito and Stéphane Ducasse. *Selecting Semi-permanent Object Candidates in Dynamically-Typed Reflective Languages*. In Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes, pages 149–149, 2022.

[Palumbo 2023] Nahuel Palumbo, Sebastian Jordan Montaño, Guillermo Polito, Pablo Tesone and Stéphane Ducasse. *Garbage Collector Tuning in Pathological Allocation Pattern Applications*. In IWST 2023: International Workshop on Smalltalk Technologies, Lyon, France, August 2023.

[Palumbo 2025a] Nahuel Palumbo and Marcus Denker. *Clean Blocks at the Opal Compiler*. In IWST 2025: International Workshop on Smalltalk Technologies, Gdansk, Poland, July 2025.

[Palumbo 2025b] Nahuel Palumbo, Guillermo Polito, Stéphane Ducasse and Pablo Tesone. *Meta-compilation of Baseline JIT Compilers with Druid*. The Art, Science, and Engineering of Programming, vol. 10, no. 1, February 2025.

[Polito 2019] Guillermo Polito, Pablo Tesone, Eliot Miranda and David Simmons. *GildaVM: a Non-Blocking I/O Architecture for the Cog VM*. In International Workshop on Smalltalk Technologies, Cologne, Germany, August 2019.

[Polito 2021] Guillermo Polito, Pablo Tesone, Stéphane Ducasse, Luc Fabresse, Théo Rogliano, Pierre Misse-Chanabier and Carolina Hernandez Phillips. *Cross-ISA Testing of the Pharo VM: Lessons Learned While Porting to ARMv8*. In Proceedings of the 18th international conference on Managed Programming Languages and Runtimes (MPLR '21), Münster, Germany, September 2021.

[Polito 2022a] Guillermo Polito, Nahuel Palumbo, Pablo Tesone, Soufyane Labsari and Stéphane Ducasse. *Interpreter Register Autolocalisation: Improving the performance of efficient interpreters*. In More VM internatlonal Workshop, 2022.

[Polito 2022b] Guillermo Polito, Pablo Tesone and Stéphane Ducasse. *Interpreter-guided Differential JIT Compiler Unit Testing*. In Programming Language Design and Implementation (PLDI'22), 2022.

[Polito 2023] Guillermo Polito, Pablo Tesone, Jean Privat, Nahuel Palumbo and Stéphane Ducasse. *Heap Fuzzing: Automatic Garbage Collection Testing with Expert-Guided Random Events*. In International Conference on Software Testing, 2023.

[Rastello 2022] Fabrice Rastello and Florent Bouchez Tichadou. Ssa-based compiler design. Springer, 2022.

[Rigo 2006] Armin Rigo and Samuele Pedroni. *PyPy's approach to virtual machine construction*. In Proceedings of the 2006 conference on Dynamic languages symposium, pages 944–953, New York, NY, USA, 2006. ACM.

[Rompf 2014] Tiark Rompf, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi and Kunle Olukotun. *Surgical Precision JIT Compilers*. In Programming Language Design and Implementation, PLDI '14, 2014.

[Seaton 2014] Chris Seaton, Michael L. Van De Vanter and Michael Haupt. *Debugging at full speed*. In Proceedings of the Workshop on Dynamic Languages and Applications, pages 1–13, 2014.

[Smith 1982] Brian Cantwell Smith. *Reflection and Semantics in a Procedural Language*. Ph.D. thesis, MIT, Cambridge, MA, 1982.

[SpiderMonkey nd] Team SpiderMonkey. *Spidermonkey*, n.d. Accessed: 2025-10-01, https://spidermonkey.dev/.

[Stallings 2011] William Stallings. Operating systems: Internals and design principles. Prentice Hall Press, USA, 7th édition, 2011.

[Thakur 2019] Manas Thakur and V. Krishna Nandivada. *PYE: A Framework for Precise-Yet-Efficient Just-In-Time Analyses for Java Programs*. ACM Transactions on Programming Languages and Systems, vol. 41, no. 3, 2019.

[Titzer 2024] B. L. Titzer. *Whose Baseline Compiler is it Anyway?* In 2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pages 207–220, Los Alamitos, CA, USA, mar 2024. IEEE Computer Society.

[Torczon 2007] Linda Torczon and Keith Cooper. Engineering a compiler. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd édition, 2007.

[Ungar 1984] Dave Ungar. *Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm*. ACM SIGPLAN Notices, vol. 19, no. 5, pages 157–167, 1984.

[Ungar 1987] David Ungar. The design and evaluation of a high performance Smalltalk system. MIT Press, Cambridge, MA, USA, 1987.

[Ungar 1992] David Ungar, Randall B. Smith, Craig Chambers and Urs Hölzle. *Object, Message, and Performance: How They Coexist in Self*. IEEE Computer (Special Issue on Inheritance & Classification), vol. 25, no. 10, pages 53–65, October 1992.

[Ungar 2005] David Ungar, Adam Spitz and Alex Ausch. *Constructing a metacircular Virtual machine in an exploratory programming environment*. In Companion to Object-Oriented Programming, Systems, Languages, and Applications conference (OOPSLA'05), pages 11–20, New York, NY, USA, 2005. ACM.

[V8 2017] Team V8. *Launching Ignition and TurboFan*, 2017. Accessed: 2025-10-01, https://v8.dev/blog/launching-ignition-and-turbofan.

[V8 2021] Team V8. *Sparkplug — a non-optimizing JavaScript compiler*, 2021. Accessed: 2025-10-01, https://v8.dev/blog/sparkplug.

[V8 2022] Team V8. *Maglev: A New JIT Compiler for V8*, 2022. Accessed: 2025-05-26, https://v8.dev/blog/maglev.

[WebKit Community 2025] WebKit Community. *JavaScriptCore - Deep Dive*. WebKit documentation, 2025. Accessed: 2025-07-29, https://docs.webkit.org/Deep%20Dive/JSC/JavaScriptCore.html.

[WebKit 2014] Team WebKit. *WebKit FTL JIT*, 2014. Accessed: 2025-10-01, https://webkit.org/blog/3362.

[Wimmer 2012] Christian Wimmer and Thomas Würthinger. *Truffle: a self-optimizing runtime system*. In Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity, pages 13–14, 2012.

[Wimmer 2013] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès and Douglas Simon. *Maxine: An approachable virtual machine for, and in, java*. ACM Transaction Architecture Code Optimization, vol. 9, no. 4, January 2013.

[Wimmer 2019] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B Kessler, Oleg Pliss and Thomas Würthinger. *Initialize once, start fast: application initialization at build time*. Proceedings of the ACM on Programming Languages, vol. 3, no. OOPSLA, pages 1–29, 2019.

[Wöss 2014] Andreas Wöss, Christian Wirth, Daniele Bonetta, Chris Seaton and Christian Humer. *An Object Storage Model for the Truffle Language Implementation Framework*. In PPPJ'14, 2014.

[Würthinger 2013] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon and Mario Wolczko. *One VM to Rule Them All*. In International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (ONWARD'13), 2013.

[Xu 2021] Haoran Xu and Fredrik Kjolstad. *Copy-and-patch compilation: a fast compilation algorithm for high-level languages and bytecode*. Proc. ACM Program. Lang., vol. 5, no. OOPSLA, October 2021.

[Xu 2023] Haoran Xu. *LuaJIT Remake: A Meta-compiled Baseline JIT with Performance Competitive to LuaJIT's Optimizing JIT*. LuaJIT Remake, May 2023. Accessed: 2025-07-29, https://sillycross.github.io/2023/05/12/2023-05-12/.

[Xu 2024] Haoran Xu and Fredrik Kjolstad. *Deegen: A JIT-Capable VM Generator for Dynamic Languages*, 2024. https://arxiv.org/abs/2411.11469.

[Yermolovich 2009] Alexander Yermolovich, Christian Wimmer and Michael Franz. *Optimization of Dynamic Languages Using Hierarchical Layering of Virtual Machines*. vol. 44, no. 12, pages 79–88, 2009.