An Efficient Implementation of a Micro Virtual Machine

Yi Lin

A thesis submitted for the degree of
Doctor of Philosophy
The Australian National University

March 2019

© Yi Lin 2019

Except where otherwise indicated, this thesis is my own original work.

Yi Lin
30 March 2019

This thesis is dedicated to my parents. The idea of starting a PhD and the fact of actually finishing it, would hardly happen for me without their support.

Acknowledgements

First I would like to thank my supervisor, Steve, for many years of mentoring me through both a master degree and a PhD. He started and leads the micro virtual machine project, and I am lucky to be a part of it for my PhD. Witnessing a virtual machine being designed and built from scratch, and joining as a part of the process is of great value to me. I learnt a lot from his expertise in runtime implementation and hardware architecture, and his serious attitude towards performance analysis and sound methodology shaped my understanding about scientific performance evaluation and will benefit me life-long. Besides, I want to express my gratitude to Steve for creating an encouraging and autonomous lab environment that makes this hard and long PhD journey enjoyable.

Then I would like to thank my advisor, Tony. The compiler work would be much more difficult, much more time consuming, or maybe impractical for me to finish if he were not around. I greatly appreciate that he spent lots of time with me looking into the very details and figuring out issues. I also would like to thank Michael, my other advisor, for his great insights on language semantics and formal verification. He closely follows the progress of the project, helps keep the project always on the right track, and provides valuable suggestions.

Then I would like to thank my parents. They encouraged me to pursue a PhD degree in the area that I am interested in, and they are being supportive full-heartedly through these many years. In the very last few months of my PhD, they even cancelled a planned trip to visit me so that I can more concentrate on the thesis writing. The idea of starting a PhD, and the fact of actually finishing it, would hardly happen for me if it were not them.

In the years when I am in the computer systems lab in ANU, there have been many fellow students who have walked alongside me, whom I owe my thanks. They have discussed problems with me, and provided useful suggestions and solutions. Especially, I would like to thank Xi for helping me so much in research, in technical stuff, and in life.

Abstract

Implementing a managed language efficiently is hard, and it is becoming more difficult as the complexity of both language-level design and machines is increasing. To make things worse, current approaches to language implementations make them prone to inefficiency as well. A high-quality monolithic language implementation demands extensive expertise and resources, but most language implementers do not have those available so their implementations suffer from poor performance. Alternatively, implementers may build on existing frameworks. However, the back-end frameworks often offer abstractions that are mismatched to the language, which either bounces back the complexity to the implementers or results in inefficiency.

Wang et al. proposed *micro virtual machines* as a solution to address this issue. Micro VMs are explicitly minimal and efficient. Micro VMs support high-performance implementation of managed languages by providing key abstractions, i.e. code execution, garbage collection, and concurrency. The abstractions are neutral across client languages, and general and flexible to different implementation strategies. These constraints impose interesting challenges on a micro VM *implementation*. Prior to this work, no attempt had been made to efficiently implement a micro VM.

My thesis is that key abstractions provided by a micro virtual machine can be implemented efficiently to support client languages.

By exploring the efficient implementation of micro virtual machines, we present a concrete implementation, Zebu VM, which implements the Mu micro VM specification. The thesis addresses three critical designs in Zebu, each mapping to a key abstraction that micro virtual machines provide, and establishes their efficiency: (*i*) demonstrating the benefits of utilizing a modern language that focuses on safety to implement a high performance garbage collector, (*ii*) analysing the design space of yieldpoint mechanism for thread synchronization, and (*iii*) building a micro compiler under the specific constraints imposed by micro virtual machines, i.e. minimalism, efficiency and flexibility.

This thesis is a proof of concept and an initial proof of performance to establish micro virtual machines as an efficient substrate for managed language implementation. It encourages the approach of building language implementations with micro virtual machines, and reinforces the hope that Mu will be a suitable back-end target. The thesis discusses the efficient implementation for micro virtual machines, but illustrates broader topics useful in general virtual machine design and construction.

Contents

Acknowledgements **vii**

Abstract **ix**

1 Introduction **1**
1.1 Thesis Statement . . . 1
1.2 Contributions . . . 1
1.3 Problem Statement . . . 2
1.3.1 Difficulties in Language Implementation . . . 2
1.3.2 Common Approaches to Language Implementation . . . 3
1.3.3 Micro Virtual Machines . . . 4
1.3.4 Implementing an Efficient Micro Virtual Machine . . . 5
1.4 Thesis Structure . . . 7
1.4.1 Scope . . . 7
1.4.2 Outline . . . 8

2 Background **9**
2.1 Frameworks for Managed Language Implementation . . . 9
2.2 The Mu Micro Virtual Machine . . . 13
2.2.1 Memory Management . . . 13
2.2.2 Thread and Stack . . . 15
2.2.3 Exceptions . . . 15
2.2.4 Foreign Function Interface . . . 15
2.2.5 Support for a Client-level JIT . . . 16
2.2.6 Boot Image Generation . . . 18
2.2.7 Local SSI Form . . . 18
2.3 Zebu VM . . . 18
2.3.1 Design . . . 19
2.3.2 Implementation . . . 20
2.4 Related Mu Projects . . . 24
2.4.1 PyPy-Mu . . . 24
2.5 Summary . . . 24

3 A Garbage Collector as a Test Case for High Performance VM Engineering in Rust **27**
3.1 Introduction . . . 28
3.2 Background . . . 29

3.2.1 Related Work 29
3.2.2 The Rust Language 30
3.3 Case Study: A High Performance Garbage Collector in Rust 31
3.3.1 Using Rust 32
3.3.2 Abusing Rust 37
3.3.3 Evaluation 39
3.4 Summary 44

4 Yieldpoints as a Mechanism for VM Concurrency 45
4.1 Introduction 45
4.2 Background, Analysis, and Related Work 47
4.2.1 Background 47
4.2.2 Analysis 48
4.2.3 Related Work 51
4.3 Yieldpoint Design 52
4.3.1 Mechanisms 52
4.3.2 Scope 54
4.4 Evaluation 55
4.4.1 Methodology 55
4.4.2 Overhead of Untaken Yieldpoints 57
4.4.3 The Overhead of Taken Yieldpoints 59
4.4.4 Time-To-Yield Latency for GC 59
4.5 Future Work: Code Patching As An Optimization 62
4.6 Summary 63

5 A Micro Compiler 65
5.1 Introduction 65
5.2 Related Work 66
5.3 Design 67
5.3.1 Optimizations 68
5.3.2 Swapstack 70
5.3.3 Code patching 75
5.3.4 Keepalives and introspection 76
5.3.5 Boot image 76
5.4 Evaluation 78
5.4.1 RPython-Mu 78
5.4.2 Back-end Code Generation 82
5.4.3 Conclusion 84
5.5 Summary 84

6 Conclusion 87
6.1 Future Work 88
6.1.1 PyPy-Mu on Zebu 88
6.1.2 Mu IR Optimizer Library 89

6.1.3 Performance Re-evaluation of GC and Yieldpoints 89
6.1.4 Verified Mu 90

Bibliography **91**

Chapter 1

Introduction

Micro virtual machines were proposed by Wang et al. to facilitate efficient implementations of managed languages by providing abstractions in a single, minimal, efficient and flexible low-level substrate. The design goals bring constraints and interesting challenges to a micro virtual machine's implementation. This thesis discusses the challenges in implementing key abstractions in a micro virtual machine, demonstrates the efficiency of our solutions, and presents a concrete micro virtual machine implementation, Zebu VM.

This thesis complements the prior work [Wang et al., 2015] on micro virtual machines, by realizing that works' *design* of a micro virtual machine specification into an efficient *implementation*. The prior work presented a *proof of existence* for micro virtual machines, while this thesis is a *proof of concept* and an initial *proof of performance* [National Research Council, 1994] for micro virtual machines as an efficient substrate for managed language implementation.

1.1 Thesis Statement

My thesis is that key abstractions provided by a micro virtual machine can be implemented efficiently.

1.2 Contributions

The thesis focuses on the approaches to and the delivery of *desired properties* of micro virtual machines, i.e., minimalism, efficiency, flexibility and robustness, across *key services* that micro virtual machines provide, i.e., code execution, garbage collection and concurrency.

The principal contributions of this thesis are: (*i*) a critical analysis of current frameworks for managed language implementation, (*ii*) a case study of utilizing a memory-/thread-safe language for a garbage collector implementation to achieve both efficiency and robustness, (*iii*) a categorization of a widely used thread synchronization mechanism for virtual machines, yieldpoints, and an evaluation across various yieldpoint implementations with respect to their efficiency and flexibility, and

(*iv*) a micro compiler design as our solution to balance minimalism, efficiency and flexibility in back-end code generation for micro virtual machines[1].

1.3 Problem Statement

1.3.1 Difficulties in Language Implementation

Engineering an efficient managed language implementation is hard, and it is becoming more difficult. (*i*) Language design tends to be higher level and friendlier to the programmer than to the machine. This leaves a larger gap between language semantics and machine instruction set architectures (ISAs) for implementers to close. Furthermore, some language features, such as dynamic typing, require sophisticated optimizations to be efficient, such as inline caching, type specialization and speculative execution. Performance suffers if the implementation is naive. (*ii*) Hardware is becoming more complex. Architectures such as multi-core and heterogeneous architectures, and instructions such as hardware transactional memory, raise more issues that the implementers must take care of, such as memory consistency in concurrent systems, but they also provide more opportunities for the implementers to exploit for better efficiency. This further complicates the implementation. (*iii*) Implementation techniques are getting more advanced. Both academia and industry continuously improve the techniques used in language implementation, which improves performance but also makes the implementation more complex. It is resource-consuming (or infeasible) to apply state-of-the-art techniques in one single implementation. Small slowdowns may add up, compromising efficiency. (*iv*) Security is becoming more critical and more challenging to deal with efficiently.

The impact of implementation difficulties is profound. One obvious consequence when the difficulties cannot be properly handled is that performance suffers. Inefficient implementations may cause the language to run orders of magnitude slower than languages such as C. Even the simple tests of the Benchmarks Game [Benchmarks Game] reveal widely popular languages such as Python and PHP to be extremely inefficient. This leads to a common misunderstanding that those languages are intrinsically inefficient. However, through efforts from academia and industry, more advanced language implementations such as PyPy [Rigo and Pedroni, 2006] and HHVM [Adams et al., 2014] make those languages faster, and in some cases, bring the performance to the same order of magnitude as C, indicating a major source of the inefficiency is the implementation. Unfortunately, most languages do not have such opportunities to draw attention from either academia or industry, and their implementations remain inefficient.

Another consequence is that implementation difficulties may obscure the language designers' understanding of the underlying implementation and levels of abstraction, unnecessarily reflecting implementation details to language design. For example,

[1]While this thesis discusses and proposes a compiler design around minimalism, efficiency and flexibility, we prioritize efficiency in this thesis and only provide performance evaluation for the compiler. Section 1.4.1 states the scope of this thesis.

some languages expose specific GC algorithms and policies to the user. PHP bears the assumption of using immediate reference counting garbage collection throughout its language design[2] [PHP GC, 2017]. This also happens for .NET languages, which assume and expose a generational GC algorithm to users [.NET GC, 2017]. This assumption imposes unnecessary constraints on the language, and prevents the implementation from using other GC algorithms which may be more efficient or more suitable in some scenarios.

1.3.2 Common Approaches to Language Implementation

To make things worse, common approaches to managed language implementations make them prone to inefficiency. Developers may build a *monolithic* implementation, which covers all aspects of the language implementation. Building a high-quality monolithic implementation demands great depth and breath of expertise from its implementers, and considerable resources if it is to deliver good performance. Notable examples with this approach include various Java virtual machines, such as Hotspot VM [Open JDK, 2017] and IBM J9. They are the results of efforts from many years of commitment and investment by large, well-resourced teams. However, many language implementations do not have such resources available, and remain inefficient.

Alternatively, developers may base their language implementation on top of existing well-engineered and performant frameworks. This approach alleviates the implementation difficulties by leveraging mature infrastructures that include rich sets of optimizations, support for multiple platforms, and other standard services. Popular back-end frameworks for managed languages include LLVM and the JVM[3]. Though the frameworks are eligible as back-ends and performant for their originally supported languages, the suitability of re-targeting them for other high-performance managed languages is questionable, especially for dynamic languages and just-in-time compiled languages [Tatsubori et al., 2010; Klechner, 2011; Castanos et al., 2012; Pizlo, 2016]. The reasons are solid, as we shall see below.

LLVM. LLVM was designed and built as an ahead-of-time optimizing compiler framework for native statically typed languages, which makes it less than ideal as a back-end for managed languages. (*i*) In a managed language implementation, the runtime is usually tightly coupled with the compiler. However, LLVM lacks a runtime implementation of features such as GC and exception handling, so the developers will still have to implement their own runtime. (*ii*) LLVM's support for integrating with a runtime is insufficient, or inefficient. Most of the arguments are around flexible and efficient support for precise garbage collectors [LLILC GC, 2017].

[2] Facebook's Hack language addresses this and other shortcomings of PHP [Hack, 2017]. In the meanwhile, PHP still carries the burden while being one of the most popular languages in web development.

[3] There are emerging frameworks, such as PyPy, Truffle/Graal and Eclipse OMR, which offer different solutions as back-end frameworks. Here we only discuss LLVM and the JVM, as they are most widely used. A detailed review of the frameworks can be found in Section 2.1.

(*iii*) LLVM's optimizing compiler is not a silver bullet to language optimization: LLVM can optimize its intermediate representation, but is not capable of language-specific optimizations [Klechner, 2011]. Both are essential for performance. (*iv*) Compilation time is an important requirement for a just-in-time compiler. However, it is not a major concern of LLVM and is known for slow compilation compared to a modern optimizing JIT.

The JVM. As opposed to LLVM, the JVM is a complete virtual machine with a high-performance JIT compiler and runtime. However, it also has pitfalls that prevent a high-performance implementation for an arbitrary language [Rose, 2014]. (*i*) Java bytecode, as the input intermediate representation (IR) for the JVM, is high-level. It lacks efficient access for unsafe and unmanaged code. This greatly constrains the viability and flexibility of implementing certain languages on top of the JVM. (*ii*) Java bytecode carries Java semantics, such as object orientation, object finalization, per-object headers and locks, the class hierarchy root `Object`, and 'all object variables are references'. This often presents semantic mismatches for an arbitrary language, and forces language developers to translate their languages into Java-like semantics when they target the JVM, no matter whether it is suitable or not for the language. As a result, it imposes unnecessary overheads. (*iii*) The JVM lacks a lightweight interface[4] for clients to access run-time variables and states, or to control VM behaviours. It makes it difficult or impossible for clients to implement run-time optimizations.

1.3.3 Micro Virtual Machines

Based on the observation that managed language implementations are difficult and that existing back-end frameworks are not suitable, the idea of micro virtual machines was proposed in Wang et al. [2015] as a foundation for managed language implementation. Micro virtual machines are analogous to micro kernels in terms of language virtual machines: they are a thin substrate that provides minimal key services to support managed language implementation. Micro virtual machines define the key services as three parts, i.e., memory management, code execution and concurrency, and provide a flexible and efficient abstraction over them.

Micro virtual machines are significantly different from existing back-end frameworks, mainly due to the principle that micro virtual machines are explicitly minimal. Micro virtual machines do not aim for omnipotence, but instead, they provide minimal support for the three key abstractions, and make guarantees for efficiency behind the abstractions. The implications of this principle are important: (*i*) The abstraction provided by the micro virtual machines draws a clear line between the front-end of the language implementation (referred to as the *client*) and the back-end, i.e., the micro virtual machine. This separation clarifies the duties of a client. If the client fails to undertake its duty, performance may be compromised. For example, in terms of code optimizations, the micro virtual machine is responsible for hardware-specific optimization as the hardware is abstracted away from the client, and the client takes

[4] JVM TI is a heavyweight API, mainly for debug use.

the responsibility for all other optimizations[5]. (*ii*) Minimalism leads to flexibility, as it imposes fewer assumptions about client languages, e.g., what languages can be implemented on top of micro virtual machines, how they are implemented at the client side and what the requirements of a implementation are. Ideally, micro virtual machines support various kinds of client languages, for example, functional/procedural/object-oriented, statically-/dynamically-typed. And micro virtual machines support different approaches to language implementations, whether they are interpreted, compiled, or a mix of both. Micro virtual machines fit in resource-constrained platforms where a low memory footprint is a requirement. (*iii*) Finally, the minimalism leads to the possibility of verifiability. A formally verified micro virtual machine raises the abstraction level of a trustable computing infrastructure from operating systems [Klein et al., 2009] to language implementation back-ends.

Following the principles and concepts of micro virtual machines, Wang et al. proposed a concrete design, the *Mu* micro virtual machine [Wang et al., 2015]. Mu is a specification that defines the semantics of the micro virtual machine[6]. Mu is designed with a requirement in mind: to efficiently support diverse managed languages and implementation approaches, with an emphasis on dynamically-typed languages. Mu attempts to avoid pitfalls of previous approaches and make distinct improvements. Mu exposes a unique level of abstractions by a low-level language-/machine-agnostic intermediate representation (*Mu IR*) with rich run-time support, such as garbage collection, exception handling, on-stack replacement, swapstack, introspection, dynamic code patching, and the capability of both just-in-time compilation and ahead-of-time compilation. With Mu, language developers can delegate low-level work to Mu while focusing on language-level compilation, optimization and runtime. A detailed introduction to Mu can be found in Section 2.2.

1.3.4 Implementing an Efficient Micro Virtual Machine

Though the concept of micro virtual machines and the specification of Mu was proposed by Wang et al., there was no efficient implementation, and the feasibility of building an efficient Mu micro virtual machine had not been proved prior to this work.

Micro virtual machines, on one hand, provide the upper language clients a flexible and efficient abstraction over key services to support complex language features and necessary optimizations. On the other hand, micro virtual machines exploit hardware architectures to implement them efficiently. The difficulties as discussed in Section 1.3.1 naturally fall to the implementation of a micro virtual machine. But the difficulties are divided so that (*i*) client developers only need to focus on language specific difficulties, and (*ii*) the efforts in solving difficulties for micro virtual machines are reusable. Nonetheless, implementing a micro virtual machine efficiently is difficult.

[5]This burden can be mitigated by providing a client-level micro virtual machine optimizer library for common optimizations. However, language-specific optimizations have to be done by the client.

[6]The specification is publicly accessible at https://gitlab.anu.edu.au/mu/mu-spec.

A micro virtual machine implements and provides some of the most complex services of a managed language implementation. Furthermore, the design of Mu imposes interesting constraints, which further makes an efficient implementation challenging. We categorize the constraints as *minimalism*, *efficiency* and *flexibility*. (*i*) Minimalism leads to properties such as faster compilation time and smaller memory footprint, which makes Mu a desirable target for just-in-time compiled languages and on resource constrained devices. Minimalism also means fewer lines of code, which makes the implementation formally verifiable. (*ii*) Mu implies efficiency, thus it demands an efficient implementation. Though efficiency can be a trade-off with minimalism, Mu seeks a good balance with effective optimizations to achieve decent performance while remaining minimal. (*iii*) Mu does not impose unnecessary restrictions, or prevent efficiency on the client side as they choose strategies to implement the clients. Mu must be flexible in order to support this. As we shall see in the following chapters, properly dealing with the implementation difficulties under these constraints is challenging.

This thesis presents an implementation of the Mu micro virtual machine named *Zebu VM*. We explicitly designed Zebu VM so that it not only implements the Mu specification but also satisfies the constraints discussed above. To be specific, Zebu implements three key services that a micro virtual machine requires: *garbage collection, code execution, and concurrency*, and Zebu is also designed to be *minimal, efficient, and flexible*. We also added another constraint to our design, *robustness*, as our prior experience in low-level system development suggests that safety and software engineering for virtual machine construction is equally important [Frampton, 2010; Lin, 2012]. This thesis presents our design for the Mu micro virtual machine's key services, and discusses how our design responds to the constraints.

Garbage collection We use the implementation of our garbage collector as a case study to justify one of the most fundamental design decisions for Zebu VM that affects overall efficiency and robustness – the choice of the implementation language. Prior work [Frampton, 2010] has shown the importance of implementation languages to virtual machine construction and that it affects the overall performance, robustness and safety. We chose a young but promising language, *Rust*, to implement Zebu. The language is efficient and memory-/thread-safe, but it has relatively restrictive semantics in order to achieve these properties. In this work, we verified that (*i*) the restrictiveness of the language does not prevent its use in a high-performance garbage collector implementation, (*ii*) its safety further benefits the implementation, and (*iii*) its performance is able to match the C language performance when used carefully. We further apply the approach applied to the garbage collector to the entire virtual machine implementation.

Concurrency Concurrency is reflected in Zebu's implementation in various places, such as its memory model, thread/stack management and thread synchronization. Among these, one of the most important mechanisms to support thread synchronization is *yieldpoints*. Yieldpoints are a general mechanism that allows

threads to pause at designated positions with a coherent view of the virtual machine, and can be used in various scenarios, for example, as a synchronization barrier for garbage collection. The design of yieldpoints affects both efficiency and flexibility of the virtual machine. We fully explore the design space for yieldpoints, and vigorously evaluate different designs on a mature platform prior to the start of Zebu's development. The results help us understand the mechanics and ultimately decide that conditional yieldpoints are so far the most suitable choice, due to their decent performance for both taken and untaken cases and lowest latency.

Code execution We carefully design Zebu's compiler, as it is the heart of a virtual machine implementation and determines performance. We take the constraints of micro virtual machines into full consideration for the compiler, i.e., minimalism, efficiency and flexibility, and come up with a succinct compiler pipeline with only necessary transformations and effective back-end specific optimizations. We evaluate our compiler along with a re-targeted ahead-of-time-compiled managed language implementation, and compare the Zebu compiler with the language's stock back-end, which targets C with LLVM. We find that, despite the fact that the re-targeting does not implement the front-end optimizations that Zebu/Mu expects, the resulting performance is good. We also show that Mu's rich run-time support allows Zebu to outperform the stock implementation in some scenarios.

This thesis discusses the design and implementation of Zebu VM, the first micro virtual machine implementation that aims for high performance. The challenge of designing Zebu is to satisfy the interesting constraints as a micro virtual machine, i.e., minimalism, efficiency, flexibility and robustness. This thesis (*i*) discusses the design decisions we faced during implementing a high performance micro virtual machine, (*ii*) demonstrates that, performance-wise, Mu as a micro virtual machine is a favourable back-end target for managed language implementation, in comparison with existing frameworks, (*iii*) encourages new micro virtual machine implementations and users, and (*iv*) sheds light on general virtual machine construction with discussions and insights in the thesis.

1.4 Thesis Structure

1.4.1 Scope

Mu is a long-running, cooperative and open source research project to explore the design and implementation of micro virtual machines. Zebu VM, a high-performance Mu implementation, is one important part of the Mu project, among a few other finished, on-going and planned projects. My thesis started and laid a solid foundation for Zebu VM. The design and implementation of Zebu VM up to the submission date *is* within the scope of my thesis, unless stated otherwise. However, the following are explicitly *not* in the scope of this thesis:

- The design of Mu micro virtual machine specification, and its feasibility for managed languages are *not* in the scope of this thesis. These are included in [Wang, 2017].
- The implementation of client languages is *not* in the scope of this thesis. This thesis refers to an RPython client for performance evaluation, which is described in [Zhang, 2015].
- This thesis only evaluates the efficiency of our design. The metrics that demonstrate the minimalism of Zebu, such as compilation time and memory footprint, are *not* included[7].

1.4.2 Outline

Chapter 2 contains a comprehensive literature review of current approaches to managed language implementation, and a more detailed overview of the Mu micro virtual machine and Zebu VM. Chapter 3 discusses our utilization of Rust as the implementation language to engineer Zebu VM, using the garbage collector as a case study to demonstrate the benefits of using a language such as Rust, and describes the difficulties encountered. Chapter 4 discusses the design of yieldpoints for thread synchronization in virtual machine implementation by evaluating characteristics of various yieldpoint designs. Chapter 5 introduces the Zebu compiler with a focus on its succinct compiler pipeline and the compiler support for Mu's rich run-time semantics. Finally, Chapter 6 concludes the thesis and describes future work.

[7] Our approach favors fast compilation time and small memory footprint. But getting them correct is heavily engineering oriented, and it is secondary to our goal, compared to performance metrics for the project.

Chapter 2

Background

This chapter includes four sections. Section 2.1 is a literature review of current approaches to managed language implementation with a focus on their benefits and shortcomings. Section 2.2 gives a detailed introduction to the design of the Mu micro virtual machine, such as its intermediate representation and APIs. Section 2.3 is an overview of the Zebu VM including its general design and some details that will not be covered in the rest of the thesis. Section 2.4 gives background information on other Mu-related projects.

This chapter discusses background topics that are general and pertinent to the entire thesis. The background and the related works that are only closely related to a specific chapter of this thesis will be covered separately in each of the subsequent chapters.

2.1 Frameworks for Managed Language Implementation

Building a language implementation on top of another is a common strategy. This approach leverages substantial investments in the lower layer, avoiding re-inventing the wheel, often with significant benefits in performance and robustness. Above all, it may allow the language implementation to build on features that would otherwise be entirely out of reach given the available resources, such as high quality code generation and advanced garbage collection.

Section 1.3.2 has briefly discussed two concrete frameworks that are popular as back-ends for managed language implementation, i.e. LLVM and the JVM. This section is a more comprehensive literature review.

C as a Target Source-to-source compilation (also known as translation) is a common approach with many software engineering benefits [Plaisted, 2013]. C, as a high-performance low-level and mostly-portable language, is often considered as a good target for language implementation to save the efforts of machine code generation [Henderson and Somogyi, 2002; Flack et al., 2003]. However, in terms of managed language implementation, C is a cumbersome target. Some run-time features require the compiler to be cooperative, but those features are not supported in C and C prevents their efficient implementation. For example, C lacks support for exception

handling [Roberts, 1989]. When targeting C, exceptions are usually implemented as an explicit check for exceptional return values, or `setjmp/longjmp` (SJLJ), both of which impose a much higher overhead than a proper implementation of exception handling with stack unwinding. Secondly, C lacks support for exact garbage collection [Baker et al., 2007; Rafkind et al., 2009]. Implementation of garbage collection with C is either conservative, or creates a shadow stack to enable exactness, neither of which is ideal. Finally, C is statically compiled, making it a poor target for a just-in-time back-end. For these reasons, C became a less practical choice once frameworks such as LLVM emerged.

Compiler Frameworks Compiler frameworks expose a lower-level intermediate representation, such as LLVM IR and C−−, which makes them more suitable for just-in-time compilation than C, since such IRs are natural targets for front-end compilers. A mature, well-engineered, and language-agnostic back-end framework such as LLVM is of great value to language implementers, making it a desired target for compilers[1], static analysis [Grech et al., 2015] [Cassez et al., 2017], verification [Zhao et al., 2012], and many other purposes. Given its popularity, we observe that leveraging LLVM in managed language implementation can help achieve reasonable performance with LLVM's just-in-time compilation support, but LLVM-enabled implementations cannot compete with a high-quality monolithic implementation. For example, multiple Python implementations that target LLVM report performance improvements over CPython [Willis, 2009; Pyston, 2017], but they are markedly slower than the state-of-the-art Python implementation, PyPy. Another example is that a few production implementations, such as JavaScriptCore and Glasgow Haskell Compiler, re-targeted to LLVM, but later moved away from LLVM or no longer use it as the default back-end. We listed the issues in Section 1.3.2.[2] This thesis' point of view is that LLVM helps managed language implementation significantly, especially when expertise or resources are limited, but LLVM was originally designed without consideration of managed language scenarios, which brings intrinsic pitfalls, therefore making it a less than ideal target when quality and performance are the main goal. Though LLVM is evolving to improve its support for managed languages such as through the introduction of `patchpoint`, `stackmap` and primitives to support garbage collection, the long list of failed projects is an evidence that it is still inadequate at the time of writing.

[1]Languages that have compilers targeting LLVM include ActionScript, Ada, C#, Common Lisp, Crystal, D, Delphi, Fortran, OpenGL Shading Language, Halide, Haskell, Java bytecode, Julia, Lua, Objective-C, Pony, Python, R, Ruby, Rust, CUDA, Scala, Swift, and Xojo [LLVM LANGS, 2017].

[2]It is worth mentioning that Zing VM starts using LLVM as the second tier JIT, and reported positive results [Reames, 2017]. Their approach elided the issues by: (*i*) restricting the source language (or restricting how the source language is implemented) in order to implement a relocating garbage collector on LLVM [Philip Reames, 2014], (*ii*) deeply customizing LLVM by embedding language-specific high-level IRs and custom passes to communicate language-specific semantics with the LLVM optimizer, and (*iii*) only using LLVM for the second-tier JIT, for which compilation time is not a main concern. The bypass solutions align with the issues we raised for LLVM being back-end for general managed languages.

High-level Virtual Machines A well-engineered reusable virtual machine can be a real asset to managed language implementation. Unlike compiler frameworks, a virtual machine provides not only compilation and code execution, but also the implementation of and integration with a runtime. Popular choices include the Java Virtual Machine (JVM) and the Common Language Runtime (CLR), both of which are mature and highly developed platforms. We observe that despite the fact that the JVM and CLR are efficient for their originally supported languages (Java and .NET languages respectively), other languages do not perform well when implemented on the VMs. Section 1.3.2 discussed issues of basing managed language implementation on top of the JVM. This thesis refers to those VMs as high-level virtual machines, as they expose intermediate representations that are relatively high level, Java bytecode and Common Intermediate Language (CIL) respectively. The problems with high-level IRs are fundamental: (*i*) High-level IRs are closer to the source languages, and inevitably carry semantics of the source language. This brings an impedance mismatching when targeting a language that the IR is not designed for. (*ii*) High-level IRs do not flexibly support various languages. The notable example is the introduction of `invokedynamic` and `MethodHandle` in JSR-292 with Java 7 [JSR-292, 2011], later `VariableHandle` in JEP-193 with Java 9 [JEP-193, 2014], and many proposals that have not yet been delivered, such as value objects [JEP-169, 2012]. This reflects the lack of flexibility from a high-level IR, which may consequently lead to IR bloat in order to support more languages. (*iii*) High-level IRs are not expressive in terms of efficiently expressing language-specific run-time optimizations, for example, inline caches [Hölzle et al., 1991] and guarded specialization [Gal et al., 2009; Costa et al., 2013]. This leaves the problem that some essential optimizations cannot be done at the front-end because they cannot be expressed in the IR, and they will not be done in the back-end VM as the VM does not know about the source language and its specific optimization. Summarizing, though reusing existing virtual machines brings great benefits to a managed language implementation, the high-level IR stands in the way, and prevents high performance implementation.

Meta Compilation Meta compilation derives an execution engine (which is usually more efficient) from an existing execution engine (which is usually easier and more straightforward to write). Examples include partial evaluation for Truffle [Würthinger et al., 2012] [Würthinger et al., 2017] and meta tracing for PyPy [Bolz et al., 2009]. Meta compilation provides a different model for managed language implementation. The traditional approach is that the language developers write an implementation while delegating part of the work to a back-end framework. Meta compilation asks the language developers to write a language interpreter (under special constraints), and the framework aids in deriving efficient machine code for the source program from the interpreter. Truffle and PyPy derive machine code differently from each other: Truffle applies partial evaluation to the interpreter with source programs, and PyPy runs source programs with the interpreter to produce traces, then optimizes and generates machine code for the trace. Both approaches are promising. Würthinger et al. reported that the JavaScript implementation based on Truffle achieves 0.83× per-

formance of V8 JavaScript engine [Würthinger et al., 2017]. Though it is still slower, it is a comparison between an implementation based on a common framework and a high-quality state-of-the-art monolithic implementation. It is a significant improvement over previous approaches. Though meta compilation provides a different but promising way to implement languages on common frameworks, it has its own limitations. (*i*) Current approaches of meta compilation rely on a high-level virtual machine, i.e., the JVM for Truffle and the PyPy virtual machine for PyPy. They also encounter some of the problems with reusing a high-level virtual machine. Würthinger et al. stated that they met difficulties when translating language semantics into the JVM semantics, such as implementing continuation and coroutines as threads, implementing tail calls with an exception trick that the underlying compiler can recognize and optimize. The implementation of features that are missing on the host VM would either require changes to the host VM, or suffer from inefficiency. These are the same problems we discussed in the previous paragraph. (*ii*) Meta compilation has longer warm-up time, in comparison with a traditional approach. Würthinger et al. reported that the warm-up times for Truffle-based implementations are an order of magnitude longer than a traditional approach, ranging from 30 seconds to 90 seconds per benchmark. This makes meta compilation not suitable for scenarios that desire peak performance within seconds. (*iii*) Utilizing these frameworks to achieve good performance is non-trivial. Niephaus et al. demonstrates an example of how heavily annotated a simple interpreter loop may have to become to deliver good interpreter performance with Graal [Niephaus et al., 2018]. Given the limitations and problems, meta compilation provides an interesting and efficient solution as an answer to reusing common frameworks for managed language implementation, and has achieved the best performance so far.

Reusable VM Components Modularizing a virtual machine implementation and exposing components for reuse is helpful. Projects include VMKit [Geoffray et al., 2010] and Eclipse OMR [Gaudet and Stoodley, 2016]. This approach focuses on providing features with generalized interfaces to each single virtual machine component, instead of a complete and semantically coherent abstraction layer. VMKit provides three components and it uses existing libraries for each, i.e., threads (pthread), compiler (LLVM), and garbage collection (MMTk [Blackburn et al., 2004]). VMKit is one of the earliest attempts to prove the concept of a universal platform for managed languages, and its concept inspired many successive approaches, including the Mu micro virtual machine. However, its design is not performance-centric [Geoffray, 2009]. Eclipse OMR provides more modules than VMKit, such as platform porting, virtual machine APIs and other utilities. It is worth mentioning that Eclipse OMR is a modularized platform that is refactored from existing IBM products, including its multi-lingual just-in-time compiler Testarossa and multiple languages built on top of it. The performance and shippability is maintained throughout the refactoring work. However, apart from originally supported languages, we are only aware of limited information about the performance of other languages built on top of Eclipse OMR. Ruby+OMR reported a moderate performance improvement over Ruby MRI in its

early development [Gaudet, 2015]. Reusable VM components, as an approach to help managed language implementation, provide pluggable and replaceable components, instead of providing a complete abstraction layer. This approach can be suitable if the components capture the right abstractions.

In summary, there are various frameworks that can be used for managed language implementation in attempt to address the issue of high cost/low performance in managed language implementation. The frameworks include earlier attempts with LLVM and the JVM, and more recent efforts of Truffle, PyPy and Eclipse OMR. Interestingly, we noticed a trend that in the recent years, both industry and academia have shifted their focus and efforts from reusing existing frameworks to proposing new frameworks. We believe the intrinsic pitfalls of existing frameworks for managed languages are becoming well understood, and there is an emerging consensus that these shortcomings need to be addressed.

2.2 The Mu Micro Virtual Machine

The Mu micro virtual machine is a thin language-/machine-agnostic abstract virtual machine that focuses on providing core services in a minimal, efficient and flexible manner. Two major differences between Mu and other proposals are that Mu is minimal and that it presents a coherent lower-level abstraction. It is feasible to base other frameworks on top of Mu[3]. The principles of Mu's design are centred around minimalism: (*i*) any feature or optimization that can be realized by the higher layer (*client*) must be; however, (*ii*) minimalism in Mu must not stand in the way of efficiency, and (*iii*) minimalism must not come at the cost of flexibility necessary to support diverse languages.

Mu was proposed in Wang et al. [2015] and Wang [2017] as an open specification [Mu Spec, 2016]. This section gives an overview of the specification, as this thesis will later focus on the discussion about realizing the specification concretely. Mu presents an interesting and unique abstraction level: it is low-level but embraces a complete abstraction for rich run-time semantics. We do not discuss designs that are common to low-level IRs, such as binary operations, comparison operations, conversion operations and control flows. This section focuses on introducing Mu specific designs.

2.2.1 Memory Management

Mu's abstract memory includes three parts: a managed heap, stacks, and global cells. Mu provides complete and coherent semantics for its abstract memory, including garbage collection for the managed heap, unified access to Mu memory and cross-referencing between the managed heap and native memory.

[3] Zhang [2015] presents preliminary work that re-targeted the PyPy's meta tracing framework to Mu.

Garbage Collection. Mu supports garbage collection (GC) with its IR without exposing details and assumptions about the underlying algorithms and policies. The Mu type system includes reference types that point to objects[4]. Reference types include ref<T> (a normal reference), iref<T> (an internal/derived reference that points to the middle of an object), weakref<T> (a weak reference that will not protect the referenced object from collection), and tagref64 (a tagged reference that serves as a union of a double, or an integer, or a reference with a tag). Mu provides NEW and NEWHYBRID[5] for allocation in the managed heap, which return a reference to the object as a ref<T> value. Other reference types are derived from ref<T> to refer to the heap. Mu guarantees that non-weak reference values (including derived values) that are reachable from the client or client-generated IR are always accessible as long as they reference a valid heap object[6]. Other than this, Mu does not expose more details or make more guarantees about GC.

Unified Access to Mu memory. Mu provides memory access operations via LOAD and STORE with specified memory order and atomic operations. Accessing Mu memory is *always* through an internal reference, iref<T>. References to stack-allocated values (through ALLOCA) and to global values are always of type iref<T>, as they are conceptually referencing the middle of a memory range, stack and global memory respectively. Accessing heap-allocated objects requires deriving an internal reference from a ref<T> through instructions such as GETIREF, GETFIELDIREF, GETELEMENTIREF and SHIFTIREF[7]. This abstraction unifies the access of Mu memory, and prevents the ambiguity of allowing multiples types being accepted by one memory accessing instruction, or bloating instruction sets for each single reference type. Furthermore, this abstraction can be zero-cost.

Cross-referencing with Native Memory. Mu allows referencing native memory in Mu code, and vice versa. Other than reference types, Mu provides an unsafe pointer type uptr<T>[8], to refer to native memory. Managing the lifetime and validity of native memory is a client-side duty. Native pointers are *untraced* by Mu, and Mu's garbage collector is unaware of native pointers. Mu also allows exposing heap references to native code by pinning the objects. PIN makes a guarantee that the object reference (or the internal reference) will stay valid until an UNPIN for the object is executed. This implies that the garbage collector will keep the object alive regardless of its reachability in the Mu and will keep the object in place even if the collector is a moving collector.

[4] Though Mu does not have the concept of object oriented programming, and is oblivious to inheritance and dispatch tables, we use the term 'object' to refer to a piece of memory allocated in Mu's abstract heap.

[5] NEWHYBRID allocates objects of variable-length *hybrid* types, while NEW allocates fixed-length objects.

[6] Mu trusts the client, and allows clients to derive reference values, including invalid references such as out-of-bound array indexing. Accessing invalid references is undefined behaviour in Mu.

[7] https://gitlab.anu.edu.au/mu/mu-spec/blob/master/instruction-set.rst

[8] Mu also provides ufuncptr to support native calls.

2.2.2 Thread and Stack

Mu supports a very flexible model for threads and stacks to allow efficient implementation of coroutines and client-level green threads (many-to-one model) [JDK Developer's Guide, 2017]. Mu makes an explicit distinction between the concepts of thread and stack: a stack is a memory area used as the context of execution while a thread is the container for execution. Under this model, the creation of stacks and threads is separate. The creation of a stack does not imply execution with the stack context at creation time; instead, only when a stack is bound to a thread does the execution start. A Mu stack can be *active* with a Mu thread as its execution container, or can be detached from any running Mu thread in a *ready* state and waiting for *resumption*. The key to this design is the *swapstack* primitive [Dolan et al., 2013] as a lightweight context switching mechanism. We will elaborate on this in later sections, as Mu exposes a more flexible model of `swapstack` [Wang et al., 2018] compared to the prior work.

2.2.3 Exceptions

Mu provides exceptions that are similar to the common **`throw-catch`** model [Steele and Gabriel, 1993; Koenig and Stroustrup, 1993]. Mu has built-in exceptions, such as division-by-zero error and invalid memory accessing, and also supports user-defined exceptions as a heap object by a `THROW` instruction. Some Mu IR instructions can optionally declare a block as the exception destination (similar to a **`catch`** block), and if an exception is thrown from the instruction or gets propagated to the instruction, Mu will transfer the control flow to the designated exception block. Potentially exception-generating instructions (PEIs) in Mu include binary operations, memory allocation and access, thread creation, stack rebinding (`swapstack`) and call instructions. It is worth mentioning that Mu allows exceptional resumption with a given stack: when the stack is bound to a thread, instead of resuming normally with arguments, an exception is given; the control flow resumes as if the instruction threw the exception. This, in combination with Mu's thread model, allows very flexible threading and control flow to implement client-level semantics.

2.2.4 Foreign Function Interface

Mu allows a mostly bi-directional foreign function interface between Mu and native code. As discussed in Section 2.2.1, Mu supports using native pointers in Mu code, and using pinned Mu references in native code. Furthermore, Mu supports calling native functions in Mu code through `CCALL`, and calling *exposed* Mu functions in native code. However, since Mu does not impose or make guarantees about object memory layout, native code can only use a pinned Mu reference as an opaque pointer.

2.2.5 Support for a Client-level JIT

Mu is designed to efficiently support dynamic languages where a client-level just-in-time compiler is essential.

Watchpoints Watchpoints are a mechanism for the client to conditionally interrupt and intervene in code execution in Mu with an assumption that the watchpoint execution is cheap in normal cases. A watchpoint is a global guard whose behaviour depends on its status: disabled (default) or enabled. The same watchpoint (identified by its ID) may be referred to by multiple instructions. Mu has two kinds of watchpoint instructions. `WATCHPOINT` is a guard, which either normally continues when disabled, or traps to the client when enabled. A special form of `WATCHPOINT` is `TRAP`, which unconditionally traps to the client. `WPBRANCH` is the other watchpoint instruction which branches to different targets based on the watchpoint status, i.e., branching instructions with targets that can be switched during execution. Watchpoints allow the client to implement efficient predicates, or interrupt IR execution for various purposes, e.g., querying execution states, triggering higher-tier recompilation, de-optimization, specializing code, etc.

Introspection Mu abstracts over machine state, and allows clients to inspect the abstract state. For example, Mu allows stack walking through abstract stack frames with a unidirectional `framecursor`. Clients may create a `framecursor` that points to the most recent Mu frame, move it monotonically towards previous frames, and inspect the current function, instruction, and variables with the cursor. Mu requires variables to be explicitly marked as `keepalive` at a given instruction so that they will be kept alive in a known location at that instruction and can be inspected by the client. Mu instructions that leave the current frame or stack, such as `CALL`, `SWAPSTACK` and `WATCHPOINT`, may optionally name `keepalive` values. This prevents Mu from storing excessive variable location information in the runtime [Stichnoth et al., 1999].

Function Redefinition Mu supports function redefinition. A function may be dynamically redefined, and when a newer version is defined, Mu guarantees that any further call to the function will be resolved to the most recent version. In practice, the client can use function redefinition to implement multi-tiered optimization and/or specialization of functions. Figure 2.1 shows one way to implement code specialization with inline caching in Mu.

Support for OSR On stack replacement (OSR) is crucial to adaptive optimization and is notoriously difficult to implement [Chambers and Ungar, 1991; Hölzle and Ungar, 1994; Lameed and Hendren, 2013]. Mu does not directly provide OSR as a primitive, but rather, following the principles of minimalism and flexibility, it provides primitives that allow the client to implement OSR at a level of abstraction above the underlying machine [Wang et al., 2018]. Besides stack walk and introspection, Mu also allows stack manipulation with `POP_FRAME_TO` and `PUSH_FRAME`. These primitives

```
1    ...
2    // inline cache for add
3    if typeof(obj1) == int and typeof(obj2) == int
4      i1 = obj_to_int(obj1)
5      i2 = obj_to_int(obj2)
6      res = i1 + i2
7    else
8      add_general(obj1, obj2)
9      profile_add_binding()
10   ...
```

(a) Monomorphic inline caching in pseudo code.

```
1    ...
2    .block %add_inline_cache:
3      WPBRANCH %add_ic_0
4        enable: %check_add_special
5        disable: %add_general
6
7    .block %check_add_special
8      %is_special = CALL check_types(%obj1, %obj2)
9      BRANCH2 %is_special
10       true: %add_special
11       false: %add_general
12
13   .block %add_special:
14     %res = CALL add_special(%obj1 %obj2)
15     BRANCH %after_add
16
17   .block %add_general:
18     %res = CALL add_general(%obj1, %obj2)
19     // client profiling code in the TRAP
20     TRAP
21       normal: %after_add
22       exception: nil
23
24   .block %after_add
25   ...
```

(b) One way to implement inline caching with Mu. Use `WPBRANCH` to enable/disable the inline cache, and use `TRAP` to profile and redefine `check_types` and `add_special` to a type-specialized version.

Figure 2.1: Example of implementing inline caching with Mu.

allow re-organizing frames for a paused stack (in the ready state), and provide the client with sufficient mechanisms to build OSR in their own implementation.

2.2.6 Boot Image Generation

Boot images are commonly used by managed runtimes to capture ahead-of-time generated code and virtual machine state to expedite start-up. In principle the machine state can typically be constructed from scratch at start-up, but the boot image is a simple optimization that will greatly reduce start-up time by pre-computing and storing the requisite bootstrap state [Alpern et al., 1999; Open J9, 2017]. Meta-circular language implementations depend on a boot images in order to bootstrap [Alpern et al., 1999; Ungar et al., 2005; Rigo and Pedroni, 2006; Blackburn et al., 2008; Wimmer et al., 2013]. Mu provides a `make_boot_image()` API for generating a boot image for the target platform. The function takes a set of white list functions, native data relocation maps, and an optional entry function if the boot image is desired to be an executable. When the boot image is loaded and resumed, it is guaranteed that the image contains (*i*) all white-listed functions in their compiled state, (*ii*) all related entities (such as values, types, functions, etc), and (*iii*) all objects in the Mu heap reachable at the time of the call. We will elaborate more in Section 5.3.5.

2.2.7 Local SSI Form

Mu IR uses a special Static Single Information (SSI) form[9] [Ananian, 1997; Singer, 2006; Boissinot et al., 2012], an extension form of Single Static Assignment (SSA). Besides the properties from SSI/SSA that every variable has exactly one definition and cannot be redefined, we further require that every block explicitly declare variables at the start of its label and the variables are strictly local to the block (we still call them SSA variables). A block terminating instruction is required to explicitly list *all* the variables being passed to the destination block, instead of only variables split by a σ-function in normal SSI form.

The benefits of the design are three-fold: (*i*) The liveness of SSA variables are explicit in the IR form, which can be exploited by the micro virtual machine to skip expensive global liveness analysis. (*ii*) Longer live ranges are broken up into smaller ones, which may allow better register allocation such as less spilling. (*iii*) This form also makes data flow explicit, and helps a formal specification of Mu. However, this design is essentially a non-minimal SSA form with potential unnecessary value merging, which causes difficulty in register coalescing. The performance implications of this design are unclear, as we shall discuss in Section 5.4.2 with a concrete example.

2.3 Zebu VM

Zebu VM is a concrete Mu micro virtual machine implementation that aims for high performance, developed as part of this thesis. Zebu VM is a proof of concept and

[9] https://gitlab.anu.edu.au/mu/general-issue-tracker/issues/18

an initial proof of performance for efficient micro virtual machines. In later chapters, we will discuss design points that are important to Zebu and are the principle contributions of this thesis. In this section, we briefly discuss Zebu and its design and implementation.

2.3.1 Design

In building Zebu, our goal is to apply state-of-the-art techniques, and when necessary, advance the state-of-the-art. Implementing a virtual machine is heavily engineering-oriented, but nonetheless important part of computer systems research. We elaborate both research questions and practical engineering concerns for our designs.

Focus on Cost-Benefit Ratio Zebu is designed with a target of around 25K lines of code in implementation while being able to achieve competitive performance, as the small amount of lines of code leads to the possibility of being formally verified practically [Klein et al., 2009]. Our experiences with Zebu prove that this target is feasible. Table 2.1 shows that the total lines of code of the core Zebu VM fall within our expectation[10]. In Chapter 5, we will further discuss the performance.

	Rust	**C/Assembly**
Compiler (x64 ASM back-end[11])	17.0K	0
GC	3.4K	0.3K
Other Runtime	1.3K	0.6K
VM	1.8K	0
Total Core Code	23.5K	0.9K

Table 2.1: Code base of Zebu (as in Git revision 211e3976). The core VM code takes 24.4K lines of code, which falls within our expectation.

We designed our compiler with a focus on cost-benefit ratio. We estimate the potential benefits (mainly performance gain) and the cost (in terms of lines of code and compilation time if applicable). For example, in an early prototype, we used a weight-based bottom-up rewrite system (BURS) for instruction selection [Boulytchev, 2007]. However, we realized that, due to the simplicity of Mu IR, BURS is over-kill for Zebu, and we replaced it with much simpler tree pattern matching that prioritizes the first match. This decision not only saves us from the effort of maintaining a code generator generator but also reduces the time spent in instruction selection, while maintaining the ability to produce good quality code for Mu. Another example is that

[10] The line counting is done with a Zebu build of the x64 ASM back-end configuration (the aarch64 back-end of 10.0K LOC is not included). The count only includes core VM code, and excludes supporting code, such as the IR data structure (3.4K LOC) and API wrappers (4.7K LOC). As Mu defines its API in C, we implement a set of wrappers to delegate the calls to our Rust code.

[11] We have two code generators for x64, an ASM one for boot image generation and a binary one for JIT. They share the instruction selector.

Zebu foregoes the opportunity of doing software instruction scheduling in its backend. We justify this decision thus (*i*) modern processors make software instruction scheduling less effective by out-of-order execution, branch prediction and instruction prefetch, (*ii*) some optimizations can be done by our client at the IR level, such as loop unrolling, and (*iii*) by skipping instruction scheduling, we further skip the necessity of providing surrounding infrastructures to support instruction scheduling, such as more information about machine instructions.

We applied the same principle for our runtime design as well. Zero-cost exception handling [Drew et al., 1995] delivers excellent performance, but it is expensive to build as it requires the compiler to cooperate and build tables for location information of variables and registers so that the runtime can use them to unwind the stack and restore execution context. However, since Mu already imposes the cost for building such tables for the requirement of supporting introspection, it is natural and beneficial to use them for exception handling. But we are undecided on the value of extending these tables to also support identifying GC roots on the stack. Recent research showed that a high performance garbage collector that is conservative with respect to the *stack* imposes insignificant overhead [Shahriyar et al., 2014]. Besides, conservativeness with the stack allows us significantly simplify our compiler implementation, and we are no longer required to carefully maintain reference locations throughout the compiler pipeline.

Follow the Standards Following defined or de facto standards is an important software engineering principle. Standardized practices enable inter-operability with utility tools or other software that follow the same standards. However, for virtual machine implementation, developers may easily deviate from the standards for their own requirements. For example, developers may use project-specific calling conventions, memory layout, binary format, etc., and sometimes this is necessary.

However, Zebu chose as a design principle to stick to existing standards wherever possible. For example, Zebu uses the standard ABI for calling convention and memory layout for target architectures [AMD64 ABI, 2017; ARM64 ABI, 2017]. Zebu generates position-independent boot images in the standard binary format for target operating systems, e.g., ELF for Linux and Mach-O for MacOS. Zebu generates debug symbol information in DWARF. This requirement benefits Zebu. (*i*) It essentially eliminates the language barrier between Mu and native, and allows direct access between Mu code and native, which efficiently supports the Foreign Language Interface required by Mu (as discussed in Section 2.2.4). (*ii*) It allows us to use standard utility tools to link, debug, inspect and profile code generated by Zebu, which is valuable for development.

2.3.2 Implementation

Figure 2.2 illustrates an overview of Zebu VM, including important components and interaction between them. Dotted lines illustrate how Zebu fits in and communicates with a language implementation.

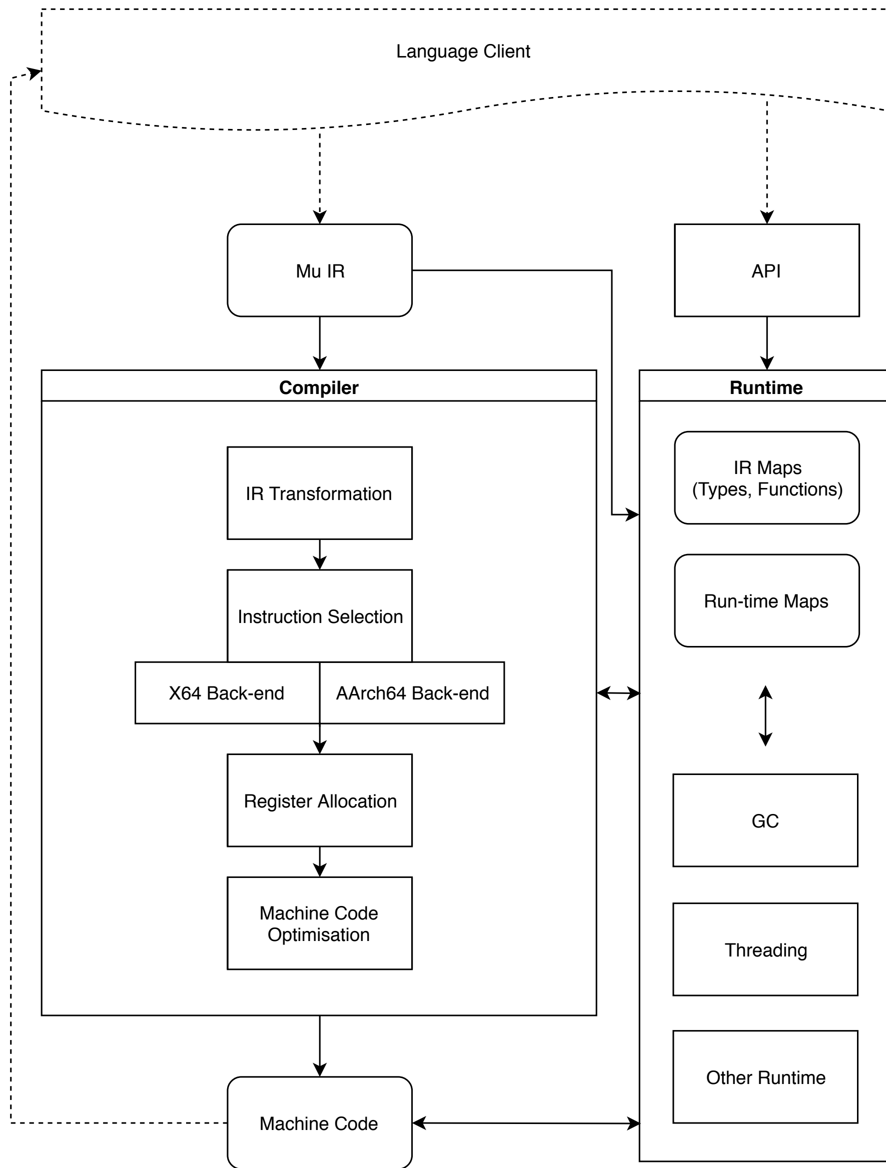


Figure 2.2: An overview of Zebu VM, illustrating important components and interaction between them. Dotted lines illustrate how Zebu fits in a language implementation.

Compiler

Zebu implements a back-end compiler with a short pipeline. The Zebu compiler executes a defined pipeline which consists of passes. There are two sets of passes, IR-level passes and machine code passes, separated by a target-specific instruction selection.

IR-level Passes are mainly for IR transformation and analysis, listed as below. The symbol $\star$ indicates a target-dependent pass, and analysis passes are omitted from the list.

1. **Return block creation** generates a single epilogue for each function, and explicitly marks it so that trace scheduling will take it into consideration.
2. **Inlining**$\star$ identifies possible inlining choices. Though inlining can be done by the language client, the micro virtual machine has target information, and is able to make better inlining choices.
3. **IR rewriting**$\star$ lowers the IR to serve two purposes: (*i*) For instructions that will introduce new blocks, this pass will rewrite the instruction, and introduce new blocks so that trace scheduling will properly lay out the blocks. (*ii*) For instructions that involves run-time calls such as heap allocation, this pass will rewrite the instruction into a fast-path sequence with a call into the runtime slow-path. Furthermore, fast-path and slow-path blocks are also explicitly marked so that trace scheduling will take them into consideration.
4. **Depth-tree generation**$\star$ is an essential IR-level pass for performance. The client feeds Zebu with a sequence of instructions. This pass uses heuristics to turn the sequence into a depth tree, in order to expose favorable patterns for instruction selection.
5. **Trace scheduling** identifies hot traces, and lays out the trace linearly. Zebu mainly uses three sources of information for trace scheduling: (*i*) Zebu considers trace hints from internally generated blocks. (*ii*) Zebu allows conditional branching instructions to specify a probability with each branch. (*iii*) When the probabilities tie, Zebu prioritizes putting the false block to the hot trace, following common practice.

Instruction Selection in Zebu is implemented by tree pattern matching that prioritizes the first match. The instruction selector invokes an abstract interface for the given target to emit machine code. The abstract interface may have different implementations. For x86_64, we implemented an assembly back-end for ahead-of-time compilation[12], and a binary back-end for just-in-time compilation. Though both back-ends share the compiler infrastructure, the instruction

[12]The assembly back-end generates assembly code that can be easily assembled and linked into a relocatable boot image in the standard binary format on the target platform by using standard tools. It also makes debugging easier.

selector needs to be aware of the current back-end in some cases. For example, the instruction selector resolves a function to a symbolic name for ahead-of-time compilation, but to an address for just-in-time compilation.

Machine-level Passes further optimize the generated code. Zebu provides an abstraction of the machine code so that some machine code passes can also be implemented in a target independent way.

1. **Register allocation** is the most important optimization at the machine-level. We implemented a graph-coloring register allocator with iterated register coalescing [Appel and Palsberg, 2003] [13]. We tuned the allocator using heuristics that consider loop depth for temporaries and moves to make decisions for freezing, spilling, and coloring nodes. The reason is that our SSA form (as discussed in Section 2.2.7) requires local variables to be local to a basic block, which essentially splits long-lived temporaries into many smaller ones. An untuned allocator makes arbitrary decisions that may fail to properly coalesce temporaries, and consequently the code contains a lot more moves than necessary.
2. **Peephole optimization** is the last pass to clean up the generated code. Currently we have only implemented target independent ones such as redundant move removal and jump removal.

Runtime

Zebu divides its managed heap into three spaces, two Immix spaces [Blackburn and McKinley, 2008] for different sized smaller objects and a freelist space for large objects. Zebu pursues a constant 1/8 ratio for the max overhead of heap object metadata, i.e., 1 byte per every 8 bytes heap space. This model reduces memory overhead for small objects, which is usually the dominant majority in managed languages. Zebu also has a special immortal space only for objects that are persisted during boot image generation; those objects are mutable and can be referenced normally, but the GC will not reclaim the memory even if they are no longer reachable (as we put them in the data segment in the boot image instead of in our managed heap).

Zebu implements a Mu thread as a Rust thread. But before a Mu stack is executed, Zebu saves the native stack pointer, and then does a special `swapstack` operation between the native stack and the Mu stack so that further execution of Mu code is always on the Mu stack. Before a thread quits, Zebu swaps back the native stack to allow proper destruction of the Rust thread. Zebu protects both ends of a Mu stack from any form of accessing to prevent over-flow and under-flow.

Zebu implements its own signal handler. The signal handler is responsible for (*i*) re-throwing Mu built-in exceptions, such as floating point errors and invalid memory access, (*ii*) INT-3 code patching guard (specific to x86_64), and (*iii*) aborting

[13] We chose a graph-coloring algorithm for reliable results and easy implementation. However, this algorithm may not be ideal for just-in-time compilation when the allocation time becomes a major concern, and a drop-in replacement may be needed in the future.

properly if none of the above met. Zebu doesn't implement `TRAP` and client-supplied `traphandler` with signals. Instead, Zebu simply emits a run-time call to prepare and call to the pre-registered traphandler as a `TRAP` instruction.

2.4 Related Mu Projects

Mu is a cooperative research project. This thesis presents the design of the implementation that focuses on performance. There are other Mu-related projects, such as Holstein VM [14] (a reference implementation), the Mu formal specification in HOL [15], and other language clients on top of Mu, GHC-Mu [16] and PyPy-Mu [17]. In this section, we only focus on PyPy-Mu, as it is closely related to this thesis for the performance evaluation in Section 5.4.

2.4.1 PyPy-Mu

PyPy-Mu is an interesting combination that allows us to to verify the feasibility and performance of Mu in a number of ways: (*i*) Python is a popular language with its official implementation known to be inefficient, and there are many implementations that improved the performance, among which PyPy provides a state-of-the-art solution. We are interested in seeing how PyPy performs with Mu as its back-end. (*ii*) PyPy has a JIT for a dynamically-typed language, which exactly matches the design goals of Mu. (*iii*) PyPy is an unconventional implementation, as it is meta-circular and features a meta-tracing code generator. It is an ambitious target to check Mu's flexibility of supporting different client implementation strategies. (*iv*) PyPy is implemented in RPython, a restricted subset of Python. PyPy relies on the ahead-of-time compilation of RPython for a boot image. We are also interested in having RPython and the boot image generation performed by Mu so that the whole implementation stack for PyPy runs on top of Mu. (*v*) Enabling RPython on Mu allows other language implementations based on PyPy and RPython to run on Mu.

We proceed this in a two-step approach. We start with enabling RPython on top of Mu, which gives us RPython as a language, and also language interpreters written in RPython, such as a SOM interpreter [RPYSOM, 2014]. Then, we enable the PyPy JIT on top of Mu to support the whole PyPy implementation stack. Currently we are approaching the end of the first step, and the work to support PyPy JIT is not ready yet. The initial work is described by Zhang [2015].

2.5 Summary

This chapter covers the background for the thesis. We started with a survey of current frameworks for managed language implementation. We pointed out the pitfalls of

[14] https://gitlab.anu.edu.au/mu/mu-impl-ref2
[15] https://gitlab.anu.edu.au/mu/mu-formal-hol
[16] https://gitlab.anu.edu.au/mu/mu-client-ghc
[17] https://gitlab.anu.edu.au/mu/mu-client-pypy

earlier approaches, and discussed more sophisticated work in recent years which have been happening in parallel with Mu and this thesis' work. The Mu micro virtual machine is one of the recent proposals to aid managed language implementation. We introduced interesting aspects of the abstractions provided by Mu, gave an overview of our Mu implementation, Zebu VM, and briefly introduced other Mu-related projects. The following three chapters will each pick one design point of Zebu, and provide an in-depth discussion.

Chapter 3

A Garbage Collector as a Test Case for High Performance VM Engineering in Rust

High performance virtual machines build upon performance-critical low-level code, typically exhibit multiple levels of concurrency, and are prone to subtle bugs. Implementing, debugging and maintaining a virtual machine can therefore be extremely challenging. The choice of implementation language is a crucial consideration when building a virtual machine. Rust's ownership model, lifetime specification, and reference borrowing deliver safety guarantees through a powerful static checker with little run-time overhead [Rust, 2010]. These features make Rust a compelling candidate for a virtual machine implementation language, but they come with restrictions that threaten expressiveness and efficiency. It is desirable to implement a virtual machine in a language that can benefit the implementation in terms of performance and robustness, however, it is uncertain that if it is feasible to deliver both requirements if the language is too restrictive.

This chapter describes our experience in constructing a prototype garbage collector for the Zebu VM in Rust, which was the first major component of the micro virtual machine that we built in Rust. Although we have subsequently built a complete prototype VM in Rust, we focus here on the garbage collector. Section 3.1 presents our motivation to utilize Rust to implement the Zebu VM. Section 3.2 briefly introduces related work and the Rust language. Section 3.3 uses our garbage collector implementation as a case study to discuss the benefits of Rust, the obstacles encountered, and how we overcame them, with a performance analysis. Section 3.4 concludes this chapter. We find that Rust's safety features do not create barriers for efficiency, and benefit our implementation.

This chapter describes work published in the paper "Rust as a Language to implement high-performance garbage collection" [Lin et al., 2016].

3.1 Introduction

Implementing an efficient and robust virtual machine is not easy, even given the narrow scope of a micro virtual machine. First, a virtual machine must manage its heap, manipulate raw memory, making it naturally prone to memory bugs. Second, a modern virtual machine is rich in concurrency (typically thread parallelism), from the application-level threading to intra-VM concurrency, such as concurrent access to run-time data, thread-local allocation, parallel garbage collection and compilation, which makes it prone to race conditions and extremely time-consuming bugs.

What makes the situation worse is that the implementation language usually does not provide help in terms of memory safety and thread safety. The imperative of performance has traditionally encouraged the use of languages such as C and C++ in virtual machine implementation. But their weak type system, lack of memory safety, and lack of integrated support for concurrency [Boehm, 2005] throws memory and thread safety squarely back into the hands of developers.

Poor software engineering leads not only to hard-to-find bugs and performance pitfalls, but decreases reuse, inhibiting progress by thwarting creativity and innovation. Unfortunately, programming languages often place positive traits such as abstraction and safety at odds with performance. However, we are encouraged: first, by prior work [Blackburn et al., 2004; Frampton et al., 2009; Frampton, 2010] that shows that in a system implementation such as a virtual machine, low-level code is the exception, not the rule; and second by the Rust programming language, which rather boldly describes itself as *a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety* [Rust, 2010].

We evaluate the software engineering of a high performance virtual machine, and our experience confirms the prior work. In particular, we confirm that: (*i*) performance-critical code is very limited in its scope, (*ii*) memory-unsafe code is very limited in its scope, and (*iii*) language-supported, high performance thread-safe data structures are fundamental to a virtual machine implementation. For these reasons, a well-chosen language may greatly benefit our micro virtual machine implementations *without* compromising performance.

Our prior experience in virtual machine implementation includes both C/C++ and high level languages. This, and the emergence of Rust led us to evaluate it as a language for high performance VM implementation. Rust is type, memory, and thread safe; all safety features that we believe will help in delivering a robust virtual machine. Rust provides high performance data structures that are essential to virtual machine development through its rich libraries (including standard and third party ones) with the same safety guarantees, which are a rare find in system programming languages. Rust also permits unsafe operations (and inline assembly[1]) in unsafe blocks, allowing us to access bare memory, and to fine-tune performance on fast paths when needed. Furthermore, Rust uses a powerful compile-time safety checker to shift the safety burden to compile time as much as possible, avoiding run-time overheads where possible. The checker is based on Rust's model of object ownership,

[1] Inline assembly is only available in nightly releases at the time of writing, and not used in this work.

lifetimes, and reference borrowing. The model eliminates the possibility of dangling pointers and races, and ensures memory safety. However, this model also restricts the language's expressiveness. Mapping semantics required of a virtual machine implementation into Rust's language model was the major challenge that we faced during the work. This is especially concerning for the runtime implementation, as it is low-level and tricky to implement, and significantly affects the overall performance of the virtual machine.

We set out our goal as to test the feasibility of using Rust to write a high performance garbage collector, as GC is one of the most correctness and performance crucial components of a virtual machine runtime. Ultimately, we found that not only was this achievable, but that we could do so with no discernable overhead compared to an implementation of the same collector written in C. The results are encouraging, and we further apply the idea to a larger scope, and implement the entire Zebu virtual machine in Rust.

This chapter uses the garbage collector in Zebu as a case study. We start by describing how we are able to *use* Rust's particular language features in our high performance collector implementation. We then discuss cases where we found it necessary to *abuse* Rust's unsafe escape hatches, avoiding its restrictive semantics, and ensuring the performance and semantics we required. Finally, we conduct a head-to-head performance comparison between our collector implementation in Rust and a mostly identical implementation in C to demonstrate that if used properly, the safety and abstraction cost from Rust is minimal, compared to an unsafe language such as C. Also we show that both implementations outperform BDW, a widely used conservative GC library.

The principal contributions of this chapter are: (*i*) a discussion of the challenges of implementing a high-performance collector in a type, memory and thread-safe language, (*ii*) a discussion of the semantic impedance between Rust's language model and the semantic requirements of a collector implementation, (*iii*) a performance comparison evaluating Rust and C implementations of the same high performance collector design, and (*iv*) a comparison with the popular Boehm-Demers-Weiser (BDW) collector implemented in C [Boehm and Weiser, 1988; Boehm et al., 1992].

3.2 Background

This section describes related prior work that utilized a higher-level safe implementation language for virtual machine implementation, and gives an introduction to the Rust language.

3.2.1 Related Work

There has been much previous work addressing the implementation of efficient virtual machines in safe languages [Alpern et al., 1999; Blackburn et al., 2004; Alpern et al., 2005; Rigo and Pedroni, 2006; Blackburn et al., 2008; Frampton et al., 2009; Wimmer et al., 2013]. The advantages of using safe languages demonstrated by these

projects motivate our work. However, our use of Rust takes some further steps: (*i*) Rust guarantees type, memory, and thread safety. Previous work uses languages that make weaker safety guarantees such as type safety and weaken memory safety (by disabling GC) and leave thread safety exposed. (*ii*) Rust has considerably more restricted semantics and expressiveness, since it performs most safety checks at compile time. This constrains runtime implementation, and invites the question of whether implementing a high performance runtime in Rust is even viable. (*iii*) Rust is an off-the-shelf language. We take the challenge to map desired features efficiently to the language semantics and we use Rust without changes to the base language. Previous work changed or augmented the semantics of the implementation languages to favor their runtime implementation [Frampton et al., 2009; Wimmer et al., 2013].

There are also projects[2] that implement garbage collectors in Rust *for Rust*. Though these projects use Rust as the implementation language, their focus is in introducing GC as a language feature to Rust. Their implementations do not reflect the state of the art of GC, and they do not report performance: the delivery of both significantly adds difficulty in a collector implemented in Rust. Our work takes on the challenge of achieving both, to deliver a high-performance advanced collector implementation.

3.2.2 The Rust Language

We now introduce some of the key concepts in Rust used in this chapter.

Ownership In Rust, variable binding grants a variable the unique ownership of the value it is bound to. This is similar to C++11's `std::unique_ptr`, but it is mandatory for Rust as it is the key concept upon which Rust's memory safety is built. Unbound variables are not allowed, and rebinding involves `move` semantics, which transfer the ownership of the object to the new variable while invalidating the old one[3]. When a variable goes out of scope, the associated ownership expires and resources are reclaimed.

References Acquiring the ownership of an object for accessing is expensive because the compiler must emit extra code for its proper destruction on expiry. A lightweight approach is to instead *borrow* references to access the object. Rust allows one or more co-existing *immutable* references to an object or exactly one *mutable* reference with no immutable references. The ownership of an object cannot be moved when it is borrowed. This rule eliminates data races, as mutable (write) and immutable (read) references are made mutually exclusive by the rule. More interestingly, this mutual exclusion is guaranteed mostly at compile time by Rust's borrow checker.

[2] A reference counted type with cycle collection for Rust: https://github.com/fitzgen/bacon-rajan-cc; a simple tracing (mark and sweep) garbage collector for Rust: https://github.com/Manishearth/rust-gc.

[3] Rebinding of `Copy` types, such as primitives, makes a copy of the value for the new variable instead of moving ownerships; the old variable remains valid.

Data Guarantees (Wrapper Types) An important feature of Rust is that the language and its library provide various wrapper types with different guarantees and tradeoffs. For example, plain references such as `&T` and `&mut T` statically guarantee a read-write 'lock' for single-threaded code with no run-time overhead, while `RefCell<T>` offers the same guarantee at the cost of run-time checks but is useful when the program has complicated data flow. Our implementation uses the following wrapper types as will be described in the next section. `Box<T>` represents a pointer which uniquely owns a piece of heap-allocated data. `Arc<T>` is another frequently used wrapper which provides an atomically reference-counted shared pointer to data of type `T`, and guarantees the data stays accessible until every `Arc<T>` to it goes out of scope (i.e., the count drops to zero). A common idiom to share mutable data among threads is `Arc<Mutex<T>>` which provides a mutual exclusive lock for type `T`, and allows sharing the mutex lock across threads.

Unsafe Rust provides a safe world where there are no data races and no memory faults. However, the semantics in safe Rust are in some cases either too restrictive or too expensive. Rust allows unsafe code, such as raw pointers (e.g., `*mut T`), forcefully allowing sharing data across threads (e.g., `unsafe impl Sync for T{}`), intrinsic functions (e.g., `mem::transmute()` for bit casting without check), and external functions from other languages (e.g., `libc::malloc()`). `Unsafe` is a powerful weapon for programmers to wield at their own risk. Rust alerts programmers by requiring unsafe code to be contained within a block that is marked unsafe, or exposed to the caller by marking the containing function as itself unsafe.

3.3 Case Study: A High Performance Garbage Collector in Rust

We describe our experience implementing an Immix garbage collector [Blackburn and McKinley, 2008] in Rust and C. We discuss the benefits of Rust, the obstacles encountered, and how we overcame them. We show that our Immix implementation has almost identical performance on micro benchmarks, compared to its implementation in C, and outperforms the popular BDW collector on the `gcbench` micro benchmark. We find that Rust's safety features do not create significant barriers to implementing a high performance collector. Though memory managers are usually considered low-level, our high performance implementation relies on very little unsafe code, with the vast majority of the implementation receiving all the benefits of Rust's safety. We see our experience as a compelling proof-of-concept of Rust as an implementation language for high performance garbage collectors, and later apply the experience to benefit the entire Zebu VM development.

```
1  #[derive(Copy, Clone, Eq, Hash)]
2  pub struct Address(usize);
3
4  impl Address {
5    // address arithmetic
6    #[inline(always)]
7    pub fn plus(&self, bytes: usize) -> Address {
8      Address(self.0 + bytes)
9    }
10
11   // dereference a pointer
12   #[inline(always)]
13   pub unsafe fn load<T: Copy> (&self) -> T {
14     *(self.0 as *mut T)
15   }
16
17   // bit casting
18   #[inline(always)]
19   pub fn from_ptr<T> (ptr: *const T) -> Address {
20     unsafe {mem::transmute(ptr)}
21   }
22
23   // cons a null
24   #[inline(always)]
25   pub unsafe fn zero () -> Address {
26     Address(0)
27   }
28
29   ...
30 }
```

Figure 3.1: An excerpt of our Address type, showing some of its safe and unsafe methods.

3.3.1 Using Rust

We now describe key aspects of how we *use* Rust's language features to construct a high performance garbage collector. In Section 3.3.2 we discuss how we found it necessary to *abuse* Rust, selectively bypassing its restrictive semantics to achieve the performance and semantics necessary for a high performance collector.

For the sake of this proof of concept implementation, we implement the Immix garbage collector [Blackburn and McKinley, 2008]. We use it because it: (*i*) is a high-performance garbage collector, (*ii*) has interesting characteristics beyond a simple mark-sweep or copying collector, and (*iii*) has a well-documented publicly available reference implementation. Our implementation supports parallel (thread-local) allocation and parallel collection. We have not yet implemented opportunistic compaction, nor generational or reference counting variants [Shahriyar et al., 2013]. We do not limit our discussion to the Immix algorithm, but rather we consider Rust's broader suitability as a GC implementation language.

Our implementation follows three key principles: (*i*) the collector must be high performance, with all performance-critical code closely scrutinized and optimized,

(*ii*) we do not use *unsafe* code unless absolutely unavoidable, (*iii*) we do not modify the Rust language in any way.

The remainder of this section considers four distinct elements of our experience of Rust as a GC implementation language: (*i*) the encapsulation of `Address` and `ObjectReference` types, (*ii*) managing ownership of memory blocks, (*iii*) managing global ownership of thread-local allocators, and (*iv*) utilizing Rust libraries to support efficient parallel collection.

Encapsulating Address Types

Memory managers manipulate raw memory, conjuring language-level objects from raw memory. Experience shows the importance of abstracting over both arbitrary raw addresses and references to user-level objects [Blackburn et al., 2004; Frampton et al., 2009]. Such abstraction offers type safety and disambiguation with respect to implementation-language (Rust) references. Among the alternatives, raw pointers can be misleading and dereferencing an untyped arbitrary pointer may yield unexpected data, while using integers for addresses implies arbitrary type casting between pointers and integers, which is dangerous.

Abstracting address types also allows us to distinguish addresses from object references for the sake of software engineering and safety. Addresses and object references are two distinct abstract concepts in GC implementations: an address represents an arbitrary location in the memory space managed by the GC and address arithmetic is allowed (and necessary) on the address type, while an object reference maps directly to a language-level object, referring to a piece of raw memory that lays out an object and that assumes some associated language-level per-object meta data (such as an object header, dispatch table, etc). Converting an object reference to an address is always valid, while converting an address to an object reference is unsafe.

Abstracting and differentiating addresses is important, but since addresses are used pervasively in a GC implementation, the abstraction must be efficient, both in space and time. We use a single-field *tuple struct* to provide `Address` and `ObjectReference`, abstracting over Rust's word-width integer `usize` to express addresses, as shown in Figure 3.1. This approach disables the operations on the inner type, and allows a new set of operations on the abstract type. This abstraction adds no overhead in type size, and the static invocation of its methods can be further marked as `#[inline(always)]` to remove any call overhead. So while the types have the appearance of being boxed, they are materialized as unboxed values with zero space and time overheads compared to an untyped alternative, whilst providing the benefits of strong typing and encapsulation.

We restrict the creation of `Address`s to be either from raw pointers, which may be acquired from `mmap` and `malloc` for example, or derived from an existing `Address`. `Address` creation from arbitrary integers is forbidden, with the single exception of the constant `Address::zero()`. This serves as an initial value for some fields of type `Address` within other `struct`s, since Rust does not allow `struct`s with uninitialized fields. A safer alternative in Rust is to use `Option<Address>` initialized as `None` to

indicate that there is no valid value. However, this adds a conditional and a few run-time checks to extract the actual address value in the performance-critical path of allocation, which adds around 4% performance overhead. We deem this tradeoff not to be worthwhile given the paramount importance of the allocation fast path and the infrequency with which this idiom arises within the GC implementation. Thus we choose to allow `Address::zero()` but mark it as `unsafe` so that implementers are explicitly tasked with the burden of ensuring safety.

Our implementation of `ObjectReference` follows a very similar pattern. The `ObjectReference` type provides access to per-object memory manager metadata (such as mark-bits/-bytes). An `Address` cannot be safely cast to an `ObjectReference`; the allocator code responsible for creating objects must do so via an `unsafe` cast, explicitly imposing the burden of correctness for fabricating objects onto the implementer of the allocator. An `ObjectReference` can always be cast to an `Address`.

Ownership of Memory Blocks

Thread-local allocation is an essential element of high performance memory management for multi-threaded languages. The widely used approach is to maintain a global pool of raw memory regions from which thread-local allocators take memory as they need it, and to which thread-local collectors push memory as they recover it [Alpern et al., 1999]. Most of the allocations are from the thread-local memory buffer that each thread reserves. This design means that the common case for allocation involves no synchronization, whilst still facilitating sharing of a global memory resource. The memory manager must ensure that it correctly manages raw memory blocks to thread-local allocators, ensuring exclusive ownership of any given raw block. Note, however that once objects are fabricated from these raw blocks, they may (according to the implemented language's semantics) be shared among all threads. Furthermore, at collection time a parallel collector may have no concept of memory ownership, with each thread marking objects at any place in the heap, regardless of any notion of ownership over the object's containing block. We make this guarantee by using Rust's ownership semantics.

Ownership is the key part of Rust's approach to delivering both performance *and* safety. We map the ownership semantics to this scenario to make the guarantee that each block managed by our GC is in a coherent state among *usable*, *used*, or *being allocated into by a unique thread*. To achieve this, we create `Block` objects, each of which uniquely represents the memory range of the block and its meta data. The global memory pool *owns* the `Blocks`, and arranges them into a list of usable `Blocks` and a list of used `Blocks` (Figure 3.2). Whenever an allocator attempts to allocate, it acquires the ownership from the usable `Block` list, gets the memory address and allocation context from the `Block`, then allocates into the corresponding memory. When the thread-local memory block is full, the `Block` is returned to the global *used* list, and waits there for collection. The Rust's ownership model ensures that allocation will not happen unless the allocator *owns* the `Block`, and, further every `Block` is guaranteed to be in one of the three states: (*i*) owned by the global space as a usable `Block`,

```
1  // thread local allcator
2  pub struct AllocatorLocal {
3    ...
4    space: Arc<Space>,
5
6    // allocator may own a block it can allocate into
7    // Option suggests the possibily of being None,
8    // which leads to the slow path to acquire a block
9    block: Option<Box<Block>>
10 }
11
12 // global space, shared among multiple allocators
13 pub struct Space {
14   ...
15   usable_blocks : Mutex<LinkedList<Box<Block>>>,
16   used_blocks   : Mutex<LinkedList<Box<Block>>>
17 }
18
19 impl AllocatorLocal {
20   fn alloc_from_global (&mut self,
21        size: usize, align: usize) -> Address {
22     // allocator will return the ownership of
23     // current block (if any) to global space
24     if block.is_some() {
25       let block = self.block.take().unwrap();
26       self.space.return_used_block(block);
27     }
28
29     // keep trying acquiring a new block from space
30     loop {
31       let new_block
32         = self.space.get_next_usable_block();
33       ...
34     }
35   }
36 }
```

Figure 3.2: Ownership transfer between the global memory pool and a thread local allocator.

(*ii*) owned by a single allocator, and being allocated into, (*iii*) owned by the global space as a used `Block`. During collection, the collector scavenges memory among *used* `Blocks`, and classifies them as *usable* for further allocation if they are free.

Globally Accessible Per-Thread State

A thread-local allocator avoids costly synchronization on the allocation fast path because mutual exclusion among allocators is ensured. This is something that Rust's ownership model ensures can be implemented very efficiently. However, parts of the thread-local allocator data structure may be *shared* at collector time (for example, allocators might be told to yield by a collector thread via this data structure). Rust will not allow for a mixed ownership model like this except by making the data

structure shared, which means that all accesses are vectored through a synchronized wrapper type, ensuring that every allocation is synchronized, thus defeating the very purpose of the thread-local allocator.

We deal with this by breaking the per-thread `Allocator` into two parts, a thread-local part and a global part, as shown in Figure 3.3. The thread-local part includes the data that is accessible strictly within current thread and an `Arc` reference to its global part. All shared data goes to the global part (with either atomic types or a safe wrapper if mutability is required). This allows efficient access to thread local data, while allowing shared per-thread data to be accessed globally.

```
1  pub struct AllocatorLocal {
2    // fields that are strictly thread local
3    ...
4
5    // fields that are logically per allocator
6    // but need to be accessed globally
7    global: Arc<AllocatorGlobal>
8  }
9
10 pub struct AllocatorGlobal {
11   // any field in this struct that requires
12   // mutability needs to be either be atomic
13   // or lock-guarded
14   ...
15 }
16
17 // statics that involve dynamic allocation
18 lazy_static! {
19   pub static ref ALLOCATORS
20     : Vec<Arc<AllocatorGlobal>> = vec![];
21 }
```

Figure 3.3: Separating a per-thread allocator into two parts. The local part is strictly thread local, while the global part can be accessed globally.

Library-Supported Parallelism

Parallelism is essential to high performance collector implementations. Aside from the design of the high level algorithm, the efficiency of a collector depends critically on the implementation of fast, correct, parallel work queues [Ossia et al., 2002]. In a marking collector such as Immix and most tracing collectors, a work queue (or 'mark stack') is used to manage pending work. When a thread finds new marking work, it adds a reference to the object to the work queue, and when a thread needs work, it takes it from the work queue. Ensuring efficient and correct operation of a parallel work queue is a challenging aspect of high performance collector implementation [Ossia et al., 2002; Blackburn et al., 2004].

We were pleased to find that Rust provides a rich selection of safe abstractions that perform well as part of its standard and external libraries (known as *crates* in

Rust parlance), and unlike past works [Alpern et al., 1999; Blackburn et al., 2004; Alpern et al., 2005; Rigo and Pedroni, 2006; Frampton et al., 2009; Wimmer et al., 2013], we are not restricted to a subset of the language semantics. The use of standard libraries is deeply problematic when using a modified or restricted language subset, as has been commonly used in the past. For example, if using a restricted subset of Java, one must be able to guarantee that any library used does not violate the preconditions of the subset, which may be extremely restrictive (such as using only the fully static subset of the language, excluding allocation and dynamic dispatch). Consequently, standard libraries are off limits when using restricted Java to build a garbage collector.

We utilize two crates in Rust, `std::sync::mpsc`, which provides a multiple-producers single-consumer FIFO queue, and `crossbeam::sync::chase_lev`[4], which is a lock-free Chase-Lev work stealing deque that allows multiple stealers and one single worker [Chase and Lev, 2005]. We use these two abstraction types as the backbone of our parallel collector with a modest amount of additional code to integrate them.

Our parallel collector starts single-threaded, to work on a local queue of GC roots; if the length of the local queue exceeds a certain threshold, the collector turns into a controller and launches multiple stealer collectors. The controller creates an asynchronous `mpsc channel` and a shared `deque`; it keeps the receiver end for the channel, and the worker for the deque. The sender portion and stealer portion are `cloned` and `moved` to each stealer collector. The controller is responsible for receiving object references from stealer threads and pushing them onto the shared deque, while the stealers steal work (`ObjectReferences`) from the deque, do marking and tracing on them, and then either push the references that need to be traced to their local queue for thread-local tracing or, when the local queue exceeds a threshold, send the references back to the controller where the references will be pushed to the global deque. When the local queue is not empty, the stealer prioritizes getting work from the local queue; it only steals when the local queue is empty.

Using those existing abstract types makes our implementation straightforward, performant and robust: our parallel marking and tracing features only 130 LOC while there are over one thousand lines of well tested code from the libraries to support our implementation. We measure and discuss the performance of our parallel marking and tracing implementation in Section 3.3.3.

3.3.2 Abusing Rust

In the previous subsection, we described how Rust's semantics affect the implementation of a high performance garbage collector. Though Rust's model is sometimes restrictive, in most cases we were able to fairly straightforwardly adapt the collector design to take full advantage of Rust's safety and performance. However, there are a few places where we found that Rust's safety model was too restrictive to express the necessary semantics efficiently, and thus found ourselves having to dive into `unsafe`

[4] https://github.com/aturon/crossbeam

code, where the programmer bears responsibility for safety, rather than Rust and its compiler.

Shared Bit/Byte Maps

Garbage collectors often implement bit maps and byte maps to represent collection state, mapping addresses to table offsets. Examples include card tables (which remember modified memory regions), and mark tables (which remember marked objects). To implement these correctly and efficiently, they are frequently byte maps (allowing atomic update). Semantics may include, for example, multiple writers but idempotent transitions: during the mark phase, the writers may only set the mark byte (not clear it). For example, an object map indicates the start of objects: in a heap where every object is 8-byte aligned, every byte in such a byte map can represent whether an 8-byte aligned address is the start of an object. In Immix, a *line mark table* is used to represent the state of every line in the memory space — an unsigned byte (`u8`) for every 256-bytes of allocated memory.

During *allocation*, the line mark table may be accessed by multiple allocator threads, *exclusively* for the addresses that they are allocating into. Since every allocator allocates into a non-overlapping memory block, they access non-overlapping elements in the line mark table. However, in Rust, if we were to create the line mark table as a Rust array of `u8`, Rust would forbid concurrent writing into the array. Ways to bypass this within the confines of Rust are to either break the table down into smaller tables, or to use a coarse lock on the large table, both of which are impractical.

On the other hand, during *collection*, the mutual exclusion enjoyed by the allocator does not exist: two collector threads may race to mark adjacent lines, or even the same line. The algorithm ensures that such races are benign, as both can only set the line to 'live' and storing to a byte is atomic on the target architecture. However, in Rust, it is strictly forbidden to modify a shared object's non-atomic fields without going through a lock. We are unaware of a reliable solution to this in stable Rust releases, which do not support an `AtomicU8` type, nor intrinsic atomic operations as in the nightly releases.

Instead, we use the work-around shown in Figure 3.4. We generalize the line mark table as an `AddressMapTable`. We wrap the necessary unsafety into the `AddressMapTable` implementation which almost entirely comprises safe code. We acknowledge also that for proper atomicity of the byte store (with respect to both the compiler and target) we should also be using an atomic operation to store the value rather than a normal assignment. Here we rely on the Rust compiler to generate an x86 byte store which is atomic. Otherwise, there are reasonable compiler optimizations that could defeat the correctness of our code [Boehm, 2011]. What is more, the target architecture might not have an atomic byte store operation. The availability of LLVM intrinsics in the non-stable nightly Rust releases would allow us to use a relaxed atomic store to achieve the correct code, as shown in the comment. This exposes a shortcoming in Rust's current atomic types where we desire an `AtomicU8` type, along the lines of the existing `AtomicUsize`. This need is reflected in the recently accepted

```
1  pub struct AddressMapTable {
2    start : Address,
3    end   : Address,
4
5    len : usize,
6    ptr : *mut u8
7  }
8  // allow sharing of AddressMapTable across threads
9  unsafe impl Sync for AddressMapTable {};
10 unsafe impl Send for AddressMapTable {};
11
12 impl AddressMapTable {
13   pub unsafe fn set (&self, addr: Address, value: u8)
14   {
15     let index = addr.diff(self.start) >> LOG_PTR_SIZE;
16     unsafe {
17       let ptr = self.ptr.offset(index);
18       // intrinsics::atomic_store_relaxed(ptr, value);
19       *ptr = value;
20     }
21   }
22 }
```

Figure 3.4: Our AddressMapTable allows concurrent access with unsafe methods. The user of this data structure is responsible for ensuring that it is used safely.

Rust RFC #1543: *'Add more integer atomic types'* [Rust RFC 1543, 2016].

3.3.3 Evaluation

The two primary objectives of our proof-of-concept implementation were to establish: (*i*) to what extent we are able to exploit Rust's safety (hopefully minimizing the amount of unsafe code), and (*ii*) the impact of Rust on performance. In this section, we discuss our evaluation of our proof-of-concept collector, focusing on these concerns.

Safe Code

Our first major challenge was to map our collector design into the Rust language. As we discuss in Sections 3.3.1 and 3.3.2, for the main part, the collector implementation can be expressed entirely in safe Rust code. As shown in Table 3.1, 96 % of 1449 lines of the code are safe. This suggests that though GC is usually considered to be a low-level module that operates heavily on raw memory, the vast majority of its code can in fact be safe, and can benefit from the implementation language if that language offers safety.

Language	Files	Lines of Code	Unsafe LOC (%)
Rust	13	1449	58 (4.0%)

Table 3.1: Unsafe code is minimal in our GC implementation.

The unsafe code amounts to 4.0 % and mainly comes from just two sources. The first is where `unsafe` is required for access to raw memory, such as dereferencing raw pointers during tracing, manipulating object headers, zeroing memory, etc. This is unavoidable in memory manager implementations. Our experience demonstrated that through proper abstraction, the unsafe code for accessing raw memory can be restricted to a small proportion of the code base. The second source of unsafety is due to Rust's restricted semantics. Rust trades expressiveness for the possibility of statically enforcing safety. Section 3.3.1 shows that for most of the cases, we are able to adapt our collector implementation to Rust's constraints. In the exceptional case described in Section 3.3.2 where Rust stands in our way, we are able to encapsulate it in a small amount of unsafe code.

Our experience demonstrates that a garbage collector can be implemented in a safe language such as Rust with very little unsafe code. Furthermore, we can report that, subjectively, the discipline imposed upon us by Rust was a real asset when we went about this non-trivial systems programming task with its acute *performance and correctness* focus.

Performance

Our second challenge was to deliver on our goal of high performance. Since at this stage we are implementing a standalone garbage collector, not yet integrated into a larger language runtime, it is hard to provide performance evaluation via comprehensive benchmarks; instead we use micro benchmarks to evaluate the collector. We are not interested in evaluating garbage collection algorithms per se (we take an existing algorithm off the shelf). Rather, we simply wish to provide proof of concept for an implementation of a high performance collector in Rust and show that it performs well in comparison to an equivalent collector written in C. To this end, we are particularly interested in the performance of critical hot paths, both for collection and allocation since the performance of the algorithm itself is already established [Blackburn and McKinley, 2008], and our prior experience demonstrates the overwhelming criticality of these hot paths to performance.

Benchmarks. To evaluate the performance of our implementation in Rust, we also implemented the collector in C, following the same Immix algorithm. We did not try to make the two implementations exactly identical, but used the features of the available language in a naturally fluent way. For most scenarios described in Section 3.3.1 and 3.3.2, it is either unnecessary or simply impossible to write C code the same way as Rust code. The C implementation allows us to set a baseline for performance in an implementation language that is known to be efficient and allows a head-to-head comparison for Rust performance. We took particular care to ensure that the performance-critical hot paths were implemented efficiently in the respective languages.

We chose three performance-critical paths of the collector to run single-threaded as micro benchmarks: allocation, object marking, and object tracing. Each micro

benchmark allocates 50 million objects of 24 bytes each, which takes 1200 MB of heap memory; we use 2000 MB memory for each run so that the GC will not collect spontaneously (we control when tracing and collection occur in the respective micro benchmarks). In each micro benchmark, we measure the time spent on allocating, marking, and tracing the 50 million objects. We use rustc 1.6.0 stable release for Rust, and clang 3.7 for C, both of which use LLVM 3.7 as back-end. We run each implementation with 20 invocations on a 22 nm Intel Core i7 4770 processor (Haswell, 3.4 GHz) with Linux kernel version 3.17.0. The results appear in Table 3.2.

	C	Rust (% to C)
`alloc`	370 ± 0.1 ms	374 ± 2.9 ms (101%)
`mark`	63.7 ± 0.5 ms	64.0 ± 0.7 ms (100%)
`trace`	267 ± 2.1 ms	270 ± 1.0 ms (101%)

Table 3.2: Average execution time with 95% confidence interval for micro benchmarks of performance critical paths in GC. Our implementation in Rust performs the same as the C implementation.

From the micro benchmark results, we can see that with careful performance tuning, the Rust implementation matches the performance of our C implementation across all the three micro benchmarks (identifying most performance critical paths in a collector implementation). In our initial implementation (without fine performance tuning), Rust was within 10% slowdown of C on micro benchmarks. We found it encouraging, considering: (*i*) our source code in Rust offers stronger abstraction than C, a low-level imperative language, and (*ii*) the source code enjoys Rust's safety guarantees. We then fine-tuned the performance, examining and comparing assembly code generated for each implementation, where necessary anticipating code generation from the Rust compiler and altering patterns in the fast path code to avoid idioms with negative performance implications. Our micro benchmarks have tiny kernels and are memory intensive, and one instruction may affect results. We found that although `rustc` is aggressive it is quite predictable, making it relatively easy to generate highly performant code. Lack of tools for finer control on the generated code such as branch hints may be a drawback of the Rust compiler, but did not hinder performance in the micro benchmarks.

Library-based Parallel Mark and Trace. We evaluate the performance scaling of parallel GC in our implementation. As described in Section 3.3.1, we quickly implemented the parallel mark and trace collector by completely basing its parallelism on existing Rust crates: `std::sync::mpsc` and `crossbeam:: sync::chase_lev`. They provide the basis for all of the concurrency in our parallel collector. This implementation approach is high-level and productive, but as we shall show, it is also performant.

We use a micro benchmark to trace 50 quad trees of depth ten to allow parallel collectors to build a reasonable local work queue for thread-local tracing and to push excessive references to the global deque. We use a large heap to avoid spontaneous

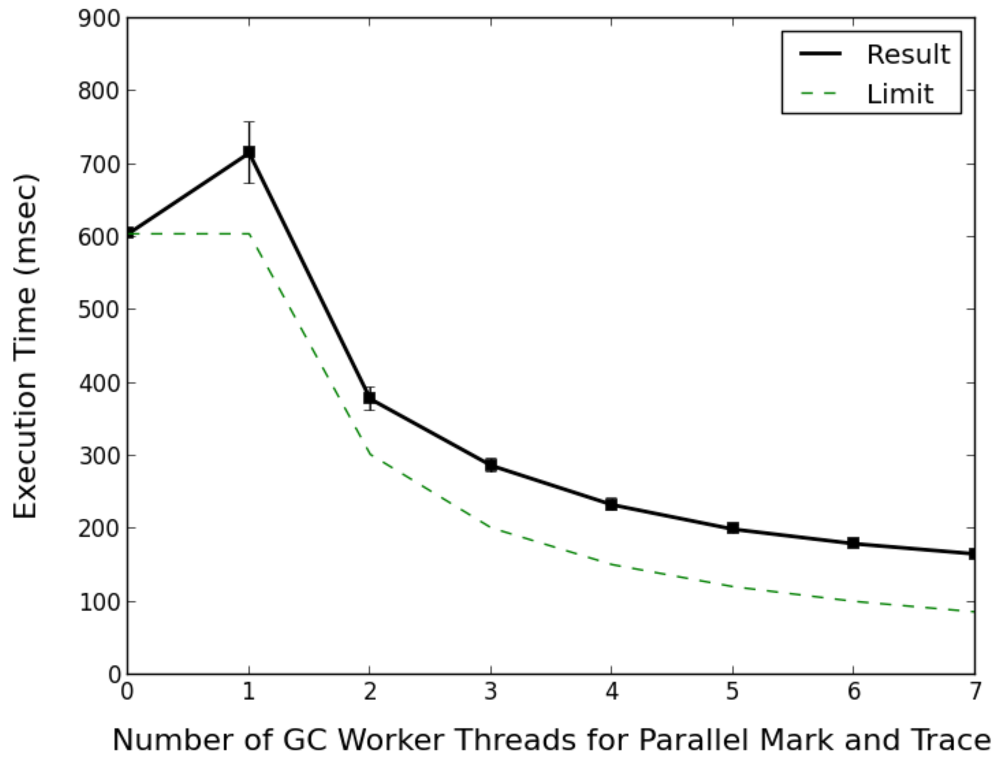


Figure 3.5: Performance scaling for our fast implemented libraries-based parallel mark and trace collector. Dotted line illustrates the performance if workloads are equally distributed among threads with no overhead.

collections during the tree allocation. We run this with twenty invocations on the same i7 Haswell machine, using from zero to seven GC worker threads, and measure the tracing time. Note that zero means no parallel GC while one to seven reflect the number of GC worker threads (with one additional controller thread). Seven workers with one controller is the full capacity of our machine (four cores with eight SMT threads). Figure 3.5 shows the results along with a line indicating hypothetical perfect scaling (which assumes workloads are equally divided among threads with no overhead compared to a single-threaded collector).

With parallel GC disabled, single-threaded marking and tracing takes 605 ms, while with one worker thread, the benchmark takes 716 ms. The overhead is due to sending object references back to the global deque through an asynchronous channel, and stealing references from the shared deque when the local work queue is empty. With two and three worker threads, the scaling is satisfactory, with execution times of 378 ms and 287 ms (52.8 % and 40.0 % compared with one worker). When the number of worker threads exceed four, the scaling starts to fall off slightly. With seven worker threads, the execution time is 166 ms, which is 23.2 % of one worker thread. The performance degradation is most likely from two sources: (*i*) GC workers start to

share resources from the same core after every core hosts one worker, (*ii*) having one central controller thread to receive object references and push them to the global deque starts to be a performance bottleneck. These results could undoubtedly be improved with further tuning. However, as it is one tricky part of the implementation, and we worked towards a working (and safe) implementation with limited time and limited lines of code by using existing libraries, the approach itself is interesting and demonstrates the performance tradeoff due to improved productivity. We believe the performance scaling is good, and that having a language that provides higher level abstractions can benefit a parallel GC implementation (and possibly a concurrent GC) greatly.

GCBench. We compare our Immix implementation in Rust with the BDW collector on the `gcbench` micro benchmark. We enable thread-local allocators and parallel marking with eight GC threads on BDW. We run on the same machine for the comparison, and use a moderate heap size of 25 MB (which is roughly 2× the minimal heap size to allow heap stress).

In a run of 20 invocations (as shown in Table 3.3), the average execution time for BDW is 172 ms, while the average for our implementation is 97 ms (79 % faster). We do not find the result surprising. Our GC implements an Immix allocator which is mainly a bump pointer allocator, while BDW uses a free list allocator. Immix outperforms freelist allocators by 16% in large benchmarks [Shahriyar et al., 2014]; we expect the performance advantage is even bigger in micro benchmarks that allocate in a tight loop. We ran our `alloc` micro benchmark for BDW, and we find that the performance difference between our allocator and the BDW allocator is similar, confirming our belief. Our GC implementation is different from the BDW collector in a few other respects, which contribute to the performance difference: (*i*) Our GC is conservative with stacks but precise with the heap, while the BDW collector is conservative with both; (*ii*) Our GC presumes a specified heap size and reserves contiguous memory space for the heap and metadata side tables during initialization, while the BDW collector allows dynamic growing of a discontiguous heap.

We also compared the two collectors with a multi-threaded version of `gcbench` (as `mt-gcbench` in Table 3.3). Both collectors use eight allocator threads, and eight GC threads. The results show that our implementation performs 3× faster than BDW on this workload. Our implementation outperforms the BDW collector on `gcbench` and `mt-gcbench` (respectively by 79 % and 2×), which suggests our implementation in Rust delivers good performance compared to the widely used BDW collector.

	BDW	Immix(Rust)	% of BDW
`gcbench`	172 ± 0.8 ms	97 ± 0.3 ms	56%
`mt-gcbench`	1415 ± 3.1 ms	466 ± 1.9 ms	33%

Table 3.3: Performance comparison between our Immix GC in Rust and BDW on gcbench and multi-threaded gcbench.

We conclude that using Rust to implement GC does not preclude high performance, and justify this with the following observations: (*i*) our implementation in Rust performs as well as our C implementation using the same algorithm in performance-critical paths, and (*ii*) our implementation in Rust outperforms the widely-used BDW collector on `gcbench` and `mt-gcbench`. This result shows the capability of Rust for high-performance GC implementations, as a language with memory, thread and type-safety.

3.4 Summary

Rust is a compelling language that makes strong claims about its suitability for systems programming, promising both performance *and* safety. We found that the Rust programming model is quite restrictive, but not needlessly so. We demonstrated the feasibility of using Rust to implement an Immix GC, matching the performance of an implementation in C. We found that the vast majority of the collector could be implemented naturally, without difficulty, and without violating Rust's restrictive static safety guarantees.

Our experience was very positive: we enjoyed programming in Rust, we found its restrictive programming model *helpful* in the context of a garbage collector implementation, we appreciated access to its large range of libraries, and we found that it was not difficult to achieve excellent performance. Based on the experience, we further apply our experience to implement the entire Zebu VM in Rust. We believe the case study of a GC implementation is sufficient to demonstrate our point: utilizing a thread-/memory-/type-safe language to implement low-level systems is beneficial without necessarily compromising performance. Though the entire VM implementation benefits from Rust, especially from the ownership model and thread safety, this thesis does not further elaborate on our implementation details in terms of utilizing Rust.

In the next chapter, we will discuss another important design point for a virtual machine implementation, *yieldpoints* for thread synchronization.

Chapter 4

Yieldpoints as a Mechanism for VM Concurrency

Yieldpoints allow a running program to be interrupted at well-defined points in its execution, facilitating exact garbage collection, biased locking, on-stack replacement, profiling, and other important virtual machine behaviors. Yieldpoints are critical to the implementation of a virtual machine as a fundamental mechanism to support thread synchronization, yet the design space is not well understood.

In this chapter, we identify and evaluate yieldpoint design choices. Section 4.1 motivates this chapter by introducing the importance of yieldpoints in a modern virtual machine implementation. Section 4.2 presents the background, analysis and related work associated with this chapter. Section 4.3 fully explores the design space of yieldpoints, from two orthogonal perspectives, *mechanisms* and *scopes*. Section 4.4 presents our methodology, and evaluation on both taken and untaken yieldpoints performance and time-to-yield latency. Note that we conducted the analysis and experiments on Jikes RVM [Alpern et al., 1999], a high-performance Java virtual machine, with real world benchmarks [Blackburn et al., 2006] for more sound performance analysis. Section 4.6 concludes this chapter with the finding that different designs of yieldpoints expose different trade-offs, however, the conditional polling yieldpoint has the most desirable characteristics: low overhead, fast time-to-yield, and implementation simplicity. The analysis and evaluation in this chapter allows us to understand the background, the design space and the trade-offs of yieldpoints, which further allows us the confidence to pick suitable designs for Zebu VM.

This chapter describes work published in the paper "Stop and Go: Understanding Yieldpoint Behavior" [Lin et al., 2015]

4.1 Introduction

A *yieldpoint* is a frequently executed check by managed application code in high performance managed run-time systems, used to determine when a thread must yield. Reasons to yield include garbage collection, user-level thread pre-emption, on-stack replacement of unoptimized code with optimized code, biased locking, and profiling for feedback directed optimization. Yieldpoints ensure that each thread is in a state

that is coherent for the purposes of the yield, such as knowing the precise location of all references in the registers and stacks for exact garbage collection, and that relevant operations such as write barriers and allocation have completed (i.e., are not in some inconsistent partial state). These properties are less easily assured if threads suspend at arbitrary points in their execution. Coherence is essential when the virtual machine needs to introspect the application thread or reason about interactions between the thread and the virtual machine or among multiple application threads. In the case of exact garbage collection, yieldpoints are known as *GC-safe points* [Jones et al., 2011]. Compilers may generate a GC *map* for each yieldpoint, allowing the run-time system to identify heap pointers precisely within the stacks and registers of a yielded thread.

To avoid unbounded waits, yieldpoints typically occur on loop back edges and on method prologs or epilogs of the application, either in the interpreter or in code placed there by the compiler. Consequently, yieldpoints are prolific throughout managed code. Yieldpoints may also be performed explicitly at other points during execution, such as at transitions between managed and unmanaged code.

Despite their important role, to our knowledge there has been no detailed analysis of the design space for yieldpoints nor analysis of their performance. This study examines both. We conduct a thorough evaluation of yieldpoints, exploring how they are used, their design space, and performance. We include designs that to our knowledge have not been evaluated before, as well as two designs that are well known. We start by measuring the static and dynamic properties of yieldpoints across a suite of real-world Java benchmarks. We measure[1] their static effect on code size as well as the dynamic rate at which yieldpoints are executed and the rate at which a yieldpoint's slow path is taken (making the thread yield). Statically, yieldpoints account for about 5 % of instructions. The Java benchmarks we evaluate perform about 100 M yieldpoints per second, of which about 1/20 000 are taken. We show that, in our measurement, among the different uses of yieldpoints, by far the most common reason for yielding is to perform profiling for feedback directed optimization (FDO), which, in Jikes RVM, occurs once every 4 ms and triggers multiple yieldpoints to be taken for sampling in each executing thread. By comparison, garbage collection occurs far less frequently, and in most benchmarks lock revocation in support of biased locking is very rare.

We examine the design space, including two major dimensions. The first dimension is the mechanism for deciding whether to yield, which may be implemented as: (*i*) a conditional guarded by a state variable, (*ii*) as an unconditional load or store from/to a guard page, or (*iii*) via code patching. The conditional yields when the state variable is set and a branch is taken, the unconditional load or store yields when the guard page is protected and the thread is forced to handle the resulting exception, while code patching can implement a branch or a trap (both unconditional). The second design dimension is the scope of the signal, which may be global or per-thread. A global yieldpoint applies to all threads (or none), while a per-thread yieldpoint can

[1] We conducted the measurement and analysis on Jikes RVM, so the numbers are specific for Jikes RVM. However, other virtual machines should follow a similar pattern, and our methodology can be applied for measuring other systems as well.

target individual threads to yield.

We identify a new opportunity for yieldpoint optimization. Rather than using code patching to turn unconditional yields on or off (which requires that *all* yieldpoints be patched) as Agesen [1998] did, we can use code patching to selectively toggle frequently executed yieldpoints. We also show that a yieldpoint implemented as an unconditional store can serve double-duty as a very low overhead profiling mechanism. If the unconditional store writes a constant that identifies characteristics of the particular yieldpoint (e.g., location or yielding thread), then a separate profiling thread can sample the stores and thus observe the yieldpoints as they are traversed [Yang et al., 2015].

We evaluate each of the design points. Among these designs, the most important tradeoff is due to the choice of mechanism, with explicit checks incurring the highest overhead in the common untaken case, around 2 %, but delivering the fastest time-to-yield, while the unconditional load or store has a lower overhead in the common case, 1.2 % at best, but has worse time-to-yield performance. The code patching yieldpoint is slightly different than the other yieldpoint designs. Code patching yieldpoints have superior common case overhead, but the cost of patching *all* yieldpoints outweighs any benefit on modern hardware. We also evaluate the tradeoffs inherent to using code patching as an optimization.

Our analysis gives new insight into a critical but overlooked aspect of garbage collector implementation, identifies a new yieldpoint optimization, and new opportunities for very low overhead profiling.

4.2 Background, Analysis, and Related Work

We now describe yieldpoints in more detail and quantitatively evaluate how yieldpoints are used.

4.2.1 Background

In principle, language virtual machines are concurrent. This is clear in the case of languages such as Java that support concurrency, but even in the case where the supported language offers no application-level concurrency, such as JavaScript, the relationship between the application code and the underlying run-time system is fundamentally concurrent. The concurrency may be explicit, with run-time services executing in discrete threads or it may be implied, with the underlying run-time services and the application interleaving their execution by time-slicing a single thread. Yieldpoints are a critical mechanism for coordinating among application threads and the run-time system.

Yieldpoints serve two complementary goals. First, they provide precise code points at which the execution state of each application thread is observably coherent, allowing the possibility of unobserved incoherent states between yieldpoints. For example, by ensuring that garbage collection only occurs at yieldpoints, we are assured that a multi-instruction write barrier will be observed in its entirety or not

at all. Second, yieldpoints reduce the cost of maintaining metadata with which the thread's state may be introspected. In general, introspection of an application thread depends on metadata (e.g., stack maps) to give meaning to the machine state of the application at any point in time. For example, the type of a value held by a machine register at a given moment will determine whether the value should be interpreted as a pointer, in which case its referent must be retained by the garbage collector, or a floating point number, in which case the value must not be altered. Because such metadata is expensive both in terms of space and in the engineering overhead of coherently generating and maintaining it, language runtimes typically only maintain such metadata for a limited set of code locations.

When yieldpoints are used to coordinate garbage collection it is typically adequate for the yield to have global scope — when activated, all application threads yield to the collector. However, when yieldpoints are used for one-to-one interactions between threads, such as for lock revocation in support of biased locking [Pizlo et al., 2011], or to support work-stealing [Kumar et al., 2012], a per-thread scope is necessary for good performance. These considerations affect the yieldpoint design space, which is discussed in Section 4.3.

Yieldpoints are either *injected* into the application execution by the interpreter or compiler, or they are *explicit*, called by the underlying runtime at key points such as transitions into and out of native code. The focus of our study is *injected* yieldpoints, which are prolific.

4.2.2 Analysis

We now present an analysis of the prevalence of yieldpoints, dynamically and statically, and the rate at which yieldpoints are taken. We use a suite of Java benchmarks and instrument a virtual machine to count yieldpoints. Because the instrumentation slows the virtual machine significantly, we use execution times for the uninstrumented virtual machine as our baseline when measuring rates. The details of our methodology are presented in Section 4.4.1.

To measure the static impact of yieldpoints on code size, we compiled a large body of Java code (the boot image for Jikes RVM) using Jikes RVM's optimizing compiler (with `O2` optimization level) and found that the resulting machine code increased in size from 13.6 MB to 14.6 MB (i.e., by 7.2 %) when a basic conditional yieldpoint was injected on each loop back edge, method prolog and epilog.

We measure the dynamic impact of yieldpoints by instrumenting the injected code to count the number of times injected yieldpoints are executed, and the number of times yieldpoints are taken for FDO profiling, lock revocation, and garbage collection. We used the execution time for uninstrumented code to determine yieldpoint rates[2].

Figure 4.1 shows the rate at which yieldpoints are executed and taken across the suite of Java benchmarks per millisecond. The green bars indicate the rate at

[2]The instrumentation slows down the execution (on average, 17.7% slow down). Because FDO sampling occurs at fixed wall-clock intervals, more of them will occur during the benchmark execution, which causes a bias for the sampling yieldpoint rate. We scaled them down proportionally.

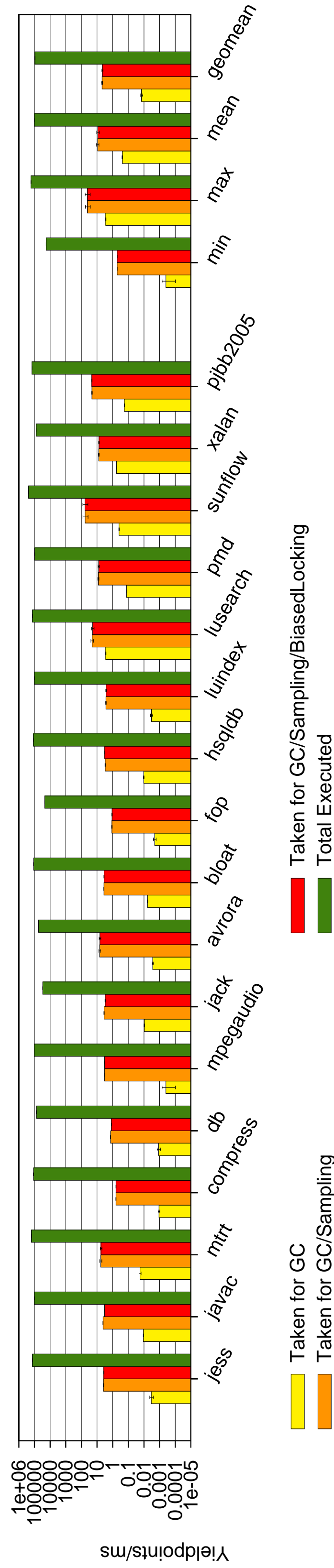


Figure 4.1: Dynamic yieldpoint rates per millisecond for Java benchmarks, showing those taken due to GC (yellow), those taken due to GC *or* sampling for FDO (orange), all taken yieldpoints (red), and all yieldpoint executions, whether taken or not (green). Counts are per thread, so multi-threaded benchmarks such as lusearch and xalan show noticeably higher take rates, reflecting their higher thread count.

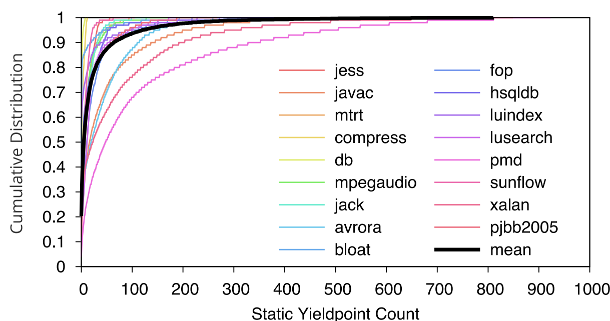


Figure 4.2: Cumulative frequency distribution of dynamic yieldpoint execution rates for each of our benchmarks, showing that a small number of static yieldpoints contribute most of the dynamic execution.

which yieldpoints are executed, taken or not. On average about 100 M yieldpoints are executed per second; about one every 10 ns, which is roughly one every 40 cycles on our 3.4 GHz machine. Of these, around 1/20 000 yieldpoints are taken. Sampling for FDO (orange and red bars) dominates the reasons for yieldpoints to be taken. Jikes RVM uses bursty sampling [Arnold and Grove, 2005], initiating sampling on each thread once every 4 ms. Once initiated, samples are taken at the next N method prologs, where N is 8. The degree of simplicity and longevity of the benchmarks affects the precise number of samples taken. Our counts are totals across all threads, so the multi-threaded benchmarks such as `lusearch`, `sunflow`, `xalan` and `pjbb` have their counts inflated in proportion to the number of threads they are running. The yellow bar indicates the number of yieldpoints due to garbage collection, and reveals that only a small fraction of taken yieldpoints are due to garbage collection. The difference between red and orange bars reflects the number of yieldpoints taken due to lock revocation, revealing that this is very rare among our benchmarks.

We further identify every single yieldpoint inserted by the compiler and maintain an execution count for each, and Figure 4.2 shows the cumulative frequency of yieldpoint execution across different benchmarks. The figure suggests that among more than 35 k yieldpoints inserted by the compiler for each benchmark, just a few hundred account for most executions. On average, just 315 yieldpoints (∼1 %) account for 99 % of all the yieldpoint executions, dynamically, and 681 yieldpoints (∼2 %) account for 99.9 % of all executions. In the worst case (`xalan`), the same percentile is 849, which is still a tiny fraction of the static yieldpoint count. This result is interesting because it suggests that, despite the pervasiveness of yieldpoint insertion and execution, less than 1 % of yieldpoints dominate the behavior. This can possibly be exploited for a more optimized yieldpoint design as will be discussed in Section 4.5.

Summarizing, Figure 4.1 shows that: (*i*) yieldpoints are executed at a very high frequency, (*ii*) they are relatively rarely taken, and (*iii*) that sampling for FDO dominates garbage collection and lock revocation as a reason for yieldpoints to be taken, while Figure 4.2 shows that a tiny fraction of yieldpoints dominate execution.

4.2.3 Related Work

To the best of our knowledge, despite their importance to language behavior and performance, no prior work has conducted a detailed study of yieldpoint design and implementation.

Agesen [1998] focuses purely on mechanisms for GC-safe points, comparing an unconditional store to a guard page (*'polling'*) with a *code patching* mechanism on SPARC machines. His code patching mechanism injects noop instructions to replace *all* yieldpoints. To trigger the yield, the run-time system patches every yieldpoint site, replacing the noop instructions at each site with a call. Agesen used a set of benchmarks comprising `specjvm`, `specjvm` candidates, and two non-trivial multi-threaded benchmarks. He reported that code patching for SPARC has a 6.6 % higher space cost than an unconditional store, on average, but delivers a 4.8 % speedup. We evaluate this design point on modern hardware and show that code patching costs dominate.

Our work differs from this prior work in multiple ways. First, we provide a detailed categorization of generalized *yieldpoint* mechanisms suited to a variety of purposes in modern run-time systems. We consider garbage collection as one use of yieldpoints, among others. The two implementations of GC-safe points measured by Agesen [1998] are what we call a *global store trap-based yieldpoint* and *global code patching yieldpoint*. Second, our methodology allows us to evaluate different yieldpoint implementations over a baseline that has no injected yieldpoints. This allows us to understand the performance overheads for each configuration. In contrast, the previous work evaluated two implementations against each other with no baseline. Our selection of benchmarks is more mature, and contains a set of real-world multi-threaded applications. Since yieldpoints are naturally designed for multi-threaded contexts, our benchmark choice enables studies such as per-thread yield latency and worst-case yield latency, which are important for real-time and concurrent garbage collection. Third, we identify code patching as an optimization over other yieldpoint designs. Finally, the previous work was evaluated on venerable SPARC machines of more than fifteen years ago: what was true then may not be true now. Our experiments evaluate and report for contemporary hardware.

Click et al. [2005] distinguish *GC-safe points* and *checkpoints* in their work related to pauseless GC algorithms. GC-safe points are the managed code locations where there is precise knowledge about the contents of registers and stacks, while checkpoints are synchronization locations for all mutator threads to perform some action. Our study projects a more detailed categorization of yieldpoints and their implementations.

Stichnoth et al. [1999] proposed an interesting alternative to the compiler injected yieldpoints discussed here. They focus on maintaining comprehensive GC maps that cover all managed code instruction locations so as to allow garbage collection to occur at any location without the need for designated yieldpoints. They report significant overhead for the resulting GC maps (up to 20% of generated code size) even after efforts to compress the maps. This may not be desirable in practice, so compiler-injected yieldpoints are widely used in language implementations.

4.3 Yieldpoint Design

In this section we categorize different implementations of *compiler injected yieldpoints* and describe the use of code patching as an optimization. Our focus is the use of yieldpoints in managed language implementations, where applications must yield occasionally to service run-time system requests. A given yieldpoint may be associated with compiler-generated information that records GC stack maps, variable liveness, etc. As an alternative to compiler injected yieldpoints, non-cooperative systems that do not rely on compiler support may use operating system signals to interrupt a native thread to 'yield' at arbitrary program locations [Boehm and Weiser, 1988]. This approach injects no code in the application, and only requires a signal handler to deal with the interrupt. However, the run-time system can make no assumptions about where the yield occurs, and this further prevents any useful information to be associated with the yielding location (such as stack maps for exact GC). For managed run-time systems it is much more desirable to be able to exploit such information, so we exclude the non-cooperative techniques from our categorization and focus on discussing compiler injected yieldpoints for managed language run-time systems.

4.3.1 Mechanisms

Because yieldpoints are frequently executed and seldom triggered, the common implementation pattern is to use the *fast-path/slow-path idiom*. The fast-path is pervasively inserted into managed application code, and does a quick check to decide whether there is any incoming request. If there is, the yieldpoint is taken and control flow goes to the slow-path which further decodes the request, and reacts accordingly. If the yieldpoint is not taken then execution continues at the next application instruction. Control transfer to the slow-path may be via a direct or indirect conditional branch, or by having the fast-path trigger a hardware trap that can be fielded by a matching trap handler.

Conditional Polling Yieldpoints This yieldpoint implementation involves a condition variable. The compiler injects a constant comparison against the value of the variable and a conditional jump to the slow path on true. In normal cases, the condition is not met, and the jump falls through to the next instruction. When the yieldpoint is enabled the jump transfers control to the slow path to execute the yield. Figure 4.3(a) shows the fast-path implementation for conditional polling yieldpoints. Jikes RVM uses this mechanism [Alpern et al., 1999].

One advantage of conditional polling yieldpoints is that they provide flexibility and allow easy implementations of yieldpoints for a finer *scope*. The compiler can generate different conditional comparison instructions for yieldpoints at various locations, and at run time the variable can be set to different values to allow a subset of the conditional comparisons to be triggered, so that only a subset of yieldpoints can be taken. For example, the compiler emits `cmp [offset] 0; jne call_yieldpoint;` for Group A and `cmp [offset] 0; jgt call_yieldpoint;` for Group B. At run-time,

```
1 .yieldpoint:
2         cmp 0 [TLS_REG + offset]
3         jne call_yieldpoint
4 .normal_code:
5         ...
```

(a) Conditional

```
1 .yieldpoint:
2         test 0 [TLS_REG + offset]
3 .normal_code:
4         ...
```

(b) Trap-based Load

```
1 .yieldpoint:
2         mov 0 [TLS_REG + offset]
3 .normal_code:
4         ...
```

(c) Trap-based Store

Figure 4.3: Thread-local polling yieldpoints

if the conditional variable is set to -1, then only Group B takes the yieldpoints.

Moreover, the condition variable can be held in a thread-local variable, allowing yieldpoints to trigger only for particular threads.

Trap-Based Polling Yieldpoints This yieldpoint implementation involves a dedicated memory page that can be protected as appropriate. The compiler injects an access (read or write) to the page as the yieldpoint fast-path (see Figures 4.3(b) and 4.3(c)). In the common case the access succeeds and will not trigger the yieldpoint. Enabling the yieldpoints is simply a matter of protecting the page (from read or write as appropriate) to make the yieldpoint instruction generate a trap. Here the slow path is the handler used to field the trap. A load yieldpoint on x86 can be implemented as a `cmp`, or `test` to avoid the use of a scratch register. The store implementation can be exploited to store useful profiling information such as the address of the currently executing method, or the address of the yieldpoint instruction itself. The Hotspot VM uses trap-based load yieldpoints on a global protected page [HotSpot VM, 2017].

Once again, the access can be to a page held in a thread-local variable, allowing yieldpoints to trigger only for particular threads.

Code Patching Yieldpoints Besides the polling mechanisms described above, code patching is another possible mechanism to implement yieldpoints. A common use is *`NOP` patching*. The compiler injects several bytes of `NOP`s at yieldpoint locations, which makes no meaningful change in the generated application code. To trigger a yieldpoint, the run-time system simply iterates through the code space or a stored

list of *all* yieldpoint code addresses, and patches code by replacing the `NOPs` with other instructions that cause control to flow to the yieldpoint slow path. Intuitively, this approach imposes the lowest fast-path overhead (both in terms of space and time), but enabling yieldpoints is costly. Agesen [1998] reported the use of this approach in 1998, and found it faster than conditional polling on a SPARC machine of that era. Our evaluation on modern hardware shows that the cost of patching the instructions dominates any potential advantage. A similar mechanism is often used for *watchpoints*, which we consider a finer-grained subtype of yieldpoints — watchpoints can be turned on and off per group, as will be discussed below.

4.3.2 Scope

Besides categorizing yieldpoints from the perspective of implementing mechanisms, we also categorize yieldpoints by different levels of scope. From coarser to finer levels, we discuss three scopes: *global*, *thread-local*, and *group-based*.

Global Yieldpoints These are turned on and off all at once to trigger a global synchronization of all application threads. Global yieldpoints are useful for global events such as stop-the-world GC. Yieldpoints of this scope can be implemented with different mechanisms: using a global conditional variable, a single global protected page, or an indiscriminate pass of patching through the whole code space.

Thread-Local Yieldpoints These can be turned on and off for a single thread or a group of threads. They can be used for global synchronization if the targeted threads include all the running threads. Yieldpoints of this scope are useful for targeted per-thread events, such as pair handshakes between two threads. Yet they provide flexibility as they can also be used for global events. As noted above, using a thread-local condition variable or thread-local protected page enables thread-local conditional polling or trap-based polling, respectively. However, there is no straight-forward implementation of a thread-local unconditional code patching yieldpoint [Agesen, 1998], since there is no easy guarantee of the patched code being executed only by certain threads.

Group-based Yieldpoints (Watchpoints) These are grouped, and can be turned on and off by group. They are also known as *watchpoints*, as we discussed in Chapter 2, Section 2.2.5 for `WATCHPOINT` in Mu. This type is useful as guards for purposes such as speculative execution. For example, in places where the compiler makes an assumption regarding type specialization or an inlining decision, it inserts a group-based yieldpoint before the specialized or inlined code. Whenever the run-time system notices that the assumption breaks, it enables that group of yieldpoints to prohibit further execution of the code under the false assumption. Code that reaches the enabled yieldpoints will take a slow-path, where the run-time compiler can make amends and generate new valid code. Code patching is the most straight-forward mechanism implementing group-based yieldpoints, since it naturally needs to know

the offset of each yieldpoint. To adapt to group-based scope, it simply records and patches yieldpoint addresses by group. Conditional polling also fits well in group-based scope by using different conditional variables or different conditions per group. Trap-based polling does not work well with group-based scope, as each group would need its own protected page, and the memory consumption for a large number of protected pages can be significant.

Summary In this thesis, we evaluate global and thread-local versions of polling yieldpoints, i.e., $[$Global, Thread-Local$] \times [$Conditional, Trap-based Load, Trap-based Store$]$ as they are most relevant to global run-time synchronization events such as garbage collection. We also include the cost of the the fast-path of code patching yieldpoints in our evaluation, which is several bytes of noop.

4.4 Evaluation

We present our methodology, and report the performance of each of the yieldpoint designs. We start by evaluating the overhead of the common *untaken* case of each of the yieldpoints. Next we evaluate the yieldpoints when they are taken with normal frequency. Finally, we measure the time-to-yield (latency) for the different yieldpoints.

4.4.1 Methodology

In this subsection, we present the software, hardware and measurement methodologies we use. We base our methodology on similar work introduced by Yang et al. [2012], adapting it to the task of measuring yieldpoints. The principal methodological contribution of this chapter is an **omitted yieldpoint** methodology, which allows us to use a system with *no injected yieldpoints* as a baseline. We describe the omitted yieldpoint methodology below.

Measurement Methodology We implement all yieldpoints in version 3.13 of Jikes RVM [Alpern et al., 1999], with a production configuration that uses a stop-the-world generational Immix [Blackburn and McKinley, 2008] collector. We hold heap size constant for each benchmark, but because our focus is not the performance of the garbage collector itself, we use a generous 6× minimal heap size for each benchmark with a fixed 32 MB nursery.

We use Jikes RVM's *warmup replay* methodology to remove the non-determinism from the adaptive optimization system. Note that the use of replay compilation has the important benefit of obviating the need for the adaptive optimization system to perform profiling, which would otherwise make our *omitted yieldpoints* methodology impossible. Before running any experiment, we first gather compiler optimization profiles from the best performance run from a set of runs for each benchmark. Then, when we run the experiments, every benchmark first goes through a complete run to warm up the runtime (allowing all the classloading and method resolving work to be

done), and then the compiler uses the pre-collected optimization profiles to compile benchmarks and disallows further recompilation. This methodology greatly reduces non-determinism from the adaptive optimizing compiler. Note that we use the replay advice from the status quo build. However, since our different builds impose little change in the run-time system, we expect the bias introduced by using the same advice to be minimal as well.

Omitted Yieldpoint Methodology To evaluate the overhead of various yieldpoint implementations, we developed a methodology with *no* injected yieldpoints, which served as a baseline against which each of the yieldpoint implementations could be compared. The methodology depends on two insights. First, we can disable two of the three systems that depend on yieldpoints: sampling for feedback-directed optimization, and lock revocation for biased locking. As mentioned above, the warmup replay methodology provides a sound basis for empirical analysis such as this, and happens to have the side effect of not requiring sampling for FDO. Biased locking is an optimization that we can readily disable, removing the need for lock revocation at the cost of modest performance losses on some multi-threaded benchmarks. Second, *explicit* yieldpoints remain in place, even when we disable *injected* yieldpoints. Empirically, explicit yieldpoints are sufficiently frequent that garbage collection — the one remaining component dependent on yieldpoints — can occur in a timely manner. We quantify the slop that removal of injected yieldpoints adds to reaching explicit GC-safe points by measuring the time taken for threads to yield and comparing it with total mutator time. The average effect on mutator time due to slower latency to reach explicit GC-safe points is 0.9 %, which is mostly due to one benchmark which triggers GC very frequently (`lusearch`, 9.1 %). For a fair comparison, in our measurements, injected yieldpoints have an empty slow path and will not bring the thread to a GC-safe point so that GC always relies on explicit yieldpoints, and the slightly longer GC-safe point latency persists for all the experiments. The obvious alternative to our approach would be to remove the need for garbage collection altogether by using a sufficiently large heap. However, this would be impractical for benchmarks such as `lusearch` which allocate prolifically, and would measurably degrade benchmark locality [Huang et al., 2004].

Hardware and Software Environment Our principal experiments are conducted on a 22 nm Intel Core i7 4770 processor (Haswell, 3.4 GHz) with 8 GB of 1600 MHz DDR3 RAM. To evaluate the impact of microarchitecture, we also use a 32 nm Intel 2600 Core i7 2600 processor (Sandy Bridge, 3.4 GHz) with 8 GB of 1333 MHz DDR3 RAM. Both processors have 4 cores with 8 SMT threads. Aside from the difference in Haswell and Sandy Bridge microarchitectures and memory speeds, the machines are extremely similar in their specifications and configuration. We use Ubuntu 14.04.1 LTS server distribution running a 64 bit (x86_64) 3.13.0-32 Linux kernel on both machines.

Benchmarks We draw the benchmarks from the DaCapo suite [Blackburn et al., 2006], the SPECjvm98 suite [SPECjvm98], and pjbb2005 [Blackburn et al.] (a fixed workload version of SPECjbb2005 [SPECjbb2000] with 8 warehouses that executes 10 000 transactions per warehouse). We use benchmarks from both 2006-10-MR2 and 9.12 Bach releases of DaCapo to enlarge our suite and because a few 9.12 benchmarks do not execute on Jikes RVM. We exclude `eclipse` from the suite since our thread-local trap-based implementation requires larger space for thread-local storage, which makes `eclipse` run out of metadata space under the default configuration.

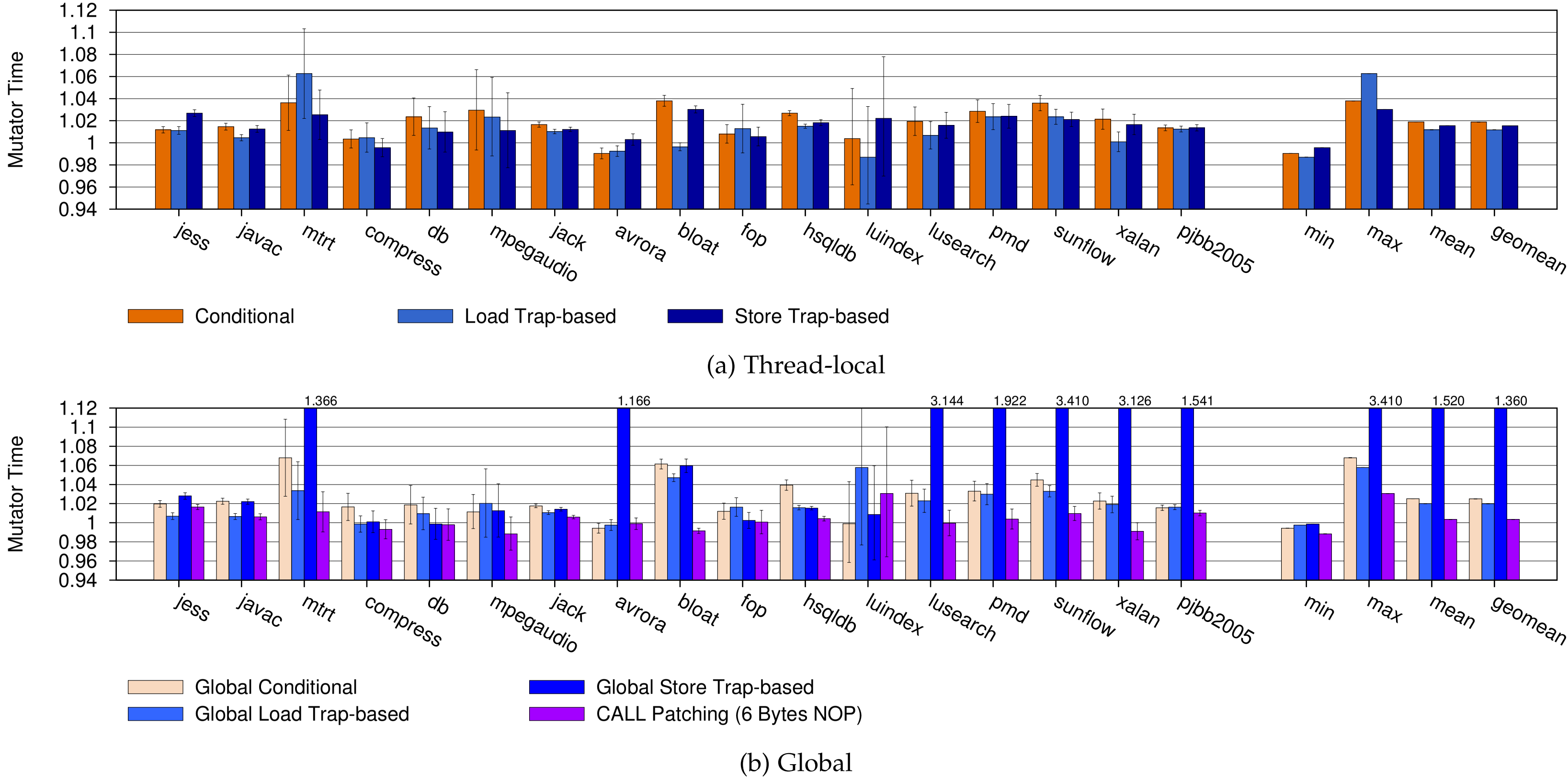
4.4.2 Overhead of Untaken Yieldpoints

We use the *omitted yieldpoint methodology* of Section 4.4.1 to measure the impact of each yieldpoint design on mutator performance in the case where the yieldpoint is *never* taken. This reflects the common case, since as Section 4.2.2 showed, only about 1/20 000 yieldpoints are actually taken. We first evaluate the overheads for thread-local yieldpoints before considering global yieldpoint designs.

Thread-Local Yieldpoints Figure 4.3 shows the code for three thread-local yieldpoint designs. Figure 4.4(a) shows the overheads on the Haswell microarchitecture. The geometric mean overheads are 1.9 % for the conditional, 1.2 % for the load trap, 1.5 % for the store trap.

Our evaluation on the Sandy Bridge hardware reveals some interesting differences between microarchitectures. The geometric means for Sandy Bridge are 2.3 % for the conditional, 2.1 % for the load trap, 1.6 % for the store trap. Thus the conditional and trap-based yieldpoints are noticeably higher and more homogeneous on the older machine. Interestingly, the load trap yieldpoint is significantly lower on the new machine, from 2.1 % down to 1.2 %, while the improvements brought by the newer micro-architecture on other implementations are marginal. This result highlights the sensitivity of these mechanisms to the underlying micro-architecture, and the consequent need to re-evaluate and rethink such designs in contemporary settings.

Global Yieldpoints Global yieldpoints are very similar to the thread-local yieldpoints shown in Figure 4.3, only rather than referring to thread-local storage (via the `TLS_REG` in Figure 4.3), they refer to a single global value for the conditional yieldpoint and a single global guard page for the trap yieldpoint. Figure 4.4(b) shows the overheads for the global yieldpoints on the Haswell microarchitecture. The geometric mean overheads are 2.5 % for the conditional, 2.0 % for the load trap, 36 % for the store trap! Each of these are higher than their thread-local counterpart. The difference between the local and global yieldpoints is moderate for the conditional and the load trap (respectively 0.6 % and 0.8 %). But for the store trap, the slowdown is extreme. The reason is obvious. It is clear from Figure 4.4(b) that all multi-threaded benchmarks account for much of the increase in store trap overhead. This is due to write contention on the guard page caused by multiple user threads trying to write to the



(a) Thread-local

(b) Global

Figure 4.4: Mutator overhead of *untaken* thread-local and global yieldpoints on the Haswell microarchitecture. The graph shows times normalized to the *no yieldpoint* baseline. The geometric mean overheads for the thread-local yieldpoints are 1.9 % for the conditional, 1.2 % for the load trap, 1.5 % for the store trap. The global yieldpoints suffer systematically higher overheads due to cache contention while the six-byte noop as code patching fast-path imposes minimal overheads (0.3 %).

same cache line. These results make it clear that aside from the additional flexibility offered by thread-local yieldpoints, they also offer substantially lower overheads.

We also measured the six-byte noop overhead, which acts as the fast-path of one implementation of code patching yieldpoint. The noops can be patched into an absolute call instruction on demand. The six-byte noop has the least overhead among all the yieldpoints we measured, 0.3 % on Haswell, and zero measurable overhead on Sandy Bridge. This only reflects the common case fast-path, it does not include the cost of performing code patching.

These results indicate a number of interesting findings. First, the conditional yieldpoint does have a reasonably low overhead but nonetheless is the worst performing among untaken thread-local yieldpoints. Second, the overhead of the code patching yieldpoint in the untaken case is (perhaps unsurprisingly) very low.

4.4.3 The Overhead of Taken Yieldpoints

In the previous section we looked at the overheads due to yieldpoints when they are *never* taken. In practice, of course, yieldpoints are taken, even if rarely. We now extend the same methodology as above, only that we allow yieldpoints to be triggered normally by the underlying profiling system that dominates yieldpoint activity (Figure 4.1). However, we implement an *empty* slow path activity: when the yieldpoint takes its slow path, we simply turn off the yieldpoint and return to the mutator rather than actually undertake profiling or any other task. Notice that in the case of the conditional yieldpoint, this means that there is very little additional overhead, whereas in the trap-based yieldpoints, the trap must still be taken and serviced before returning.

Figure 4.5 shows the results for the Haswell micro-architecture. The geometric mean overheads for the yieldpoints are 1.9 % for the conditional, 2.7 % for the load trap, 3.0 % for the store trap. Notice that the total overheads are now dramatically evened out compared to the *untaken* results seen in Figure 4.4. The conditional has non-measurable extra overhead for taking yieldpoints. However, though clearly faster than conditional yieldpoints in untaken cases, trap-based yieldpoints are now slower due to the overhead associated with servicing the traps. These results undermine the advantage of load and store trap-based yieldpoints when yieldpoints are required to be taken frequently.

4.4.4 Time-To-Yield Latency for GC

A third performance dimension for yieldpoint implementations is the time it takes for *all* mutator threads to reach a yieldpoint. This is of course dominated by the *last* thread to come to a yieldpoint. Intuitively, a trap-based yieldpoint will perform worse on this metric than a conditional polling yieldpoint because it is subject to the vagaries of the operating system servicing the trap and any scheduling perturbations that may

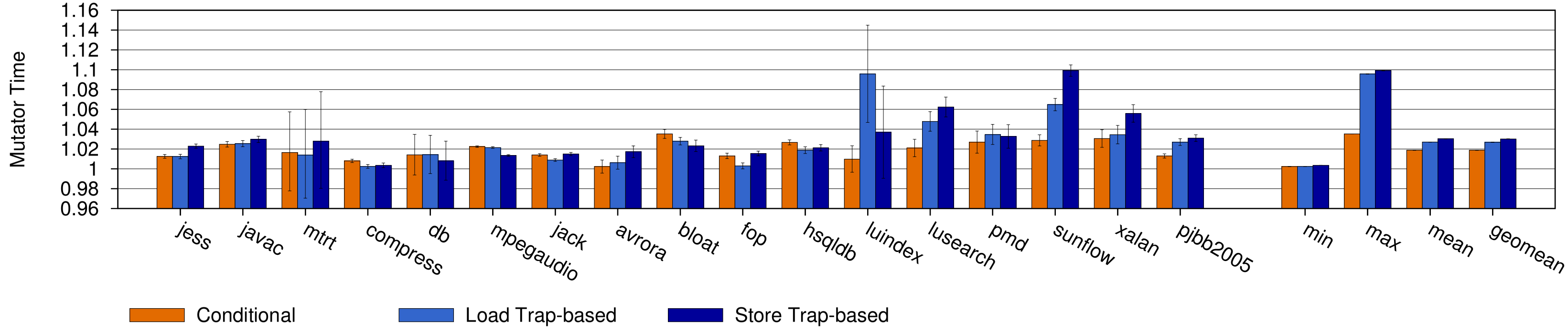
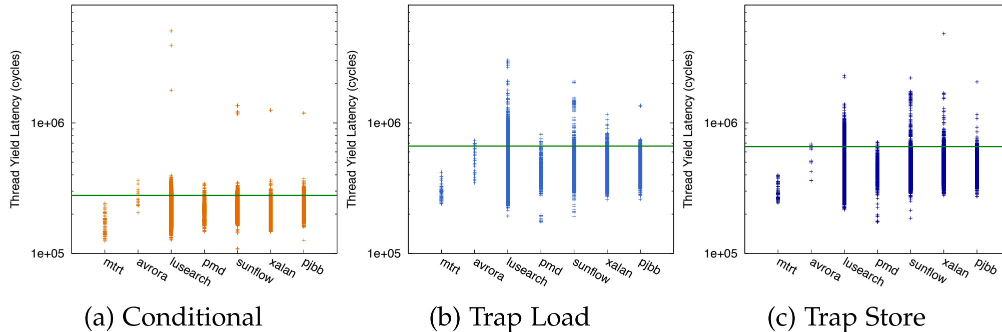


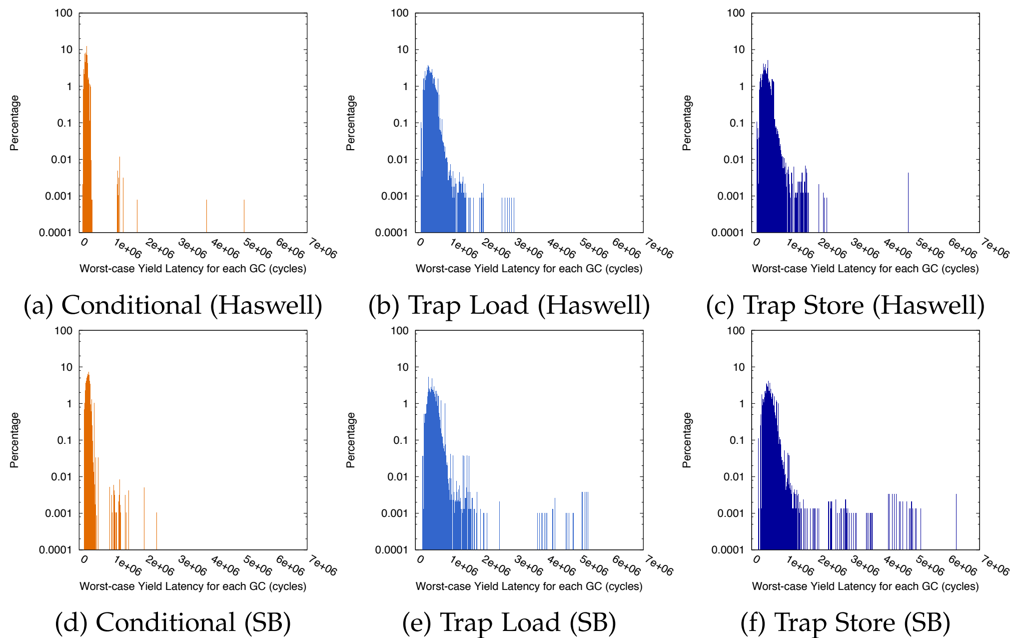
Figure 4.5: Mutator overhead of *sometimes taken* yieldpoints (thread-local) on the Haswell microarchitecture. The graph shows times normalized to the *no yieldpoints* baseline. The geometric mean overheads for the yieldpoints are 1.9 % for the conditional, 2.7 % for the load trap, 3.0 % for the store trap.

induce. We measure the time-to-yield latency for each GC by using thread-local polling yieldpoints with multi-threaded benchmarks.[3]



(a) Conditional (b) Trap Load (c) Trap Store

Figure 4.6: Time-to-yield latency for polling yieldpoints, measured in cycles (log-scale y-axis), for each of the multi-threaded benchmarks.



(a) Conditional (Haswell) (b) Trap Load (Haswell) (c) Trap Store (Haswell)

(d) Conditional (SB) (e) Trap Load (SB) (f) Trap Store (SB)

Figure 4.7: Time-to-yield worst-case latency distribution for each GC. The conditional yieldpoint has a much tighter distribution, and the newer Haswell microarchitecture produces tighter distributions than its older Sandy Bridge counterpart.

Figure 4.6 shows the thread yield latency (in cycles) for each GC. Every point in the figure shows the latency from when the collector initiates the yield to when *each* thread reaches a yieldpoint. The horizontal line indicates the 95th percentile among the data points (278 k cycles, 665 k cycles and 659 k cycles, respectively, for the three implementations on Haswell). Conditional polling has a substantially lower average

[3] For single-threaded benchmarks the only application thread yields immediately on a failed allocation, so latency is not affected by the particular yieldpoint implementation, so we exclude the single-threaded benchmarks.

time-to-yield, but is also more tightly grouped. This is unsurprising, since trap-based implementations require a system call to protect the polling page, and require signal handling to take yieldpoints, while conditional polling involves a simple change on the value of the polling flag, and a call.

Figure 4.7 shows the distribution of *worst-case* thread yield latency across all of our multi-threaded benchmarks. Worst-case thread yield latency is the time from when the collector initiates the yield to when the *last* thread reaches a yieldpoint. We can see that on both machines, conditional polling has a much tighter distribution and lower latency. We examined the worst (rightmost) results in each scenario and found that the majority are from two benchmarks `sunflow` and `lusearch`. 15 out of the worst 30 results are from `sunflow` and 10 out of 30 are from `lusearch`. For the best-case time-to-yield latency (i.e., the fastest time from GC initiation to all threads yielding) there is a clear distinction between conditional and trap-based polling. On Haswell, conditional polling has the lowest yield latency of 109 k cycles while trap-based polling is 174 k cycles for both load and store.

From these measurements, we see that conditional polling yieldpoints have a markedly better time-to-yield latency than trap-based yieldpoints on average and at the 95th percentile. However, the worst-case time-to-yield latency is not well correlated with yieldpoint implementation, but rather affected by the operating system and the benchmarks.

4.5 Future Work: Code Patching As An Optimization

We note that the unconditional code patching yieldpoint presents a severe tradeoff. The common case cost of a noop-patched yieldpoint is very close to zero. However, we measured the cost of patching and found that when patching is performed at every timer tick, it adds on average 13.4 % overhead when all yieldpoints are patched. We then measured the effect as we reduced the number of yieldpoints patched, and found that it fell to 0.6 % when 681 ($\sim$99.9 % in Figure 4.2) are patched and just 0.3 % when 315 ($\sim$99 %) are patched. This observation led us to consider code patching as a possible optimization over conditional or trap-based yieldpoints.

When used as an optimization, code patching selectively overwrites only the most frequently executed yieldpoints with no-ops. When a yieldpoint is triggered, the optimized yieldpoints are rewritten to their original state (or to unconditional yields). Once the yield is complete, the most frequently executed yieldpoints are once again elided. The choice of which yieldpoints to optimize will depend on the cost-benefit tradeoff between the patching cost and the cost of executing the unoptimized yieldpoint. If the 300 or so most heavily executed yieldpoints could be successfully identified and patched, it seems possible that the optimization would be almost entirely effective and yet introduce only a tiny overhead due to patching. Possible refinements to this optimization include parallelizing the code patching (also applicable to the code patching yieldpoint), aborting patching if the yield succeeds before all are patched, and ordering the patching so that the most frequently executed

yieldpoints are patched first.

4.6 Summary

Yieldpoints are the principal mechanism for thread synchronization in a virtual machine to determine when a thread must yield. In this chapter, we have identified and evaluated a range of yieldpoint mechanisms. We find that the trade-off between common-case fast path execution and overheads in the uncommon case can be severe. While an unconditional trap-based poll has low overhead in the common case, it is costly when the yield occurs, resulting in slightly worse performance than a simple conditional test, on average. An unconditional code patching yieldpoint presents an even move extreme trade-off, with near zero common case overhead but substantial patching overheads at every yield (as discussed in Section 4.5). We highlight the micro-architectural sensitivity of these mechanisms, indicating the need for virtual machine implementers to reassess their performance assumptions periodically. We also identify that code patching presents an interesting opportunity for an optimization, replacing a few of the most frequently executed yieldpoints with no-ops at times when yields are not required.

The study in this chapter allows us the confidence to pick suitable mechanisms for implementing thread synchronization for Zebu VM, which is one major part of the concurrency requirement for a micro virtual machine. In this chapter, we concluded that conditional polling has the most desirable characteristics for Zebu VM as GC yieldpoints: low overhead, fast time-to-yield, and implementation simplicity. Furthermore, we studied group-based yieldpoints, which are widely used in Zebu VM as *watchpoints*. We will further discuss our implementation of watchpoints in the next chapter (Chapter 5, Section 5.3.3).

In the next chapter, we will discuss our compiler design, with a focus on meeting the requirements of a micro virtual machine, i.e., minimalism, efficiency and flexibility.

Chapter 5

A Micro Compiler

Micro virtual machines promise to simplify managed language implementations but they depend critically upon an efficient execution engine. Such an execution engine faces interesting constraints, including efficiency, language-neutrality, minimality, support for features such as exact garbage collection, on-stack replacement, swapstack, introspection, dynamic code patching, and capability of both just-in-time and ahead-of-time compilation. Building a compiler under these constraints is challenging and interesting.

This chapter presents our implementation of a compiler for the Zebu micro virtual machine. Section 5.1 motivates our compiler design, and discusses constraints imposed by the micro virtual machine upon the compiler's design and the rich run-time behaviours that the compiler must support. Section 5.2 discusses related work for this chapter. Section 5.3 presents solutions addressing some interesting design points. In Section 5.4, we evaluate the performance of our compiler from two angles: (*i*) a full-stack scenario – we retarget PyPy's implementation language RPython to Mu [Zhang, 2015] to evaluate a real world managed language running on Zebu VM, in comparison with its stock back-end (C), and (*ii*) a back-end specific scenario – we translate highly optimized LLVM IR into equivalent Mu IR and evaluate our performance against the LLVM back-end. We show that the results for both are promising. They suggest that micro virtual machines not only simplify managed language implementations, but that they can also be implemented efficiently.

5.1 Introduction

Compiler design is a key factor to the performance of a language virtual machine. While the efficiency of the generated code is fundamental to a compiler's design, there are usually other constraints, for example, real-time credibility, power efficiency, compilation time, memory footprint, etc. For the Mu micro virtual machine and our implementation, Zebu VM, we face quite different but interesting constraints. We expect an optimizing compiler that allows decent code quality and easy integration with the runtime in both ahead-of-time and just-in-time compilation mode. We also expect the design of the compiler to fulfil the design principles of Mu: minimalism, efficiency, and flexibility. We elaborate on each point, and discuss how they affect our

design.

Minimalism. Minimalism is the most important concern for our design, as it defines a *micro* virtual machine. We make minimalism a must throughout our design. Minimalism leads to some desirable properties, such as faster compilation time, smaller memory footprint, and the feasibility of being formally verifiable. For minimalism, our compiler explicitly avoids the inclusion of traditional front-end optimizations that can be done by the client above Zebu VM, such as common sub-expression elimination and constant propagation. Further, we argue that some back-end optimizations such as software pipelining can be omitted because modern hardware makes them less effective. In Section 5.3.1, we will discuss our choices of compiler optimizations in detail.

Efficiency. Efficiency is often at odds with minimalism, especially for a compiler. The more effort we spend optimizing, the more efficient the generated code may be. When optimizations are missing, the code quality may suffer. However, there are *diminishing returns* for back-end optimizations: a small set of optimizations are the most effective ones while other optimizations provide limited improvement upon them [Lee et al., 2006]. For our compiler, it is not our aim to achieve the best performance possible. Instead, our goal is a sweet spot that represents a good balance between efficiency and minimalism before reaching diminishing returns.

Flexibility. Mu does not impose unnecessary restrictions on the client. Zebu must reflect this flexibility. We identify key mechanisms and reuse them to support this flexibility. For example, we implement a more general `SWAPSTACK` primitive, in comparison with the prior work [Dolan et al., 2013], and our compiler reuses the primitive to implement many of the Mu semantics, such as thread creation and destruction, and stack manipulation (re-ordering stacks and swapping stacks). Another example is the yieldpoint, as discussed in Chapter 4. Zebu use yieldpoints both for garbage collection and `WATCHPOINTS`.

In the following sections, we will present related work, discuss our design under these constraints, and evaluate our implementation.

5.2 Related Work

Prior work has discussed compiler designs under specific constraints, such as energy efficiency [Saputra et al., 2002; Lorenz et al., 2002; Haj-Yihia et al., 2015], real-time constraints [Hong and Gerber, 1993; Weil et al., 2000; AbouGhazaleh et al., 2006], embedded environments [Kerschbaumer et al., 2009] and resource-constrained environments [Shaylor, 2002; Debbabi et al., 2004; Gal et al., 2006]. KVM JIT [Shaylor, 2002] and Hotpath VM [Gal et al., 2006] are lightweight dynamic Java compilers, both translating only a subset of bytecodes with low memory consumption (60 and 150

kilobytes respectively), and showing speed-up against an interpreter. E-Bunny [Debbabi et al., 2004] is a non-optimizing Java compiler with one-pass stack-based code generation. It features multi-threading support for the VM, efficient compilation time, and low memory footprint, and shows performance improvement over KVM. Prior work prioritized resource constraints and consequently compromised performance. Micro virtual machines require different constraints compared to prior work, seeking balance among minimalism, efficiency and flexibility. Thus performance is one of our clear goals. This work proposes a compiler design that fulfils the principles of a micro virtual machine compiler.

Dolan et al. proposed *swapstack* as a primitive for lightweight context switching with compiler support. Swapstack unbinds a thread from one context and rebinds it to another context. They showed that their lightweight context switching mechanism can be implemented fully in user space with the support of the compiler in only a few instructions. This is impossible for library-based approaches, including `setjmp/longjmp`, `swapcontext` or customized assembly code, which have no information from a compiler and must conservatively save all registers. The design of Mu's thread sub-system is greatly influenced by this work. Mu not only pervasively uses swapstack, but also provides richer semantics, such as `KILL_OLD`[1] and `THROW_EXC`[2]. This imposes different implementation constraints compared to Dolan et al.

Bootstrapping a virtual machine (especially a meta-circular virtual machine) with a pre-compiled boot image is common [Alpern et al., 1999; Ungar et al., 2005; Blackburn et al., 2008; Wimmer et al., 2013]. The popular approach is to use a specific boot image writer to carefully turn live code and objects into a persistent image, and then load the image with a boot image runner. Our work is significantly different: we expect Zebu to be a part of the client implementation, and we do not intend to make any assumption on how the client is going to utilize the boot image; so instead, we generate the boot image as a standard position-independent binary (executable or dynamic library) for the target platform so that the client can run or link against the boot image in their preferred way.

5.3 Design

In the following subsections, we discuss some points of our compiler design that address the constraints. To be specific, we will discuss five design points: selecting optimizations, implementing a general swapstack mechanism, allowing efficient code dynamism with code patching, supporting client introspection for abstract execution state, and generating native and relocatable boot images.

---

[1] A swapstack instruction with `KILL_OLD` will kill the current stack after swapping.

[2] A swapstack instruction with `THROW_EXC` will raise the argument as an exception object with the target stack after swapping.

5.3.1 Optimizations

For Mu client languages, Mu IR is a dividing line that ensures only necessary information flows beyond it. It is a clean separation between the language and the hardware. The Mu implementation knows the underlying machine, but is not aware of the source language. This naturally separates compiler optimizations in the whole implementation stack into three groups: language-level optimizations, IR-level optimizations, and machine-level optimizations (Table 5.1).

We follow these principles to pick optimizations for Zebu. (*i*) For optimizations that are specific to the client language, Zebu does not have the knowledge to apply those optimizations effectively, and must not perform them due to our principles. (*ii*) For optimizations that can be done either by Zebu or the client, the client should be the one doing it, as the minimalism of Mu requires pushing work to the client unless impractical. (*iii*) For optimizations at the machine-level, they can only be done in Zebu, and Zebu must be responsible. If any machine specific optimization is missing in Zebu, the entire implementation will miss the opportunity and incur a corresponding performance penalty. It does not necessarily mean we have to include all machine level optimizations, but we need to be vigilant when ruling out any of them.

The most important optimizations performed by Zebu are instruction selection and register allocation. We match IR trees with defined patterns, and prioritize the first match. It is sufficient for our IR (e.g., Figure 5.1) and much more lightweight than weight-based bottom-up rewriting pattern matching [Fraser et al., 1992]. For register allocation, we currently adopt simple graph coloring that supports register classes and coalescing [George and Appel, 1996] with careful tuning to favor allocation and coalescing in inner loops. There are more advanced graph coloring algorithms [Smith et al., 2004; Cooper and Dasgupta, 2006], linear scan algorithms [Wimmer and Mössenböck, 2005; Wimmer and Franz, 2010], or other algorithms [Lueh et al., 2000; Greedy RA, 2011; Eisl et al., 2016], which are worthy of further consideration.

```
1    .typedef @point = struct<@int64, @int64>
2    ...
3    %iref_obj = GETIREF <@point> %ref_obj
4    %iref_y = GETFIELDIREF <@point> %iref_obj 1
5    %y = LOAD <@int64> %iref_y
```

(a) Common Mu pattern for field access.

```
1    mov [%ref_obj + 8] -> %y
```

(b) Resulting code after pattern matching.

Figure 5.1: Mu IR heavily relies on pattern matching.

Following our principles, we design the compiler pipeline around those key optimizations, focusing on machine-level optimizations with very few IR-level optimization (such as depth tree generation to get IR trees of deeper depth for better pattern

	Description	**Solution**	**Examples of optimizations**[a]
Language level	Language specific optimizations, which requires intimate knowledge of language-level semantics. It is impossible and counter-productive for Mu to do effective optimizations at this level [Castanos et al., 2012].	We expect the client to optimize before lowering the abstraction to Mu IR level (and losing the information).	array bound check elimination, null pointer check elimination[b], inline caching[c], call devirtualization[d], lock elision[e], type specialization[f]
IR level	Mu IR-to-IR transformations and optimizations. These include some very general compiler optimizations. However, in pursuit of minimalism, such optimizations are kept out of Mu.	The clients can implement this. Alternatively, a standalone optimizer library can be provided *alongside* Mu.	dead code elimination, constant folding, common sub-expression elimination, scalar replacement of aggregates, vectorization[g], code hoisting, loop unrolling
Machine level	Machine specific optimizations. Mu is solely responsible for these optimizations.	Zebu implements those (or chooses to miss the opportunities).	trace scheduling, instruction selection, register allocation, peephole optimization, instruction scheduling

Table 5.1: Compiler optimizations in a language implementation with Mu. The client and Mu are responsible for a different set of optimizations.

[a] Most of the optimizations mentioned in the table are discussed in Appel and Palsberg [2003].
[b] Kawahito et al. [2000]
[c] Hölzle et al. [1991]
[d] Ishizaki et al. [2000]
[e] Nakaike and Michael [2010]
[f] Gal et al. [2009]
[g] Nuzman and Henderson [2006]

matching). We avoided including common IR-level optimizations. Instead we plan to provide an optimizer library available to the client, but above Mu to aid the client in performing common optimizations over Mu IR without impinging on the minimality and verifiability of Mu.

Section 5.4 presents an evaluation of our compiler. It shows that (*i*) our choices for back-end optimizations in the Zebu compiler achieve promising performance, and (*ii*) though pushing common IR level optimizations to the client makes Zebu minimal, when the client is not cooperative in implementing those optimizations, it imposes overheads. Providing an optimizer library along with Zebu seems a necessity.

5.3.2 Swapstack

Dolan et al. proposed the idea of implementing swapstack as a special calling convention with compiler support to allow lightweight context switching. The work exploits compiler optimizations to reduce the cost of context switching by storing less state in paused contexts and by passing parameters between contexts in registers. It inspired the Mu's design of swapstack.

However, the requirements for `SWAPSTACK` in Mu are different and more general in a number of ways. (*i*) Besides a normal resumption when the invoking context passes arguments to the invoked context (`PASS_VALS`), the resumption can also be exceptional as the invoked context resumes with an exception thrown from the paused frame (`THROW_EXC`). (*ii*) `SWAPSTACK` supports the `KILL_OLD` semantics, i.e., the instruction not only swaps to a new stack, but also kills the old stack. We cannot kill the old stack before switching to the new stack. After switching to the new stack, we still cannot kill the old stack before preparing swapstack arguments, as the argument values may get spilled on the old stack. This implies that we cannot implement `SWAPSTACK` as simply as a special calling convention as in the prior work, because for us, there is not a single point where the transition between stacks happens. (*iii*) `TRAP` and `WATCHPOINT` provide the client the flexibility to choose the resumption (normally or exceptionally, on the original stack or on a different stack). Thus, at compile time, Zebu does not know the resumption point for `TRAP` and `WATCHPOINT`, and their implementations are naturally more general and heavyweight than `SWAPSTACK`. (*iv*) `NEWSTACK` takes an entry function as its argument, and allows execution with the function from its entry when the stack is swapped to or is started with a new thread. However, the function has to be compiled with the default Zebu calling convention (the same as the target ABI defines), as it can also be called normally. This implies that our swapstack calling convention needs to be compatible with the default Zebu calling convention, otherwise we would need an adaptor so those functions can be called normally. (*v*) Thread creation also implies a swapstack from its native stack (e.g., the `pthread` stack) to a designated Mu stack, and we will need to be able to swap back to the native stack for proper thread deconstruction.

For the requirements of swapstack in Mu, we proposed a more general swapstack implementation,[3] shown as pseudo code in Figure 5.2.

[3] This work is done in cooperation with Gariano, and is also described in [Gariano, 2017].

Stack State

We define two states of a Mu stack: the *active* state when a stack is an active context that is bound to a thread, and the *ready* state when a stack presents a paused context that is no longer bound to any thread and is ready for resumption. `prepare_stack` and `restore_stack` are two primitives that transform stacks between the two states, which we use for all swapstack cases. Figures 5.2(a) and 5.2(b) show the primitives, and Figure 5.2(c) shows the stack layout in the two states.

`prepare_stack` brings a stack from the active state to the ready state. `prepare_stack` first bumps the stack pointer to reserve a return area on stack which is used to pass extra parameters when the context gets resumed and some parameters cannot fit in registers. The size of the return area is determined at compile time by inspecting the resumption signature of the instruction (the size is zero if all the parameters are passed by registers). The macro then pushes the resumption address (the code to execute after resumption) and the frame pointer, and saves the current stack pointer with the current stack reference (`stackref`). All the necessary information to allow resumption can be retrieved from a `stackref`.

The other macro `restore_stack` does the opposite – bringing a paused context from the ready state to the active state. It unloads return values from the invoking context and restores the invoked context to the active state for resuming execution.

Calling Convention

Our compiler treats swapstack as a special calling convention in the same way as the prior work [Dolan et al., 2013] to exploit the existing register allocation pass to only save live registers as stored contexts. However, different from the prior work, we need to define two calling conventions, *hot resumption* and *cold resumption*, to meet Mu's requirement of a more general swapstack mechanism.

Hot resumption happens when both the invoking and the invoked context are executing Zebu code (runtime code or compiled user code), where we can safely assume argument passing by registers. This is the case for the `SWAPSTACK` instruction. Cold resumption takes place when either of the contexts is executing non-Zebu code. This happens when we swap from a native stack to a Mu stack for `NEWTHREAD`, or when we swap after returning from a `TRAP` to the client. We cannot assume argument passing by registers, as foreign code will not comply with our calling convention; as a result, we pass arguments on the stack. We use the default Zebu calling convention for hot resumption (the standard ABI for target platforms), with the exception that return values are passed in argument registers [Dolan et al., 2013]. Cold resumption is a compatible calling convention with no register arguments (and return values), and it requires extra instructions to save and later load arguments from the stack to argument registers. Using default calling conventions allows arguments from swapstack to naturally feed to a normal function prologue, thus resuming at a stack from a function entry does not need extra adapters to marshal arguments.

Resumption

For Mu swapstack instructions, at compile time, the compiler knows whether the instruction is intended to resume normally or exceptionally. For the two different cases, the compiler generates different code sequences, as shown in Figures 5.2(d) and 5.2(e).

The first part of the generated code is identical for both cases, which is the sequence to initiate a stack swapping. It first transforms the current stack to the ready state (`prepare_stack`), and prepares arguments for resumption. This step also marks the instruction to clobber all general registers, and this will cause live values to be spilled and saved to the current stack by the register allocator. Note that preparing resumption arguments may still need the old stack, as some values may reside on the stack. After preparing resumption arguments, general registers contain values for resumption, and should be preserved until the execution reaches the resumption address. At this point, the old stack is paused in a ready state, live values are spilled, and resumption arguments are loaded — we are ready to switch to the new stack. Stack switching is simply loading the stack pointer from the target `stackref`, and now we are on a new stack in the ready state. If the instruction requires the `KILL_OLD` semantics, the compiler will insert a runtime call here with a safe calling convention that will not clobber argument registers to kill the old stack, and the compiler omits the sequence for `prepare_stack` as the old stack will not be resumed.

For normal resumption, as the new stack is in ready state, we simply pop the frame pointer and the resumption address, and jump to the resumption address where the `restore_stack` sequence awaits. `restore_stack` will unload return values if appropriate, and collapse the return area to make the stack back to the active state. For exceptional resumption, we need to tidy up the stack layout so that we can throw an exception as if there were a `THROW` instruction.

At the end, for both cases, the compiler emits a resumption point with `restore_stack` so that others can resume execution on the old stack that we just put it to pause.

Mu Instructions with Swapstack

The following discusses how we use our swapstack mechanism to implement related Mu instructions.

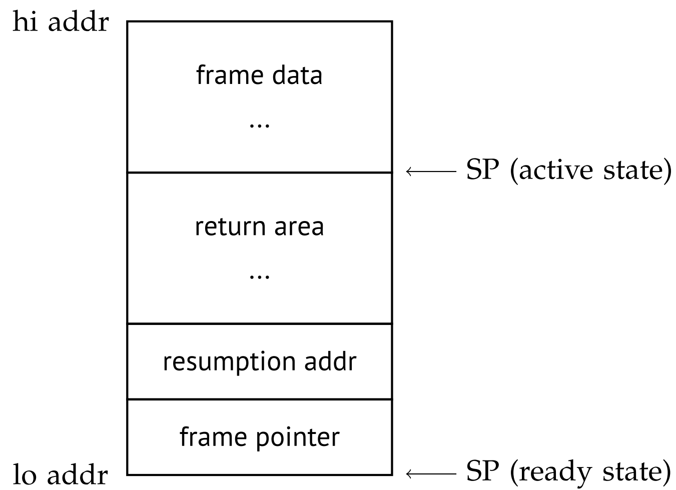
SWAPSTACK. `SWAPSTACK` always uses the hot resumption calling convention. The compiler emits code for swapstack, which is the most lightweight swapstack in Zebu. The swapstack implementation in Dolan et al. [2013] has the same semantics as our `SWAPSTACK` with normal resumption and no `KILL_OLD`, for which, our implementation is as lightweight as the prior work (See Figure 5.2(d)). However, our `SWAPSTACK` is more flexible. When `KILL_OLD` is supplied, we need to kill the old stack after we have prepared the arguments and have left the old stack. If the old stack's destruction needs a run-time call, a special safe calling convention is required to avoid clobbering argument registers of the hot resumption calling convention. In Mu, `SWAPSTACK`

```
.macro prepare_stack
  reserve return area on stack
  push resumption address
  push frame pointer
  store SP to current stackref
```

(a) Prepare stack to ready state.

```
.macro restore_stack
resumption:
  unload return values
  collapse return area
```

(b) Resume stack normally.



(c) Stack layout for ready and active state.

Figure 5.2: Swapstack implementation in Zebu.

```
1    prepare_stack (if NOT KILL_OLD)
2    prepare arguments for resumption
3    load SP from new stackref
4    kill old stack (safe cc) (if KILL_OLD)
5
6    // prepare resumption (running on the new stack)
7    pop frame pointer
8    pop resumption address
9    jmp resumption address
10
11   // resumption point (when this stack gets resumed)
12   restore_stack
```

(d) Swapstack normal resumption.

```
1    prepare_stack (if NOT KILL_OLD)
2    prepare arguments for resumption
3    load SP from new stackref
4    kill old stack (safe cc) (if KILL_OLD)
5
6    // prepare resumption (running on the new stack)
7    prepare stack for unwinding
8    throw exception
9
10   // resumption point (when this stack gets resumed)
11   restore_stack
```

(e) Swapstack exceptional resumption.

Figure 5.2: Swapstack implementation in Zebu (continued).

supports resumption at the target stack by throwing and propagating an exception. Figure 5.2(e) presents our implementation.

TRAP and WATCHPOINT. When a trap to the client is required (`TRAP` or enabled `WATCHPOINT`), cold resumption is needed to swap to a Mu stack after returning from client code. The Zebu runtime needs to prepare stacks for resumption as the client asks, and does the swapstack in the runtime library code.

NEWSTACK and NEWTHREAD. We use cold resumption for both, as we are swapping to a Mu stack from a native stack. This is also done in the runtime library code.

5.3.3 Code patching

Allowing dynamism in the generated code during execution is essential to efficient dynamic language implementation, which is one of Mu's goals. Mu provides watchpoints and function redefinition to support code dynamism. These primitives give clients the flexibility to efficiently implement optimizations, such as inline caching and guarded specialization.

Both watchpoints and function redefinition can be implemented naively, for example, implementing `WPBRANCH` as a conditional branch, or implementing any Mu call as an indirect call to support function redefinition. However, this defeats the very purpose of providing these primitives to support efficient code dynamism. Zebu implements these features with code patching. When the compiler emits code, it generates patchpoint tables for callsites, and for `WPBRANCH`. When a function is redefined, or a watchpoint is enabled/disabled, Zebu will query related patchpoints, and eagerly patch call/jump offsets. Zebu also implements lazy function resolution based on the same patching mechanism.

As Mu only requires patching offsets, the implementation is straightforward. We use the following two mechanisms to ensure that our patching is correct and matches Mu's semantics.

`INT3`-based Patching. We use `INT3`-based code patching on x86 to ensure that the instruction being patched will not be executed in an incoherent state. The first byte of the instruction will be patched as a one-byte instruction `INT3`, and then the remaining bytes will be patched one by one. During patching, in case the instruction gets executed, the `INT3` will prevent execution for the rest of the instruction, and trap to Zebu's signal handler in which the instruction will be re-executed. When the patching is done, the first byte is re-rewritten to the valid first byte for the instruction. This is a well-known technique to patch multi-byte instructions atomically [INT3 PATCHING, 2013].

Global Guard Page. Mu allows a group of watchpoints with the same ID to be enabled and disabled atomically. For example, when a watchpoint is enabled,

we expect all the `WPBRANCH` instructions associated with the watchpoint to have a consistent behaviour of branching to the enabled destination. However, patching the instructions takes time, and is not atomic (as discussed in Section 4.5), which leaves a possibility that patched instructions branch to the new destination (the enabled destination) while unpatched ones still branch to the old destination. To avoid this inconsistency, we insert a memory load instruction to access a per-watchpoint guard page before watchpoint instructions as a barrier. When the group of watchpoints is enabled or disabled, before patching the watchpoint instructions, the guard page is protected so the load prevents the watchpoints being executed while being patched. Only after the patching is finished, the guard page is unprotected and the execution of watchpoints is again allowed. In this way, the watchpoints' behaviour *appears* to be changed globally atomically.

5.3.4 Keepalives and introspection

Mu provides introspection of execution states at designated points with `keepalive` and `Framecursor`. Allowing arbitrary instruction and SSA variable introspection would overly limit the compiler and its ability to optimize code. For this reason, Mu limits introspection to `keepalive` variables.

The implementation naturally falls out in Zebu. The Zebu compiler tags `keepalive` variables as an extra use of the given variables at those instructions. The extra use will prevent the compiler from optimizing away the variable (such as the intermediate values in Figure 5.1). Thus a kept-alive variable is always held in an explicit location (in a register or on the stack). The extra use may keep the variable live longer and result in register pressure. As a result, kept-alive values have longer lifetime, and are more prone to be spilled. Thus using `keepalives` has a potential performance overhead, and the client needs to be aware of this and only use it when necessary.

Zebu builds a stack map and a callsite table to support stack unwinding for both zero-cost exception handling and stack introspection with `keepalives`. As `keepalive` appears only in call-like instructions (which have corresponding entries in the callsite table for exception handling to support unwinding, such as exception destination, stack argument size and callee saved register locations), we augment the entries in the callsite table to also include keepalive variables. The register allocator cooperates to dump a register assignment map for compiled functions. As a result, we are able to resolve from client-supplied SSA variables to physical registers, and then to stack locations that store the registers.

5.3.5 Boot image

Managed languages are often criticized for having long start-up times to initialize the runtime and load standard libraries. Boot images are commonly used by managed runtimes to capture ahead-of-time generated code and virtual machine state to expedite start-up. Mu supports boot image generation at the client's request. Zebu builds *native* and *relocatable* boot images in the standard binary format (as executables

or libraries) for the target platform/OS. Zebu ensures that the boot image persists the states at the point when boot image generation is requested. The boot image we produce includes three persisted dimensions from a running Zebu instance: code, reachable heap objects and a `VM` instance.

Code. The Zebu compiler supports different code emitters for a single target. We implement a binary back-end for the JIT, and an assembly back-end for boot image generation. When genearting the boot image, Zebu invokes the compiler in ahead-of-time (AOT) mode to compile *white listed* functions (functions that are required at boot time need to be compiled into the boot image). The resulting assembly files will later be linked as a part of the boot image. We do not allow run-time modification of AOT generated code after booting.

Heap. Zebu traverses its heap for boot image generation in the same way as it does for garbage collection. The traversal returns a transitive closure of live objects in the heap along with a map to help relocate references. For native pointers in our heap, we rely on the client to supply a relocation map, as Zebu is oblivious of native pointers and the native heap. The assembly back-end then cooperates to dump the objects (and their metadata) as relocatable data. The persisted heap is considered to be a non-moving, non-reclaimable immortal space, which is a part of the new heap upon booting. The boot image objects can be modified by the running program and traced by the garbage collector.

VM. Zebu allows the client to query (or modify if possible) declared Mu entities such as types, values, functions and global cells. For example, a client may want to declare common types, and create a boot image that contains those so that later it can use the types without declaring them again. Zebu, written in Rust, naturally stores those entities as Rust objects in Rust data structures (such as `HashMaps`). A Zebu `VM` instance contains all the information about a running VM. We persist the entities that the client is interested in by partially persisting the `VM` instance in the boot image. Thus when the VM instance is loaded from the boot image, it contains the same data as when we dump it. Note that we only persist data that is client-supplied (such as declared types) or necessary for us to execute the code (such as stack maps), and we do not dump the whole stateful VM. Run-time tables such as stack maps contain addresses and need to be preserved in a relocatable way. Running threads and stacks are not preserved. We implement a tool called *Rust Object Dumper And Loader (rodal)*,[4] and use it to generate a `Dump` trait implementation for all the Rust types we want to persist. Rodal dumps the objects as defined in their corresponding `Dump` trait, and generates assembly similar to a persisted Zebu heap. When the `VM` is loaded from the boot image, those are normal Rust objects that can be mutated and destroyed.

[4]In early development, we dump Rust objects as text using the Rust serialization interface. Gariano implemented rodal to allow dumping Rust objects in binary to significantly speed up the start-up time.

5.4 Evaluation

In this section, we present our performance evaluation. We first evaluate a retargeted RPython implementation that runs on top of Zebu with a set of micro benchmarks (including a SOM language interpreter). This evaluates how Zebu performs with a retargeted real-world client. Then we pick two micro benchmarks from our suite, acquire highly optimized Mu IR for the micro benchmarks, and evaluate their performance on Zebu. This evaluates the performance of the Zebu compiler in isolation, independently of any Mu client. We now discuss both in detail.

5.4.1 RPython-Mu

Although the design and implementation of a client language on top of Mu is not within the scope of this thesis, it is hard to evaluate Zebu effectively without a concrete language client that is robust and performant enough to run some reasonably complex workloads. We use the RPython-Mu client [Zhang, 2015], which is the initial step of our PyPy-Mu approach as discussed in Section 2.4.1.

Methodology

We elaborate on our methodology before presenting the results.

Measurement RPython is a restricted subset of Python as a statically typed managed language that is ahead-of-time compiled. RPython originally targeted C as its back-end. We retarget it to Mu, and run it on top of Zebu. We disabled garbage collection for our measurement of the compiler, and use sufficient heap. We measure the time spent executing each benchmark kernel (excluding start-up time). We run 10 invocations for each benchmark, and report the average with 95-percentile confidence intervals. We compare the performance of three builds:

(*i*) **RPython-C.** The stock implementation that uses C as the back-end. The generated C source code is further compiled by Clang at `O3` optimization level. We use this as our comparison baseline.

(*ii*) **RPython-C (no back-end opt).** We turn the back-end compilation for C into two steps: we use Clang to apply `O3` optimizations at the front-end to generate highly optimized LLVM IR, and then use `llc` at `O0` to generate the machine code with no back-end optimization. We do not intent to compare our performance with this build. But this allows us to tease apart front-end and back-end optimizations, and provides us with a lower bound baseline to understand the performance of RPython-Mu.

(*iii*) **RPython-Mu.** Our retargeted implementation. We must emphasize that though the design of Mu requires the client to apply front-end optimizations to be efficient, the retargeting work did not implement any optimizations specifically for Mu other than a simple peephole optimization that includes one

pattern. Though the retargeting still takes advantage of existing RPython optimizations, it leaves a large blank area of missing optimizations: some common IR-level optimizations by the stock C back-end are now missing, as Zebu is not designed to do optimizations at this level (as discussed in Section 5.3.1), and they are not implemented in the client.

Hardware and Software Our experiments are conducted on a 22 nm Intel Core i7 4770 processor (Haswell, 3.4 GHz) with 8 GB of 1600 MHz DDR3 RAM. We use Rust 1.21 (`3b72af97e 2017-10-09`) to compile Zebu. We use Clang 4.0 to assemble and link the assembly files generated by Zebu, and also use it to compile the C source code for the stock C back-end. We use Ubuntu 16.04.2 LTS server distribution running a 64 bit (x86_64) 3.13.0-88 Linux kernel.

Benchmarks We source the following micro benchmarks for the evaluation: (*i*) micro benchmarks that test language features, such as `alloc` and `except`, (*ii*) micro benchmarks that implement algorithms, such as `fib` and `quicksort`, (*iii*) benchmarks from the computer language benchmarks game[5] [Benchmarks Game] such as `btree`, `fannkuchredux`, `nbody`, `pidigits` and `spectralnorm`, and (*iv*) a SOM language interpreter running its standard test suite (`som`).

Results

Figure 5.3 shows the results of our measurement with the RPython client. The performance of RPython-Mu varies significantly from benchmark to benchmark, with a geometric mean of 14.1 % slower than the stock RPython-C implementation.

The results of RPython-Mu are co-influenced by the following factors: (*i*) The rich run-time semantics of Mu IR makes it a better target than C, allowing more efficient implementation in some cases. For example, Mu provides abstractions for exceptions and stacks, and internally Zebu can implement zero-cost exception handling and implicit stack overflow checking with guard pages, while the stock implementation can only implement them as explicit checks. (*ii*) The missing front-end optimizations hinder performance. Though the RPython compiler implements language-level optimizations, it makes an assumption that their stock back-end (C/LLVM) will be responsible for certain optimizations, for example, loop unrolling, loop strength reduction, vectorization, and load elimination. However, the design of Mu explicitly disclaims responsibility for those IR-level optimizations, and our retargeted client compiler did not implement the optimizations either. Thus, the performance is compromised at a certain level. Furthermore, lack of front-end optimizations will result in less efficiency in our back-end code generation as well, as we face higher register pressure and suboptimal instruction patterns. (*iii*) Our back-end intrinsically implements fewer optimizations than LLVM (as we aim for minimalism and do not intend to match LLVM). The effectiveness of our limited set of optimizations affects the

[5] We use the Python implementation of the benchmarks as our RPython benchmarks. Due to the difference of the two languages, only a subset can be run as RPython.

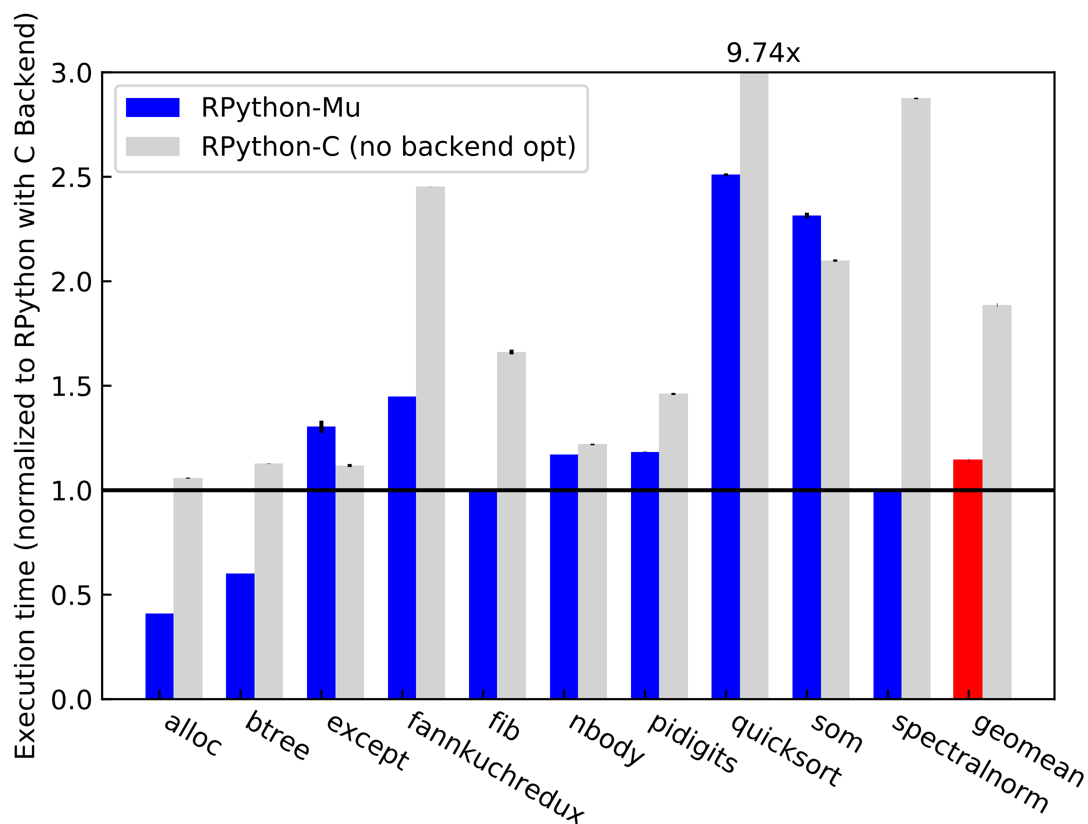


Figure 5.3: Performance of RPython-Mu (lower is better). The baseline and the RPython-C (no back-end opt) results illustrate a range of how back-end optimizations affect the overall performance for the C back-end. RPython-Mu results show our compiler performance.

performance, and this is what we try to demonstrate in our performance evaluation and will further demonstrate in the next section.

The performance of four benchmarks is dominated by the run-time system instead of code quality from the compiler. `alloc` and `btree` are allocation-intensive, for which allocator performance dominates. Section 3.3 has already established and demonstrated the efficiency of our allocator, and the results that we are faster than the stock implementation are within our expectation. `fib` is a recursive algorithm that is call-intensive. The stock implementation does a stack overflow check and a return value check in every invocation. We found that our zero-cost exception handling and implicit stack overflow checks compensate for the possible inefficiency from other sources and make our performance on par with the stock implementation. `except` explicitly measures the time for exception handling after an exception is thrown. Our implementation uses zero-cost exception handling, which favors the fast path performance where no exception is thrown but imposes higher overhead when an exception is thrown and we need to unwind and restore the stack. The stock

implementation always checks the return value for exceptional cases, which results in similar performance whether an exception is thrown or not.

For most of the benchmarks, our execution time is within 1.5× of the stock implementation, despite the fact that we clearly miss some effective front-end optimizations. For example, for `fannkuchredux` and `nbody`, the lack of auto vectorization makes us unable to utilize vector instructions on the target architecture. We also notice that, very frequently, LLVM is able to reduce the number of indexing variables for loops from multiple to one, which greatly helps register allocation in the loops. However, such an optimization is missing for the RPython-Mu client compiler. Despite these shortcomings, our compiler still delivers promising performance in comparison with `O0`-level RPython-C (no back-end optimization). Though we deliver reasonable performance for most of the benchmarks, two benchmarks are exceptional, `quicksort` and `som`, which are respectively 2.5× and 2.3× of execution time of the stock implementation. We discuss these in more detail now.

The behavior of `quicksort` is interesting. We implement the benchmark with a pre-initialized global array to sort. As the `quicksort()` and `partition()` functions always use this global array as argument, the RPython compiler specializes the functions to always load from the global on *every* use instead of simply referencing the parameter. This results in excessive loads from the global array for both the C back-end and the Mu back-end. The count of load instructions nearly matches the count of computation instructions in `partition()`, which becomes a clear bottleneck for performance. The C compiler (LLVM) can exploit the fact that the global array is non-volatile, hoist the redundant loads into one and remain efficient. However, for Mu, Zebu does not implement this optimization, and furthermore, this optimization is *not always* valid in Mu's semantics, as a global may be accessed and modified by other threads. We would expect the client to be aware of this semantic difference, and apply optimizations before handing code to Zebu.

`som` is the largest benchmark we include in our evaluation, featuring 3.4K lines of RPython code, which is a SOM language interpreter that executes its test suite. Surprisingly, RPython-Mu is slower than RPython-C with no back-end optimizations, unlike all other benchmarks (excluding `except` for which the difference in exception handling mechanisms determines performance). We believe this is for multiple reasons. (*i*) The interpreter loop uses a cascading **`if-else`** instead of **`switch-case`** which is much more efficient. The LLVM back-end for the stock implementation is able to identify the pattern, and turn it into a jump table look-up. However, for RPython-Mu, we lack this optimization. For each interpreter loop, we need to do a comparison and branch for every operator code, instead of a single lookup and branch. (*ii*) `som` is also a call-intensive benchmark. Over-specializing code across functions results in code that Zebu cannot optimize, similar to the problem with `quicksort`.

In summary, we argue that Zebu delivers reasonable performance in comparison with its stock implementation: on average 14.1 % slower, and mostly within 1.5× of execution time of the stock implementation. Further, we believe this conveys a clear message that RPython-Mu is missing optimizations Mu is not responsible for,

and some of the optimizations greatly affect performance. We argue that Zebu itself already includes sufficient optimizations to achieve a reasonable performance for back-end code generation. We will further discuss this in the next section.

5.4.2 Back-end Code Generation

In the previous section, we evaluated and discussed the performance of RPython-Mu, which involves both the RPython compiler and also the Zebu compiler. It gives us a good indication of how a retargeted language implementation performs on Zebu (despite the fact the retargeted implementation is not ideal as it misses lots of front-end optimizations that Zebu expects). In this subsection, we specifically evaluate back-end code generation of the Zebu compiler.

Methodology

In order to evaluate the back-end, we rule out the influence of a language client by providing equivalent and highly optimized IR to LLVM and Zebu. We achieve this by deriving optimized LLVM IR from C implementations of the benchmarks (with `O3` optimization), and then deriving Mu IR from the LLVM IR by a translation tool that does simple one-to-one mapping. We feed the highly optimized LLVM IR to the LLVM back-end, and feed the derived Mu IR to Zebu. In this way, the only factor that affects the performance is the back-end code generation, i.e., the Zebu compiler vs. LLVM back-end.

However, it is not always practical to derive equivalent Mu IR from LLVM IR, as the two IRs are different. For example, LLVM may generate platform-specific intrinsics in its IR that Mu does not know. Another example is that, LLVM's `getelementptr` is a super instruction that matches to different Mu instructions (`GETFIELDIREF`, `GETELEMENTIREF` and `SHIFTIREF`), based on their actual uses, which we find it hard to translate automatically (without further analysis and semantic information). In this subsection, we show evaluation on two micro benchmarks from the previous suite, `quicksort` and `spectralnorm`, for which we managed to derive equivalent Mu IR with limited manual alternation.

We use the same measurement methodology and the same hardware/software environment as the previous experiments (described in Section 5.4.1).

Results

Table 5.2 shows the average execution time for two micro benchmarks that are implemented as equivalent and highly optimized IR for each system. We also include the execution time of the same benchmarks written in RPython that we used in the evaluation in the previous subsection as a comparison. We found that in our evaluation, Zebu delivers reasonable performance in comparison with LLVM back-end code generation (2% and 25% slower), despite Zebu's minimalist implementation. We view this result as promising. The result also shows that RPython is an efficient

	LLVM	Zebu	RPython-C	RPython-Mu
`quicksort`	443 ± 1.8 ms	552 ± 3.5 ms (125%)	548 ± 21 ms (124%)	1375 ± 59 ms (310%)
`spectralnorm`	5620 ± 30 ms	5740 ± 47 ms (102%)	5671 ± 257 ms (101%)	5597 ± 45 ms (100%)

Table 5.2: Average execution time with 95% confidence interval for back-end code generation. We also include data from two RPython builds on the same benchmark for a comparison.

language in comparison with C (24% and 1% slower than C/LLVM in the two micro benchmarks).

It is worth mentioning that, for `spectralnorm`, Zebu matches the LLVM performance in terms of back-end code generation, and RPython-Mu also matches RPython-C. `spectralnorm` is a computation-intensive benchmark with nested loops. Our register allocator with heuristics that favor better allocation for inner loops helps us in this case. The RPython results demonstrate that the front-end opitimizations missing from RPython-Mu are not important here. We will focus on discussing `quicksort`, for which the four systems behave quite differently.

`quicksort` is one of the benchmarks in the RPython-Mu evaluation where we are much slower than the stock implementation (2.5× RPython-C). We have mentioned that the RPython compiler over-specializes code and yields code that is intrinsically inefficient for Zebu. With highly optimized IR, the performance difference drops from 2.5× down to 25%. By comparing the generated code, we find that the remaining slowdown is from the following sources: (*i*) The register allocation is different. As we have discussed in Section 2.2.7, our local SSI form breaks down long-lived variables into smaller live ranges. In practice, we found that though the shorter live ranges eliminate the two spilling occasions that happen for LLVM (Zebu does not spill registers for `quicksort()` with `partition()` inlined), it essentially introduced more register shuffling (especially near call instructions), i.e. eight more move instructions. We are uncertain about the performance implications of this difference. (*ii*) Zebu is not completely eliminating unnecessary branch instructions. We have a peephole optimization that eliminates instructions that branch to the next instruction, or branch to another branching instruction. Either of the two optimizations may create opportunities for the other, and it would be more effective if we alternate the two optimizations until the code is stable. However, we have not implemented this yet, and we noticed several jump instructions that could have been eliminated otherwise. (*iii*) Zebu is not fully exploiting x86_64 addressing modes as our depth-tree generation obscures some opportunities. Zebu turns sequential IR inputs into a depth tree, and finds tree patterns for addressing modes. However, our tree generation heuristics only make computation instructions whose result is used exactly once the child of another instruction node. If the result is used more than once, we compute and store

the result. This heuristic works well in most cases as it avoids redundant computation. However, for example, if a memory operand derived from base and offset is used multiple times, we will still compute the derived address, store it in a variable, and use the variable wherever the memory operand is used. This essentially introduces one more instruction to compute the address, and one more live register. A better approach would be to implement a special heuristic for addressing-related instructions to exploit addressing modes on instructions if possible and avoid computing addresses unnecessarily. (*iv*) Zebu does not fully perform some performance tricks for x86_64 code generation. For example, it is preferable to use `lea` instead of `add` in some cases to allow concurrent micro-op execution. In summary, we believe the gap between Zebu and LLVM on `quicksort` can be further closed with fine tuning.

5.4.3 Conclusion

In this section, we evaluated our Zebu compiler under two scenarios. First, we presented the performance with a retargeted RPython implementation that runs on top of Zebu. Our retargeted implementation is faster or on par with the stock implementation in 4 out of the 10 benchmarks, and within 1.5× of the execution time of the RPython-C implementation in 8 out of 10 benchmarks. We identified that our retargeted implementation lacks essential compiler optimizations that Zebu requires to be efficient. Then, we conducted a head-to-head comparison between Zebu and the LLVM back-ends by feeding them with equivalent and optimized IR. We evaluated with two benchmarks drawn from the previous set. We showed that on one benchmark, we match LLVM performance, and on the other, we are 25% slower (for the same benchmark in RPython, we are 2.5× slower than the baseline). We also believe this gap can be further closed relatively easily. Through our evaluation in the two scenarios, we find that the Zebu compiler, which is designed with specific constraints, delivers reasonable performance even in comparison with a state-of-the-art heavy-weight compiler framework, LLVM. Some of the benchmarks also show that the rich run-time support from Mu actually gives us an edge over an implementation based on LLVM.

5.5 Summary

Code execution is a key part of implementing a micro virtual machine. Designing a compiler for Zebu is challenging, as we face some interesting constraints. In this chapter, we elaborated our approach to a micro compiler design that focuses on minimalism, efficiency and flexibility. We discussed five design points: selecting optimizations, implementing a general swapstack mechanism, allowing efficient code dynamism with code patching, supporting client introspection for abstract execution state, and generating native and relocatable boot images. We evaluated our compiler with a retargeted RPython implementation and also evaluated our compiler alone with highly optimized IR. We find the results in both scenarios encouraging, and our compiler delivers reasonable performance in most cases. However, we also find

some improvements we can make to the compiler, and confirmed that our retargeted implementation missed some essential IR-level optimizations. This suggests the necessity of providing an optimizer library along with Mu micro virtual machines.

It is worth mentioning that our evaluation is based on a statically typed managed language (RPython), and many aspects of our compiler are not yet evaluated. As the PyPy-Mu project progresses more, we will be interested to see how Mu and our implementation fit in as the back-end for a trace-based just-in-time compiler for a dynamic language. We will discuss this further in future work in the next chapter.

Chapter 6

Conclusion

Building an efficient managed language implementation is difficult, requiring abundant expertise and resources. To alleviate the difficulty, the developers may choose to base their implementation on existing frameworks. However, the most widely used frameworks were not designed to facilitate managed language implementation, and carry intrinsic pitfalls that prevent efficiency. The Mu micro virtual machine was proposed by Wang et al. to address this issue. Mu is a minimal abstract machine that encapsulates three key services that are both important and hard to build in a managed language implementation, i.e., garbage collection, concurrency and code execution. Mu is an open specification that defines the abstract machine. This thesis discussed the development of an efficient implementation of the Mu micro virtual machine.

We aim for minimalism, efficiency, flexibility and robustness for our design and implementation. We pick one design point from each of the key services that Mu provides and discuss how our design answers to our constraints: (*i*) utilizing a memory-/thread-safe language to implement a high-performance garbage collector and the virtual machine, (*ii*) analysing a widely used thread synchronization mechanism, yieldpoint, and exploring and evaluating its design space, and (*iii*) proposing and evaluating a micro compiler design for our micro virtual machine. Our implementation not only delivers efficiency, but also fulfils other properties that we desire for a micro virtual machine.

The implementation language is important for system development, as it affects the overall properties of the implementation. We choose Rust for our implementation language. Rust is an emerging language that provides not only memory and thread safety but also efficiency, which are always desirable for system developers. However, the safety offered by the language restricts its semantics and expressiveness. Whether it is a suitable language for implementing a virtual machine, and whether it remains efficient in our scenarios, is unknown. We use our garbage collector as a case study to demonstrate that though Rust has restricted semantics, it is possible to write a high performance garbage collector in Rust and use the language to make safety guarantees in most of the code. We further apply this approach to implement the entire micro virtual machine in Rust.

Concurrency is essential for an efficient managed language implementation, including both language-level concurrency, and intra-VM concurrency. Yieldpoints are

an important and popular mechanism to support thread synchronization, as they provide known locations and coherent virtual machine states when a thread yields. Despite the fact that yieldpoints are widely used, they were not fully analysed and evaluated in the literature prior to this work. We presented a thorough discussion and categorization on their design space, and conducted performance evaluation on different yieldpoint implementations. We showed the intrinsic characteristics of yieldpoints, such as frequency, distribution and static code size, and also showed the taken and untaken performance and time-to-yield latency for each implementation. We found that the conditional yieldpoint has the best taken performance and time-to-yield latency, and is flexible and easy to implement. We also show that the code patching yieldpoint with group based scope (also known as watchpoints) can serve as efficient guards in managed language implementation. We utilize both designs in our micro virtual machine implementation.

The compiler design is one key part of our micro virtual machine implementation. Our compiler faces interesting constraints, including minimality, efficiency, flexibility, language neutrality, and support for very rich run-time features. Building a compiler under these constraints is challenging and interesting. We presented our compiler, and discuss on how the constraints affect our design, and how we implement some key mechanisms that lay the foundation for our micro virtual machine. We evaluated our compiler in two scenarios: with a retargeted language client, and alone for back-end code generation. We found that though our compiler is minimal, it delivers promising performance in most cases, even in comparison with the state-of-the-art compiler framework, LLVM. We find the results encouraging.

In all, this thesis presents our design of a Mu micro virtual machine implementation. We discussed three major design points, one for each key service Mu provides, and demonstrated our performance for each case. This thesis provides a concrete, practical and efficient design for a micro virtual machine implementation, and further established the concept of micro virtual machines as a solid foundation for managed language implementation.

6.1 Future Work

Micro virtual machines and Mu are an ongoing topic of research. The Zebu VM presented in this thesis is an important part of the research. There are many interesting future works that Zebu leads to. We briefly discuss a few.

6.1.1 PyPy-Mu on Zebu

As discussed in Section 2.4.1, the first step to get PyPy running on Mu is finished: RPython was retargeted to Zebu with the capability of running non-trivial programs, and the performance was evaluated in this thesis (Section 5.4.1). The next step is to retarget the PyPy meta-tracing just-in-time compiler to Zebu, with possible utilization of many just-in-time features that Mu provides, such as watchpoints and stack introspection. We are eager to see how Zebu performs for a dynamic language.

Future work for Zebu on this track is to provide back-end and run-time support for the retargeted PyPy JIT, and to establish efficiency in just-in-time scenarios (generated code quality, run-time performance and compilation time).

6.1.2 Mu IR Optimizer Library

Micro virtual machines provide abstractions over some common services that all managed languages would require, and relieves the developers from implementing all the low-level details by themselves. However, the design philosophy of micro virtual machines to pursue minimalism pushes some non-trivial optimization tasks to the client side. It is beneficial for micro virtual machines for implementation and verification purposes, however, it adds burden to the client developers, as they are still required to implement those optimizations that are common to various languages. As a solution to this, we plan to provide an optimizer library for Mu IR (similar to `opt` in LLVM framework). The idea of having libraries to further facilitate the clients' development was originated from Wang [2017], and our performance evaluation on a retargeted language implementation further confirms its necessity. The optimizer library is not a part of the micro virtual machine implementation, as it is optional, and does not necessarily follow any constraint of a micro virtual machine. The client may choose to use the library, or implement the optimizations by themselves (including ignoring the optimizations). We plan to reuse Zebu's code (such as data structures for Mu IR and the compiler pass manager) to implement the optimizer library, and allow the client to choose optimizations they would like to include and the order to run. We will be interested to re-evaluate the experiments in Section 5.4.1, and see how they perform when the missing front-end optimizations are performed by the optimizer.

6.1.3 Performance Re-evaluation of GC and Yieldpoints

We evaluated our prototype GC and yieldpoints mechanisms as exploratory work before we designed and implemented the counterparts in Zebu. Both works are informative and shaped the implementation for Zebu, but the evaluations had limitations. Having RPython and the full Python in the future running on Zebu would allow us opportunities to re-evaluate both systems.

In Section 3.3.3, we evaluated the performance of our prototype garbage collector with a focus on comparing the implementations in Rust and in C. We evaluated with micro benchmarks (including `gcbench`) to demonstrate the efficiency. As the prototype was a standalone garbage collector and was not integrated with a language runtime (as the integration would require significant efforts [JEP-304, 2018]), we were unable to measure its performance with sophisticated benchmarks. Zebu's GC was evolved from the prototype. With Zebu being able to run more sophisticated Python benchmarks in the future, it would be interesting to re-evaluate the GC performance on Zebu to show the GC performance with a safe language (Rust) in larger macro benchmarks.

In Section 4.4, we performed a head-to-head comparison between different yieldpoint mechanisms on JikesRVM. The results showed the characteristics of different yieldpoints on different micro architectures, and helped us quantitatively understand yieldpoint behaviors. However, as they were measured on JikesRVM, the numbers are biased towards JikesRVM. For example, VM-specific policies such as where the compiler inserts yieldpoint and the purpose of yielding may affect yieldpoint bahaviors. An re-evaluation for yieldpoints on Zebu would show the characteristics of yieldpoints across different virtual machines, and how VM policies affect yieldpoint behaviors.

6.1.4 Verified Mu

Trustworthy computing systems are desirable. Works in Klein et al. [2009] provided a trust-able operating system kernel, seL4, as the bottom layer of a trustworthy computing stack. A trust-able language virtual machine could further extend the stack towards the application layer.

The Mu micro virtual machine is designed to be verifiable, which adds a further advantage of the micro virtual machine approach in comparison with the existing approaches discussed in Section 2.1. First, Mu's semantics are non-ambiguous, which makes it possible to reason about the semantics and to provide a formal specification. Initial works[1] have been progressing on this direction to provide a formal specification for Mu in the HOL interactive theorem prover. Secondly, the minimalism makes the scope and the lines of code of the virtual machine small enough to be practically verifiable. This thesis has further confirmed this point by implementing an efficient Mu within limited lines of code. Potential future work on the verification track could be a verified implementation of Mu. Furthermore, initial works have been started to explore the possibility of running micro virtual machines on top of the seL4 micro kernel. This investigates the viability of combining micro virtual machines as another trust-able layer with the trust-able operating system kernel seL4.

[1] The piece of work and the works described later in this section do not involve the author of this thesis, and are conducted by multiple researchers that we do not elaborate here.

Bibliography

AbouGhazaleh, N.; Mossé, D.; Childers, B. R.; and Melhem, R., 2006. Collaborative operating system and compiler power management for real-time applications. *ACM Transactions on Embedded Computer Systems.*, (2006). doi:10.1145/1132357.1132361. (cited on page 66)

Adams, K.; Evans, J.; Maher, B.; Ottoni, G.; Paroski, A.; Simmers, B.; Smith, E.; and Yamauchi, O., 2014. The Hiphop virtual machine. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14. doi:10.1145/2660193.2660199. (cited on page 2)

Agesen, O., 1998. GC points in a threaded environment. Technical report, Sun Microsystems Laboratories. http://dl.acm.org/citation.cfm?id=974974. (cited on pages 47, 51, and 54)

Alpern, B.; Attanasio, C. R.; Cocchi, A.; Lieber, D.; Smith, S.; Ngo, T.; Barton, J. J.; Hummel, S. F.; Sheperd, J. C.; and Mergen, M., 1999. Implementing Jalapeño in java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99. doi:10.1145/320384.320418. (cited on pages 18, 29, 34, 37, 45, 52, 55, and 67)

Alpern, B.; Augart, S.; Blackburn, S. M.; Butrico, M.; Cocchi, A.; Cheng, P.; Dolby, J.; Fink, S.; Grove, D.; Hind, M.; McKinley, K. S.; Mergen, M.; Moss, J. E. B.; Ngo, T.; Sarkar, V.; and Trapp, M., 2005. The Jikes Research Virtual Machine project: Building an open source research community. *IBM Systems Journal*, (2005). doi:10.1147/sj.442.0399. (cited on pages 29 and 37)

AMD64 ABI, 2017. System V application binary interface, AMD64 architecture processor supplement, draft version 0.99.7. https://www.uclibc.org/docs/psABI-x86_64.pdf. (cited on page 20)

Ananian, C., 1997. *The Static Single Information Form*. Master's thesis, Princeton University. (cited on page 18)

Appel, A. W. and Palsberg, J., 2003. *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edn. ISBN 052182060X. (cited on pages 23 and 69)

ARM64 ABI, 2017. Procedure call standard for the ARM 64-bit architecture (AArch64). http://infocenter.arm.com/help/topic/com.arm.doc.ihi0055b/IHI0055B_aapcs64.pdf. (cited on page 20)

Arnold, M. and Grove, D., 2005. Collecting and exploiting high-accuracy call graph profiles in virtual machines. In *International Symposium on Code Generation and Optimization*. doi:10.1109/CGO.2005.9. (cited on page 50)

Baker, J.; Cunei, A.; Pizlo, F.; and Vitek, J., 2007. Accurate garbage collection in uncooperative environments with lazy pointer stacks. In *Proceedings of the 16th International Conference on Compiler Construction*, CC'07. http://dl.acm.org/citation.cfm?id=1759937.1759944. (cited on page 10)

Benchmarks Game. Computer Language Benchmarks Game. http://benchmarksgame.alioth.debian.org. (cited on pages 2 and 79)

Blackburn, S. M.; Cheng, P.; and McKinley, K. S., 2004. Oil and water? High performance garbage collection in Java with MMTk. In *International Conference on Software Engineering*. doi:10.1109/icse.2004.1317436. (cited on pages 12, 28, 29, 33, 36, and 37)

Blackburn, S. M.; Garner, R.; Hoffman, C.; Khan, A. M.; McKinley, K. S.; Bentzur, R.; Diwan, A.; Feinberg, D.; Frampton, D.; Guyer, S. Z.; Hirzel, M.; Hosking, A.; Jump, M.; Lee, H.; Moss, J. E. B.; Phansalkar, A.; Stefanović, D.; VanDrunen, T.; von Dincklage, D.; and Wiedermann, B., 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. doi:10.1145/1167515.1167488. (cited on pages 45 and 57)

Blackburn, S. M.; Hirzel, M.; Garner, R.; and Stefanović, D. pjbb2005: The pseudoJBB benchmark. http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005. (cited on page 57)

Blackburn, S. M. and McKinley, K. S., 2008. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08. doi:10.1145/1375581.1375586. (cited on pages 23, 31, 32, 40, and 55)

Blackburn, S. M.; Salishev, S. I.; Danilov, M.; Mokhovikov, O. A.; Nashatyrev, A. A.; Novodvorsky, P. A.; Bogdanov, V. I.; Li, X. F.; and Ushakov, D., 2008. The Moxie JVM experience. (cited on pages 18, 29, and 67)

Boehm, H.-J., 2005. Threads cannot be implemented as a library. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. doi:10.1145/1065010.1065042. (cited on page 28)

Boehm, H.-J., 2011. How to miscompile programs with "benign" data races. In *USENIX Conference on Hot Topics in Parallelism*. http://www.usenix.org/events/hotpar11/tech/final_files/Boehm.pdf. (cited on page 38)

Boehm, H.-J.; Demers, A. J.; and Shenker, S., 1992. Mostly parallel garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. doi:10.1145/113445.113459. (cited on page 29)

Boehm, H.-J. and Weiser, M., 1988. Garbage collection in an uncooperative environment. (1988). doi:10.1002/spe.4380180902. (cited on pages 29 and 52)

Boissinot, B.; Brisk, P.; Darte, A.; and Rastello, F., 2012. SSI properties revisited. *ACM Transactions on Embedded Computer Systems.*, (2012). doi:10.1145/2180887.2180898. (cited on page 18)

Bolz, C. F.; Cuni, A.; Fijalkowski, M.; and Rigo, A., 2009. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09. doi:10.1145/1565824.1565827. (cited on page 11)

Boulytchev, D., 2007. BURS-based instruction set selection. In *Proceedings of the 6th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics*, PSI'06. http://dl.acm.org/citation.cfm?id=1760700.1760740. (cited on page 19)

Cassez, F.; Sloane, A.; Roberts, M.; Pigram, M.; Suvanpong, P.; and de Aledo, P., 2017. *Skink: Static analysis of programs in LLVM intermediate representation*. doi:10.1007/978-3-662-54580-5_27. (cited on page 10)

Castanos, J.; Edelsohn, D.; Ishizaki, K.; Nagpurkar, P.; Nakatani, T.; Ogasawara, T.; and Wu, P., 2012. On the benefits and pitfalls of extending a statically typed language JIT compiler for dynamic scripting languages. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12. doi:10.1145/2384616.2384631. (cited on pages 3 and 69)

Chambers, C. and Ungar, D., 1991. Making pure object-oriented languages practical. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '91. doi:10.1145/117954.117955. (cited on page 16)

Chase, D. and Lev, Y., 2005. Dynamic circular work-stealing deque. In *ACM Symposium on Parallelism in Algorithms and Architectures*. doi:10.1145/1073970.1073974. (cited on page 37)

Click, C.; Tene, G.; and Wolf, M., 2005. The pauseless GC algorithm. In *ACM/USENIX International Conference on Virtual Execution Environments*. doi:10.1145/1064979.1064988. (cited on page 51)

Cooper, K. D. and Dasgupta, A., 2006. Tailoring graph-coloring register allocation for runtime compilation. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06. doi:10.1109/CGO.2006.35. (cited on page 68)

Costa, I.; Alves, P.; Santos, H. N.; and Pereira, F. M. Q., 2013. Just-in-time value specialization. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code*

Generation and Optimization, CGO '13. doi:10.1109/CGO.2013.6495006. (cited on page 11)

Debbabi, M.; Gherbi, A.; Ketari, L.; Talhi, C.; Tawbi, N.; Yahyaoui, H.; and Zhioua, S., 2004. A dynamic compiler for embedded Java virtual machines. In *Proceedings of the 3rd International Symposium on Principles and Practice of Programming in Java*, PPPJ '04. http://dl.acm.org/citation.cfm?id=1071565.1071584. (cited on pages 66 and 67)

Dolan, S.; Muralidharan, S.; and Gregg, D., 2013. Compiler support for lightweight context switching. *ACM Transactions on Architecture and Code Optimization*, (2013). doi:10.1145/2400682.2400695. (cited on pages 15, 66, 67, 70, 71, and 72)

Drew, S.; J. Gough, K.; and Ledermann, J., 1995. Implementing zero overhead exception handling. (1995). (cited on page 20)

Eisl, J.; Grimmer, M.; Simon, D.; Würthinger, T.; and Mössenböck, H., 2016. Trace-based register allocation in a JIT compiler. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '16. doi:10.1145/2972206.2972211. (cited on page 68)

Flack, C.; Hosking, A. L.; and Vitek, J., 2003. Idioms in Ovm. Technical report, Purdue University. (cited on page 9)

Frampton, D., 2010. *Garbage Collection and the Case for High-level Low-level Programming*. Ph.D. thesis, Australian National University. (cited on pages 6 and 28)

Frampton, D.; Blackburn, S. M.; Cheng, P.; Garner, R. J.; Grove, D.; Moss, J. E. B.; and Salishev, S. I., 2009. Demystifying magic: High-level low-level programming. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. doi:10.1145/1508293.1508305. (cited on pages 28, 29, 30, 33, and 37)

Fraser, C. W.; Hanson, D. R.; and Proebsting, T. A., 1992. Engineering a simple, efficient code-generator generator. *ACM Lett. Program. Lang. Syst.*, (1992). doi:10.1145/151640.151642. (cited on page 68)

Gal, A.; Eich, B.; Shaver, M.; Anderson, D.; Mandelin, D.; Haghighat, M. R.; Kaplan, B.; Hoare, G.; Zbarsky, B.; Orendorff, J.; Ruderman, J.; Smith, E. W.; Reitmaier, R.; Bebenita, M.; Chang, M.; and Franz, M., 2009. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09. doi:10.1145/1542476.1542528. (cited on pages 11 and 69)

Gal, A.; Probst, C. W.; and Franz, M., 2006. HotpathVM: An effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE '06. doi:10.1145/1134760.1134780. (cited on page 66)

Gariano, I., 2017. *AArch64 Mu: An Efficient Implementation of a Micro Virtual Machine*. Honours thesis, Australian National University. (cited on pages 70 and 77)

Gaudet, M., 2015. Experiments in sharing Java VM technology with CRuby. http://rubykaigi.org/2015/presentations/MattStudies. (cited on page 13)

Gaudet, M. and Stoodley, M., 2016. Rebuilding an airliner in flight: A retrospective on refactoring IBM Testarossa production compiler for Eclipse OMR. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages*, VMIL '16. doi:10.1145/2998415.2998419. (cited on page 12)

Geoffray, N., 2009. The VMKit project: Java (and .Net) on top of LLVM. https://llvm.org/devmtg/2008-08/Geoffray_VMKitProject.pdf. (cited on page 12)

Geoffray, N.; Thomas, G.; Lawall, J.; Muller, G.; and Folliot, B., 2010. VMKit: A substrate for managed runtime environments. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '10. doi:10.1145/1735997.1736006. (cited on page 12)

George, L. and Appel, A. W., 1996. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, (1996). doi:10.1145/229542.229546. (cited on page 68)

Grech, N.; Georgiou, K.; Pallister, J.; Kerrison, S.; Morse, J.; and Eder, K., 2015. Static analysis of energy consumption for LLVM IR programs. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '15. doi:10.1145/2764967.2764974. (cited on page 10)

Greedy RA, 2011. Greedy register allocation in LLVM 3.0. http://blog.llvm.org/2011/09/greedy-register-allocation-in-llvm-30.html. (cited on page 68)

Hack, 2017. The Hack language specification. https://github.com/hhvm/hack-langspec/blob/master/spec/04-basic-concepts.md#reclamation-and-automatic-memory-management. (cited on page 3)

Haj-Yihia, J.; Yasin, A.; and Ben-Asher, Y., 2015. DOEE: Dynamic optimization framework for better energy efficiency. In *Proceedings of the Symposium on High Performance Computing*, HPC '15. doi:10.1145/513829.513832. (cited on page 66)

Henderson, F. and Somogyi, Z., 2002. Compiling Mercury to high-level C code. In *Compiler Construction: 11th International Conference*, CC '02. doi:10.1007/3-540-45937-5_15. (cited on page 9)

Hölzle, U.; Chambers, C.; and Ungar, D., 1991. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91. http://dl.acm.org/citation.cfm?id=646149.679193. (cited on pages 11 and 69)

Hölzle, U. and Ungar, D., 1994. A third-generation SELF implementation: Reconciling responsiveness with performance. In *Proceedings of the Ninth Annual Conference on Object-oriented Programming Systems, Language, and Applications*, OOPSLA '94. doi:10.1145/191080.191116. (cited on page 16)

Hong, S. and Gerber, R., 1993. Compiling real-time programs into schedulable code. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93. doi:10.1145/155090.155106. (cited on page 66)

HotSpot VM, 2017. HotSpot VM. http://openjdk.java.net/groups/hotspot/. (cited on page 53)

Huang, X.; Blackburn, S. M.; McKinley, K. S.; Moss, J.; Wang, Z.; and Cheng, P., 2004. The garbage collection advantage: Improving program locality. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*. (cited on page 56)

INT3 PATCHING, 2013. x86: introduce int3-based instruction patching. https://patchwork.kernel.org/patch/2825904/. (cited on page 75)

Ishizaki, K.; Kawahito, M.; Yasue, T.; Komatsu, H.; and Nakatani, T., 2000. A study of devirtualization techniques for a Java just-in-time compiler. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00. doi:10.1145/353171.353191. (cited on page 69)

JDK Developer's Guide, 2017. Java threads in the Solaris environment. https://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqe/index.html. (cited on page 15)

JEP-169, 2012. JEP 169: Value objects. http://openjdk.java.net/jeps/169. (cited on page 11)

JEP-193, 2014. JEP 193: Variable handles. http://openjdk.java.net/jeps/193. (cited on page 11)

JEP-304, 2018. JEP 304: Garbage collector interface. http://openjdk.java.net/jeps/304. (cited on page 89)

Jones, R.; Hosking, A.; and Moss, E., 2011. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edn. ISBN 1420082795, 9781420082791. (cited on page 46)

JSR-292, 2011. JSR 292: Supporting dynamically typed languages on the Java platform. https://jcp.org/en/jsr/detail?id=292. (cited on page 11)

Kawahito, M.; Komatsu, H.; and Nakatani, T., 2000. Effective null pointer check elimination utilizing hardware trap. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX. doi:10.1145/378993.379234. (cited on page 69)

Kerschbaumer, C.; Wagner, G.; Wimmer, C.; Gal, A.; Steger, C.; and Franz, M., 2009. SlimVM: A small footprint Java virtual machine for connected embedded systems. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09. doi:10.1145/1596655.1596678. (cited on page 66)

Klechner, R., 2011. Unladen Swallow retrospective. http://qinsb.blogspot.com.au/2011/03/unladen-swallow-retrospective.html. (cited on pages 3 and 4)

Klein, G.; Elphinstone, K.; Heiser, G.; Andronick, J.; Cock, D.; Derrin, P.; Elkaduwe, D.; Engelhardt, K.; Kolanski, R.; Norrish, M.; Sewell, T.; Tuch, H.; and Winwood, S., 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09. doi:10.1145/1629575.1629596. (cited on pages 5, 19, and 90)

Koenig, A. and Stroustrup, B., 1993. The evolution of c++. chap. Exception Handling for C++. MIT Press. ISBN 0-262-73107-x. (cited on page 15)

Kumar, V.; Frampton, D.; Blackburn, S. M.; Grove, D.; and Tardieu, O., 2012. Work-stealing without the baggage. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*. doi:10.1145/2384616.2384639. (cited on page 48)

Lameed, N. A. and Hendren, L. J., 2013. A modular approach to on-stack replacement in LLVM. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13. doi:10.1145/2451512.2451541. (cited on page 16)

Lee, H.; von Dincklage, D.; Diwan, A.; and Moss, J. E. B., 2006. Understanding the behavior of compiler optimizations. *Softw. Pract. Exper.*, (2006). doi:10.1002/spe.v36:8. (cited on page 66)

Lin, Y., 2012. *Virtual Machine Design and High-level Implementation Languages*. Master's thesis, Australian National University. (cited on page 6)

Lin, Y.; Blackburn, S. M.; Hosking, A. L.; and Norrish, M., 2016. Rust as a language for high performance GC implementation. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2016. doi:10.1145/2926697.2926707. (cited on page 27)

Lin, Y.; Wang, K.; Blackburn, S. M.; Hosking, A. L.; and Norrish, M., 2015. Stop and go: Understanding yieldpoint behavior. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15. doi:10.1145/2754169.2754187. (cited on page 45)

LLILC GC, 2017. GC Support in LLILC. https://github.com/dotnet/llilc/blob/master/Documentation/llilc-gc.md. (cited on page 3)

LLVM LANGS, 2017. LLVM Wikipedia. https://en.wikipedia.org/wiki/LLVM. (cited on page 10)

LORENZ, M.; WEHMEYER, L.; AND DRÄGER, T., 2002. Energy aware compilation for DSPs with SIMD instructions. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems*, LCTES/SCOPES '02. doi:10.1145/513829.513847. (cited on page 66)

LUEH, G.-Y.; GROSS, T.; AND ADL-TABATABAI, A.-R., 2000. Fusion-based register allocation. *ACM Transactions on Programming Languages and Systems*, (2000). doi:10.1145/353926.353929. (cited on page 68)

MU SPEC, 2016. Mu specification. https://gitlab.anu.edu.au/mu/mu-spec. (cited on page 13)

NAKAIKE, T. AND MICHAEL, M. M., 2010. Lock elision for read-only critical sections in Java. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10. doi:10.1145/1806596.1806627. (cited on page 69)

NATIONAL RESEARCH COUNCIL, 1994. *Academic Careers for Experimental Computer Scientists and Engineers*. https://www.nap.edu/catalog/2236/academic-careers-for-experimental-computer-scientists-and-engineers. (cited on page 1)

.NET GC, 2017. .NET GC class. https://msdn.microsoft.com/en-us/library/system.gc(v=vs.110).aspx. (cited on page 3)

NIEPHAUS, F.; FELGENTREFF, T.; AND HIRSCHFELD, R., 2018. GraalSqueak: A fast Smalltalk bytecode interpreter written in an AST interpreter framework. In *Proceedings of the 13th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS '18. doi:10.1145/3242947.3242948. (cited on page 12)

NUZMAN, D. AND HENDERSON, R., 2006. Multi-platform auto-vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06. doi:10.1109/CGO.2006.25. (cited on page 69)

OPEN J9, 2017. Open J9. https://www.eclipse.org/openj9/oj9_resources.html. (cited on page 18)

OPEN JDK, 2017. Open JDK. http://openjdk.java.net/. (cited on page 3)

OSSIA, Y.; BEN-YITZHAK, O.; GOFT, I.; KOLODNER, E. K.; LEIKEHMAN, V.; AND OWSHANKO, A., 2002. A parallel, incremental and concurrent GC for servers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. doi:10.1145/512529.512546. (cited on page 36)

PHILIP REAMES, S. D., 2014. Practical fully relocating garbage collection in LLVM. https://www.playingwithpointers.com/llvm-dev-2014-slides.pdf. (cited on page 10)

PHP GC, 2017. PHP manual: garbage collection. http://php.net/manual/en/features.gc.php. (cited on page 3)

Pizlo, F., 2016. Introducing the B3 JIT compiler. https://webkit.org/blog/5852/introducing-the-b3-jit-compiler/. (cited on page 3)

Pizlo, F.; Frampton, D.; and Hosking, A. L., 2011. Fine-grained adaptive biased locking. In *International Conference on Principles and Practice of Programming in Java*. doi:10.1145/2093157.2093184. (cited on page 48)

Plaisted, D. A., 2013. Source-to-source translation and software engineering. doi:10.4236/jsea.2013.64A005. (cited on page 9)

Pyston, 2017. Pyston 0.6.1 released, and future plans. https://blog.pyston.org/2017/01/31/pyston-0-6-1-released-and-future-plans/. (cited on page 10)

Rafkind, J.; Wick, A.; Regehr, J.; and Flatt, M., 2009. Precise garbage collection for C. In *Proceedings of the 2009 International Symposium on Memory Management*, ISMM '09. doi:10.1145/1542431.1542438. (cited on page 10)

Reames, P., 2017. Falcon: an optimizing Java JIT. https://llvm.org/devmtg/2017-10/slides/Reames-FalconKeynote.pdf. (cited on page 10)

Rigo, A. and Pedroni, S., 2006. PyPy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06. doi:10.1145/1176617.1176753. (cited on pages 2, 18, 29, and 37)

Roberts, E. S., 1989. Implementing exceptions in C. Technical report, Systems Research Center. (cited on page 10)

Rose, J., 2014. Evolving the JVM: Principles and directions. http://www.oracle.com/technetwork/java/jvmls2014goetzrose-2265201.pdf. (cited on page 4)

RPYSOM, 2014. RPySOM: The Simple Object Machine Smalltalk implemented in RPython. https://github.com/SOM-st/RPySOM. (cited on page 24)

Rust, 2010. The Rust programming language. https://www.rust-lang.org. (cited on pages 27 and 28)

Rust RFC 1543, 2016. Rust RFC 1543: Add more integer atomic types. https://github.com/rust-lang/rfcs/pull/1543. (cited on page 39)

Saputra, H.; Kandemir, M.; Vijaykrishnan, N.; Irwin, M. J.; Hu, J. S.; Hsu, C.-H.; and Kremer, U., 2002. Energy-conscious compilation based on voltage scaling. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems*, LCTES/SCOPES '02. doi:10.1145/513829.513832. (cited on page 66)

Shahriyar, R.; Blackburn, S. M.; and McKinley, K. S., 2014. Fast conservative garbage collection. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14. doi:10.1145/2660193.2660198. (cited on pages 20 and 43)

Shahriyar, R.; Blackburn, S. M.; Yang, X.; and McKinley, K. S., 2013. Taking off the gloves with reference counting Immix. In *OOPSLA '13: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. doi:10.1145/2544173.2509527. (cited on page 32)

Shaylor, N., 2002. A just-in-time compiler for memory-constrained low-power devices. In *Proceedings of the 2nd Java™ Virtual Machine Research and Technology Symposium*. http://dl.acm.org/citation.cfm?id=648042.744884. (cited on page 66)

Singer, J., 2006. Static program analysis based on virtual register renaming. Technical report, University of Cambridge, Computer Laboratory. (cited on page 18)

Smith, M. D.; Ramsey, N.; and Holloway, G., 2004. A generalized algorithm for graph-coloring register allocation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04. doi:10.1145/996841.996875. (cited on page 68)

SPECjbb2000. SPECjbb2000 (Java Business Benchmark) documentation. https://www.spec.org/jbb2005/. (cited on page 57)

SPECjvm98. SPECjvm98, release 1.03. http://www.spec.org/jvm98. (cited on page 57)

Steele, G. L., Jr. and Gabriel, R. P., 1993. The evolution of Lisp. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II. doi:10.1145/154766.155373. (cited on page 15)

Stichnoth, J. M.; Lueh, G.-Y.; and Cierniak, M., 1999. Support for garbage collection at every instruction in a Java compiler. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99. doi:10.1145/301618.301652. (cited on pages 16 and 51)

Tatsubori, M.; Tozawa, A.; Suzumura, T.; Trent, S.; and Onodera, T., 2010. Evaluation of a just-in-time compiler retrofitted for PHP. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '10. doi:10.1145/1735997.1736015. (cited on page 3)

Ungar, D.; Spitz, A.; and Ausch, A., 2005. Constructing a metacircular virtual machine in an exploratory programming environment. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05. doi:10.1145/1094855.1094865. (cited on pages 18 and 67)

Wang, K., 2017. *Micro virtual machines as a solid foundation for language development*. Ph.D. thesis, Australian National University. (cited on pages 8, 13, and 89)

Wang, K.; Blackburn, S. M.; Hosking, A. L.; and Norrish, M., 2018. Hop, Skip, & Jump: Practical on-stack replacement for a cross-platform language-neutral VM. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '18. doi:10.1145/3186411.3186412. (cited on pages 15 and 16)

Wang, K.; Lin, Y.; Blackburn, S. M.; Norrish, M.; and Hosking, A. L., 2015. Draining the swamp: Micro virtual machines as solid foundation for language development. In *1st Summit on Advances in Programming Languages*, SNAPL '15. doi:10.4230/LIPIcs.SNAPL.2015.321. (cited on pages ix, 1, 4, 5, 13, and 87)

Weil, D.; Bertin, V.; Closse, E.; Poize, M.; Venier, P.; and Pulou, J., 2000. Efficient compilation of ESTEREL for real-time embedded systems. In *Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '00. doi:10.1145/354880.354882. (cited on page 66)

Willis, N., 2009. Unladen Swallow: Accelerating Python. https://lwn.net/Articles/332038/. (cited on page 10)

Wimmer, C. and Franz, M., 2010. Linear scan register allocation on SSA form. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10. doi:10.1145/1772954.1772979. (cited on page 68)

Wimmer, C.; Haupt, M.; Van De Vanter, M. L.; Jordan, M.; Daynès, L.; and Simon, D., 2013. Maxine: An approachable virtual machine for, and in, Java. *ACM Trans. Archit. Code Optim.*, (2013). doi:10.1145/2400682.2400689. (cited on pages 18, 29, 30, 37, and 67)

Wimmer, C. and Mössenböck, H., 2005. Optimized interval splitting in a linear scan register allocator. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05. doi:10.1145/1064979.1064998. (cited on page 68)

Würthinger, T.; Wimmer, C.; Humer, C.; Wöss, A.; Stadler, L.; Seaton, C.; Duboscq, G.; Simon, D.; and Grimmer, M., 2017. Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '17. doi:10.1145/3062341.3062381. (cited on pages 11 and 12)

Würthinger, T.; Wöss, A.; Stadler, L.; Duboscq, G.; Simon, D.; and Wimmer, C., 2012. Self-optimizing AST interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12. doi:10.1145/2384577.2384587. (cited on page 11)

Yang, X.; Blackburn, S. M.; Frampton, D.; and Hosking, A. L., 2012. Barriers reconsidered, friendlier still! In *ACM SIGPLAN International Symposium on Memory Management*. doi:10.1145/2258996.2259004. (cited on page 55)

Yang, X.; Blackburn, S. M.; and McKinley, K. S., 2015. Computer performance microscopy with SHIM. In *International Symposium on Computer Architecture*. doi: 10.1145/2749469.2750401. (cited on page 47)

Zhang, J., 2015. *MuPy: A First Client for the Mu Micro Virtual Machine*. Honours thesis, Australian National University. (cited on pages 8, 13, 24, 65, and 78)

Zhao, J.; Nagarakatte, S.; Martin, M. M.; and Zdancewic, S., 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12. doi:10.1145/2103656.2103709. (cited on page 10)