AArch64 Mu: An Efficient Implementation of a Micro Virtual Machine

Isaac Oscar Gariano

A sub-thesis submitted in partial fulfilment of the degree of
Bachelor of Science (Honours) at
The Research School of Computer Science
Australian National University

November 2017

© Isaac Oscar Gariano

Typeset in Tex Gyre Schola, `Latin Modern Mono`, and Latin Modern Sans,
by XƎTEX and LaTeX 2_{ε}.

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

Isaac Oscar Gariano
5th November 2017

To Oswin,
whom I left, so I could write this.

Acknowledgements

Firstly, and above all, I wish to give my enormousness gratitude to Professor Steve Blackburn for giving me this project. Your wisdom, knowledge, experience and feedback has been immensely valuable. I thank you from the bottom of my heart for supervising and accepting me, you are far better supervisor than I could have hoped for

To Professor Tony Hosking and Doctor Michael Norrish, I thank you for assistance and guidance throughout this project, as well as agreeing to review this thesis. To Yi Lin, I thank you for creating Zebu, and assisting me with its modification; I also thank you greatly for accepting my input and contributions. To Kunshan Wang, though we have never met, your creation of Mu, and help with understanding and implementing it has been invaluable, thank you.

To my friends and colleagues, Zixian Cai, Pavel Zakopaylo, and John Zhang, I thank you for making use of, and encouraging my work, as well as providing assistance and feedback with both my technical, and written work.

I wish to thank the Australian National University, and specifically its Research School of Computer Science, for providing me with the facilities and teaching that helped me with my research; I also wish thank them for accepting me into my chosen degree program. I also wish to thank the University of Queensland for teaching me most of what I know about computer science, and how large and diverse the field is, and giving me the inspiration to pursue research. I also wish to thank the people and residents at John XXIII College for accepting me, and allowing me to live there throughout my degree.

Finally, I wish to thank my parents, Tamara Young and Doctor Antonia Carlos Gariano, for raising me throughout my life, and providing me with the necessary love, support, and opportunity to do this research.

Abstract

Implementing programming languages is difficult, implementing efficient programming languages is even more challenging. The Mu micro virtual machine was designed to address this problem by providing important abstractions to language implementers. Mu provides a minimal, language agnostic framework for language implementation, yet it provides powerful abstractions over memory, concurrency (threads and 'stacks'), exceptions, and hardware (such as dynamic code generation), that are particularly useful for modern managed languages. The first proof of concept implementation of Mu achieved key design issues including usability and correctness. This thesis is concerned with the next stage of Mu development, proof of performance.

In particular, I wish to demonstrate that:
the abstraction level provided by Mu is appropriate for the generation of efficient code.

I am part of a team that is developing a new 'fast' implementation of Mu, code-named *Zebu*. It is written in the medium-level systems programming language *Rust*. Zebu includes a compiler which outputs native machine-specific assemblies . I have created a new RISC based back-end (targeting ARM's 64-bit architecture state, 'AArch64') to complement the x86-64 back-end (developed by PhD student Yi Lin). I have also improved the general implementation of Zebu and implemented additional features of Mu. I have made Zebu a reasonably efficient (when compared to LLVM) implementation of Mu, as well as one providing a significant proportion of Mu's features.

In this thesis, I will detail how I have implemented parts of, and improved, Zebu, with a particular focus on the efficient implementation of some of Mu's most interesting features. I will further detail the simple, yet effective, optimisations I have implemented, and evaluate their effect on performance, with a particular focus on Zebu's potential suitability as a JIT back-end for managed languages.

I hope that this work can assist with further research into the idea of micro virtual machines, and in particular, to further improve Zebu. Though a large section of the Mu specification has been implemented, there is still a lot of work to be done; currently Zebu is only an AOT compiler, and there are many advanced features of Mu that have not been implemented yet, such as On Stack Replacement features.

Contents

Chapter 1

Introduction

This thesis argues that the Mu micro virtual machine [Wang et al. 2017] provides a level of abstraction amenable to efficient language implementation. It does so by presenting an implementation of Mu, *Zebu*—running on the ARMv8-A AArch64 RISC architecture [ARM Limited 2017]—and comparing and evaluating its performance.

1.1 Problem Statement

Though there are a large number of programming languages, some of which are implemented well, they often suffer from performance problems or undesirable semantics due to tight coupling of the language design and implementation (such as PHP's copy-on-write implementation [Tozawa et al. 2009]). One of the major reasons for these problems is the difficulty, expertise, and time required in implementing languages well. This is particularly evident with the implementation of hardware specific code and optimisations, concurrency, and garbage collection. I hope that providing abstractions that handles these concerns for language developers, will allow them to focus on higher level design and implementation. The in-development Mu 'micro virtual machine' is designed to be such an abstraction layer; it is intended to be an easy target for language developers, to be used as a JIT back-end, and to produce efficient code.

Though the design of Mu may appear to fulfil its objectives [Wang et al. 2015; Zhang 2015], this hasn't as yet been experimentally verified; in particular whether it can realise the goal of providing back-end level performance was previously untested. In order to experimentally verify and confirm this, Yi Lin, A PHD student at the Australian National University, has created a Mu implementation, *Zebu*, with efficiency as its primary design goal. It originally only supported Linux and Mac on x86-64, but I have since expand it to support Linux on Aarch64 and Javad Ebrahimian Amiri has added support for Rumprun (running on the SeL4 microkernal) on x86-64. Zebu was originally inefficient and its performance had not been properly evaluated; these problems needed to be fixed in order for Zebu to properly answer the question of whether the design of Mu is efficient or not.

1.2 Research Content

In this thesis I will present, describe, review and evaluate my work on answering the question of Mu's performance. In order to evaluate Mu's performance potential, I have extended Zebu with an AArch64 back-end, and evaluated its performance. This approach was chosen over alternatives (such as implementing Mu on-top of an already existing performant system (e.g. LLVM), or providing a theoretical analysis of Mu's semantic) so that Zebu could be made to be a real-world, practical and efficient implementation of Mu, whilst also minimizing code-size and compilation time, to allow it to eventually be extended to perform fast JIT compilation.

In particular, this thesis will discuss:
- The general architecture of Zebu, and argue that it is appropriate and reasonable (see § **3**).
- How I have extended Zebu with additional features and implemented my AArch64 back-end, together with optimisations (see § **4**).
- My experimental evaluation of the correctness and performance of Zebu, as well as the reasons for these results and their meaning (see § **5**).
- An overview of the work I have conducted, what it means and its limitations, together with ideas and plans for future work (see § **6**).

1.3 Results

Through thorough testing, I can conclude that Zebu adequately implements the features of subset of Mu it supports.

I have conducted an experimental performance test with a set of 10 RPython benchmarks; in 7/10 of these benchmarks, the performance of Zebu is at most 30% slower than Clang with its highest optimisation level, also in 7/10, Zebu is faster than Clang with its lowest back-end optimisation level. The results of experiment allow me to conclude that:
- Zebu's (and hence Mu's) performance is a reasonable choice as a managed-language back-end, in particular it is comparable with using C & Clang.
- Zebu's performance could be improved with additional optimisations, without needing to redefine the semantics of Mu.

1.4 Summary

The work and results presented in this research, suggest that Mu can be implemented to be efficient. These results point to Mu being a performant target for language developers, and I hope that this will lead to more efficient, and less bug-ridden, language implementations; in particular I hope that new efficient languages can be developed and built upon this system with ease, and take advantage of the abstractions and value provided by Mu.

However, since I have only really compared a portion of Mu's features (namely those that are implemented in Zebu and those that have counterparts in LLVM), there is work yet to be done. In particular, whether these unexplored features can be implemented efficiently, and if so how, and how their performance compares to alternative implementations and abstractions, needs to be shown. In addition, the results presented in this thesis do not evaluate the utility of Mu's abstractions to language developers, nor the performance effect of different mappings of higher-level language constructs to Mu. I aim for this work to encourage, and be a valuable aid to further such research, as well as further development and implementations of Mu

Zebu is still slow compared to LLVM, a high-performance well-funded mature state-of-the-art compiled tool-chain for static languages; further research is needed to improve this, as well as complete the implementation of Mu. However the performance of Zebu answers the research question.

1.5 Conventions

To aid in readability, code is often interspersed in text in a `monospace` font, with `teal` indicating types, `orange` variables, `maroon` literals, `blue` 'keywords', and `green` for comments. In addition `italics` is used for things that arn't known at compile-time or otherwise arn't a constant. A convention used in the pseudo-code presented is that lines starting with a '#' are to be evaluated at compile-time (i.e. when code is generated), and not run-time. I have adopted the convention (as has [ARM Limited 2013]) that stacks grow downwards, towards lower valued-addresses.

Chapter 2

Background

Virtual Machines can make language implementation easier, by offloading low-level, platform-specific and core runtime features to a layer of abstraction that can deal with these things in isolation from the higher level language features. The VM's intermediate representation (its 'machine code') then becomes the translation target for the high level language. This principle is quite common, especially in managed languages (ones with garbage collection); however VMs are usually tightly coupled with a specific language (or class of languages), are large, and complicated. Mu, being a micro virtual machine, takes a different approach.

2.1 Mu

Mu is a 'micro virtual machine', it is a managed, minimal, and language agnostic abstraction layer for language implementations [Wang et al. 2015; Wang et al. 2017; The Mu Micro Virtual Machine Project]. Mu, unlike large monolithic VMs (such as the JVM), does not have a standard library, and is not tied to a particular language, or class of languages, in addition it does not have a large runtime system, nor is it designed to perform heavy IR level optimisations. Mu, unlike some other VMs (like LLVM [Lattner and Adve 2004]), is specifically designed to be a specification, or 'design', rather than an implementation. This allows for diverse implementations, that focus on different things, as well as allowing the design to be focused more on idealised (and formal) semantics, and not be limited by implementation difficulties, or behaviour that particular ones might have. For this to work well however, some behaviour has been made *undefined* (e.g. having an uncaught exception, or out of bounds access to an array), *implementation-defined* (e.g. the layout of objects), and *conditionally-supported* (e.g. `tagref64`); this kind of approach is common amongst low-level programming languages to enhance performance (e.g. C++ [for Standardization 2014]), but less so in VMs (e.g. the CLI [International 2012]), due to issues such a security problems this can cause, however Mu leaves it up to clients to handle these.

2.1.1 Abstractions

Mu abstracts over three key areas, hardware, memory and concurrency. These where chosen due to their complexity, especially when interacting with each other, which imposes a significant burden on language implementers, despite their internals not being very language specific.

Mu abstracts over hardware, as do all VMs, by providing a hardware independent intermediate instruction set, with the aim of offloading hardware specific optimisations (such as combining multiplys and adds to a fused-multiply-add instruction). It also provides dynamic code generation (in the form of 'IR Bundles' that can be created at runtime), this is particularly useful for building JITs, such as those used to make dynamic languages efficient, In addition, it does not rely on, nor expose, specific hardware details (e.g. SIMD capabilities).

Mu abstracts over memory by only defining the structure of objects, not their layout, as well as providing allocation instructions (as with most other managed VM's). Memory created using Mu's instruction set is managed by Mu itself (i.e. clients don't destroy it, or specify how allocation or zero-initialisation is done), allowing implementations to automatically perform garbage collection. Automatic memory management is at the core of most modern managed languages, as it limits memory waste and helps eliminate bugs; however the general principle behind it (i.e. objects must live as long as all references to them) is often the same for these languages, so rather than have each language-implementer re-invent and re-implement memory management, Mu provides an abstract implementation-agnostic framework.

Mu abstracts over concurrency, by providing (call) 'stacks' (similar to fibres/contexts), as well as simple OS-independent threads. These features allow for simple and efficient implementation of higher-level concurrency features, like green-threads and coroutines; they are also particularly useful for OSR (on stack replacement) [Wang et al. 2017] (which is useful for dynamic language JITs for (de-)optimising running code), where a stack can swap to another one, so that the original one can be modified safety (due to it not being executed), in fact Mu provides primitives for doing this on non-executing 'stacks'.

Mu also provides a memory model (based on the C11 one [for Standardization 2011]) and associated atomic operations (like compare-exchanges and atomic-read-modify-writes), this is similar to the library features provided by many modern languages (like C++). Mu has a simple exception handling mechanism, which provides for throwing and catching references to heap objects; this allows various different language-level exception-handling techniques (e.g. finally blocks or catching only certain kinds of exceptions) to be handled by clients in terms of this language-agnostic model.

Though these abstractions are on their own already reasonably difficult to implement, their interaction is extremely challenging, especially if an efficient concurrent garbage collector is desired.

2.1.2 Interface

Mu is intended to be used in one of two ways: as a library used by a client written outside of Mu (by interfacing with Mu's C bindings defined in `mu_api.h`), or by code within Mu itself (using Mu's instruction set). In the former case, a 'client' starts a Mu implementation, and then creates and uses a `MuCtx` to interact with the running VM (e.g. to operate on Mu's memory or start new Mu threads). Code written in Mu itself, however, uses the Mu instruction set to interact with Mu.

Mu IR is grouped into bundles, which define top-level Mu entities (types, global cells, functions, function-versions, constants, etc.); a bundle can reference entities declared in itself, or any previously 'loaded' bundle. When a new bundle is needed, the client, using the C API or Mu instruction set (as applicable), creates a new `MuIRBuilder`/`irbuilderref` which maintains a Mu IR AST, new nodes are then created in it (in any order), and the bundle is loaded into the VM; once loaded, a bundle's declared entities can be used immediately.

A function-version (defined as a set of blocks of Mu instructions) is then executed whenever the associated function is called (assuming it is not redefined by another version), either by another function's version, or when a stack (such as a newly created one) with a frame created from the function is executed (such as by starting a new thread with it).

A 'boot-image' can be created by the C API, which dumps a set of 'white-listed' Mu-entities to a file, this can then be 'executed' (in an implementation-defined manner). Executing a boot-image will start a new-thread with a predefined 'primordial' function or stack. This allows for the building of libraries or executables in Mu, avoiding the need to re-compile them.

2.1.3 Prior Work

Mu was initially implemented as an interpreter in Scala, codenamed *Holstein* [Wang 2017], it was intended as a functionally complete 'reference' implementation, and so it did not in any way consider performance. Research has also been done in the utility of using Mu for modern languages, including a JavaScript [Wang 2015] and BF (Brainfuck) [Hall 2016] client, in addition a Haskell GHC backend [Hall et al. 2017] is currently being developed. Of particular interest is the PyPy client [Zhang 2015; Zhang et al. 2017], which implements a Mu AOT back-end to RPython (to complement the default C one); a Mu JIT back-end is also being developed (to complement the default assembly one). This client is the most complete, and up to date client, and is used by this thesis for correctness and performance testing.

2.2 Other Systems

The LLVM compiler framework and CLI virtual machine are two systems that at first glance may seem to be very similar to Mu, despite being very different from each-other. As these systems are state-of the art and share many features and design goals with Mu, they are useful and valuable to learn from.

2.2.1 LLVM

LLVM (Low Level Virtual Machine) is a compiler framework designed for unmanaged languages (such as C and C++) [Lattner and Adve 2004], unlike Mu it is not targeted for managed-languages. LLVM is used as a back-end for many languages, such as Swift, C/C++ (using Clang) and Rust (the language Zebu is written in, see § **3.1**). It provides many powerful, yet expensive optimisations, making compilation slower.

Though LLVM (and languages built on-top of it) is a common target for managed languages, its support is limited, in particular it does not provide an inbuilt garbage collector[1], it does not provide a JIT[2], and its exception handling features are complicated and targeted towards C++ like languages.

LLVM's SSA (single-static-assignment) form IR is mostly machine-independent and language-agnostic; it is known to allow for highly efficient code (in fact Clang, which uses LLVM IR, is one of the fastest C compilers); for these reasons, Mu's IR is based on LLVM's, with the hope that even with the addition of higher-level abstractions, it will inherit LLVM IR's quality.

2.2.2 CLI

The CLI (Common Language Infrastructure) ([International 2012]) is a managed-virtual machine designed to support various languages as diverse as Python, F♯, and C++, by providing the CIL (Common Intermediate Language), a stack-based IR, garbage collection, a rich type-system, and JIT capabilities.

The CLI is large and monolithic, it is heavily based on a particular implementation, Microsoft's CLR (Common Language Runtime), as well as being designed around specific kinds of languages (ones similar to C). In particular, the CLI provides a large *required* standard library, which together with its high-level and complicated type-system, is designed to allow interoperability between language libraries. The CLI focuses on language-security (preventing things such as out-of-bounds array access) and follows the 'compile-once run-anywhere' principle, by providing a byte-code that is then JIT'ed as needed.

[1]It does provide some IR level features to assist with integrating with a GC.
[2]Like any AOT compiler, it can in theory be used as a JIT, but doing so is very slow.

Mu, on the other hand, is designed to be minimalist, aims specifically for performance, and to be further language-agnostic. As such it does not provide the language-interoperability features of the CLI, nor does it have a byte-code (it is up to the Mu implementation to determine what form Mu boot-images take).

2.3 Summary

Though there is extensive prior work in VMs, work on micro virtual machines and Mu specifically is limited and still in its infancy. As the only current complete implementation of Mu, *Holstein*, was not in any way designed to be efficient, a new, distinct approach to building micro VMs is needed, such as that taken by Zebu.

Chapter 3

Zebu Architecture

The design of Mu requires implementations to not only provide a means of executing Mu IR, but provide runtime services, both to users of the C API and to code written in Mu IR itself.

Before I began work on this thesis Zebu (with an x86-64 back-end) was already implemented by Yi Lin, as such, most of the design and implementation of the high level architecture was already done by him. Unless otherwise stated, the majority of work described in this chapter is that of Yi Lin.

3.1 Zebu

Zebu[1] [Lin et al. 2017] is an open-source[2] library, which 'boot-images' and 'clients' link to. Boot-images use internal runtime functions, and clients use C API bindings to interface with Zebu. The Zebu library contains the internals for code compilation, the client interface implementation, as well as runtime support (e.g. exception handling routines). This approach is needed as almost everything that can be done in the C API can also be done in Mu code, in particular Mu code can dynamically create and load new Mu IR. To maximise performance, Zebu generates optimised machine code for the input Mu IR directly, as opposed to offloading to another VM or interpreting.

The modern systems programming language *Rust* [The Rust Project Developers 2017] was used due to its approach of 'zero cost abstractions', which provide high level abstractions (e.g. generics and virtual methods) and compile-time safety checking (e.g. 'borrow' checking), whilst maximising efficiency by making most of these features statically checked. Rust also provides low level *unsafe* features (like pointers) that make it a better alternative than C for low-level programming (such as direct memory modification and interfacing with assembly code). This makes Rust a prime choice for high-performance VM implementation, as well as the VM's runtime (e.g. garbage collection [Lin et al. 2016]).

[1]The version discussed in this thesis is the one on the '`isaacs-thesis`' Git branch.
[2]Licensed under the Apache License, Version 2.0

Zebu defines a C thread-local variable[3], `mu_tls`, that contains a pointer to a `MuThread` struct instance. The `MuThread` type contains information about the current thread, including its current stack (see § **4.5**), it also points to a global instance of the `VM` struct. The `VM` struct contains state about the running Mu VM, such as information about all the Mu IR entities, the vm's options, and exception handling information.

Code emitted by Zebu is position-independent, as such accessing the `mu_tls` variable is non-trivial to do within assembly (doing so requires calling a 'tlsdesc' function loaded from memory, and using the returned value to compute an offset from the value of the `TPIDR_EL0` system register). As the `mu_tls` variable is used often[4] and its value is constant within a thread, a callee-saved register (`X28`) is assumed to always contain the `mu_tls` value. The `mu_tls` register only needs to be updated upon thread creation (this is done by calling into a C function that returns the value of `mu_tls`); since it is a callee-saved register, any calls to C or Rust functions will preserve its value, in addition, code generated by Mu never modifies it, thus respecting the ABI.

3.1.1 API Implementation

The Mu C API is implemented in C compatible Rust code that forwards to proper Rust code (e.g. by wrapping C pointers & array size arguments into a `Vec`). The IR builder interface is implemented by creating a temporary representation of the IR, and then when `load` is called, creating Zebu AST nodes and loading them into the `VM` struct[5]. The other parts of the API that are implemented are done by calling simple methods on the `VM` struct.

3.1.2 Compilation

When a function-version[6] is compiled, its AST is passed through several passes that perform operations such as inlining functions, rearranging the AST into a tree form, and generating Phi blocks (blocks with the `MOVE` pseudo-Mu instruction, which sets one SSA variable to another). Once these high-level passes have finished, it then goes through a back-end specific instruction selection phase which generates the actual assembly-code[7] for each Mu instruction (see § **4**), afterwoods it performs the back-end independent passes of register allocation, peep-hole optimisations and code-emission (i.e. emitting the assembly code to a file).

[3]Rust's thread-locals are slow as they are implemented using a map.
[4]it is also planned to be used for GC allocation, whose performance is critical
[5]These parts where primarily implemented by Kunshan Wang, an ANU PhD student.
[6]A particular implementation/version of a Mu function.
[7]We plan on adding another 'back-end' to instruction selection that generates machine-code directly, for use with JIT.

3.1.3 Boot Image Building

When a client calls the `make_boot_image` function on a `MuCtx`, Zebus compiles the Mu IR entities, given in the `whitelist` argument, to a native executable or dynamic library. It compiles all the function-versions for white-listed functions, and records the primordial function (if given) in the current `VM` struct instance, it then emits a file called `context.S`. The `context.S` assembly file contains the state of the Mu heap, as well as the `VM` struct instance.

The heap is dumped by traversing each object reachable from a global cell (including the global cell's contents), and dumping it, giving each one a label (and a label with the mangled name for global cells), replacing references to objects to assembly labels (see **Figure 3.1**). The `VM` struct instance is also dumped in a similar manner (using the `RODAL` Rust library [Isaac Oscar Gariano 2017]), by traversing the needed Rust objects referenced by the `VM`, unneeded ones are dumped with their minimal default values (like an empty map for a `HashMap`). This is needed as a lot of the information in `VM` struct is either used by the runtime or potentially usable by the client (e.g. if it wants to load more IR bundles referencing previously declared entities)[8]. This approach however does increase boot image building time, and prevents linking multiple Mu boot images together (due to having multiple declarations of the `VM` struct[9]), however it could allow self-modifying boot images (i.e. during runtime that load new Mu IR, and then call `make_boot_image` again[10]).

Zebu then executes Clang to assemble-and link together (as position independent code) the `context.S` file, the `main.c` file (if it's being compiled as an executable, see **§ 3.1.4**), the assembly files for each compiled function version, and Zebu itself (dynamically, by default).

3.1.4 Boot Image Execution

When a boot image is executed, it starts at the C `main` function defined in `main.c`, which overrides the C standard library `free` and `realloc` functions (to handle the case where Rust calls these on an object dumped in `context.S`)[11], it then calls a rust function `mu_main` that loads the dumped `VM` struct (performing some initilisation) and starts a new thread for the primordial function (with the primordial thread-local). The `mu_main` function then waits (by joining) until there are no Mu threads running, and then exits the program.

[8]These features are not properly implemented, and have not been tested as they arn't currently needed.

[9]We can't simply merge them, as Mu entities may have distinct ID's and the client made the boot image depend on these values (e.g. if it stores an ID on the heap).

[10]This won't actually work yet as the new boot image will only contain the code for newly defined functions, and their is no `make_boot_image` instruction yet.

[11]an alternative, and more portable approach, is to use Rust's *unstable* custom allocator feature

```
.typedef Node = struct<int<8> ref<Node>>
.typedef List = struct<int<64> ref<Node>>
.global my_list <List> = {1, .new<Node>{2, 'NULL'}}
```

(a) Pseudo Mu-IR & HAIL

```
    …
    .balign 8
GCDUMP_0_0xFFFFAAA10008:
    .byte 0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0
    .xword 0

    .balign 8
    .globl __mu_my_list
__mu_my_list:
GCDUMP_1_0xFFFFAAA00008:
    .byte 0x1,0x0,0x0,0x0,0x0,0x0,0x0,0x0
    .xword GCDUMP_0_0xFFFFAAA10008
    …
```

(b) `context.S` file excerpt

Figure 3.1: Heap dumping example of a sized linked list equal to `[2]`.

3.2 Limitations

Though Mu is a 'micro virtual machine', its feature set is large and implementing it is not easy. Some features it provides are complicated, some are currently unused by clients, and others are not particularly relevant to the problem of generated code performance.

In particular, I have not yet implemented the following features: vectors, SSA variables of array or struct type, function exposing, function redefinition, atomic read-modify-writes, futex instructions, watchpoints, traps, OSR features (frame cursors and 'keep alive's), dynamic top-level-definition lookup and IR-loading, as well as exceptions for division by zero and null reference/pointer accesses. The only properly working client we currently have, an RPython AOT-compiler [Zhang et al. 2017], does not use these features, so we have not chosen to implement them yet. Where relevant however, this thesis will discuss how the implementation of missing features could affect the implementation and/or performance of current features, as well as how they could be implemented.

In addition, some other features have not been implemented in the x86-64 back-end, but the AArch64 back-end, which is the primary focus of this thesis, implements a superset of the x86-64 back-end's features. The garbage collector has not been fully implemented yet, so it has been disabled, instead I implemented allocation with glibc's `calloc` function.

Chapter 4

The AArch64 Back-End

The implementation details of the AArch64 back-end are of crucial importance to performance, in addition, parts of this implementation were quite interesting and challenging, especially due to the high-level abstractions provided by Mu.

The instruction set of Mu provides the operations expected in even the most low-level imperative programming language or VM, whilst also providing instructions that interact with the VM runtime, which provide the core abstractions of Mu. Most of Mu's instructions are not unusual, but they are essential for any practical VM, and are the most commonly emitted instructions by high-level languages, as such their implementation is crucial to performance. However, Mu's high level instructions are typically implemented as runtime calls into Zebu. This chapter describes how the AArch64 back-end implements the Mu instruction set, both in the code it generates, and the Zebu runtime functions it calls.

The AArch64 implementations for most Mu instructions where derived from the x86-64 back-end (implemented by Yi Lin), however I have heavily modified them to optimize more, and to work with the AArch64 architecture. I have completely rewritten exception handling (it was previously extremely slow, and relied heavily on `dlsym`), I have also added support for Mu's threads and stacks. Yi Lin has updated the x86-64 back-end to reflect these improvements, in a very similar way to the AArch64 implementation presented here.

In general, the approach for both architectures has been to follow the platforms ABI conventions as close as possible (for AArch64 this is the ARM 64-bit, System V Revision 4 variant, procedure call standard [ARM Limited 2013]). This is done to maximise compatibility with C code (allowing Mu functions to be called as if they are C functions), as well as making it easier to integrate with existing debuggers.

4.1 Type System

The implementation of Mu's type system is crucial and fundamental to the AArch64 back-end. Mu provides a wide variety of parameterised fundamental types, in particular it provides integer[1], floating-point, unmanaged-pointer, managed-reference, struct, hybrid, array, vector, 'opaque' reference types and a 64-bit tagged-reference type. Implementation of these types (i.e. their size, layout, alignment, register allocation etc.) is the same as the corresponding type in the ABI, if there is one. The opaque-reference and managed-reference to object and function types are implemented as if they were unmanaged pointers. Hybrids are treated like C style structs with a flexible array member (i.e. the Mu type `hybrid<`T_1`, …, `T_n`, Y>` is implemented like the C type `struct{`T_1`; …; `T_n`; Y[];}`). The 64-bit tagged-reference type, tagref64 can hold a double precision floating point value, a tagged managed-reference, or a 52-bit integer, it is internally treated like a 64-bit integer (this was chosen over treating it like a 'double', since most operations on it heavily involve integer arithmetic). The vector types are currently unimplemented, but when they are they will probably be implemented as machine-specific SIMD vector types. The currently implemented 'opaque' reference types are merely pointers to their associated Rust data structures (`stackref` as a `*mut MuStack`, `thread` as a `*mut MuThread` and `irbuilderref` as a `*mut CMuIRBuilder`).

AArch64 has 64-bit registers, with instructions that operate on the full 64-bits, or the lower 32-bits (usually reading & setting the upper 32-bits as zero); however Mu requires support for other integer sizes, specifically 1, 8, and 16-bits. To accommodate this, we assume that the upper unused bits of a register always have undefined values and can be freely overridden, this is the approach the ABI takes; if an operation requires the upper bits to have a specific value (e.g. a signed division would want them to be copies of the sign bit), it will modify them accordingly.

4.2 Basic Instructions

Some of Mu's low-level instructions required some care and thought in implementing, others however where straightforward and simple. Most of the instructions, including arithmetic, comparison, addressing, conversion, and branch instructions where implemented in a standard way. Memory operations (loads, stores, compare-exchanges and fences) are implemented in the same way as Clang (with the `-O3` and `-fPIC` flags) implements the equivalent C++ (using the standard library for atomics), these are mostly implemented with single machine instructions, however compare-exchanges require a loop, as do atomic loads and stores of 128-bit integers. Heap memory allocation (`NEW` & `NEWHYBRID`) is implemented as a simple call to the C standard `calloc` function[2]. Stack memory allocation (`ALLOCA` & `ALLOCAHYBRID`) is implemented by simply growing the stack by the appropriate amount (as well as a call to the C standard `MEMSET` function to perform the required zeroing). The `uvm.native.pin`, `uvm.native.unpin`, and `uvm.native.get_addr` intrinsics [3] are

[1]Not all are required for implementations, currently only 1–64-bit and 128-bit integers are supported in the AArch64 back-end.

[2]A bump-pointer based allocator was previously used, but it was unexpectedly slow.

[3]Referred to as "common instructions" in the Mu specification.

implemented as a simple move, while the (non-moving) GC is disabled in the current prototype.

Arithmetic instructions for sizes other than 32 and 64-bits where non-trivial to implement, especially since Mu supports returning flags from instructions. In particular, appropriately extending the operands and/or results was needed for some operations and flags (e.g. signed-division required the operands to be sign-extended), the carry and overflow flags for addition & subtraction also required some logical machinery to compute. The AArch64 architecture only provides flags for its addition, subtraction, logical-and, and logical-and-not instructions, so other instruction's flags needed to be computed with a comparison, in particular the carry and overflow flags for a Mu multiply, require the computation of the upper part of the multiplication first.

4.3 Functions

All Mu code is contained in functions, so they are at the core of Zebu's instruction generation. Mu functions are very similar to LLVM functions, so they have been implemented accordingly, they have a *prologue* and *epilogue*, whose behaviour is in accordance with the ABI. Zebu's Mu IR passes replace the Mu RET instruction with a branch to the epilogue block.

The prologue pushes a 'frame-record' (frame-pointer[4] & return-address[5] pair) to the stack, sets the frame-pointer to point to this, grows the stack by a fixed amount to accommodate spills, saves all used callee-saved registers to the stack, saves `XR`[6] (if used), and unloads all the arguments. The 'epilogue' reverses this, it loads all the return values, restores the stack-pointer (to a fixed offset from the frame-pointer), pops the callee-saved registers, pops the frame-record, and returns. This approach is a straightforward yet naïve implementation of the calling convention, in particular:

- It always pushes a frame-record, even if the function is a leaf.
- It does not allow for the frame-pointer to be used for other purposes, if its unneeded.
- It always loads arguments passed on the stack, even if never used, or doing so would result in another spill.
- Stack argument space is only ever used to initially load arguments, it is not used as temporary storage space.
- It always saves and restores callee-saved registers, even if they are not used in every invocation of the function.

However, Zebu's back-end and register allocator are not implemented in a way that would make fixing these problems simple, so it has been left for future research to

[4]I.e. register X29.
[5]The value in the link-register, i.e. X30.
[6]The indirect return location register, an alias for `X8`.

see what performance improvement would be gained by doing so.

Calling a (C or Mu) function is done by first evaluating the reference to the function to be called (if not a constant), then (if needed) reserving space on the stack for the return value (storing the location into `XR`) and argument values (if needed), loading the arguments into registers and the stack, and then calling the function (with a `BL` or a `BLR` instruction). Once the new function returns, the return values are loaded (from registers and/or the stack), and the stack-space allocated before the call is deallocated.

The calling convention defined in the ABI does not support multiple return values (since C doesn't), yet Mu does. To overcome this, if there is more than one return value, the function is implemented and called by wrapping the return values into a struct, and returning that intsead. This is the same thing that Rust does when marking a function returning multiple values as `extern "C"`. This approach does however have several drawbacks:

- When returning multiple-values, if the total size (after padding) is greater than 16-bytes (and it is not a HFA, 'homogeneous floating point aggregate'[7]) it will return them on the stack, even if they could all fit in multiple argument registers.
- When returning multiple values of total-size less than 16-bytes (and not a HFA), the results are packed into registers, which need to be combined and extracted (e.g. returnning two 8-bit integers would pack them into the lower 16-bits of a register, but Zebu treats each register as containing a separate SSA values).
- When returning multiple values in registers (when not a HFA), floating point values will be placed in integer registers.

A better approach would be to return values in the same was as they are passed (but this would break the calling convention).

Tail-calls are implemented in the normal way, the reference to the function to call is evaluated, the arguments are loaded (in registers or on the current functions stack argument space), the value of `XR` is restored, callee-saved registers are restored, the frame record is popped, the stack-pointer is restored to what it was on entry, and the function is branched to (with a `B` or a `BR` instruction). Unfortunately, if the current function has insufficient stack argument space to hold the tail-call's arguments, this approach will not work; the following are two possible solutions:

- Increase the stack argument space, however when the tail-called function returns to the current functions caller, the calling convention requires the stack-pointer to be the value it was when the current function was called. Changing the calling convention to instead require functions to pop all of their stack-arguments before returning would solve this problem, and allow tail-calls to be implemented properly in all cases.
- Implement the tail call like a normal call followed by a return, this limits ef-

[7]Defined by the ABI as an aggregate (struct or array) consisting entirely of 1–4 floating point values of the same size.

ficiency and makes frame iteration more complex (as the Mu specification requires tail-calls to not create a new frame, but a call would, i.e. it would need to skip a frame when iterating).

Overall, the implementation of functions has been complicated, and made less efficient due to the strict adherence to the ABI; however there is a lot of optimisation opportunities that could be taken, without breaking ABI compatibility.

4.4 Exceptions

Mu's exception handling is simple, yet powerful; Zebu's implementation is both highly performant and uses only minimal code. Exception handling has been implemented based on the so-called 'zero cost' principle, i.e. code that potentially throws an exception should not suffer from a performance penalty if no exception is thrown, at the expense of greater overhead when if one is. This has been achieved through the use of a '*callsite*'-table that stores static information needed to find catch blocks and unwind the stack, the only code that then needs to be executed at runtime, is the code to throw and catch the exception itself, as opposed to having to explicitly check whether an exception was thrown after each call (this is the approach RPython's C back-end uses).

The *callsite*-table is a table mapping the return address of a potentially exception instruction[8], to the address of the associated exceptional destination[9] (if there is one), the size of stack arguments (since the exceptional destination expects the stack-pointer to have the same value as if the call returned normally), a reference to a table identifying the location on the stack of all saved callee-saved registers (there is one such table per function-version, the locations are relative to the frame-pointer), and the Mu ID of the function version (used only for printing of back-traces). Some of this information (specifically the callsite's and exceptional destination's addresses) is only known when the associated code is loaded into memory, as such a slightly different version of the table is created at compile-time and the real table is computed (using the `dlsym` function) at run-time (boot-image startup).

When an exception[10] is thrown, the runtime assembly function `muentry_throw_exception` is called, this function pushes a frame record, and all the callee-saved registers to the stack. The exception and the base of the callee-saved area (pointing to the frame-record) is then passed to the `throw_exception_internal` Rust function. The `throw_exception_internal` function 'walks' the stack (by using the frame-records) untill it finds a callsite with an exceptional destination (using the callsite-table), in which case it calls the `exception_restore` assembly function. If it doesn't find a catch block, it 'unwinds' a frame by loading all of the frame's stored

[8]Currently this is the `BL`/`BLR` or `BR` instruction used to implement the `CALL` or `SWAPSTACK` Mu instructions.

[9]I.e. catch block.

[10]A `ref<void>` value

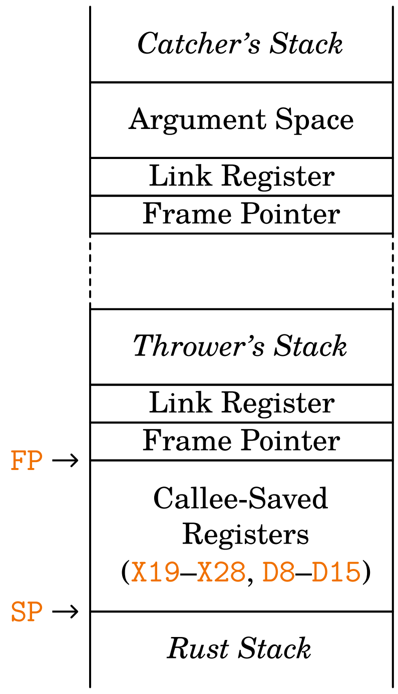

Figure 4.1: Stack layout for exception throwing

callee-saved registers (using the callsite-table to locate them) and places them into the frame-cursor[11], as well as updating its frame-record (see **Figure 4.2** for details). The `exception_restore` function restores the saved callee-saved registers from the frame-cursor, pops the frame-record preceding the frame-cursor, updates the stack-pointer to the value given by `throw_exception_internal`, and branches to the exceptional destination. This whole process works due to the layout of the stack (see **Figure 4.1**).

The exceptional-destination loads the exception through the mu_tls variable, and then executes the client specified code to handle the exception.

4.5 Stacks

One of the most powerful and interesting Mu features is stacks (these are highly usefull for OSR [Wang et al. 2017]), though similar features have been implemented before ([Dolan et al. 2013]), Mu's are interesting due to both being low-level, and yet supporting exceptional continuation. A 'stack' (in the sense of the Mu specification) is an 'execution-context'; it abstracts over the state of a function, it contains all the live SSA variables and 'alloca' cells, and function call information, as well as an 'instruction pointer'.

A stack in Zebu is represented by a Rust struct, `MuStack`, which contains handles

[11]The passed in pointer to the temporary frame-record & callee-saved register space.

```
throw_exception_internal(exception, frame_cursor):
    mu_tls→exception = exception

    // Acquire a read lock, blocking if a thread is modifying
    // the table (e.g. if we are JITing)
    callsite_table = READ-LOCK(
        mu_tls→vm→compiled_callsite_table);

    current_frame_pointer = frame_cursor
    (previous_frame_pointer, callsite) = *current_frame_pointer

    LOOP:
        IF previous_frame_pointer = 0 ||
                callsite ∉ mu_tls→vm→callsite_table:
            print_bactrace()
            // Fatal error, we have reached a native frame, or
stack-bottom
            PANIC

        callsite_info = callsite_table[callsite]

        IF callsite_info.exceptional_destination != 0:
            catch_address = callsite_info.
                exceptional_destination
            stack_pointer = current_frame_pointer + 16 +
                callsite_info.stack_args_size
            // will not return
            exception_restore(
                catch_address, frame_cursor, stack_pointer)

        // Restore callee-saved registers
        FOR (target_offset, source_offset) ∈
                callsite_info.callee_saved_registers:
            *(frame_cursor + target_offset) =
                *(previous_frame_pointer + source_offset)

        // Move up to the previous frame
        current_frame_pointer = previous_frame_pointer;
        (previous_frame_pointer, callsite) =
            *current_frame_pointer

        // Update the frame-record
        *frame_cursor = (previous_frame_pointer, callsite)
```

Figure 4.2: Pseudo-code for `throw_exception_internal`

to ‘mmaped’ memory as well as a saved stack pointer value. The stack pointer stored here is not kept up to date whilst the stack is active. A stack has a constant, fixed size (currently 4 MiB) together with two fully protected pages before and after the stack area, so that a stack overflow can be detected when either page is read or written two (this will cause a segmentation-fault). However, stack overflows are not in anyway exposed to language clients, as there is nothing in the Mu specification referencing them, instead Zebu will simply terminate if there is one. This is a significant limitation, as clients have no way of communicating to Zebu how large stacks need to be, this limits what can be implemented in Mu, whilst also wasting memory. A better solution might be to resize stacks (and move them around if necessary), whenever the stack overflows, however this will not be simple to implement, especially due to the possibility of having things outside of the stack referencing things inside the stack (e.g. the result of an `ALLOCA` may be stored in a global cell).

The top of a stack that is ready to be activated (i.e. stack that is newly created, or previously ‘swapped’ out of) contains a frame-record, the stored frame-pointer will point to the base of the previous frame (if there is one), which will be somewhere in the lower part of the stack. Arguments (those that arn’t allocated to registers) are passed in the area before the frame-record, in the same way as normal function calls. This layout (see **Figure 4.3**) tries to minimise the amount of data saved, as well as respect the calling convention. Swapping to a newly created stack will call a function, which may expect arguments on the stack in the normal way, but otherwise the layout of the stack is unspecified by the ABI[12]. In the case of a stack that is swapped out of (i.e. not a brand new one), any state that needs saving (except the frame and stack pointers), must be explicitly saved by it on the stack (the register allocator will handle this by generating spills). This simple stack layout means that code, such as a `SWAPSTACK`, does not need to know or care how the stack came to be in its current state, whether it was newly created, or recently swapped out of.

The currently executing stack is stored in a field accessible through the mu_tls thread local.

4.5.1 Stack Creation & Destruction

Stacks are created with the `uvm.new_stack` intrinsic, that takes a reference to the function to be executed when swapped-to. I have implemented this as a call into the Rust runtime that takes the function, and total size of stack arguments (computed at compile-time based on the functions signature). The runtime function dynamically creates a new `MuStack`, initialising it by ‘mmaping’ memory for the new stack, reserving space for stack arguments, pushing the address of the function, and a null frame-pointer to the new stack, it then returns a pointer to the new

[12]AArch64 does not push the return address on the stack, it is passed in the link-register (X30).

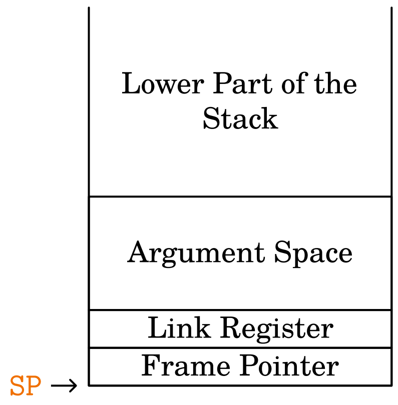

Figure 4.3: Layout of a stack that is ready to be activated (if the stack is brand new, the lower-part will be empty)

stack (the lifetime of this is not managed by Rust). When a stack is swapped-to (see § **4.5.2** or a thread-created with it (see § **4.6.1**), the link-register is explicitly set to 0, so that if this stacks function tries to return[13] (i.e. by branching to this register), Zebu will 'segfault'.

The `uvm.kill_stack` intrinsic and the `KILL_OLD` clause in the `SWAPSTACK` instruction destroy a stack by calling into the `muentry_kill_stack` Rust runtime function. The `muentry_kill_stack` function merely deallocates the referenced stack, and lets Rust do any other clean-up (such as deallocating the stack's memory), this works as `MuStack` implicitly implements the Rust `Drop` trait. As the `MuStack` is allocated in Rusts' heap, the GC cannot automatically free it when it becomes unreachable (i.e. to prevent memory leaks the client is expected to specifically kill stacks). If we were to instead allocate it on the GC's heap, we would have to implement a 'finalizer' like mechanism to call clean-up code (the `Drop::drop` function), however there is no such feature of Mu.

4.5.2 Stack Swapping

Stacks can be swapped using the `SWAPSTACK` instruction:

```
(r₁, …, rₙ) = SWAPSTACK stack_new
              (RET_WITH<T₁, …, Tₙ> | KILL_OLD)
              (PASS_VALUES<Y₁, …, Yₘ>(a₁, …, aₘ) | THROW_EXC(e))
```

It is roughly implemented by evaluating each argument, updating the `mu_tls→stack` field, swapping-out of or killing the current stack, updating the stack pointer, and branching or throwing an exception. For details see **Figure 4.4**.

To kill the current stack, a call to `muentry_kill_stack` (see § **4.5.1**) is made, how-

[13]This is undefined behaviour according to the Mu specification.

ever it needs to be made on the new stack, since it will un-‘mmap’ the old stacks memory, since Rust functions may freely access their own stack.

To swap out of the current stack, stack space is reserved for the return values (the same ammount of space that would be used when *passing* arguments of type T_1, ..., T_n to a function), a resumption address (the address of the first instruction after the branch/exception throw to the new stack), and the frame-pointer are also pushed to the stack.

To pass values to the new stack, arguments are passed using the normal calling convention, except that stack arguments are passed starting bellow the frame-record at the bottom of the stack (at the stack pointer, plus 16). Since argument registers are defined as ‘temporary’ (i.e. not callee-saved) registers by the ABI, the call to the Rust `muentry_kill_stack` function may not preserve this value. Luckily, there are sufficient callee-saved GPRs and FPRs for the arguments to be moved into these before the call, they are then moved to the argument registers after the call.

A `BR` instruction is used to branch to the new stacks code, as opposed to a `BLR` or `RET` instruction, since those confuse the branch predictor (and causes the `SWAPSTACK` to be slower).

Throwing an exception to the new stack (when there is a `THROW_EXC(e)` clause) is done by reserving space for callee-saved registers, saving the `mu_tls` value in this space, and passing the base of this area and `e` to `throw_exception_internal`. This is similar to how exceptions are thrown (see § **4.4**), however there are no callee-saved register values that need to be preserved, since a newly created stack dosen’t have any state it needs saving, and a swapped-out of stack saves all the state it needs itself.

Rather than saving all registers on the stack before swapping, the `BR`/`B` instructions that branch to/throw to the new stack are marked as overwriting all machine registers. This forces the register allocator to spill-store any (normal) registers whose values are to be used when the stack restores. This differs from a normal call instruction, `BLR`/`BL`, which defines only non-callee-saved registers.

This swap-stack implementation was designed to allow for high-performance stack swapping, it does so by allowing the register allocator to only save things that are needed after the swap, and minimises instructions and calls to run-time functions, similar to that suggested by [Dolan et al. 2013].

4.6 Threads

Many systems have threads, but Mu’s are tightly coupled with stacks, and so their implementation needs to match, making Zebu’s implementation somewhat differ-

```
#for i = 1, …, m:
    evaluate $a_i$

$stack_{old}$ = mu_tls→stack
mu_tls→stack = $stack_{new}$

#if there is a RET_WITH clause:
    SP -= SAS<$T_1$, …, $T_n$>
    LR = ADR(&callsite)
    Push-Pair(LR, FP)
    $stack_{old}$→sp = SP

SP = $stack_{new}$→sp

#if there is a PASS_VALUES clause:
    place all stack arguments in the normal way,
        (starting from SP+16 upwards)

#if there is a KILL_OLD clause:
    place all register arguments in corresponding callee-saved
        registers
    BL muentry_kill_stack($stack_{old}$)
    move callee-saved registers to argument registers
#else there is a PASS_VALUES clause:
    place all register arguments in the normal way

#if there is a THROW_EXC clause:
    $SP_{old}$ = SP
    SP -= 144 // Total size of callee-saved registers
    // This call will not preserve any registers
    B throw_exception_internal(e, $SP_{old}$)
    callsite: ;
#else there is a PASS_VALUES clause:
    LR = 0 // Segfault if it tries to return through this
register
    (FP, resumption) = Pop-Pair()
    // Execute the new stacks code
    BR(resumption) // This call will not preserve any registers
    callsite: ;

#if there is a RET_WITH clause:
    #if i in 1, …, n:
        $r_i$ = load argument i from stack or register
    SP += SAS<$T_1$, …, $T_n$>
```

Figure 4.4: 'Pseudo-Assembly' for SWAPSTACK implementation.
Where SAS<T_1, …, T_n> is the total size (in bytes) of stack arguments to be passed back to the current stack (i.e. the size of stack arguments of functions taking arguments of type (T_1, …, T_n)).

ent from others. Mu threads abstract over OS-threads, a thread will start executing the code of a given stack when created, and will continue executing code (possibly swapping to different stacks) until the thread exits. Currently the support for threads in the Mu specification is limited, they can only be created and exited (but only from within themselves), but they do have thread local state. Major features that are missing include the ability to join[14], cancel (i.e. kill them from another thread), and suspend and resume[15] threads.

The user specified thread-local [16] is got/set by clients, by loading/storing through the `mu_tls` thread-local variable.

4.6.1 Thread Creation

Threads are created using the `NEWTHREAD` instruction:

```
    thread = NEWTHREAD stack [THREADLOCAL(threadlocal)]
             (PASS_VALUES<Y₁, …, Yₘ>(a₁, …, aₘ) | THROW_EXC(e))
```

(if the `THREADLOCAL` clause is absent, the value of `threadlocal` defaults to a null pointer).

A `NEWTHREAD` instruction with a `THROW_EXC` clause is implemented as a call into a Rust function which calls `mu_thread_launch` (passing in the `stack`, `threadlocal` and `e` values). The `PASS_VALUES` implementation is similar, except it does not pass an exception argument, and it passes the arguments a_1, …, a_m, to the new stack. Stack arguments are passed below `stack`'s stack-pointer and register arguments are placed above it (note: space is first reserved for all argument registers, even unused ones); see **Figure 4.5**. All register arguments are passed through the stack, so their values can be preserved across the call into Rust code, so a different internal thread-entry function is not needed for each function signature.

The `mu_thread_launch` Rust function starts a new Rust thread, whose start function calls the `muthread_start_normal`/`muthread_start_exceptional` assembly functions, passing them the stack-pointer of the `stack`, the address to store the native stack-pointer to, the new value of `mu_tls` and (if applicable) the exception to throw to the new-thread. The `muthread_start_normal` & `muthread_start_exceptional` functions swap from the native stack to the given Mu stack, by pushing a frame-record, storing the stack-pointer to passed in location, and setting the stack-pointer to the passed in value. After swapping stacks, the `muthread_start_normal` sets the `mu_tls` register to the passed in value, pops all the argument registers from the stack, pops a frame-record, sets the link-register to null, and returns (to the just popped return address). The `muthread_start_exceptional` function on the other hand, allocates space for a frame-cursor, storing the `mu_tls`

[14]One could emulate this using some kind of global status indicator in global cell for each thread.
[15]This can be emulated by stack-swapping, then thread-exiting to save the state of, and suspend the thread; resumption can then be emulated by creating a new thread from the suspended stack.
[16]A `ref<void>` value.

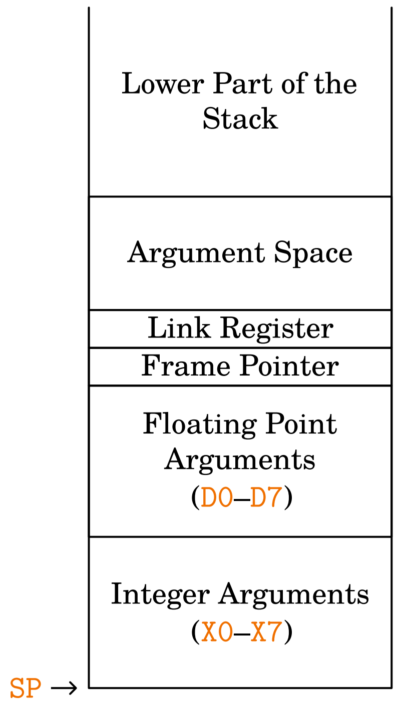

Figure 4.5: Layout of a thread's stack before it is started

value into it, and branching to `throw_exception_internal`, similar to how exceptions are normally thrown (see § **4.4**).

4.6.2 Thread Exiting

A thread can exit itself with the `uvm.thread_exit` intrinsic, this loads the native stack-pointer from the `mu_tls` variable and passes it to the `muentry_thread_exit` assembly function. The `muentry_thread_exit` function implements a simple swap back to the native stack, it restores the stack-pointer to the passed value, pops callee-saved registers from the native stack, pops the frame-record and returns (to the native-stacks code).

4.7 Other Instructions

4.7.1 Tagref64

The `tagref64` intrinsics (checking the type of the value held, extracting the held value, and creating a new instance) are implemented in the same was as the Scala implementation in Holstein, with the operations merely converted directly to assembly (by translating to C++, and then using Clang).

The `tagref64` type is quite limited, and not particularly useful to clients. It would be more useful to have a tagged reference type together with the ability to create unions of arbitrary types (like C unions, but GC aware). However such an approach could cause data races in concurrent environments (especially with a concurrent GC), and so will need to be designed and implemented with care.

4.7.2 Metacircular Client Interface

One of the most powerfull features of Mu, is that ability to generate code at runtime, this feature is directly a part of the instruction set with the metacircular client interface. This interface provides intrinsics that perform essentially the same operations as various things in the C API, allowing a Mu client to be written in Mu itself, and operate on the current VM (if one where to use the C API from within Mu, they would be creating a new Mu VM).

I have implemented the Metacircular IR-Builder interface by transforming each instruction in the ir-building phase to an equivalent `CCALL`, that calls the implementation of the corresponding C API function. The only special case, was that of the `@uvm.irbuilder.new_ir_builder` instruction, which created a `*mut CMuVM` from the running `VM`, obtained from the `mu_tls` thread-local variable. This was then used to create a new C `*mut CMuIRBuilder`, this marshalling is probably unnecessary, but removing it would likely require some non-trivial rewrites to the API's implementation.

4.8 Optimisations

Zebu implements a small and simple, yet effective set of optimisations, most of these are ingrained directly in the back-end, eliminating the need for separate passes, and thus reducing code-complexity and build-time. Imediates are used as machine-instruction operands whenever possible, in addition when Zebu generates instructions internally to perform computations (such as computing the size of an object before allocating), the back-end will generate minimal and efficient code, by performing constant-propagation, and transforming complicated arithmetic to simpler ones (e.g. emitting shift's instead of divisions). However, arithmetic given directly in Mu-IR is not optimised by the back-end if unnecessary, rather it is held to be the responsibility of the 'client', or higher-level compilation passes. Another minor optimisation performed is manual zeroing with `ALLOCA`'s that allocate at most 64-bytes (as opposed to doing a more-expensive `memset` call).

There is a small and simple peep-hole optimisation pass which elides redundant assembly moves and branches. In particular, if a branch branches to another branch, or to the following instruction, it is removed.

4.8.1 Memory Addresses

For efficient code generation, instead of generating assembly code directly for memory addressing instructions, an internal Rust address structure is maintained to describe addresses. This structure contains `base` and `offset` (an SSA variable or constant) values a `scale` (a constant) and flag indicating whether the offset is `signed`; this represents the address with value `base + offset×scale`. The `scale` is usually 1, it is used to contain the size of shifts when indexing arrays (using the `GETELEMIREF` or `SHIFTIREF` instruction). The `base` value and `offset` is re-computed, by emitting simple arithmetic instructions, as needed (e.g. if using `GETELEMIREF` or `SHIFTIREF` whose element size doesn't equal the current `scale` value). If the address value is used for a memory-assembly instruction (load, store or pre-fetch) it is converted to an optimised assembly address operand (based on the address modes supported by the instruction), which will potentially cause instructions to be emitted to do the computation (e.g. if the `offset` or `scale` values are outside the range supported by the instruction). If the address is used in any other assembly instruction, its full value is calculated into a register.

4.8.2 Comparisons

Comparison operations work on a two-stage principle, first the comparison instructions are emitted (which set processor flags), then the comparison-operation's result value is set acordingly. The second stage is omitted when operations can be performed directly with respect to the processor's flags (i.e. they don't need to examine a value in a register), this is done with Mu select and condition-branch instructions, which emit simple AArch64 conditional-select and branch instructions.

4.8.3 Missing Optimisations

There are many more optimisations that can be performed, identifying them, choosing the appropriate ones, and manner in which to implement them, is not an easy task. I have identified several optimisations that are simple to do, but are not implemented yet:

- Using the zero-register whenever an instruction operand is zero, this is already done sometimes, however not all instructions, and instruction-variants, support the zero-register as an arbitrary argument. To simplify implementing this optimisation, it might be best to emit moves from the zero-register to the operand's register, and then allow it to coalesce, however this will not-work with the current version of the register allocator.
- Combining load/store instructions that operate on adjacent memory locations, into a single load/store (e.g. combining tow 64-bit loads with offset 0 and 8 (respectively), into a single 128-bit load to two-registers, with offset 0), this should probably be done in the peep-hole phase (to allow it to work on assembly-instructions generated from unrelated Mu-instructions), however it will need to be done carefully, as to not violate the conditions of Mu's memory model.

- AArch64 provides some branch-instructions that perform comparisons on registers internally (as opposed to needing preceding compare instructions), these check if the register is zero/non-zero, or check whether a subset of their bits are all-set/not-all-set. There are a moderate set of of possible Mu comparisons that could utilise these (e.g. various comparisons with 0, or checks against −1), so pattern matching them may not be simple to implement.
- AArch64 conditional branch instructions can check against arithmetic status-flags, this could be used to do a conditional branch based on the flag results of a Mu arithmetic instruction, without having to explicitly put the flag results into a register. However this could be complicated due to the sometimes complicated methods by which Mu flags are computedsee § **4.2**.
- Division, the slowest of AArch64's scalar integer instructions, can be simplified when some of it's operands are known, some such optimisations can be done on the IR level (e.g. transforming an unsigned-division by a power of 2 to a logical-shift-right), however some can take advantage of AArch64 specific instructions, that don't have a direct Mu counterpart (e.g. a signed-multiply-high can be used to implement division by a constant, by getting the high 64-bits of a product with a 'magic-number' and then dividing it by a power of two (using a shift)). Implementing such division optimisations is quite difficult to do correctly, especially with signed-division.
- More peep-hole optimisations could be performed, to take advantage of AArch64's instruction set, such as combining multiply and add instructions into a multiply-add, or using the operand 'extending' variant of the add instruction, instead of shifting an operand first.

4.9 Summary

The concurrency and exception handling instructions were particularly interesting, and great care needed to be taken to avoid unnecessary inefficiency; in addition, implementing some of the other instructions was not simple. Most of the Mu instruction set was straightforward to implement, as they were not particularly high-level or novel. There are however a large range of Mu instructions that have not yet been implemented, in particular some of the most interesting Mu features, like the OSR API, and dynamic code generation features (e.g. dynamic global cell lookup) have not yet been implemented (see § **6.1.1**).

Though some optimisations have been implemented, there is still a lot left to do, in particular optimising Mu's higher-level instructions in particular cases (such as a swap-stack, whose resumption point is known), has not yet been explored.

Chapter 5

Evaluation

Though I have done and presented lots of work in creating my AArch64 back-end, it must be evaluated in order to fulfil its goal of helping answer the question of Mu's performance; in this chapter I will present an evaluation of its correctness and performance.

5.0.1 Correctness

Before evaluating performance, it is necessary to evaluate the correctness of my implementation of the Mu instruction set on AArch64, as the performance of an incorrect implementation is meaningless. Zebu includes an extensive suite of correctness tests, which where mostly developed by Yi Lin and John Zhang, with some additional ones and modifications done so by myself.

Most of the simpler (and older) tests where written in Rust, using its inbuilt testing mechanism, these create internal Zebu AST nodes and compile them directly; these are used to test the correctness of instructions, independently of the Mu API implementation. The other tests where written using `pytest`, some tested specific instruction implementations (by either calling the C API directly or passing text form Mu IR[1] through `muc` [Isaac Oscar Gariano 2016]), others tested the working of the RPython Mu back-end [Zhang et al. 2017] by compiling high-level RPython code to Mu. This large set of (about 341) tests all pass on the AArch64 back-end[2], though no testing suite is perfect, this helps show that Zebu is reasonably correct.

However, there are known problems, for example though PyPy (which is written in RPython) compiles on-top of Zebu, it does not fully work, e.g. it does not correctly pass command-line arguments, it segfaults, and causes malloc corruption. These problems could indicate faults in Zebu and/or the RPython to Mu translator (though the Mu IR it generates has been verified as valid according to the Mu specification, it still could cause undefined behaviour). There are also back-end compilation-failing bugs that could theoretically occur, but haven't yet, such as branching to a block that is too far away, for its offset to be used as an immediate

[1]Zebu does not support the text form directly, as it has been deprecated.
[2]some don't pass the x86-64 back-end due to unimplemented Mu features

operand to an assembly branch instruction.

5.0.2 Experimental Performance Results

In order to answer the research question into Mu's performance, an experimental evaluation is needed; so I have conducted one, evaluating the performance of Zebu's AArch64 back-end.

To evaluate performance, I have run a suite of RPython benchmarks that compile and run RPython code using its default C back-end, as-well as its prototype Mu back-end [Zhang et al. 2017]. The benchmarks used are included in the `mubench` framework [Cai et al. 2016], `mubench` itself was used to compile, run, and time the benchmarks[3]. The benchmark used are varied and measure a wide variaty of things, some measure particular operations (such as allocation or function-calling), others are leading generic language benchmarks (ones taken from [Gouy]), and one measures a moderately sized, real-world program (a programming language interpreter).

The benchmarks used are as follows:

- **alloc**: this benchmarks allocation by creating a list of 5,000,000 elements, and assigning each element to a new instance of a trivial class.
- **except**: this calls a function that recursively call itself. After a call depth of 26,000[4] it throws an exception which is then caught at the top of the call chain.
- **fib**: this calculates the 36th Fibonacci number, using naïve recursion.
- **quicksort**: this uses the quick-sort algorithm to sort a list of 1,000 random integers.
- **som**: runs the test-suite for a SOM (The Simple Object Machine) interpreter [Marr et al. 2016].
- Several benchmarks taken from 'The Computer Language Benchmarks Game' [Gouy], translated from Python to RPython:
  - **btree**: this creates various binary trees of depth 15.
  - **fannkuchredux**: this performs a 'fannkuch-redux' of a 10 element permutation.
  - **nbody**: this performs 50,000,000 n-body computations, using a 'symplectic-integrator'.
  - **pidigits**: this calculates and prints the first 1,000,000 digits of Pi.
  - **spectralnorm**: this approximates the spectral norm of an infinite matrix, using 1,000 iterations.

Each benchmark was compiled 3 times, using RPython's C back-end (with the GC turned off and Clang as the C compiler), with both minimum and maximum LLVM back-end optimisation levels (i.e. the `-O0` and `-O3` command line flags), and with

[3]By running `mubench` in 'pipeline' mode with the `mubench.yml` file included with Zebu.

[4]Though this depth results in only very small execution times (< 50ms), higher-numbers cause stack-overflows.

RPython's Mu back-end (using Zebu). The LLVM front-end optimisations where set to maximum (the `-O3` flag); this was done due to lower-optimisation levels producing LLVM IR that is wildly inferior to the Mu IR, in particular front-end optimisation level `-O0` causes C variables to be placed in 'alloca' cells pn the stack, as oppesed to SSA form which used by `-O3` LLVM IR, as well as the generated Mu IR. The choice of only using `-O3` LLVM IR as a comparison point is also due to my *hypothesis* that the Mu IR generated by the RPython back-end is at a similar level of optimisation as the generated C-code, which in-turn is so low-level that the `-O3` LLVM IR is effectively the same as the C code; this hypothesis seems to be reasonable based on manual comparison of the generate C code, `-O3` LLVM IR, and Mu IR. The optimisation flag passed to Clang is used to control both the IR front-end and assembly back-end's optimisations. In order to disable back-end optimisation only, I created a Python script, dubbed `clang-O3-O0`, which acts as a drop in replacement to Clang (by setting the `CC` environment variable to the path of `clang-O3-O0`). Whenever the `clang-O3-O0` script is called to compile a file to a (`.o`) object file, clang is first called to compile the file to a (`.ll`) LLVM IR file with `-O3`, it then compiles the LLVM IR file to an object-file using `-O0`. When not compiling to a (`.o`), `clang-O3-O0` merely passes its command-line arguments to Clang.

Each benchmark was run 50 times, with time measured within the benchmark itself (thus excluding any start-up overhead). Start-up overhead was excluded due to the Zebu version having a significant such overhead (primarily due to unnecessary heap-space initialisation), since Zebu aims to be a JIT (and not an AOT compiler), AOT only performance costs are not relevant.

See **Figure 5.1** for a graph of the normalized-results, see **Figure 5.2** for a table of the non-normalised results.

The benchmarks were run on a Softiron 'OverDrive 1000' (with 8GB of RAM, and an AMD opteron A1100 SOC (with 4 ARM Cortex-A57 Cores)), running Ubuntu 17.10 and Linux Kernal '14.13.0-16-generic'. Version 4.0.1 of Clang/LLVM was used by the RPython C back-end, and Zebu internally (see § **3.1.3**).

5.0.3 Performance Analysis

The performance results are quite varied and inconsistent; they require some careful thinking and investigation in order to understand the reasons why, as well as their meaning.

There are 2 benchmarks where Zebu outperforms Clang `-O3`: 'btree' and 'fib':

- I believe the main reason why 'fib' is faster on Zebu is due to function-call overhead. The majority of the work done in the benchmark is function-calling, with just some addition, subtraction and an if statement in each function. In RPython's C back-end, after each function call there is an explicit check for excep-

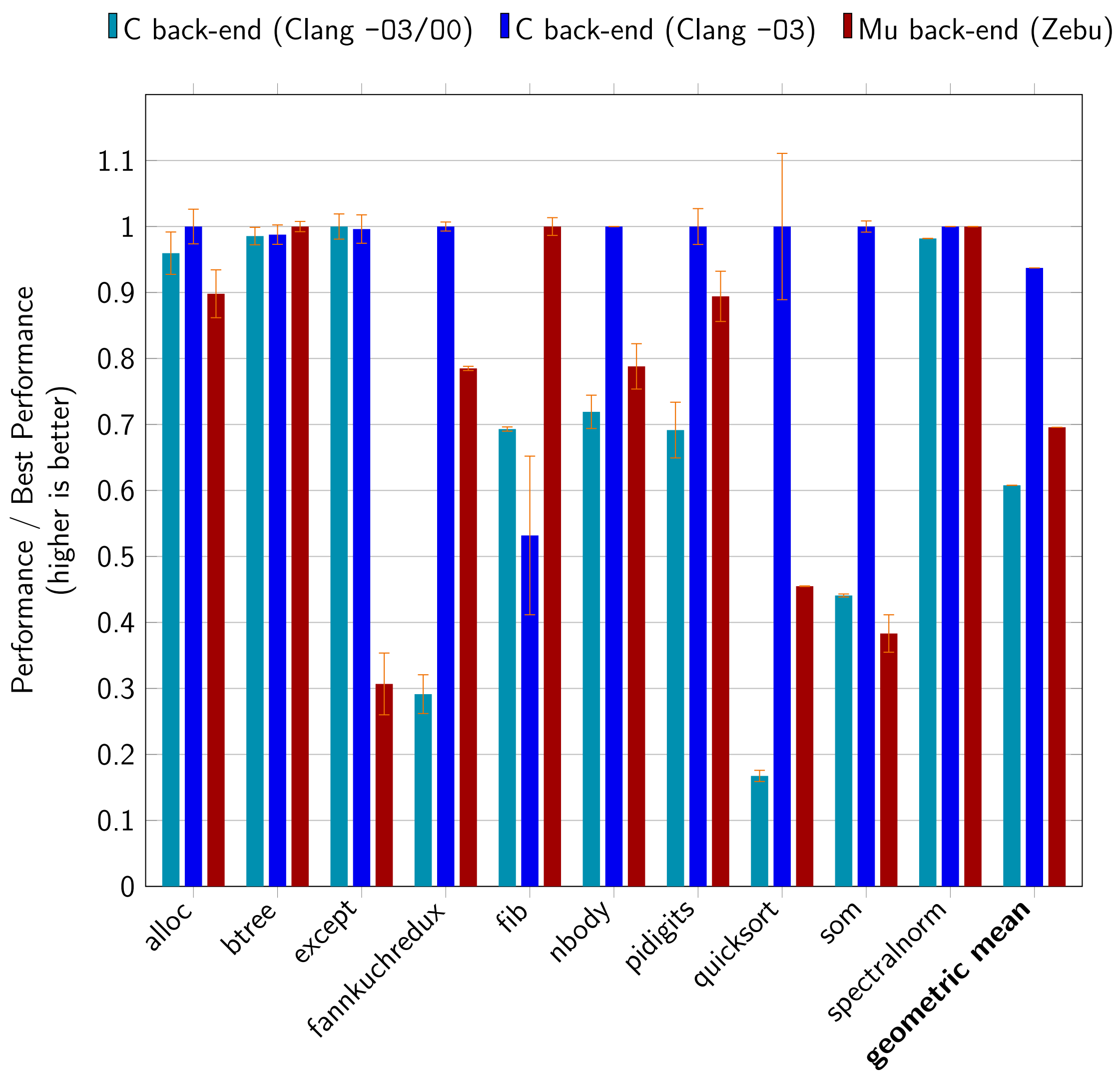

Figure 5.1: Normalized mean (over 50 runs) of each RPython benchmark's execution time, relative to the best performing back-end's execution time (for the given benchmark). Overall, Zebu is the best performing back-end in 2/10 benchmarks, it is tied with the best-performing in 1/10 cases, and is the second best in 4/10 cases. The geometric mean of the benchmarks' normalized means is 60.8%, 93.7%, and 69.6% for the three back-ends (Clang -O3/O0, Clang -O3, and Zebu, respectively).

Benchmark	Clang -O3/O0	Clang -O3	Zebu
alloc	0.650	0.623	0.694
	± 0.020	± 0.016	± 0.023
btree	1.069	1.066	1.053
	± 0.014	± 0.016	± 0.008
except	0.014	0.014	0.045
	± 0.000	± 0.000	± 0.001
fannkuchredux	1.904	0.555	0.707
	± 0.016	± 0.004	± 0.002
fib	0.600	0.782	0.416
	± 0.001	± 0.050	± 0.006
nbody	0.429	0.309	0.392
	± 0.008	± 0.000	± 0.011
pidigits	0.374	0.258	0.289
	± 0.011	± 0.007	± 0.010
quicksort	10.882	1.822	4.005
	± 0.015	± 0.202	± 0.001
som	0.138	0.061	0.158
	± 0.000	± 0.001	± 0.002
spectralnorm	0.720	0.707	0.707
	± 0.000	± 0.000	± 0.000

Figure 5.2: Mean benchmark run-time and error (standard-deviation) in seconds.

tions (by loading a global variable); in the beginning of each function there is a check for stack-overflows. In RPython's Mu back-end, these checks need not be performed as Zebu implements exception-handling in a 'zero-cost' way (see § **4.4**); in addition, Zebu uses the CPU's memory-protection hardware to check for stack-overflows (see § **4.5**), which is done implicitly. The 'fib' benchmark is particular odd, as it is the only case where Clang with `-O3/O0` outperforms `-O0`, in fact the standard-deviation of the performance results for `-O3/O0` is extremely high, whereas that of `-O0` is very low.

- The 'btree' benchmark is designed to be a GC stress-test, since the GC has been turned of for both RPython back-ends, and the results are mostly the same across the three back-ends, this benchmark is not particularly interesting.

Zebu is slower thang Clang `-O3` in 6 benchmarks, 'alloc', 'except', 'fankuchredux', 'nbody', 'pidigits', 'quicksort', and 'som':

- The slowdown for the 'alloc' benchmark is likely due to the way Zebu allocates memory, as it calls `calloc` (due to Mu's requirement that allocated memory be Zeroed), whereas the C back-end uses `malloc`. A proper GC allocator should far outperform `malloc`([Lin et al. 2016]), in particular zeroing GC memory can be done much more efficiently than `calloc`([Yang 2011]).
- The 'som' benchmark is by far the most complicated benchmark, as it runs lots of different pieces of code (the test-suite it runs is designed to check various features of the interpreter), making it very difficult to determine why the performance is so bad (it is about 161% slower on Zebu than Clang `-O3`). Zebu's performance is worse in the Clang `-O3/O0` case (by 15%), this suggests that there are either some inherent inefficiencies in the design of Mu, or more likley (due the code size), the RPython to Mu back-end needs to be further optimised. The 'som' benchmark is the only one that uses 128-bit arithmetic, this could also suggest that the implementation or optimisation of this needs to be improved.
- The slowdown in 'except' could be due to the the 'zero-cost' way in which exceptions are implemented (see § **4.4**). RPython's C back-end unwinds the stack by simply returning from each function until it reaches a catch, this is simpler than Zebu's unwinding (which requires lots of table lookups).
- The reason for the performance slowdown in the other becnhmarks ('fankuchredux', 'nbody', 'pidigits', and 'quicksort') is harder to tell, requiring a detailed analysis. It is likely due to a combination of inefficient generated Mu IR and a lack of many optimisations in Zebu's back-end. However, an attempt at manual optimisation[5] of Zebu's generated code (to make it more similar to Clang `-O3`'s) did not yield much (if any) improvement.

Zebu has equal performance in 1 benchmark, namely 'spectralnorm':

- The 'spectralnorm' benchmark makes the most use of floating-point operations, which are not in any-way optimized by Zebu. This suggests that on AArch64,

[5]See § **4.8.3** for examples of such optimisations.

there aren't many ways to optimise simple floating point operations[6].

Though Zebu is usually slower than Clang `-O3`, in most cases it does outperform Clang `-O3/O0`, suggesting that the main reason for poor performance is Zebu's implementation, as opposed to inefficient input Mu IR. In fact, on close inspection, the Mu IR generated by RPython's back-end is very close the LLVM IR generated by Clang `-O3`, it is only needing a few minor optimisations. However, one notable difference between the LLVM and Mu IR, is how string constants are handled, the RPython Mu back-end implements them as hybrids, and placing references to them in global cells, necessitating an extra load to access the string, whereas the LLVM-IR puts them as array constants, whose address does not need an indirection to compute. Fixing this would require a major change to the RPython Mu back-end (it would need to change the type of the string to a fixed-size array, since a hybrid can't be stored in a global-cell).

The overall results are promising, the performance of Zebu is usually not significantly slower than Clang's, and due to the high similarity between LLVM-IR and Mu IR, it should be theoretically improvable by simply implementing Clang's optimisations in Zebu directly, as opposed to needing to come up with entirely new optimisation strategies.

5.0.4 Summary

My evaluation of the correctness and performance of Zebu shows that it is both reasonably efficient and correct; this shows that Mu itself is designed in a way that makes quality implementations possible.

My performance analysis left many question unanswered, such as the exact reasons for the observed results and what methods could be used for improving them. More detailed investigation and experimentation is needed in order to fully answer these questions.

Though the evaluation of correctness and performance used a variety of tests, there is still work to be done in further analysing Zebu (and hence Mu). In particular, my testing has only used a single language client (the RPython one) due to it being the only one that works with the latest version of Mu. When further clients are implemented, it would be valuable to use them as a basis for evaluation of Zebu, this would be particularly useful if they utilise Mu features that are not used by the RPython client (such as threads and stacks).

[6]Clang `-O3`, by default, limits floating-point optimisations to minimise loss of precision, I have not tested the result of turning on less-precise optimisations.

Chapter 6

Conclusion

My work presented in this thesis has hopefully answered the research question of whether or not Mu's abstraction level is appropriate for performance, however this research is by no means complete, and other avenues of research are worth exploring.

6.1 Future Work

No research is ever complete, in particular their is far more work to be done, with both Zebu and Mu. Here I wish to present some concrete ideas for further exploration.

6.1.1 Further Zebu Development

Many features of Mu, and enhancements to Zebu have not yet been done; in particular extending Zebu to be an efficiently-compiling JIT compiler is a major goal, and would help investigate the suitability of Mu for dynamic managed languages. Development and implementing a garbage collector that appropriately interacts with Mu's abstractions is another interesting research direction. One major limitation of Zebu, though not a feature of Mu per-se, is its inability to compile Mu 'libraries' that can be linked to other libraries, dynamically loaded from within Mu, or the generation of libraries based on JIT-ed code; implementing such features is likely to be extremely difficult, especially with the need to merge Mu IR top-level entities, but doing so would help make Zebu a more practical and useful compiler & virtual-machine.

Many instructions have not yet been implemented in Zebu; doing so would be very useful in order to evaluate their performance. Here I present some ideas on how they could be implemented efficiently:

- SSA variables of a struct type could be implemented by simply 'expanding' out such an SSA variable to a list SSA-variables, such an expansion could be done on the IR level form. This approach has the advantage that it does not place unncesarry structural constraints on struct's, however if the ABI is to be continued-to be respected, passing structs to functions would be non-trivial (they may require merging their values into a specific sequence of registers, or placing the value on the stack); if the ABI is not to be respected however, they could be passed and returned as if passing or returning their fields directly.

- Frame-cursor creation and iteration could be implemented using the same methods as exception implementation (see § **4.4**). Frame popping is simple to implement by unwinding the stack to point to the appropriate frame-record. Pushing a frame using a frame-cursor is complicated due to the ABI, whose argument passing and value returning conventions are different, this could be fixed by JIT-ing 'adapter' frames that convert between these different conventions (as proposed by [Wang et al. 2017]), this complexity is further compounded by the possibility of needing to return through `XR` (whose value is passed in to a function call). However, an alternative and more efficient approach would be to modify the way values are returned to be the same as argument passing (this also requires callees to be responsible for popping stack-arguments, which would also simplify tail-calls (see § **4.3**)), this would eliminate the need to JIT any code.
- Function exposing can be implemented by simply placing the so called 'cookie' into a dedicated register, and recomputing the mu_tls register, then branching to the Mu function's code. To get the cookie, it would be loaded within a Mu function prologue, and stored in an internal SSA variable (freeing the 'cookie' register and allowing the register allocater to determine how best to save the SSA variable), the `uvm.native.get_cookie` intrinsic would then be implemented as a simple load from the cookie register. This approach however would require some special care to presserve argument registers (and the link-register) through the function call required to evaluate mu_tls.
- Trap instructions could be implemented as a simple swap-stack, that swaps to a newly created stack, passing in neccesary information. The entry function for the new stack would then be a runtime-function that branches to the client provided trap handler (and passing in any necessary information).
- A watchpoint-branch instruction could be implemented as a no-op when disabled (the default), when enable it would then be automatically overridden to be a branch to the enable destination (this is possible as all AArch64 instructions are 32-bits, which can be atomically stored).

There are many more features of Mu that have yet to be implemented, and further research needs to be done in order to determine efficient ways to do so.

6.1.2 Further Mu Research

This thesis was only interested in investigating one aspect of Mu's design, namely that it is appropriate for efficient code; however the other goals of Mu should also be investigated further. In particular, how reasonable and easy it is to use Mu as a target for various languages, and language features, as well as whether they would benefit from this should be further explored. It is also interesting to consider what effect additional features and changes to Mu's design would have on its performance and ease of use is worth further investigation.

6.2 Summary

In summary, though my implementation and evaluation of Zebu (on AArch64) does point towards Mu being efficient, it is by no means conclusive or exhaustive. In particular, I have only evaluated performance on a single architecture, and with a single implementation strategy; what effect implementing Mu differently would have has not been explored. This work has also no discussed the full performance implications of all of Mu's features, nor the performance effect of different mapping from higher-level languages to Mu IR.

In conclusion, though this thesis ends here, answering the question of Mu's performance is not finished, and far more research is yet to be done.

Bibliography

ARM Limited. 2013. *Procedure Call Standard for the ARM 64-bit Architecture (AArch64)*. ARM Limited. (pp. 3, 17)

ARM Limited. 2017. *ARM Architecture Reference Manual*. ARM Limited. (p. 1)

Cai, Z., Zhang, J., et al. 2016. mubench. `https://gitlab.anu.edu.au/mu/mu-perf-benchmarks`. (p. 34)

Dolan, S., Muralidharan, S., and Gregg, D. 2013. Compiler support for lightweight context switching. *ACM Trans. Archit. Code Optim. 9*, 4 (Jan.), 36:1–36:25. (pp. 22, 26)

for Standardization, I. O. 2011. Information technology — Programming languages — C. Standard (Dec.), International Organization for Standardization, Geneva, CH. (p. 6)

for Standardization, I. O. 2014. Information technology — Programming languages — C++. Standard (Dec.), International Organization for Standardization, Geneva, CH. (p. 5)

Gouy, I. The computer language benchmarks game. `http://benchmarksgame.alioth.debian.org/`. (p. 34)

Hall, A. M. 2016. MuBF. `https://gitlab.anu.edu.au/mu/mu-client-bf`. (p. 7)

Hall, A. M., Yong, N., and Zakopaylo, P. 2017. ANU Haskell Compiler. `https://gitlab.anu.edu.au/mu/mu-client-ghc`. (p. 7)

International, E. 2012. Common Language Infrastructure (CLI). Standard (June), Ecma International, Geneva, CH. (pp. 5, 8)

Isaac Oscar Gariano. 2016. muc. `https://gitlab.anu.edu.au/mu/mu-tool-compiler`. (p. 33)

Isaac Oscar Gariano. 2017. Rodal. `https://gitlab.anu.edu.au/mu/rodal`. (p. 13)

Lattner, C. and Adve, V. 2004. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04 (Washington, DC, USA, 2004), pp. 75–. IEEE Computer Society. (pp. 5, 8)

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 (New York, NY, USA, 2016), pp. 89–98. ACM. (pp. 11, 38)

Lin, Y., Gariano, I. O., et al. 2017. Zebu. `https://gitlab.anu.edu.au/mu/mu-impl-fast`. (p. 11)

Marr, S., Zhang, J., et al. 2016. RPySOM. `https://github.com/microvm/RPySOM`. (p. 34)

The Mu Micro Virtual Machine Project. The Mu Micro Virtual Machine.

The Rust Project Developers. 2017. The Rust Programming Language. `https://doc.rust-lang.org/stable/book/first-edition/`. (p. 11)

Tozawa, A., Tatsubori, M., Onodera, T., and Minamide, Y. 2009. Copy-on-write in the php language. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09 (New York, NY, USA, 2009), pp. 200–212. ACM. (p. 1)

Wang, K. 2015. js-mu. `https://gitlab.anu.edu.au/mu/obsolete-js-mu`. (p. 7)

Wang, K. 2017. Holstein. `https://gitlab.anu.edu.au/mu/mu-impl-ref2`. (p. 7)

Wang, K. et al. 2017. Mu Specification. `https://gitlab.anu.edu.au/mu/mu-spec`. (p. 1)

Wang, K., Blackburn, S., Hosking, A., and Norrish, M. 2017. Hop, Skip, & Jump: A Practical On-Stack-Replacement API for A Cross-platform Language-neutral Virtual Machine. (unpublished). (pp. 6, 22, 42)

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 T. Ball, R. Bodik, S. Krishnamurthi, B. S. Lerner, and G. Morrisett Eds., *1st Summit on Advances in Programming Languages (SNAPL 2015)*, Volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)* (Dagstuhl, Germany, 2015), pp. 321–336. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. (pp. 1, 5)

Yang, X. 2011. Locality Aware Zeroing: Exploiting Both Hardware and Software Semantics. (p. 38)

Zhang, J., Cai, Z., et al. 2017. PyPy-Mu. `https://gitlab.anu.edu.au/mu/mu-client-pypy`. (pp. 14, 33, 34)

Zhang, J. J. 2015. MuPy: A First Client for the Mu Micro Virtual Machine. (p. 7)