Patrick Ferris

Computer Science Tripos — Part II — Project

OPTIMISATIONS ACROSS SOFTWARE AND HARDWARE USING RISC-V

Pembroke College

May 8, 2020

Declaration of Originality

I, Patrick Ferris of Pembroke College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I, Patrick Ferris of Pembroke College, am content for my dissertation to be made available to the students and staff of the University.

Signed:

[illegible]

Date: 7th May 2020

Acknowledgements

This project would not have been possible without the support of:

- David Allsopp, my supervisor, for always making time to have meetings, inviting me to speak to OCaml compiler developers and for keeping me motivated with his enthusiasm. He also started my interest in functional programming as my Part IA supervisor in Foundations of Computer Science.
- Anil Madhavapeddy, my Director of Studies for two years, who helped me see the potential for doing good as a computer scientist and has inspired me ever since to become a better one.

Proforma

Candidate Number:	2386G
Title:	Optimisations across Software and Hardware using RISC-V
Examination:	Computer Science Tripos Part II, June 2020
Word Count:	11851[1]
Line Count:	1626[2]
Project Supervisor:	Mr. D. Allsopp
Project Originators:	Dr. A. Madhavapeddy & Mr. D. Allsopp

The Original Aims of the Project

The original aim was to show how RISC-V could be extended to enable OCaml-specific instructions and how the OCaml compiler could be modified to use these instructions to prove theoretically that lower-power and better performing devices were possible. In particular, the goal set out in the proposal was to reduce the number of instructions for functions which check for the value 0 (OCaml 1) and return true or false.

Summary of Work Completed

The project was a success. The particular function described was optimised and the compiler modified to produce the required instructions. In addition to this, other OCaml-specific optimisations were made such as arithmetic add and subtract, whilst a small portion of the proposed bit manipulation extension was added to show how the compiler could be modified in the future. An instruction execution log analysis tool was built which led to further discoveries of possible instruction-fusions, reducing the number of effective instructions further.

Special Difficulties

None.

[1]Calculated using Texcount: https://app.uio.no/ifi/texcount/
[2]Calculated using cloc and git diff –stat for the modifications: http://cloc.sourceforge.net/

Contents

List of Figures

List of Listings

List of Tables

1 | Introduction

The hardware-software interface is described by instruction set architectures (ISAs). Software developers work in high-level abstractions, whilst compiler writers are translators and transformers from the abstract to this interface. For most of the 20th and 21st centuries, computer architects have had a large say in how compiler writers and programming language designers can work by issuing a concrete, largely inflexible interface through proprietary ISAs. This dissertation looks at how an open-source specification for a modular, flexible, reduced instruction set architecture, RISC-V, can open doors to greater control and manipulation over the hardware-software interface allowing for better collaboration from both sides [1]. In particular, I argue that the greater flexibility and ease of developing custom hardware, enables a future where we can achieve faster and more power-efficient computers simultaneously. This is realised through adding extensions and modifications to compilers targeting domain-specific or even programming-language-specific hardware. The compiler and language used in this dissertation is OCaml.

1.1 Motivation

Power-consumption is the dominant metric in modern processor design. Clock frequencies have been stagnating partially due to Dennard scaling and meeting the power dissipation requirements on increasingly denser circuits is proving challenging [2]. The computer processor industry has moved from pipelining, to superscalar, to multi-core and now to application specific integrated circuits (ASICs) in search for performance and power efficiency. Even in 1991, in Mark Weiser's seminal paper on ubiquitous computing, a future where computers seamlessly integrate into everyday lives, the importance of power-consumption was understood [3].

> "The technology required for ubiquitous computing comes in three parts: cheap, low-power computers that include equally convenient displays, software for ubiquitous applications and a network that ties them all together."

For true ubiquitous and seamless integration, environments like those described in "An Architecture for Interspatial Communication" will require many domain-specific, low-power embedded devices [4]. When combined with the specificity and therefore efficiency and security of unikernels, such as MirageOS[1] on the OCaml platform, an extension to the OCaml compiler for generating OCaml-specific, custom instructions could improve system performance and power-consumption making Weiser's future a possibility.

Moreover, in the age of climate change, having more energy-efficient hardware has never been more important. Studies in 2014 by Whitehead et al. placed the proportion of the UK's total electricity consumption caused by the information and computer technology (ICT) industry at 10% in 2007, whilst globally contributing 2% to anthropogenic carbon-dioxide emissions (similar to the aviation industry) [5]. Monitoring of remote locations

[1] `https://mirage.io/` (Accessed: 6th April 2020)

for more accurate climate data can be achieved through low-power devices requiring less maintenance and longer in-field service. Reducing the power consumed in data-centres through highly optimised, domain-specific processors running customised ISAs could help reduce energy consumption even further.

1.2 Related Work

Application-specific hardware is not new. At the most pervasive level is the use of graphics processing units (GPUs) for graphics-related tasks. There, tasks like vector operations and parallel computations using multiple threads run more efficiently both in terms of performance and power-consumption [6]. Other examples include work done by Luca Benini et al. on the Parallel Ultra-Low Power[2] (PULP) platform with digital signal processing (DSP) extensions for DSP-specific hardware [7]. RISC-V allows more developers to explore designs as it relates to software development, without the licensing imposed by proprietary ISAs. Many mobile devices, constrained by power budgets, make use of many hardware-accelerated modules. For instance, the Myriad 2 is an always-on vision processing unit for mobile devices aimed at providing more sustained performance [8].

1.3 Notes on this Dissertation

The general structure of this dissertation follows the conventional five chapters of introduction, preparation, implementation, evaluation and conclusion. Chapter 2 outlines the design of key codebases including the RISC-V GNU Toolchain and the OCaml compiler. Chapter 3 describes the modifications made to both to support custom instructions. Chapter 4 evaluates the resulting code, explains the testing framework and explains the analysis which inspired some of the implementation choices and Chapter 5 summarises the key findings of the project.

It is important to note the large amount of mutual information between the preparation and implementation chapters. When modifying and extending large existing code-bases, the majority of the battle is understanding what came before you, leading to a preparation chapter similar in size and density to the implementation chapter.

Most of the code fragments are either OCaml code or RISC-V assembly. To help understanding, some of the OCaml functions have been given explicit type annotations in order to help reason about their arguments and what they return.

[2]The project is a collaboration between ETH Zurich and Università dí Bologna: https://pulp-platform.org/projectinfo.html

2 | Preparation

This chapter outlines the background knowledge and existing projects which formed a part of the project (§ 2.1), the starting point for the project (§ 2.2), the core deliverable goals (§ 2.3) and finally the key software techniques used across the project (§ 2.4).

2.1 Background

There are two existing and deployed codebases that are extended in this project — the RISC-V GNU Toolchain and the OCaml compiler [9]. More specifically, the RISC-V port of the OCaml compiler [10].

2.1.1 RISC-V Instruction Set Architecture

An Instruction Set Architecture (ISA) is a human-readable abstraction of a computer. Started by the Parallel Computing Laboratory at the University of California, Berkeley in 2010, the RISC-V ISA is a universal, standardised, open specification [11]. RISC-V is a reduced instruction set architecture as opposed to a complex one. In combination with its universality it follows a set of core principles.

- **Simplicity** — in the base ISA most instructions aim to complete in one clock cycle akin to the micro-operations of *x86*. Ease of programming is also achieved (especially for compiler writers) thanks to simplicity, avoiding things like the branch delay slots of the MIPS ISA or overly long instructions which can be much harder to pipeline.
- **Extensibility** — there are two general approaches to ISA design with respect to how new features are added, either incrementally or in a modular way. *x86* for instance is built in an incremental way in order to have binary backwards compatibility. However, this also leads to excessive bloating and archaic instructions persisting past their usefulness (for example the 16-bit `AAA` instruction for binary-coded decimal addition) [12]. RISC-V is modular — a core, base ISA (which has been frozen) offers the bare minimum required to execute software. Optional extensions provide flexibility, but also allow processor designers to omit unneeded instructions which is important for low-power, embedded devices. A great example is the ONiO.zero which implements only the base ISA (with a reduced set of registers) and the compressed instruction extension to deliver a battery-less, power-harvesting microcontroller [13].
- **Flexibility** — in order to be universal, RISC-V must be able to be used in a variety of settings. Whether it is in application-specific integrated circuits (ASICs), small embedded devices or high-performance super-scalar processors, it must perform well.
- **Stability** — for a large user-base to adopt the ISA, it must have a solid foundation on which to build processors. This is the so-called RV32G and RV64G ISAs, for 32-

bit and 64-bit respectively, consisting of a set of base instructions (I), multiply and divide (M), atomicity (A), single-precision floating point (F) and double-precision floating point (D) [14].

With respect to this project there are two main advantages to using RISC-V as a target instruction set. It is an open-source specification which means there are not the same licensing restrictions that would be seen if using *x86* or similar. Its simplicity and modularity leave a large space for adding custom instructions. Some examples have already demonstrated the possibility of doing this including the "RI5CY" core developed by the PULP Platform team and now the OpenHW Group. "RI5CY" is a simple four-stage RV32IM(F)C core modified to include general-purpose extensions like auto-incrementing load and stores, as well as signal-processing specific instructions like SIMD instructions over eight bit vectors [15].

The RISC-V instruction set contains four main types of instruction: register-to-register (R-type), immediate (I-type, also used for loads), store (S-type) and Upper immediate (U-type). The core design principles behind the instruction format include:

- Register specifiers are in the same place for all instructions — this allows the registers to be read before instruction decoding has even begun because for example the destination register (`rd`) is always in bit position 11 to 7.
- The most significant bit of an immediate is always the highest bit in the instruction encoding — this is to allow sign extension to happen before instruction decoding. All RISC-V immediates are sign-extended using the most significant bit and this can provide simpler instruction patterns. Consider, for example, the following C code "`n & 0xFFFFFF00`" for dropping the lowest byte. On a RISC-V 64-bit architecture the hexadecimal (-256) would be sign-extended to `0xFFFFFFFFFFFFFF00`, the resulting assembly is therefore `andi a0, a0, -256` (assuming `n` is in register `a0`). On MIPS architecture, logical operations' immediates are zero-extended [16]. This means that to perform the same calculation, the assembly would first use a load-immediate to sign-extended -256 and then the register-to-register `and` instruction to perform the operation [12].
- Learning from past mistakes — RISC-V is a synthesis of lessons learned over years of ISA design. One notable decision made was to leave sufficient opcode space for future extensions and customisations. In fact, the base ISA only takes up $\frac{1}{4}$ of the available opcode space [17].

There are two additional instruction types which build on the original four base types, branch (B-type) instructions use the S-type encoding where possible and jump instructions (J-type). A diagram of the B-type instruction format is given in Figure 2.1. Of particular note, is the most significant bit being in bit position 31 following the principle stated above and the least significant bit of the immediate (`imm[0]`) is left out. This is because the relative branching offset is performed in multiples of 2 bytes. The RISC-V architecture is word-aligned — on a 32-bit architecture this amounts to instructions being stored at multiples of 4 bytes. However, RISC-V allows for the 16-bit compressed

31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode

Figure 2.1: *RISC-V B-type instruction formats [1].*

instruction extension hence the branch offset must be multiples of 2 bytes. This small optimisation increases the reachable range of the immediate.

One of the core principles of the RISC-V ISA design was to achieve universality through modularity and extensibility. Beyond the core modules described as the G variant, proposals exist for more exotic extensions such as bit manipulation (B) or vector instructions (V). Compilers that support targeting these extensions can optionally compile programs to use them when targeting hardware which supports them.

Compressed instructions have a one-to-one mapping to normal instructions and are decoded separately in the front-end of a typical CPU pipeline. For example, in the lowRISC Ibex core this happens in the instruction fetch stage [18]. They allow for much smaller binaries which can be important when compiling for small, embedded devices. It does however add slightly more complexity, as the invariant of having a fixed-length of 32 bits per instruction no longer holds. Most processors make use of the least-significant bits to distinguish between 32-bit and 16-bit instructions to decide how to increment the program counter, or perform multiple fetches to account for this [18].

The current port of the OCaml compiler targets RV64G. There is no support for 32-bit architectures or compressed instructions. Compressing instructions is usually left to the assembler to identify opportunities for compression (e.g. `addi a0, a0, 1`). However, there are some side cases where the OCaml compiler makes assumptions about instruction size, like in the jump tables used for pattern matching over constant variants. This is discussed in greater depth in § 3.5.2.

2.1.2 Processor Performance and Energy Consumption

A processor executes a sequence of instructions and manipulates data. The defacto standard for processor performance is often related to the Iron Law of Performance [19]:

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}} \times \frac{\text{instructions}}{\text{program}}$$

Processor implementation and ISA design govern how cycles per instruction, CPI, and seconds per cycle, frequency, interact. Notably if there is no impact to CPI or frequency, but the number of instructions is decreased, it is likely that a performance increase will be seen.

Primarily, this report is focused on the energy consumption of processors. One of the biggest contributing factors to this is instruction fetch energy, the energy required to retrieve instructions from cache at the front-end of a pipelined processor. John Arends et al. note that typically only 25% to 30% of dynamic instructions are data-oriented (loads and stores) resulting in there being approximately 3 instructions per data reference (later in § 4.4 there is evidence for this in OCaml code) [20]. Andrew Waterman also remarks on how compressed instructions in the RISC-V ISA fetch fewer total bits, reducing total instruction fetch energy [21].

By reducing the total number of instructions without impacting CPI or frequency, it is possible to see both performance improvements and a reduction in power consumed by the processor. This is in contrast to other techiniques such as cold scheduling, which for an optimal, energy-efficient scheduling of instructions saw a performance degradation of 6% [22].

2.1.3 RISC-V GNU Toolchain

The RISC-V GNU Toolchain is a collection of specifications, programs and software for cross-compiling C (and C++) code to RISC-V — this includes the standard GNU tools like `gcc, g++, objdump` and `as` (the assembler) [23]. There is another collection of tools for RISC-V development which includes the RISC-V ISA simulator (Spike) and the proxy kernel [24]. The importance of these tools and the subsequent necessary modifications is explained in § 3.1.1, but a short overview of some of the main tools is given below.

- Binutils — the collection of tools for the final stages of compilation including the assembler, linker and debugger [25]. The assembler needs modifications to accept the new opcodes for custom instructions.
- Spike — a RISC-V ISA simulator which functionally emulates a core with a cache model which can have one or more hardware threads (harts). Internally, Spike defines the instructions of the ISA using C. It can be configured to execute many of the RISC-V ISA extensions and also be modified to execute custom instructions. This will be the main tool for analysing the effectiveness of the custom instructions proposed later [26].
- Proxy Kernel — is an application for running statically-linked ELF binaries by providing an execution environment in which system calls and IO-related methods are proxied to the underlying host operating system [27]. This allows statically compiled OCaml applications to be run on the Spike simulator.

2.1.4 OCaml

OCaml is a general-purpose programming language. It is a direct descendent of the Caml programming language — an implementation of the ML-style languages. Some of its key features include a powerful type system that is statically type-checked, type inference, a flexible and robust module system and an object and class system [28].

OCaml Internals

Understanding the OCaml data representation is crucial to making improvements to common code fragments. OCaml has a uniform memory representation where everything is a word-sized value. The values are either immediates, represented as unboxed integers or non-immediates, which are pointers to blocks stored in either the OCaml heap or the C heap [29]. Boxing is the process of wrapping additional meta-data around an object and in OCaml this is its length, garbage-collection information (colour) and a tag-byte for knowing what the block is. After this is a series of word-sized fields containing more values. Constant variants (those without parameters), booleans and ASCII characters all map to unboxed integers.

```
(* Examples where the constant constructor maps to an integer *)
type 'a option = None | Some of 'a
type 'a list = [] | (::) of 'a * 'a list
type 'a tree = Leaf | Branch of 'a * 'a tree * 'a tree

type int_complex = {real: int; imag: int}
(* Result will be allocated to the heap *)
let add_complex x y =
  {real = x.real + y.real; imag = x.imag + y.imag;}
```

Listing 1: *The variants described above all have one constant constructor which will map to the integer 0 (the OCaml representation being 1). Allocation of non-immediate values to the heap — here an integer record.*

To distinguish between pointers (memory addresses) and immediate integers, OCaml uses a tagged-bit memory representation where the least-significant bit is set to 1 to indicate an immediate integer and a 0 to indicate a pointer. This means that on a 32-bit machine, OCaml only has 31-bit integers and to find the same representation of an integer it must be bit-shifted left by one place and incremented by one.

Blocks in the heap start with a one-word header, either 32 or 64-bit depending on the architecture, which contains information about the length of the value (22 or 54 bits), 2 bits for a colour which is used in garbage collection and 8 bits for a multi-purpose tag byte [28, 30]. Figure 2.2 shows how these relate to the OCaml and C heaps for allocating larger values in memory.

Certain frequently used data-types have a particular data representation which impacts the assembly generated to distinguish between values. Variants with constant constructors are represented by integers. They are enumerated in the order they appear in the type declaration. For non-constant constructors (e.g. `type t = Foo of int`), they are represented as blocks where the tag-byte encodes the enumeration. Listing 1 shows how some data is allocated to the heap and also how some common variants partially map to integers. Polymorphic variants are more flexible versions of variants where constructors are not tied to any single particular type. This makes it easier to extend or reuse code thanks to their polymorphic nature [31]. For their data representation, a hash of the polymorphic variant constructor name is stored in the first field of the block (the values section in Figure 2.2). The hashing algorithm is defined in the `caml/hash.c` file. At compile time these hashed values can be loaded as immediates and the appropriate field

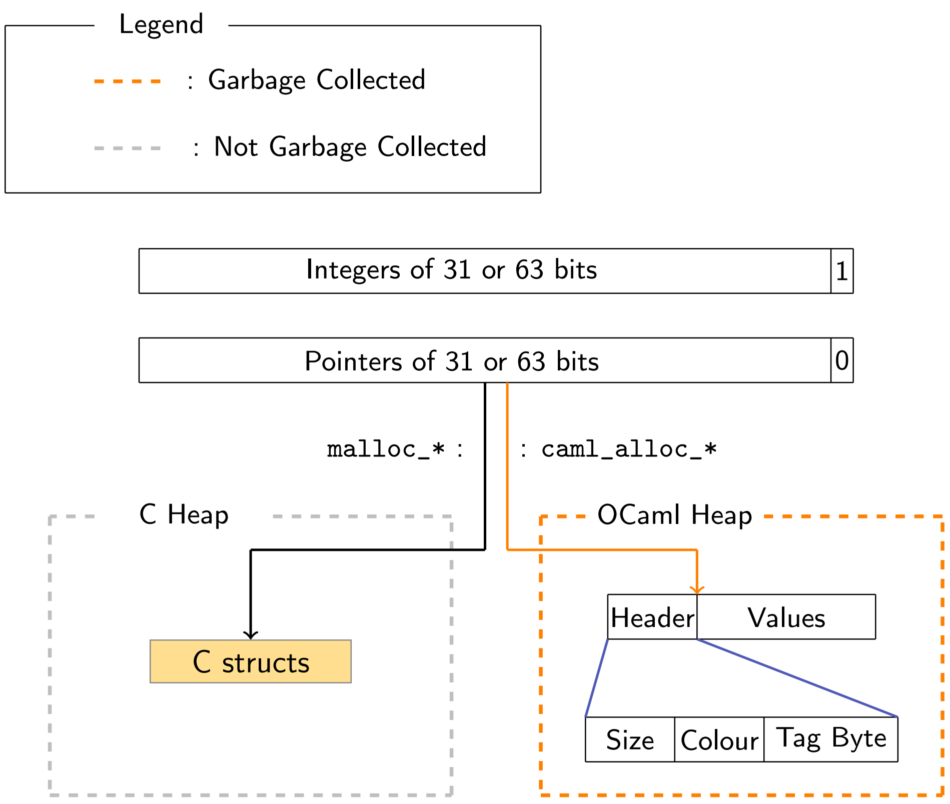


Figure 2.2: *The runtime data representation of OCaml values and the structure of heap-allocated blocks. Note that the pointer of the block starts on the first field for faster access to the data, for accessing the header an offset of -4 or -8 bytes can be specified.*

read from, which is where the pointer points, to check if they are equal [30].

Class System

The class system is explained in depth in a practical sense in the *Real World OCaml* book and from a theoretical perspective in the paper by Didier Rémy and Jérôme Vouillon [28, 32]. What follows is a brief summary of the salient points.

Objects are implemented similarly to a record of methods with instance variables, introducing the possibility of a restrictive form of row polymorphism [33]. If some function requires an object with a `prints` method of type `string -> unit` then any object whose set of methods contains a matching to this can be used in place of that.

When thought of from the class system it becomes clear that inheritance enables this implicitly as the set of methods of the subclass are a superset of the superclass. If a function explicitly types an argument as needing to be of the superclass type then OCaml allows you to explicitly coerce the subclass to the superclass [34].

Objects can be created with a reference to the current object (often bound to a variable called `self`) allowing methods to call one another in a mutually recursive style [30]. More importantly, is the use of late-binding which enables open recursion. Given a class `A` and a subclass `B`, open recursion is the ability for a method `foo` of class `A` to invoke the `foo` method implementation of class `B` which is defined after class `A`. In the backend

```
let print_str :
  < prints : 'a -> 'b; .. > -> 'a -> 'b =
  fun obj str -> obj#prints str

let o :
 < prints : string -> unit;
   printi : int -> unit >
 = object
 method prints s = print_string s
 method printi i = print_int i
end

print_str o "Hello" (* Prints Hello *)
```

Listing 2: *The polymorphism attainable through the OCaml object system.*

of the compiler the class and object system enables large code sharing amongst different architectures, with easily managed open recursion. The reference to `self` is dynamically bound whenever a method is called [35]. This becomes clearer with an example.

Listing 3 can emit a small programming language with integer addition, subtraction and negation (a unary operator). `Selectgen` is the general selector for emitting the program. `Nosubselect` can be thought of as a different architecture, inheriting most of the capabilities for the general selector, but does not support the binary subtraction operator. Instead it overrides the `emit_expr` method (indicated by the `method!`) and pattern matches on subtraction expressions, negating the second argument and returning an addition. Crucially, the calls to `self#emit_expr` in `selectgen` refer to the `emit_expr` of `nosubselect` because `self` is late-binding (dynamic dispatch). This is in contrast to the module system which is early-binding. Functors (the module equivalent of functions) are used to provide implementations at a later point [28]. The object and class system encapsulate open recursion in a cleaner and less cumbersome way for programmers.

2.2 Starting Point

The body of work submitted for this project is largely modifications on top of existing tools and pieces of software. From a theoretical standpoint, this is the first time I have worked on a large-scale, functional program such as the OCaml compiler and in particular low-level assembly and C programming [1]. The existing code bases and bodies of work are:

- RISC-V GNU Toolchain and Tools: this represents the large collection of RISC-V related tools including Spike and the GNU tools such as the assembler and compiler.

[1]The relevant courses from across the Computer Science Tripos include: Foundations of Computer Science (IA), Compiler Constructions (IB), Computer Design (IB), Programming in C and C++ (IB), Optimising Compilers (II Lent) and Comparative Architectures (II Lent)

```
type expr = Bop of  op * expr * expr | Neg of expr | Int of int
and op = Add | Sub

class selectgen = object (self)
  method emit_expr = function
    | Int i ->  print_int i
    | Neg e -> print_string "-"; self#emit_expr e
    | Bop (op, e1, e2) ->
      print_string "(";
      self#emit_expr e1; self#emit_op op; self#emit_expr e2;
      print_string ")"

  method emit_op = function
    | Add -> print_string " + "
    | Sub -> print_string " - "
end

class nosubselect = object (self)
  inherit selectgen as super
  method! emit_expr = function
    | Bop (Sub, e1, e2) ->
      self#emit_expr (Bop(Add, e1, Neg e2))
    | e -> super#emit_expr e
end
```

Listing 3: *An example of the class and object system in OCaml making use of open recursion and inheritance that is similar in design to the Cmm to Mach transformation (see § 3.1) in the OCaml compiler*

- RISC-V OCaml Port[2]: largely written by Nicolás Ojeda Bär, this fork of OCaml contains the code for the RISC-V backend. It is the 4.07.0 version that my code extends.
- RISC-V and OCaml: work carried out at the Indian Institute of Technology Madras (IITM) by K. C. Sivaramakrishnan et al. helped ease the path into getting started with OCaml and RISC-V and in particular building a cross-compiler[3].

2.3 Requirements Analysis

The project proposal (see Appendix D) outlined the core deliverable for this project which can be broadly separated into two parts.

[2]At the start of this project the RISC-V port was an impressive, unofficial project. It has now been merged upstream making RISC-V an official backend for OCaml https://github.com/ocaml/ocaml/pull/9441

[3]https://github.com/kayceesrk/riscv-ocaml

1. Modify and extend the OCaml compiler to produce binaries using custom instructions.
2. Evaluate the binaries to prove from a theoretical point-of-view that they use fewer instructions and therefore less power.

The complexity of the problem is greater than that implied by (1) — adding the instructions was further subdivided into increasingly more challenging tasks. In the first instance, adding operations for specific tasks like bit-manipulation (e.g. logical-and) by implementing parts of existing RISC-V ISA extension proposals. Secondly, developing custom OCaml-specific instructions to optimise operations and control-flow. Thirdly, exploratory analysis for potential instruction fusion (the combining of frequently occurring adjacent instructions) opportunities.

Only the original `is_none` optimisation was part of the project proposal, but for a better narrative it comes later in the implementation chapter.

2.4 Software Techniques

The project took on three distinct phases of work. An initial understanding phase where key tools and open-source projects that were used could be understood in greater detail. The second phase focused on delivering the core goal of the project, optimising one specific function within the OCaml compiler. The third and final phases were exploring possible extensions: other optimisations, the garbage collector, the runtime and dynamically analysing the instruction stream for further possible improvements.

RISC-V development is new and as such many tools remain undocumented and lacking the finesse of industry standard tools for development. This resulted in more time being devoted to understanding the RISC-V GNU Toolchain and Spike ISA simulator than originally expected. All software used in this project make use of open-source, permissive licenses. Full details and version numbers can be found in Appendix C.

2.4.1 Docker

Docker[4] is a tool for creating containers which are isolated environments for running applications with the intent of being able to run any application on any host computer. This project was developed using Docker in order to improve reproducibility and make the required setup more explicit. It acts as a limited form of self-documentation as other developers can read the `Dockerfile` and get an understanding of the tools required, the modifications made and how to begin developing themselves.

[4]`https://www.docker.com/` (Accessed: 24th April 2020)

2.4.2 Version Control and Continuous Integration

Both the dissertation and the various tools and frameworks built as part of this project were version controlled using Git. As it became more apparent that the tooling surrounding OCaml and Spike may be useful for others (IITM for example) it made it all the more important to use a good test framework[5] and continuous integration through Github Actions[6].

[5]Alcotest — a very lightweight, functional-style testing framework `https://github.com/mirage/alcotest` and ocaml-mdx for simulating end to end tests from the command line `https://github.com/realworldocaml/mdx`

[6]`https://github.com/features/actions`

3 | Implementation

This chapter focuses on building optimisations into the OCaml compiler and cross-compiling to RISC-V. The compiler pipeline is described (§ 3.1), leading to an explanation of adding custom instructions and modifying existing components. The optimisations are presented as to how the compiler could be extended to use existing RISC-V proposal extensions (§ 3.4.1), optimising idiomatic OCaml patterns (§ 3.5.1) and finally using analysis to discover opportunities for further improvements (§ 3.5.2).

3.1 OCaml Compiler Pipeline

OCaml can be compiled to byte-code (using `ocamlc`) and interpreted in a runtime written in C (with `ocamlrun`) or compiled to high-performance native code (`ocamlopt`) [30]. The latter is the relevant system of interest for producing executables for a RISC-V architecture.

The compiler pipeline takes OCaml source code and performs a series of transformations and optimisations to produce assembly-code. These transformations include phases like type-checking, common sub-expression elimination and register allocation. As is common practice in most compiler pipelines, this is made more manageable by passing through a series of intermediate representations that progressively remove higher-level abstractions (e.g. closures, `if-then-else` constructs etc.).

The source code is first parsed for syntax-checking and the parse-tree is formed before a second pass performs the type-checking, producing a typed-tree. The code then moves through a series of intermediate representations (IRs) removing higher-level abstractions and performing common optimising compiler improvements relating to code-size and efficiency. Each IR occurs along a spectrum of abstraction from the high-level OCaml code to the explicit instructions of the machine code. Understanding where each IR comes in the pipeline, helps determine the appropriate place to add custom instruction optimisations.

Lambda is an IR based on the lambda calculus first introduced by Alonzo Church [36]. Many translations occur between the typed-tree and the Lambda IR, including from typed terms to lambda terms (erasing type information) and optimising pattern-matching to if-statements or switch-cases (desugaring the higher-level constructs).

Clambda follows the Lambda IR and differs primarily by making closures explicit, uncurrying functions and making indirect and direct function calls explicit.

Cmm is the last IR which is fully machine-independent and follows similar principles to C-- [37]. The transformation from Clambda to Cmm introduces explicit memory representations for OCaml such as, the tagged-bit to distinguish between immediate integers and heap pointers for garbage collection discussed in § 2.1.4 (see listing 12).

Mach is the first machine-dependent IR. During the transformation from Cmm to Mach, the OCaml compiler backend offers more flexibility for handling different target instruc-

```
(* From asmcomp/cmm_helpers.ml *)
let int_const dbg n =
  if n <= max_repr_int && n >= min_repr_int
  then Cconst_int((n lsl 1) + 1, dbg) (* ... *)

(* From asmcomp/cmmgen.ml *)
let transl_constant dbg = function
  | Uconst_int n ->
      int_const dbg n (* ... *)
```

Listing 4: *Converting integer constants to the OCaml tagged-bit representation in the translation from Clambda to Cmm*

tion sets. To do this, the compiler exploits inheritance and method overriding through the OCaml class and object system. In particular it defines a new interface for a virtual class called `selector_generic`. A virtual class indicates that an object cannot be instantiated with a call to `new`. Instead, ISA specific selectors inherit from a base selector implementation and override and define functions specifically for that architecture. A good example is checking if an integer immediate is valid or not. Within the RISC-V architecture, I-type instructions have a 12-bit immediate field which limits the immediates to 2^{12} possible values. This is spread over positive and negative integers $[-2048, 2047]$ [1]. The IBM PowerPC architecture has 16-bit immediates which enables values in the range $[-32768, 32767]$ [38]. Hence, the implementation of the virtual `is_immediate` method will be different depending on the target architecture.

Linear is the final IR before emitting textual assembly code. There are many Mach to Mach transformations for compiler optimisations such as liveness analysis, dead code elimination and register allocation. The transformation to Linear also removes higher-level constructs like `if-then-else` and replaces them with pseudo conditional branch statements. Linear is very similar to a generalised instruction set and the architecture-specific `emit.mlp` file completes the pipeline.

3.1.1 Cross-compiling OCaml

Cross-compiling is the process of compiling source code for a target architecture which is different from the architecture on which the compiler is running. Although this can be achieved through virtualisation tools like QEMU, it can often be much slower and difficult to set up. Understanding how an executable OCaml program is compiled, is necessary for knowing where modifications will be needed and also where potential improvements can be made.

To make use of exising tools and for more flexibility in the types (and sizes) of programs that can be cross-compiled, it makes sense to do this using the OCaml build tool, Dune [39]. Using the OCaml package manager, opam[1], with a custom repository based on an

[1]`https://opam.ocaml.org/` (Accessed: 24th April 2020)

existing one[2], which modifies the opam files of packages to cross-compile, larger and more feature-rich programs can be compiled.

3.2 Modifying the RISC-V GNU Toolchain and Spike

From the compiler pipeline outlined in § 3.1, it is clear that the OCaml compiler produces textual assembly which then relies on an appropriate C compiler, assembler and linker to produce the final executable. If the compiler is producing custom instructions, the standard assembler will not suffice for producing binaries as the new instructions would not be recognised.

Defining a new instruction is one part a manual process and one part automatic. Part of the toolchain defines an opcode tool where the opcode can be specified and the appropriate mask and match codes will be generated. § 2.1.1 explains in more detail the RISC-V instruction format. When defining a new instruction, the same style must be adhered to which respects the types (e.g. R-type) and does not clash with other opcodes.

The RISC-V Binutils repository[3] stores the code for the assembling steps of the cross-compiler. In particular the `opcode` directory describes the possible instructions and how they should be converted to their binary format. The mask and match codes allow the assembler to identify the instruction. For example, the I-type instruction uses the bits in positions 14 to 12 and 6 to 0 to distinguish between I-type instructions. The match code is the instruction encoding derived from these bit positions. This is xor-ed with the instruction which should set the one values to zero and the zero values also to zero. The mask is `0x707f` (ones in the bit positions for I-type instructions) which is bitwise and-ed with the instruction to extract only those bit positions and setting everything else to zero. The resulting value should be 0 in the case of a match.

31 20	19 15	14 12	11 7	6 0	
imm[11:0]	rs1	funct3	rd	opcode	I-type
imm[11:0]	rs1	000	rd	0001011	`ocvali`
000000000000	00000	000	00000	0001011	mask
000000000000	00000	111	00000	111111	match

Figure 3.1: *RISC-V I-type instruction format shown with the custom instruction* `ocvali` *introduced in § 3.5.2. The mask and match codes show how to check for the instruction [1].*

Spike also needs modifying for recognising instructions and invoking their C code, emulating the low-level hardware. As an example, the I-type and-immediate instruction (`andi rd, rs1, imm12`) performs a bitwise logical and of `rs1` with `imm12` and places the result in register `rd`. It is defined in a header file as `WRITE_RD(insn.i_imm() & RS1)`. `WRITE_RD` is a macro for placing the result in the destination register of the instruction.

[2] https://github.com/mirage-shakti-iitm/opam-cross-shakti
[3] https://github.com/riscv/riscv-binutils-gdb

`insn.i_imm()` extracts the top 12 bits of the instruction by shifting the instruction 20 bits to the right and `RS1` is another macro for extracting the contents at bits 19 to 15 (see Figure 3.1) [40].

3.3 OCaml Runtime and Garbage Collection

Figure 3.2 shows a section of the compilation of an OCaml program. It introduces the concept of the OCaml runtime. A runtime is an execution environment providing important methods and tools for running an OCaml program including memory management through the garbage collector and also a set of primitives for building a standard library for performing things like IO. This enables polymorphic comparison functions using structural equality through calls to the C methods (e.g. `caml_equal`).

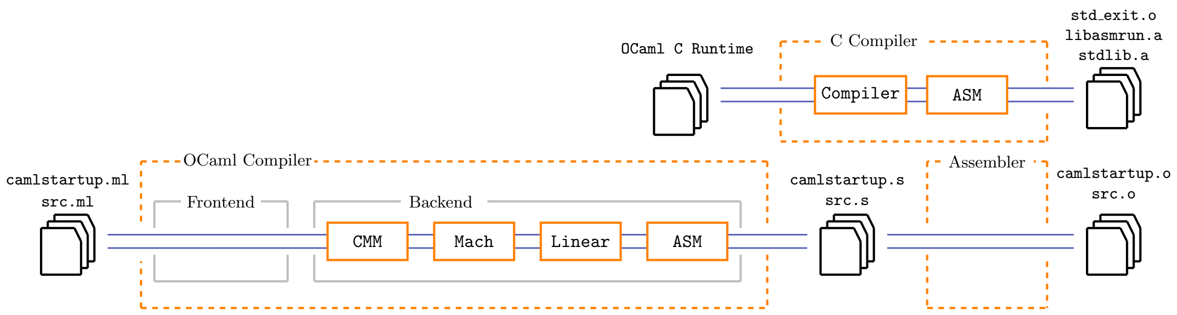


Figure 3.2: *A diagram explaining the compilation pipeline of a typical OCaml program*

The standard library and garbage collector are tools and methods used across all OCaml programs. More concretely, an OCaml program for parsing text may use more string methods and another program for computing Fourier Transforms may use floats and arithmetic operations but both will likely use comparison operators and certainly be managed by the garbage collector to some capacity. Optimising these parts of an OCaml program follows the principle of making the common case fast, often used in computer architecture design [19].

The OCaml compiler performs many optimisations (like dead-code elimination), but is restricted by the flexibility of the target architecture. With the ability to create new custom instructions this is no longer the case.

3.4 Modifying the Compiler Backend

Altering the assembly that the OCaml compiler produces, based on the target architecture is common and occurs between the Cmm and Mach transformation as mentioned in § 3.1. The compiler implements an object-oriented approach to make use of open recursion.

When compiling for a RISC-V architecture the `Selection` module, with a simple interface describing only the `fundecl` function which transforms Cmm to Mach, is instantiated with a RISC-V implementation which constructs a new `Selector` object, inheriting from the `Selectgen` base object. Figure 3.3 shows this. The `fundecl` method is then called and from here it is apparent why the open recursion is so powerful. The method is implemented in the parent class, but subsequent calls to methods such as `select_operation` are called on the inheriting object.

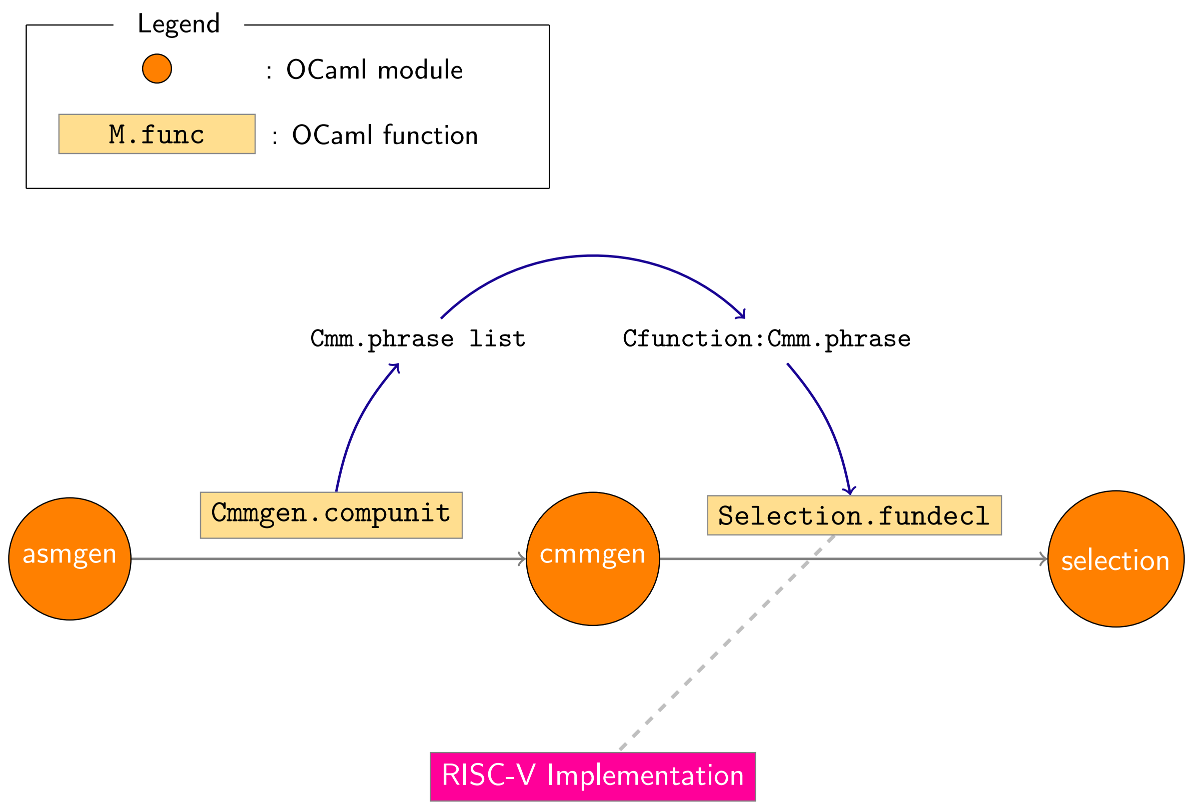


Figure 3.3: *A flow diagram highlighting the important modules and function calls in the backend of the OCaml compiler taking Clambda IR code to Mach IR code through Cmm IR code.*

Using method overriding in the `Selector` object, RISC-V specific customisations can be inserted. This could be for simple operations that are available in the RISC-V ISA such as fused multiply and add for single precision floating point numbers (`fmadd.s`). The relevant Cmm is pattern-matched and the RISC-V specific code is emitted using the `Ispecific of Arch.specific_operation` constructor.

```
(* asmcomp/riscv/selection.ml *)
method! select_operation op args dbg =
  match (op, args) with
    | (Caddf, [Cop(Cmulf, [arg1; arg2], _); arg3]) ->
      (Ispecific (Imultaddf false), [arg1; arg2; arg3]) (* ... *)
```

Listing 5: *Emitting specific operations in the Mach IR — note the method overriding such that the parent selector uses the child's implementation. A match-all pattern is typical at the end of overriden methods, passing control back to the parent with a call to* `super#select_operation op args dbg`.

3.4.1 Flexibility through Extensions

The modularity of the RISC-V instruction set demands a certain amount of flexibility in the OCaml compiler. The extensibility of the ISA has allowed others to create domain-specific instructions. For example, the digital signal processing (DSP) specific instructions Benini et al. added to their RISC-V core for higher performance and energy-savings as endpoints in IoT networks [7].

To highlight the possibility of extending the OCaml compiler with future RISC-V customisations, some of the proposed bit manipulation extensions have been implemented [41]. For the majority of cases this involves adding more cases to the `select_operation` method of the RISC-V selector object. Although, some thought must be given to how the compiler changes the underlying operations from source code to Cmm. For example, logic-with-negate-and (`andn`) takes two arguments negating the second and bitwise anding the result with the first operand.

```
let andn a b = a land lnot b
let orn a b  = a lor lnot b
let xorn a b = a lxor lnot b
```

(a) *Semantics of the logic-with-negate instructions in OCaml.*

```
(and a (or (xor b -1) 1))
(or a (or (xor b -1) 1))
(or (xor a (xor b -1)) 1)
```

(b) *Simplified Cmm output for the functions.*

Listing 6: *Mapping OCaml logic-with-negate functions to the Cmm IR noting the additional operations to deal with the tag-bit.*

The logical operators are infix functions over integer values in OCaml. In normal binary representation, it is sufficient to use a bitwise exclusive-or with the binary number consisting of all ones (in two's complement this is -1), to achieve a logical negation. However, OCaml needs the additional logical-or with 1 to undo the zeroing of the tagged-bit. The logic-with-negate-xor is slightly optimised, the following is written in a Cmm-style prefix notation.

$$\begin{aligned}\texttt{a lxor lnot b} &= \texttt{or (xor a (or (xor b -1) 1)) 1}\\ &= \texttt{or (xor a (xor b -1)) 1}\end{aligned}$$

Taking the logical-xor of two OCaml integers means we can safely remove two specific operations if they exist on the operands, either a logical-or with 1 or a tagging of an integer with a shift left and an addition of 1. Simply pattern-matching on these logical bitwise operations and outputting a specific Mach instruction for logic-with-negate is not enough because of this tag-bit. For `andn` the negation of the second argument will unset the tagged bit and when and-ed with the first argument the result will be without a set tag-bit so an additional logical-or with 1 is necessary. For `orn`, it is simpler. The first argument will be a tagged-bit integer and the second when negated, will have an unset tagged-bit which will return once the `or` operation is applied. Finally, `xorn` is similar to orn, in that negating the second argument ensures that the tag bit is unset. The xor operation sets the tag bit and so no additional logic is required.

Extended logical and bitwise operations are common in high-performance algorithms including Fourier Transforms and cryptographic functions. Another option might be to add these a primitives (like logical operations) and produce optimised assembly for RISC-V with the B extension and the normal assembly for all other architectures.

3.4.2 Optimised Arithmetic

The data representation in OCaml has trade-offs. The tagged-bit format enables the garbage collector to quickly check whether something is or is not in the heap. This is important in a functional language where immutable (and likely short-lived) objects are common. However, it does make some straightforward code use more instructions. For example, the code for adding two integer values looks like this.

```
let add : int -> int -> int =
  fun a b -> a + b
```

(a) *A simple curried add function.*

```
camlAdd__add_1002:
L100:
    add     a2, a0, a1
    addi    a0, a2, -1
    ret
```

(b) *The resulting RISC-V assembly code for the* add *function.*

Listing 7: *The two instructions which are needed to perform OCaml integer addition.*

OCaml integer arithmetic must incorporate additional operations due to the tagged-bit representation. If OCaml integers are represented in their full form this becomes clearer. When adding or subtracting to OCaml integers a and b, their underlying representation is $2i+1$ and $2j+1$ for some integer i and j.

$$\begin{aligned}
a+b &= (2i+1)+(2j+1)\\
&= 2(i+j)+2\\
a-b &= (2i+1)-(2j+1)\\
&= 2(i-j)
\end{aligned}$$

This shows how both need the additional add operation to get the correct integer. Using the existing architecture-specific flexibility offered by the class system based backend of the compiler, optimising add and subtract operations in OCaml is possible with two new instructions that incorporate the additional adding or subtracting of one.

A direct follow-up question is whether or not floating-point arithmetic would benefit from a similar optimisation. Floating-point numbers are heap allocated and stored in a normal format within an OCaml block. Since pointers start at the block's fields, floating-point

```
(* asmcomp/riscv/selection.ml *)
method! select_operation op args dbg =
 match (op, args) with
  | (Caddi, [Cop(Caddi, [arg1; arg2], _); Cconst_int 1]) ->
   (Ispecific Iocadd, [arg1; arg2]) (* ... *)

(* asmcomp/riscv/emit.mlp *)
let emit_instr i = match i.desc with
 | Lop(Ispecific sop) -> match sop with
  | Iocadd ->
  `ocadd {emit_reg i.res.(0)}, {emit_reg i.arg.(0)},
    {emit_reg i.arg.(1)}`
```

Listing 8: *Emitting an optimised addition first in Mach IR then as actualy assembly. Note the use of the backticks in* `mlp` *files is because they are preprocessed by the compiler for better assembly generation.*

operations are more constrained with additional loading of values rather than actual arithmetic [42].

3.5 Common Patterns

3.5.1 Idiomatic OCaml

Another method of trying to optimise the common case is to look at idiomatic OCaml code, extract the shared coding patterns and improve the assembly generated for these. A very common practice is pattern-matching and in particular checking for a particular base case for a given data type. Such examples include checking for the empty list, checking for 0 and checking for `None` for a given `'a option`. These OCaml values map to the immediate integer 1 (OCaml's 0 representation). The RISC-V assembly that is then generated is shown in Listing 9b.

The RISC-V assembly shows the OCaml data representation being used. The check for 0 becomes a check for 1 (`(0 lsl 1) + 1`). It also reveals a limitation of this data representation — boolean values (often represented as 1 and 0) are 3 and 1 for true and false respectively. Without this representation, the assembly could use the pseudo-instruction `seqz` (set if equal to zero) to achieve the same result in a single instruction.

Unlike the arithmetic or bit manipulation extensions and optimisations, this example is more difficult as it involves replacing a block of control flow operations. The Cmm to Mach transformation produces a sequentialisation of Mach instructions with pseudo-registers allocated. Figure 3.3 shows the entrypoint to this transformation as `emit_fundecl`. From here the function body is passed to `emit_tail`, where it proceeds to emit expressions, select condition operators, allocate pseudo-registers etc. to a growing linked-list of type `Mach.instruction` which is a mutable member variable to the `selector_generic`

```
let is_none : 'a option -> bool
  = function
    | None   -> true
    | Some _ -> false
```

(a) *A common* `is_none` *function.*

```
camlTest__is_none_1002:
L101:
    li      a1, 1
    beq     a0, a1, L100
    li      a0, 1
    ret
L100:
    li      a0, 3
    ret
```

(b) *The resulting RISC-V assembly code for the* `is_none` *function.*

Listing 9: *An OCaml* `is_none` *function which results in checking for the OCaml value 1 (which is 0) and loading 3 (true) or 1 (false) into* `a0` *and returning.*

class.

Figure 3.4 shows how the `is_none` function is generated in its Mach instruction representation. This level of understanding is necessary in order to override the `emit_tail` method to output different logic without breaking the current logic. What would be useful in this instance, is an OCaml equivalent to the RISC-V pseudo-instruction `seqz` (set equal to zero) which is in reality an unsigned set less than immediate. For OCaml the zero is a one, so an immediate "is equal" instruction would be useful and below is the semantics of **OC**aml **EQ**ual **I**mmediate.

$$\texttt{oceqi rd, rs1, imm12} \rightarrow \texttt{rd} = \texttt{rs1 == imm12}$$

To return the correct OCaml value, the 1 or 0 which results from the equality check needs to converted to tagged-bit form. This usually involves an `slli` and an `addi` instruction but this can be combined given its ubiquity in OCaml. The logical shift left and an immediate add of 1 can be combined into the single `ocvali` instruction. Given the extra instruction encoding space, the value for adding is parameterised in the immediate field. The full semantics for the instruction are given below.

$$\texttt{ocvali rd, rs1, imm12} \rightarrow \texttt{rd} = \texttt{(rs1 \ll 1) + imm12}$$

Producing this assembly involves overriding the `emit_tail` method of the generic selector class, something other target architectures do not do. This is because the control-flow is being altered, which is more complex than removing or modifying operations. Figure 3.4 provides the flow to follow in constructing the optimised version of the code. Whenever an instruction is emitted, the resulting effect is for the instruction sequence to be augmented with a new operation, and the return register (if any), to be returned from the emit call. This is returned as a `Reg.t array` and for the operations in the original code this is a single, integer pseudo-register. To make this concrete, consider the `econd`

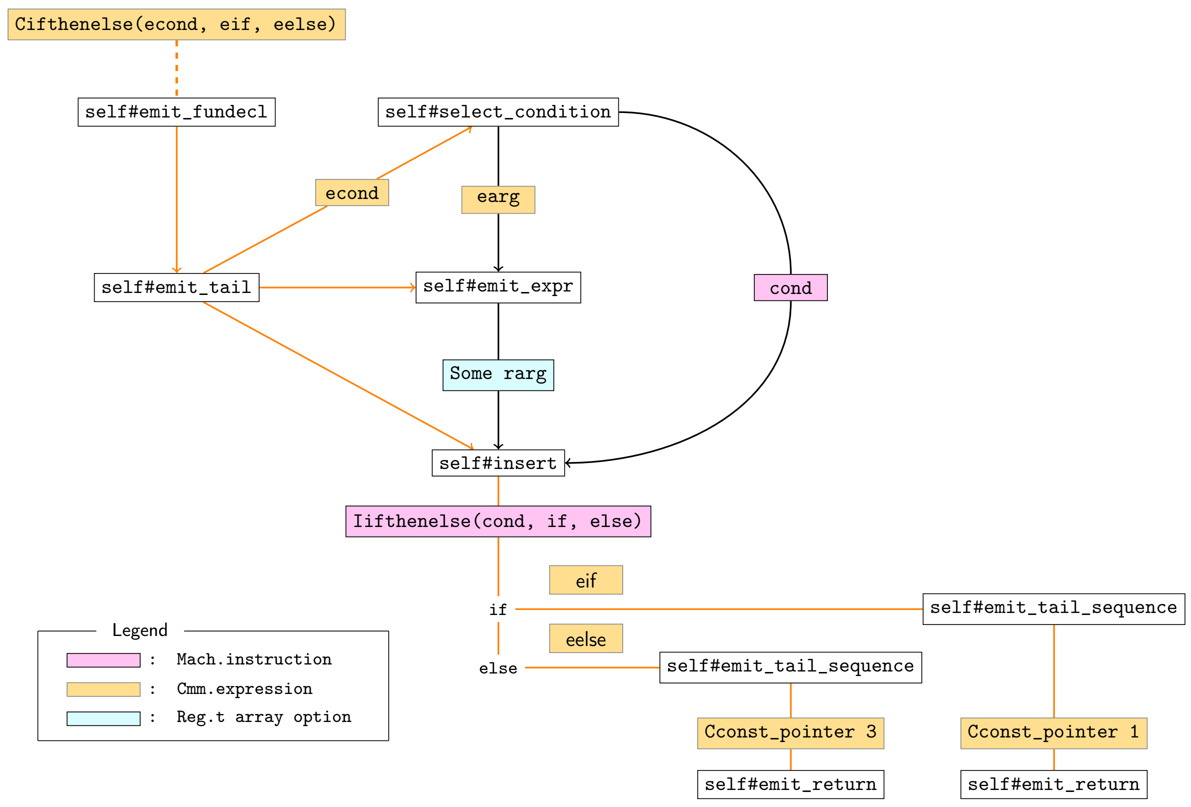


Figure 3.4: *This annotated control-flow-like diagram shows how the unaltered OCaml compiler deals with an if-then-else statement like that found in the* `is_none` *function.*

```
camlTest__is_none_1002:
L101:
    oceqi    a1, a0, 1
    # a1 = a0 == 1 ? 1 : 0
    ocvali   a0, a1, 1
    # a0 = (a1 << 1) + 1
    ret
```

Listing 10: *The resulting customised RISC-V assembly code for the* `is_none` *function*

argument to the Cmm expression `Cifthenelse`. For this example, it is a negated equals comparison between a `Cvar ident` and `Cconst_int 1`.

However, with the custom instructions there is no need to emit any of this code (like the loading of the immediate 1 into a register). Instead, using the `env_find` function from the `Selectgen` module, the register associated with the argument can be obtained. A fresh return register (of type integer in the RISC-V architectural sense) can be created by calling the super class' implementation of the `regs_for` method. Next, the two new instructions can be inserted passing the argument register, the fresh return register and the register produced by the instruction to the appropriate specific instructions. Finally, the effect of emitting a return instruction must be recreated by calling the `insert_moves` method along with the processor specific information for calling conventions to know which register a result should end up in.

The `ocvali` custom instruction can be used wherever a value is needed in the tagged integer format. This happens frequently, especially for converting 0 and 1 to 1 and 3. A method called `tag_int` in the transformation from Clambda to Cmm generates the shift and addition. This should remain for other target architectures, but the operation selection method can be overridden to match on this pattern and produce the desired custom instruction code. Given that the `ocvali` instruction can produce values for different immediate integers and providing the Cmm `Caddi` operation is within the immediate range described in the RISC-V `proc.ml` file, this instruction can be substituted for those two instructions.

The C portions of the OCaml runtime use a standard data-representation without a tag-bit, so any numerical values (including true and false) must be converted back to this format before being passed back to the OCaml program. The standard library of functions can make use of the `ocvali` instruction for converting values to their tagged-bit format. For example, the polymorphic comparison functions all use the macro `Val_int` to return their result (`Val_int` aliases `Val_long` which peforms the shift and add). Instead of GCC compiling the shift and add instructions, it can instead inline the `ocvali` instruction.

```c
typedef long intnat;
typedef unsigned long uintnat;
// Before using C code
#define Val_long(x) ((intnat) (((uintnat)(x) << 1)) + 1)
// After with inline assembly
#define Opt_val_long(x) \
  ({ intnat val_long_res; \
  asm("ocvali %0, %1, 1" : \
  "=r" (val_long_res) : "r" ((uintnat) (x))); \
  (intnat) val_long_res; })
```

Listing 11: *A possible optimised version of the* `Val_long` *macro. Sometimes it is used with constants or in a case statement where values are needed at compile time so uses of* `Val_long` *need changing by hand. The* `O2` *optimisation level is required to inline these properly.*

3.5.2 Common Instruction Patterns

The previous example focused on a high-to-low level approach where idiomatic OCaml patterns were considered and their resulting RISC-V assembly analysed for potential optimisations. In this section the converse low-to-high approach is used. The motivation for this arose from evaluation of the dynamic instruction streams (see § 4.4). One common pair of instructions seen was `slli` and `add`. At first glance reminiscent of the `ocvali` instruction for converting to the tagged integer form. However, here the shift is given an immediate value of 2 whilst the add instruction is between registers.

Focusing on the pattern.ml test case, three forms of variants are pattern matched. Constant variants have constructors of arity zero, non-constant variants have constructors with arguments and polymorphic variants are untied to any particular type and can equally be used with or without arguments. This type of pattern-matching is ubiquitous in OCaml programming [43]. Pattern-matching is a higher-level abstraction of nested if-then-else statements, making them fast is crucial for performance in OCaml programs. Left unoptimised the if-then-else approach may be compiled to a series of branch statements and if the last pattern is frequently the matching clause, then this can result in many wasteful instructions and will rely heavily on appropriate branch predicting algorithms to help improve efficiency [44].

One optimisation which the OCaml compiler implements is using a jump table. This is a pattern which allows for skipping of conditional statements by moving to an appropriate instruction which jumps to related code in the pattern-match. Since constant variants are represented as enumerations (see § 2.1.4), they form the basis of an index which can be used to jump to the right program counter.

```
type const = A | B | C | D

let print_const = function
  | A -> print_string "A\n"
  | B -> print_string "B\n"
  | C -> print_string "C\n"
  | D -> print_string "D\n"
```

(a) *Pattern-matching over constant variants.*

```
L100:
  srai    a1, a0, 1
  la      t0, L101
  slli    t1, a1, 2
  add     t0, t0, t1
  jr      t0
L101:
  j       L107
  j       L106
  j       L105
  j       L104
```

(b) *Jump table in RISC-V assembly.*

Listing 12: *Pattern-matching over constant variants with more than three constructors generates a jump table.*

The assembly in Listing 12 converts the OCaml value to a regular integer by shifting right. It then loads the base address of the jump table before taking the integer values and converting them to multiples of 4 (instructions are 32-bit) and adding this to the base address as an offset to get to the corresponding jump.

A simple optimisation would be to combine the shifting left by 2 and the adding. This optimisation can also be used in the nonconstant variant form. The enumeration there takes place in the tag byte of the header. The value passed to the function will be the pointer to the variant starting at the first field as shown in Figure 2.2. From here, to access the tag-byte, the load-byte instruction can be used after offseting to the beginning of the block header i.e. lbu a1, -8(a0). This can then be combined as above, with the appropriate shift left, add and jump. Note the shift right is unnecessary as the integer is not stored in the OCaml format.

The jump table approach is only used for variants with more than three constructors, but this is common. A very clear and appropriate example would be the `selectgen.ml` file in the asmcomp directory of the compiler where large pattern-matching expressions over the Cmm expression and operation types is used. Mixing constant and nonconstant constructors can still make use of the optimisation as the assembly generated checks to see if the argument is an integer (`andi t0, a0, 1`) before branching to the appropriate jump table. For completeness, polymorphic variants cannot be optimised in the same way. As mentioned in § 2.1.4, polymorphic variants use a stored hash of the variant name at runtime in the first field of the block. At compile time, assembly is generated which loads the hash into a register and a decision-tree of branch statements guides the control flow to the approrpiate code for the correct polymorphic variant. The hash function theoretically uniformly distributes the names over the key space (here 64-bit values) and so it would be more complex to map these to a jump table offset.

This is a specific use-case of a much more common problem: *load effective address*. In *x86* architectures there is a separate instruction, (`lea`), for doing this [45]. When working with arrays, it is common to index a particular element of the array. As they are stored as a contiguous block of memory, the index needs to move across multiple memory addresses to get to the next element. On a 64-bit architecture, the fields of an OCaml block are 8 bytes and so the indexing is in multiples of 8. With integers already being represented as tagged-integers, the shift required is only by 2 and not 3. See Appendix B for more information. The jump table code is generated by the `emit.mlp` file which produces assembly. The addressing of other objects can be matched in the `select_operation` method.

```
(* From asmcomp/riscv/emit.mlp *)
let emit_instr i =
 match i.desc with
  | Lswitch jumptbl ->
   let lbl = new_label() in
   ` la {emit_reg rt1}, {emit_label lbl}\n`;
   ` oclea {emit_reg rt1}, {emit_reg i.arg.(0)}, {emit_reg rt1}\n`;
   ` jr {emit_reg rt1}\n`;
   `{emit_label lbl}:\n` (*...*)

(* From asmcomp/riscv/selection.ml *)
method! select_operation op args dbg =
 match (op, args) with
  | (Cadda, [arg2; Cop(Clsl, [arg1; Cconst_int 2], _)])
  | (Cadda, [Cop(Clsl, [arg1; Cconst_int 2], _); arg2]) ->
   (Ispecific Ioclea, [arg1; arg2]) (*...*)
```

Listing 13: *Making use of the load-effective address (*`oclea`*) instruction for array-indexing and jump-table indexing.*

Additionally, the instruction-fusion analysis also highlighted many immediate adds followed by stores and similarly many loads followed by immediate adds. By inspecting the executables, the cause was generally manipulating the stack pointer and storing (or loading) the return address for subroutines — function prologues and epilogues. Whenever a function calls another function, it must store the return address on the stack (along with any other arguments) and so the caller must allocate stack space before invoking the callee. This is very common and to analyse compressing this action, two new instructions were added: **OC**aml **f**unction **ent**er and the **OC**aml **f**unction **ex**it.

```
// OCFET
require_rv64;
sreg_t stack_pointer = sext_xlen(RS1 + insn.i_imm());
MMU.store_uint64(stack_pointer + (-insn.i_imm()-8), READ_REG(X_RA));
WRITE_RD(stack_pointer);

// OCFEX
require_rv64;
WRITE_REG(X_RA, MMU.load_int64(RS1 + insn.i_imm() - 8));
WRITE_RD(sext_xlen(RS1 + insn.i_imm()));
```

Listing 14: *The* `ocfet` *instruction first tries to store the return address and the computed stack pointer offset in case there is a store fault and the instruction is re-run. This highlights that in practice this optimisation might be difficult to implement in hardware, but it would suggest that either a compression scheme from one to two instructions may be useful and failing that, extending OCaml to include the normal compressed format would reduce must of these pairs of instructions to* `c.addi` *and* `c.sd` *or* `c.ld`*. Also note that this is a 64-bit only instruction given the assumption made on the address size (8).*

All of the OCaml-RISC-V customisations were built into the compiler under a new `-riscv` flag. This meant code could be compiled normally or letters passed in, to enable different optimisations. For example, `-riscv aj` would add the arithmetic and jump table optimisations. This is potentially a solution for future work if it is decided to add some full extensions upstream to the compiler.

3.6 Analysis Tools

A large portion of this project focused on building tools that work with Spike to analyse execution logs in order to find promising macro-fusion opportunities (see § 4.4). What follows is a brief overview of the capabilities of `ospike`, a command-line tool, to analyse Spike execution logs along with some of the key design choices.

The general requirements were for the tool to work from standard input and to parse instructions at different levels of granularity. For example, an add followed by a load — should the registers matter, should the address (i.e. the function) they belong to matter etc. The tool also had to handle different address ranges so noisy instructions from the proxy kernel or interrupt handlers could be ignored. Lastly, it had to be able to compare groups of instructions to find common pairs (or more). To do this a simple `Buffer` module with a fixed capacity is filled from the execution log and drops instructions as it reaches capacity. The instructions implement custom hashing and comparing functions, allowing for different levels of granularity. The `Buffer` module implements the requirements of the Core.Hashtbl.`Key` module[4] and importantly folds over the custom hash functions for the instructions.

The parsing makes use of OCaml's first-class modules to provide comparable implementations at runtime. Using a functor which expects a module of type `LineParser` the Parser.S module is implemented. Different parsing strategies can be used and tested with ease. These implementations, which are first-class module, are then stored in a hashtable and from the command-line a particular implementation can be selected [30, 46]. This makes it easier to compare different regular expression packages and a standard library string manipulation implementation[5].

[4]Core is an alternative standard library implementation for OCaml developed by Jane Street `https://github.com/janestreet/core`

[5]In practice a version of `ospike` was compiled with debugging information and the different implementations profiled using `gprof`. Initially only the regular expression implementation existed but analysis showed the string implementation to be approximately five times faster.

3.7 Repository Overview

Test	Description
`riscv-ocaml/asmcomp`	The `riscv-ocaml` repository is the cross-compiling, 4.07.0 version of the OCaml compiler based on the port by Nicolás Ojeda Bär and K. C. Sivaramakrishnan. The `asmcomp` directory contains the assembly compiler portion of the OCaml compiler and the code for specific target architectures. Of particular importance are the `selectgen.ml` and `./riscv/selection.ml` files which implemented the Cmm to Mach transformation.
`riscv-binutils-gdb`	The code for assembling programs using the GNU assembler maintained by the RISC-V foundation — modifications were needed in order to target new custom instructions in the RISC-V ISA.
`riscv-isa-sim`	The functional, non-cycle accurate RISC-V instruction set architecture simulator also maintained by the RISC-V foundation which also needed modification to support custom instructions.
`opam-cross-shakti`	The `opam` repository, extended to build Yojson, a low-level parser for JSON written in OCaml.
`ospike`	A command-line tool developed for analysing the Spike log of executed instructions built on top of OCaml streams and "hashable" queues. The `lib` directory contains the `ospike` library and the `src` directory contains the command-line tool.
`riscv-benchmarks`	A collection of custom test cases which used idiomatic OCaml patterns along with some test cases from the Sandmark test-suite (see § 4.1) and a Yojson weather program to simulate a real-world weather reporting system and simple functions over this data. This repository is cross-compilable using `dune` and a custom Opam repository with cross-compiling compatible packages for RISC-V.

Table 3.1: Important repositories of code which made up this project.

4 | Evaluation

The main goals in this chapter are to highlight the testing framework used (§ 4.1), the impact of the compiler modifications on dynamic instruction counts (§ 4.3) and provide the analysis that lead to some of the optimisations by inspecting common instruction pairings and the most frequent instructions (§ 4.4).

Evaluating the changes is based on the simplicity that comes with a reduced instruction set approach to ISA design. The instructions are simple enough that any given instruction should only require a single CPU cycle. This was a critical factor in considering which instructions could be feasibly replaced. The hardware for doing a shift and add for example is relatively simple, and could be achieved in a single cycle. No instructions were replaced by those that would take more cycles and as such the relative, dynamic instruction count is a good proxy for performance and energy consumption. This is similar to the analysis by Andrew Waterman [21].

4.1 Test Framework

The testing framework must cross-compile to RISC-V and be a representative sample of possible common OCaml programs. The approach taken it to combine some custom test cases along with the more common test cases taken from the Sandmark[1] library of OCaml compiler benchmarks. In order to cross-compile, two `opam` switches were installed for cross-compiling with and without the customisations. `Opam` switches are isolated, concurrent installation prefixes containing their own list of available and installed packages. Commonly one version of the OCaml compiler is linked to a switch as the compiler used to build and install the packages [47]. Having a unified approach to package management and compiler versioning is useful for compiler development. In the Rust programming language for example, two tools are needed, cargo[2] and rustup[3]. Table C.1 describes each of the custom test cases.

4.2 Execution Log

Spike, the RISC-V ISA functional simulator, models a RISC-V core and cache memory system. A functional simulator emulates the instruction set architecture on an instruction-by-instruction basis, producing a trace of program execution. It is not meant to be cycle-accurate. In this sense, Spike is a tool for checking compilation correctness and providing evidence for potential optimisations without going through complicated processes like modifying existing cores written in Verilog and synthesising to an FPGA [48]. It is important to be aware that the improvements seen in the functional simulator do not necessarily map to improvements in a true pipelined, branch-predicting processor. They do, however, act as a good indication of what is possible. Spike also offers data and instruction cache emulation to help achieve more real-world program execution.

[1]`https://github.com/ocaml-bench/sandmark` (Accessed: 5th April 2020)
[2]The Rust package manager: `https://doc.rust-lang.org/cargo/`
[3]The Rust toolchain (compiler) manager: `https://rustup.rs/`

Test	Description
`riscv/gc.ml`	Using the standard library's `Map` module, this extends it to create one which tracks it's current size. It then allocates many strings to the Map and when reaching capacity will remove strings too. The maps and strings are immutable objects and so many are created with short life spans which causes the garbage collector to be called more frequently.
`riscv/someornone.ml`	Similar to the `zerotypes` test, this case randomly generates lists of elements of type `int option` and then maps over it using the `is_none` function.
`riscv/pattern.ml`	Testing pattern matching over constant, non-constant and polymorphic variants which is very common in OCaml programs.
`riscv/intfloatarray.ml`	Testing the standard library's polymorphic comparison operators on lists of type `int list`, `float list` and `(int * float) list`.
`riscv/sorting.ml`	Similar to the above test case only performing merge sort using the polymorphic comparison.
`riscv/zerotypes.ml`	Testing the `is_none` operator over `int option`, `int list` and `int list list` (for the empty list check).
`sandmark/numerical/fft.ml`	Computes the Cooley-Tukey fast Fourier transform algorithm. In tests a small size of 1024 was used.
`yojson/yj.ml`	Using the Yojson library a simple json writing and parsing tool for fictional weather data was implemented as a more real-world example.

Table 4.1: Explanation of the custom test cases designed to exploit some of the common OCaml patterns and polymorphic code, as well as providing some real-world examples to analyse the impact of the OCaml compiler modifications.

For running the tests a proxy kernel is placed between the test and the simulator to provide system call functionality. The software emulates a memory system, including page faults and during testing it became apparent that this was going to dominate the instruction counts. To avoid losing information, only the instructions at the address range accessed through taking the `objdump` of the executables were considered. Since the executables are statically compiled this also includes standard library functions and the runtime (e.g. the garbage collection code). Spike and the proxy kernel use the same addresses as those in the static executable making it easy to only look at relevant OCaml code.

4.3 Quantitative Evaluation

The test cases were compiled using default parameters for `ocamlopt` (the native-code compiler for OCaml) and the C compiler used `-O2` optimisation. Given the deterministic nature of the functional simulator, one run per test is sufficient in calculating total number or instructions. Spike was configured to use identical, set-associative, instruction and data cache models which were 16KB in size (128 sets, 2 ways and cache line sizes of 64 bytes).

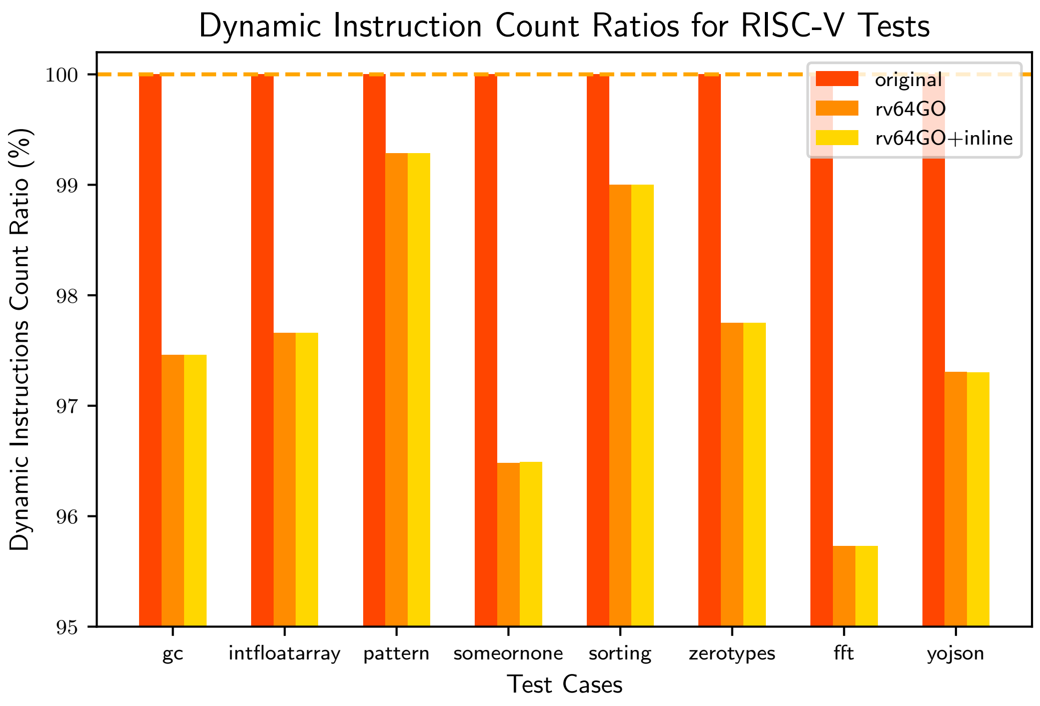


Figure 4.1: *RISC-V test cases plotted for the three versions of the compiler — with modifications (`rv64GO`), with modifications and inlining (`rv64GO+inline`) and without modifications (original). The percentage shows the reduction in dynamic code size as compared to the original compiler.*

From this analysis a few things are apparent. Firstly, the optimisations reduce the number of instructions which was part of the core deliverables for this project. The next question is by how much and why for the different test cases.

In the more real-world examples like the Fast-Fourier Transform (FFT) and the Yojson weather analyser, different combinations of the optimisations played an important role. For the FFT test `oclea`, `ocvali` and `ocadd` all occurred frequently. This makes sense given the use of oclea for array indexing, the ocvali instruction is used in the bit-reversing loop of the test and ocadd instruction is used in the *butterfly*[4] portion of the algorithm. The modifications managed to reduce the number of instructions by 4.3%. For the Yojson test, the same instructions proved useful in reducing the number of instructions and in addition to this the ocvali was useful in generating abstract data types from integers. It saw a reduction of 2.7%. Whilst small, these numbers are significant. Asanović et al. found that macro-op fusion code reduced the effective instruction count for RV64GC by 5.4%, figures similar to what has been shown here.

Inline assembly does not tend to make a significant difference in the OCaml runtime, this is possibly due to the inability of GCC to optimise as well with this instruction because it is inlined rather than being semantically defined in the compiler. This makes an interesting point about the runtime and garbage collection in OCaml. For the fft.ml test, the executed instructions from mostly OCaml code (including the standard library) make up approximately 54% of the executed instructions and for gc.ml it drops to just 43%. This is an important point, the reductions detailed above are only coming from OCaml code, yet a substantial amount of the program is spent in the runtime and garbage collection (written in C). A small test reusing the FFT test, shows that there were still 0.71% of the instructions that could have been compressed to one instruction due to loading of addresses and function prologues and epilogues in C code. Pushing through these optimisations would likely reduce dynamic instruction size even further.

4.4 Common Instructions and Instruction Fusion

Instructions that appear next to each other in the dynamic instruction log are ripe for instruction fusion. With the flexibility of the RISC-V ISA this can be achieved through the compiler, instruction stream analysis and the appropriate custom instructions. Although, typically this is done as part of the processor pipeline where fetched instructions are fused before being executed like the *x86* compare and branch instructions [50].

Figure 4.2 shows, for each of the test cases (described in Table C.1), the proportion of instructions that made up the OCaml part of the program. As expected a majority of the executed instructions are only a small subset of the total number of instructions. In fact, many of the commonly cited cases in the literature can be seen in this small group of test cases. As found by Andrew S. Waterman in his proposal for the compressed instructions, addi, ld and sd make up some 44.23% of all static instructions in SPEC CPU2006 [51]. Dynamically the same cumulative percentage also includes fld (floating-point double-word), lw and add [17]. The dominance of addi is partly explained by its ability to generate constant values and also effectively load immediates into registers. The figures also support the previous claim that approxmately 30% of the instructions are data-oriented loads and stores.

Other patterns in the data are explainable, the garbage collection test case sees many loads and stores. Calling the OCaml garbage collector invokes a hand-crafted function called caml_call_gc in an assembly file (asmcomp/riscv.S) which wraps a call to the garbage collector which saves and loads all registers to the stack when invoking garbage collection.

[4]A small computation for computing complex numbers used in the FFT algorithm [49].

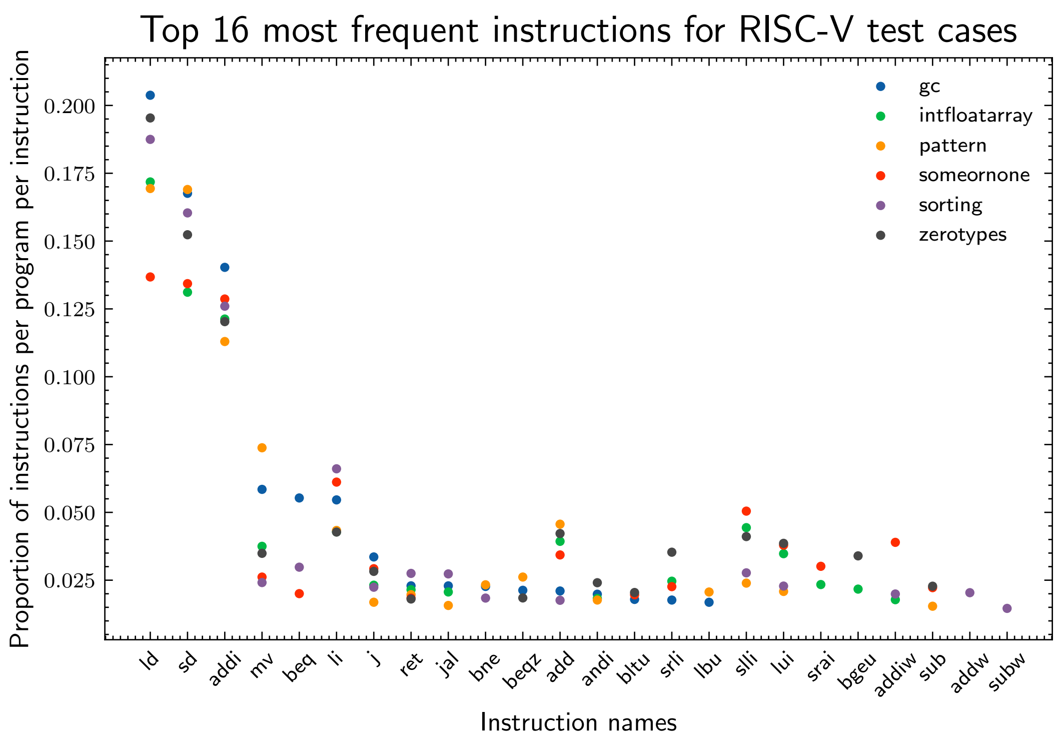


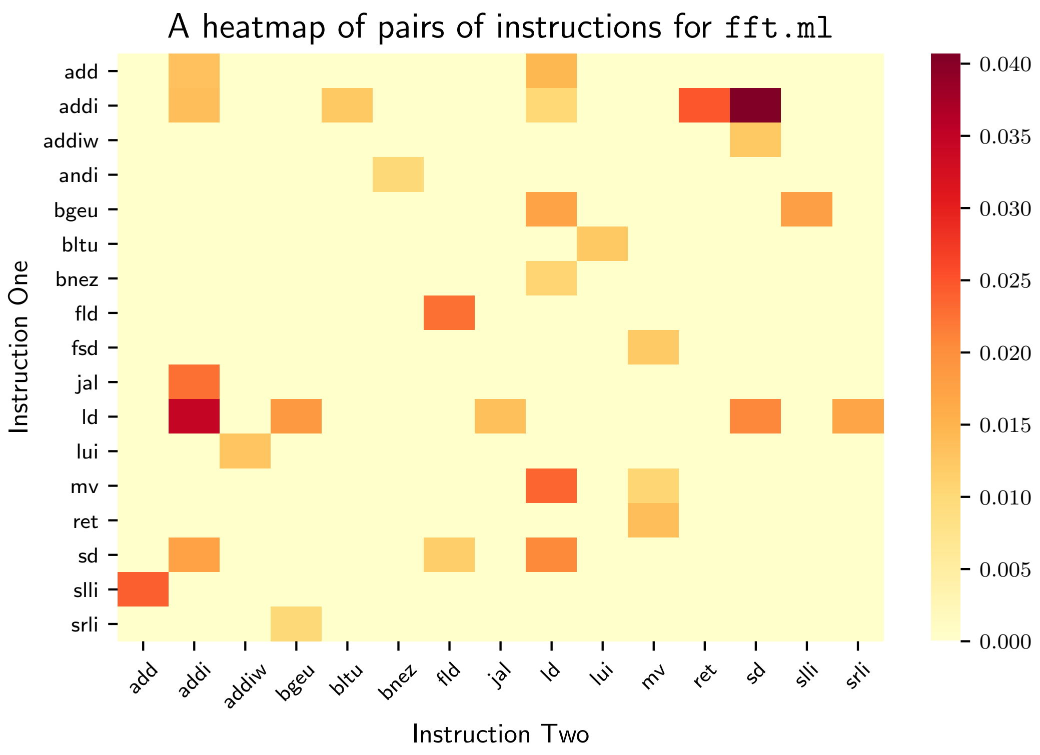
Figure 4.2: *A plot showing the top 16 most frequent instructions for each of the test cases, the numbers have been normalised to sum to one so the y-axis shows the proportion of the program spent executing that instruction. This is only the instructions that make up the statically compiled OCaml executable (the test case code, imported modules and runtime) — so no additional instructions, for example waiting for page tables to load, is included.*

For analysing the opportunities for instruction fusion, it makes sense to analyse the co-occurrence of different instructions within the dynamic instruction stream before trying to trace back to the original OCaml code which caused them to appear in the first place. Figure 4.3 shows a heatmap for commonly occurring pairs of instructions. Note that ordering is significant and hence the heatmap is not symmetric. Again, a few instructions dominate the graph.

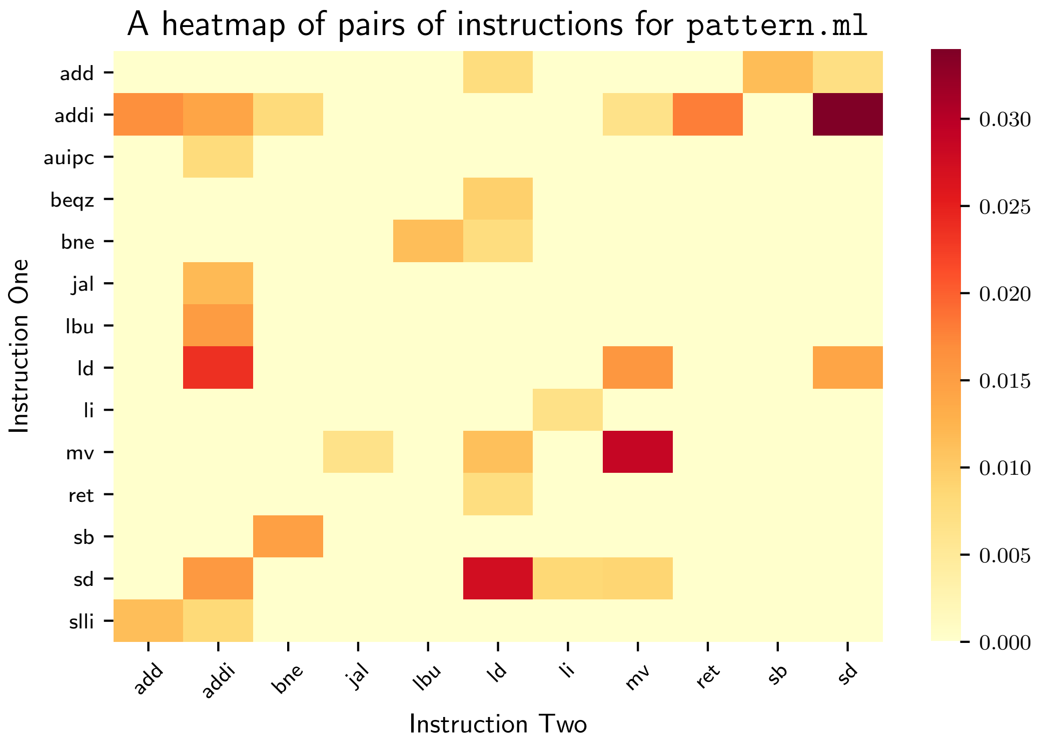
The data from Figure 4.3 shows the most common pairs of instructions from the execution of `pattern.ml` and `fft.ml` when the address range was constrained to just that generated by the files `pattern.ml`, `utils.ml`, the OCaml startup and exit code and the encoding of curried functions and similar OCaml constructs. Some interesting and reasonable conclusions can be made from this data.

- *Loads and stores are often followed by other loads and stores* — some of these multiple loads and stores are a consequence of currying functions. A curried function is a function which accepts only one argument and may return another function when applied. Listing 15 shows explicit currying examples and also an example of how currying allows for partial application and therefore partial body evaluation. In example 2, the Lucas number[5] inside the `add_lucas` function is computed once for the entire list [52]. For this to be possible the compiler builds primitive versions of

[5]Lucas numbers are similar to Fibonacci numbers except with different starting numbers



(a) *Heatmap for* `fft.ml`



(b) *Heatmap for* `pattern.ml`

Figure 4.3: *A heatmap showing the normalised counts of instructions executed one after another in the* `pattern.ml` *and* `fft.ml` *test cases. Note that stores followed by stores and loads followed by loads have been removed as these were by far the most common.*

curried functions and stores closures of partially evaluated curried functions. It will store addresses for partially applied functions along with arguments or intermediate results and return this closure, for example the curried Lucas function returns the address of the addition part, along with its arity and the value of the computed parameter. This can account for some of the series of loads and stores. If the entirety of the OCaml code was analysed this number of loads followed by loads and stores followed by stores would likely be higher as calling the C runtime for things like garbage collection often results in registers being stored and reloaded upon return.

- *Loads are often followed by unconditional jumps* — this is also to be expected as it is likely an address that should be jumped to that is being loaded.
- *Shifting left is often followed by an addition* — Some of these have been accounted for in the description of converting to the OCaml integer format. Upon closer analysis of the generated code, another use case is in jump tables made during pattern matching. This was the inspiration for the optimisations suggested in § 3.5.2.
- *Add-store and load-add pairs were common* — after analysing compiled programs, one of the biggest causes of this was functions calling functions requiring stack space to be allocated and return addresses to be stored (and the inverse). This was the inspiration for the `ocfet` and `ocfex` instructions in § 3.5.2.

```
(* A simple recursive function *)
let rec lucas n =
  if n = 0 then 2 else if n = 1 then 1
  else lucas (n - 1) + lucas (n - 2)

(* Exploiting the partial evaluation *)
let add_lucas l =
  let luc_num = lucas l in
  fun n -> luc_num + n

List.map (add_lucas 30) [1; 2; 3; 4; 5]
```

Listing 15: *Illustrating the power of currying for partial body evaluation that reduces computation in certain situations similar to that described in Xavier Leroy's "The ZINC Experiment: An economical implementation of the ML language" [52]. With the power of currying and first-class citizenship, functions and invoking functions from within functions is very common in OCaml.*

Results from the instruction stream analysis (loading addresses and stack-space allocation to a certain extent) support what Asanović et al. argue in their 2016 paper “The Renewed Case for the Reduced Instruction Set Computer: Avoiding ISA Bloat with Macro-Op Fusion for RISC-V” [53]. They also discovered frequent pairings for indexed loading or loading 32-bit immediates into registers using `lui` and `addi` (seen in Figure 4.3). Rather than optimising in the compiler, a less invasive approach would be to use the macro-op fusion at the hardware level to descrease the effective instruction count.

5 | Conclusion

The two main goals outlined in § 2.3 were to optimise the `is_none` function described in the project proposal and to have a system of evaluation for the modifications to the compiler. These goals were achieved in § 3.5.2 and the evaluation in § 4.3. The analysis resulted in a better understanding of the OCaml internal data representation, which led to more ideas such as modifying the garbage collector and runtime and reducing the number of instructions in the pattern-matching jump table. This was a direct consequence of the instruction fusion analysis in § 4.4. The modifications were made possible in large part thanks to the open recursion enabled through OCaml's object-oriented system seen in § 2.1.4, which supports the idea of a multi-paradigm, general-purpose programming language.

This project has also outlined a possible workflow (see Figure 5.1) for exploring this area of customising the RISC-V ISA for a particular compiler and programming language.

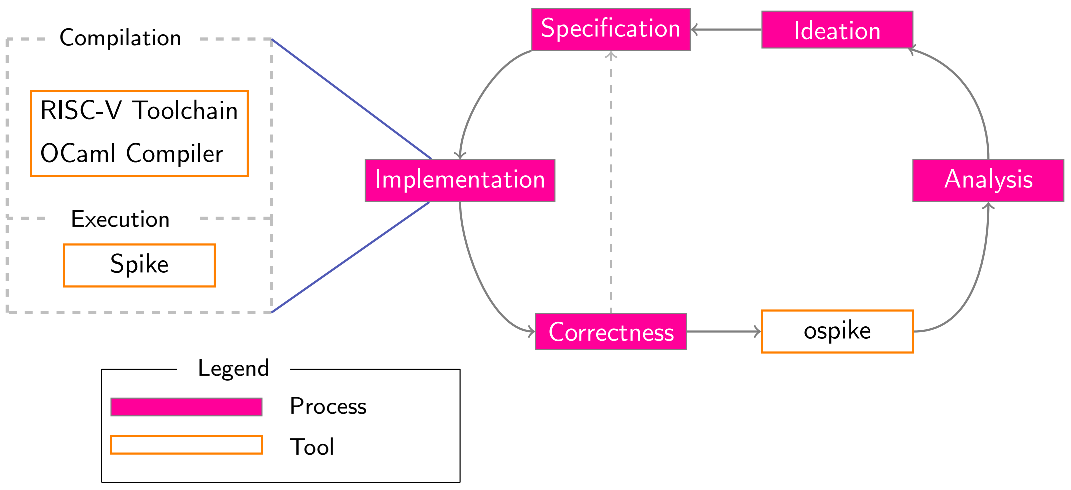


Figure 5.1: *A workflow for discovering optimisations — going from real-world programs, to finding patterns, creating and specifying custom instructions, simulating them and checking correctness and analysing their effect. The compiler portion of the code happens to be OCaml (with the required C compiler) but this could be any language and compiler targeting RISC-V.*

The impact on performance and energy consumption are the most important findings. The modfications made to the OCaml compiler resulted in the best case (for the tests used) reduction of 4.3% in dynamic instruction count. With special care being given to not alter CPI or clock frequency, the performance is directly proportional to this reduction in dynamic instruction count. At the same time, with only simple hardware being necessary to implement the language-specific instructions, the total energy consumed is also likely to decrease based on the theory outlined in § 2.1.2. When scaled horizontally across many devices (in a data-centre for instance) and over long periods of time (years), the resulting energy saved could be impressively large.

The framework established as part of this project, the compiler customisations, RISC-V tool modifications and the `ospike` analysis tool, should enable even more novel insights into how OCaml programs behave. It demonstrates how RISC-V, as the hardware-software interface, can be moulded to help reduce power-consumption without sacrificing performance.

5.1 Future Work

Below are some possible extensions to the work achieved in this project.

- *Closures, currying and the runtime* — analysis of some common patterns across OCaml code showed that curried functions appear fairly frequently, containing stores and loads which could be optimised or compressed. Compressing multiple stores and loads is often discussed and the analysis in this project showed that it could greatly reduce static executable sizes, although it goes against the one to one mapping of compressed to expanded instructions set out by the RISC-V Foundations [1].
- *Modifications to GCC* — OCaml relies heavily on its runtime and garbage collector where loading effective addresses, stack-pointer manipulation followed by return address restoring and loading larger immediates all take place — extending the OCaml compiler modifications to GCC would likely reduce the number of instructions further. If similar reduction percentages were seen with the C code, dynamic instruction count could see a reduction approaching 9%.
- *RV32G and Compressed Instructions* — Before supporting other extensions it makes sense to add support for 32-bit RISC-V and for compressed instructions which can drastically reduce file sizes. For the most part this is handled by the assembler, however, some data structures like the jump table assume fixed 32-bit instructions, but the jumps may be compressed to 16-bits which would break this construction.
- *A Custom Shakti Core* — running statically-compiled executables on Spike shows that the compilation process works and that, theoretically, performance gains can be met along with lower power-consumption. However, it would be interesting to modify existing core designs and in particular the 64-bit, 5-stage pipeline, Shakti C-64 processor in order to run more rigorous testing [54]. From there, if synthesised to an FPGA, more evaluating metrics like power-consumption would lead to more reliable results. In addition to this, adding macro-op fusion to this core would also make for an interesting project.

5.2 Lessons Learned

This project combined many areas which were fairly new to me, including functional programming at scale, programming-language design, compilers (in particular assembly generation) and computer architecture. Toy languages such as "*Slang*"[1] introduce concepts such as parsing and interpreting a language, whereas this project was more focused

[1]The lambda calculus inspired language from the Part IB Compiler Construction course — `https://github.com/Timothy-G-Griffin/cc_cl_cam_ac_uk/tree/master/slang`

on low-level native assembly generation along with data representation. This project explored the software-hardware interface, a notoriously difficult divide and how RISC-V may help bring flexibility to it. Understanding the RISC-V ISA meant understanding modern RISC-V core implementations and general computer architecture design. The reliance of the OCaml compiler on a C compiler and an assembler meant more time was spent learning the internals of GCC and modifying C and C++ code to support the customisations fully, than was originally intended. The lack of a similarly powerful type system and the ability to arbitrarily cast values to different types in C, only strengthened my enjoyment in working with OCaml.

This project has been possible thanks to open-source development. In a world where devices need to be less power-intensive, more secure and still perform well, the various open-source projects explored have offered a glimpse into what this future could look like. OCaml offers secure code with its powerful type system, whilst showing how operating system design can be made more light-weight and power-efficient thanks to MirageOS [55, 56]. Match that with the modular, flexible and open RISC-V ISA specification enabling more open-source core implementations like the Shakti-T, which can catch temporal and spatial memory attacks; the full-stack, from software to hardware and across the interface seems more harmonious in achieving performance and security at a lower power [57].

Bibliography

[1] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic. The risc-v instruction set manual v2.2. https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf, 2017. Accessed: 2020-03-21.

[2] Mark Bohr. A 30 year retrospective on dennard's mosfet scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.

[3] Mark Weiser. The Computer for the 21st Century. *Scientific american*, 265(3):94–105, 1991.

[4] Anil Madhavapeddy, KC Sivaramakrishnan, Gemma Gordon, and Thomas Gazagnaire. An architecture for interspatial communication. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 716–723. IEEE, 2018.

[5] B. Whitehead, D. Andrews, A. Shah, and G. Maidment. Assessing the environmental impact of data centres part 1: Background, energy use and metrics. *Building and Environment*, 82:151 – 159, 2014.

[6] Song Huang, Shucai Xiao, and Wu-chun Feng. On the energy efficiency of graphics processing units for scientific computing. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–8. IEEE, 2009.

[7] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini. Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2700–2713, 2017.

[8] Brendan Barry, Cormac Brick, Fergal Connor, David Donohoe, David Moloney, Richard Richmond, Martin O'Riordan, and Vasile Toma. Always-on vision processing unit for mobile applications. *IEEE Micro*, 35(2):56–66, 2015.

[9] RISC-V Community. RISC-V GNU Toolchain. https://github.com/riscv/riscv-gnu-toolchain, February 2020.

[10] Nicolás Ojeda Bär. RISC-V Port of the OCaml Compiler. https://github.com/nojb/riscv-ocaml, February 2020.

[11] RISC-V Foundation. RISC-V Origin: Research at Berkeley. https://riscv.org/risc-v-history, February 2020.

[12] D. Patterson and A. Waterman. *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon LLC, 1 edition, 2017.

[13] ONiO. ONiO.zero: Solving the power bottleneck for Internet of Things. https://www.onio.com/technology.html, 2020. Accessed: 2020-04-25.

[14] RISC-V Foundation. RISC-V Genealogy Report and Collapsible Tree. https://riscv.org/risc-v-genealogy/, February 2020.

[15] P. Davide Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand, and L. Benini. Slow and steady wins the race? a comparison of ultra-low-power risc-v cores for internet-of-things applications. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–8, Sep. 2017.

[16] Charles Price. MIPS IV Instruction Set. https://www.cs.cmu.edu/afs/cs/academic/class/15740-f97/public/doc/mips-isa.pdf, 1995. Accessed: 2020-04-02 (page A-32).

[17] Andrew Shell Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, UC Berkeley, 2016.

[18] lowRISC. Ibex: a two-stage RV32 Core. https://ibex-core.readthedocs.io/en/latest/instruction_fetch.html#instruction-fetch, March 2020. Accessed: 2020-04-02.

[19] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.

[20] Lea Hwang Lee, Bill Moyer, and John Arends. Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In *Proceedings of the 1999 international symposium on Low power electronics and design*, pages 267–269, 1999.

[21] Andrew Waterman. Improving Energy Efficiency and Reducing Code Size with RISC-V Compressed. Master's thesis, EECS Department, University of California, Berkeley, May 2011.

[22] Mahmut Kandemir, N Vijaykrishnan, and Mary Jane Irwin. Compiler optimizations for low power systems. In *Power aware computing*, pages 191–210. Springer, 2002.

[23] RISC-V Foundation. RISC-V GNU Toolchain. https://github.com/riscv/riscv-gnu-toolchain/, March 2020. Accessed: 2020-03-19.

[24] RISC-V Foundation. RISC-V Tools. https://github.com/riscv/riscv-tools, March 2020. Accessed: 2020-03-20.

[25] RISC-V Foundation. RISC-V GNU Toolchain - Binutils. https://github.com/riscv/riscv-binutils-gdb/, March 2020. Accessed: 2020-03-19.

[26] RISC-V Foundation. Spike - The RISC-V ISA Simuator. https://github.com/riscv/riscv-isa-sim, March 2020. Accessed: 2020-03-20.

[27] RISC-V Foundation. The RISC-V Proxy Kernel. https://github.com/riscv/riscv-pk/, March 2020. Accessed: 2020-03-20.

[28] Y. Minsky, A. Madhavapeddy, and J. Hickey. *Real World OCaml*. O'Reilly, 1 edition, 2014.

[29] Simon Colin, Rodolphe Lepigre, and Gabriel Scherer. Unboxing mutually recursive type definitions in ocaml. *arXiv preprint arXiv:1811.02300*, 2018.

[30] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. The OCaml system release 4.10. https://caml.inria.fr/pub/docs/manual-ocaml/, February 2020. Accessed: 2020-03-20.

[31] J. Garrigue. Code Resuse through Polymorphic Variants. 2000.

[32] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.

[33] Didier Rémy. Type checking records and variants in a natural extension of ml. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 77–88, 1989.

[34] L. White. Advanced Functional Programming - Chapter 8: Row Polymorphism. https://www.cl.cam.ac.uk/teaching/1415/L28/rows.pdf, April 2020. Accessed: 2020-04-03.

[35] Jonathan Aldrich and Kevin Donnelly. Selective open recursion: Modular reasoning about components and inheritance. *SAVCBS 2004 Specification and Verification of Component-Based Systems*, page 26, 2004.

[36] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

[37] N. Ramsey, S. P. Jones, and C. Lindig. The C - - Language Specifiction version 2.0. https://www.cs.tufts.edu/~nr/c--/extern/man2.pdf/, February 2005. Accessed: 2020-03-20.

[38] H. Blanchard. Introduction to Assembly on the PowerPC. https://developer.ibm.com/technologies/linux/articles/l-ppc/, July 2002. Accessed: 2020-03-20.

[39] Jane Street Open Source. Dune: A Composable Build System for OCaml. https://dune.build/, March 2020. Accessed: 2020-03-25.

[40] N. Srivastava. Adding Custom Instruction to RISCV ISA and Running it on Gem5 and Spike. https://nitish2112.github.io/post/adding-instruction-riscv/, July 2017. Accessed: 2020-04-04.

[41] C. Wolf. RISC-V Bitmanip Extension. https://github.com/riscv/riscv-bitmanip/blob/master/bitmanip-0.92.pdf/, November 2019. Accessed: 2020-04-06.

[42] R. W. M. Jones. A Beginner's Guide to OCaml Internals. https://rwmj.wordpress.com/2009/08/04/ocaml-internals/, 2009. Accessed: 2020-04-16.

[43] F. Le Fessant and L. Maranget. Optimizing Pattern Matching. *ICFP*, 2001.

[44] James E Smith. A study of branch prediction strategies. In *25 years of the international symposia on Computer architecture (selected papers)*, pages 202–215, 1998.

[45] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf, 2016. Accessed: 2020-04-23.

[46] Jacques Garrigue. First-class modules and composable signatures in objective caml 3.12.

[47] OCamlPro, OCamlLabs. The opam manual. https://opam.ocaml.org/doc/Manual.html. Accessed: 2020-05-02.

[48] Lieven Eeckhout. Sampled processor simulation: A survey. In *Advances in COMPUTERS*, volume 72 of *Advances in Computers*, pages 173 – 224. Elsevier, 2008.

[49] Paul Heckbert. Fourier transforms and the fast fourier transform (fft) algorithm. 1995.

[50] Christopher Celio, Palmer Dabbelt, David A. Patterson, and Krste Asanović. The Renewed Case for the Reduced Instruction Set Computer: Avoiding ISA Bloat with Macro-Op Fusion for RISC-V, 2016.

[51] John L Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[52] Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA, 1990.

[53] Christopher Celio, Palmer Dabbelt, David A Patterson, and Krste Asanović. The renewed case for the reduced instruction set computer: Avoiding isa bloat with macro-op fusion for risc-v. *arXiv preprint arXiv:1607.02318*, 2016.

[54] Neel Gala, Arjun Menon, Rahul Bodduna, GS Madhusudan, and V Kamakoti. Shakti processors: An open-source hardware initiative. In *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*, pages 7–8. IEEE, 2016.

[55] H. Mehnert. Leaving Legacy Behind: Reducing carbon footprint of network services with MirageOS unikernels. https://media.ccc.de/v/36c3-11172-leaving_legacy_behind, 2019. Accessed: 2020-04-20.

[56] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, et al. Jitsu: Just-in-time summoning of unikernels. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 559–573, 2015.

[57] Arjun Menon, Subadra Murugan, Chester Rebeiro, Neel Gala, and Kamakoti Veezhinathan. Shakti-t: A risc-v processor with light weight security extensions. In *Proceedings of the Hardware and Architectural Support for Security and Privacy*, pages 1–8. 2017.

A | Intermediate Representations

Understanding the entire pipeline of the OCaml compiler is crucial in developing ideas for optimisations, the following small example shows the transformation from the typedtree representation which comes after the parsetree obtained directly from OCaml source code, to the Cmm intermediate representation code. Many of the IRs shown have been cleaned up of excess declarations of things like global constants or the roots of the garbage collector for readability. The function being compiled is the same as that in Listing 9.

```
typedtree:
Texp_function
Nolabel
[
 <case>
  pattern (test.ml[2,24+10]..test.ml[2,24+14])
   Tpat_construct "None"
   []
  expression (test.ml[2,24+18]..test.ml[2,24+22])
   Texp_construct "true"
   []
 <case>
  pattern (test.ml[3,48+10]..test.ml[3,48+16])
   Tpat_construct "Some"
   [
    pattern (test.ml[3,48+15]..test.ml[3,48+16])
    Tpat_any
   ]
  expression (test.ml[3,48+20]..test.ml[3,48+25])
   Texp_construct "false"
   []
]

lambda:
(seq
 (let (is_none/1002 = (function param/1004 (if param/1004 0a 1a)))
  (setfield_ptr(root-init) 0 (global Test!) is_none/1002))
 0a)
```

Listing 16: *The Typedtree and Lambda intermediate representations for the* `is_none` *function.*

The Listing above contains the typedtree and the lambda IR code for the `is_none` function. There are a few points to mention. Firstly, the internal representation of boolean values `true` and `false` are no different than `type boolean = False | True`. This is further evidenced by the subsequent transformation to integers 0 and 1 in the lambda representation

```
clambda:
(seq
 (let
  (is_none/1002
   (closure
    (fun camlTest__is_none_1002 1  param/1004 (if param/1004 0a 1a)) ))
  (setfield_ptr(root-init) 0 (global camlTest!) is_none/1002))
 0a)

cmm:
(function{test.ml:1,14-73} camlTest__is_none_1002 (param/1004: val)
 (if (!= param/1004 1) 1a 3a))
```

Listing 17: *The Clambda and Cmm intermediate representations for the* `is_none` *function.*

The transformation from Clambda to Cmm shows the tagged-bit integer representation being applied. Similarly we also see the conversion of the condition to checking for not 1, that is not 0 and returning false (1a) or true (3a).

B | Unboxed Float Arrays

Without any optimisations or compiler options being set, the `float` data-type in OCaml is represented as a full, double-precision floating point number. This requires it to be stored as a block with a single field containing the number. For numerical computations (such as the `fft.ml` test), it is common to store arrays of floats and perform arithmetic operations on them. If no special treatment was given to objects of type `float Array.t`, this would result in double indirection — first to get to the field of the array which would contain a pointer to a block containing the float. Instead, OCaml unboxes floats in the array and uses the special tag byte `Double_array_tag` to tell the Garbage Collector that the fields are floats [28].

```
let square_arr arr len =
  for i = 0 to len do
    arr.(i) <- arr.(i) *. arr.(i)
  done
```

(a) *A function for squaring each element of an array.*

```
camlTest__square_arr_1002:
L102:
 li      a2, 1
 bgt     a2, a1, L100
L101:
 ld      a4, -8(a0)
 srli    a5, a4, 9
 bleu    a5, a2, L103
 oclea   a6, a2, a0
 fld     ft0, -4(a6)
 oclea   s3, a2, a0
 fld     ft1, -4(s3)
 fmul.d  ft2, ft1, ft0
 oclea   s6, a2, a0
 fsd     ft2, -4(s6)
 mv      s7, a2
 addi    a2, a2, 2
 bne     s7, a1, L101
L100:
 li      a0, 1
 ret
L103:
 call    caml_ml_array_bound_error
```

(b) *The resulting RISC-V assembly, compiled with the modified compiler to use the* `oclea` *instruction.*

Listing 18: *An example where the* `oclea` *instruction can be used to index into a float array. This would work for any array in OCaml as the fields have uniform width.*

Regardless, fields of a block (the elements of an array) are uniform in size (8 bytes on 64-bit). This means indexing should happen at multiples of 8 from the memory address the pointer starts at. As OCaml integers are already shifted to the left one place and have the tag-bit set, they enumerate the odd numbers. The quickest way to convert the odd numbers to multiples of eight is to shift left two more places and load the fields at an offset of −4 bytes.

Array indexing (loading from offsets of a particular size) is common across many programming languages. For the OCaml runtime many garbage collection function use arrays and could benefit from a `oclea` instruction, however, C uses a more conventional representation of integers so the shift (on a 64-bit architecture) is by 3 (for multiples of 8). Two solutions exist: either create separate instruction or peform macro-op fusion in the processor pipeline to handle this [50].

Stepping through the assembly in provided, the first check is for a length of zero, `a2` holds the index value. From label `L101` the header of the array block is loaded into `a4`, and the top 54 bits extracted using the shift-right instruction to get the length of the fields. The index is compared with the length in case a runtime out-of-bounds error must be called. Otherwise the index is multiplied by 4, and the field is read using an offset of four bytes.

C | Software Licensing and Versions

The following summarises the software used in this project. All pieces of software use open-source licensing.

Software	**License**	**Version**
`riscv-isa-sim`	BSD-3-Clause	Commit `2710fe5`
`riscv-pk`	BSD-3-Clause	Commit `8c12589`
`riscv-gcc`	GPLv2	`v9.2.0`
`riscv-binutils-gdb`	GPLv2	Commit `82dcb86`
`riscv-ocaml`	LGPLv2.1	`v4.07.0`
`Yojson`	BSD-3-Clause	`v1.7.0`
`fft.ml` from the Sandmark test suite	MIT	Commit `02aeacd`
`Re2`	BSD-3-Clause	`v1.7.0`
`Alcotest`	ISC	`v1.1.0`
`Mdx`	ISC	`v1.6.0`
`Core`	MIT	`v0.12.4`

Table C.1: Software used in this project and their license and version number (or commit if no version was available).

The exact RISC-V code for the OCaml compiler that this project extends from is K. C. Sivaramakrishnan's work on top of Nicolás Ojeda Bär's original port to RISC-V.

D | Project Proposal

What follows on the subsequent pages is the original project proposal.

Computer Science Tripos - Part II - Project Proposal

OPTIMISATIONS ACROSS SOFTWARE AND HARDWARE USING RISC-V

Patrick Ferris
Pembroke College
pf341

October 25, 2019

Project Supervisor:	Mr. D. Allsopp
Project Originators:	Dr. A. Madhavapeddy & Mr. D. Allsopp
Project Overseers:	Dr. R. Mullins & Prof. P. Lio'
Director of Studies:	Dr. A. Prorok

1 Introduction and Description of Work

With the ubiquity of small, embedded devices and a surge towards greener solutions to counteract climate change, power-consumption is likely to become an increasingly relevant metric by which to evaluate programs. A hurdle that stands in the way of producing low-power code has always been the trade-off with performance and one bottleneck has been the inflexibility of modern instruction set architectures (ISAs).

RISC-V[1] looks to blur the divide between the software and hardware worlds by being a highly extensible but, at its core, simple ISA which anyone can manipulate to suit their needs. This flexibility enables software developers to produce RISC-V assembly code targeting their specific domain with the potential to reduce code size, improve performance and lower power consumption simultaneously.

OCaml[2] is a member of the ML family of programming languages based on the functional programming paradigm, but with object-oriented additions. This project will look at modifying the OCaml compiler to emit custom instructions to replace common patterns. This should minimise instruction fetching thus reducing power consumption. An initial candidate instruction pattern relates to OCaml's data representation format in which to distinguish between integers and pointers, the least significant bit is set to 1 or 0 respectively [1]. This allows integers (along with other

[1] `https://riscv.org`
[2] `https://ocaml.org`

types which map to it) to be unboxed at runtime to improve efficiency but this check stills needs to take place.

Below is the OCaml function for testing if the argument is `None` or `Some v` for a value v. The `None` value in OCaml maps to 0 whereas `Some v` is represented as a block with a pointer (the tag bit being set to 0). We can see the impact of this data representation with the `li a1 1` instruction as this is the integer 0 in OCaml.

```
(* val foo :
    'a option -> bool = <fun> *)
let foo = function
    | None   -> true
    | Some _ -> false
```

Listing 1: An OCaml function checking for `None`

```
camlInstr__foo_1002:
L101:
    li      a1, 1
    beq     a0, a1, L100
    li      a0, 1
    ret
L100:
    li      a0, 3
    ret
```

Listing 2: RISC-V assembly code for `foo`

It is this pattern of instructions that I propose would make an ideal candidate for optimising through a custom instruction. Consider an instruction `check_int rd, rs` which could perform a check on the least significant bit of `rs` and if it is a 1 it would place a 3 (OCaml's truth value) in `rd` otherwise it would place a 1 (OCaml's false value). There is not necessarily a unique solution for the custom instruction. How flexible the proposed solution is, is an important factor to consider as it will be proportional to how often it can be used.

2 Description of Starting Point

OCaml: Through the Part IA *Foundations of Computer Science* course and the Part IB *Compiler Construction* course I have some familiarity with the ML family of languages and in particular OCaml. I have begun reading, but have not finished, the *Real World OCaml* book [1].

Compilers: As part of the Computer Science Tripos in Part IB we took the *Compiler Construction* course which mainly focused on parsing and lexing and only briefly discussed backend aspects to the compiler such as assembly emission, optimisations and instruction scheduling.

RISC-V: There is an increased interest in this open-source ISA and this can be seen in university courses changing to use more of it including *Computer Design* in Part IB. I will continue to learn more about this ISA by reading *The RISC-V Reader* [2].

Related Work: Andrew Waterman's work in improving energy efficiency and performance by developing the compressed RISC-V variant for the most frequent instructions based on his analysis bares many similarities with the proposed work [4]. The work will rely on Nicolás Ojeda Bär's fork of the OCaml compiler which can target RISC-V[3].

[3]https://github.com/nojb/riscv-ocaml

3 Success Criteria

The project will be deemed a success if:

- The compiler can produce working examples of programs using the custom instructions.
- Using the metrics outlined in 3.1, show that theoretically the programs run using less power and analyse the impacts on code-size and performance.

3.1 Evaluation

The programs can be run on the Spike[4] RISC-V ISA simulator and the instruction stream analysed. As a proxy for power consumption, the number of fetched instructions can be used and then a comparison of the amount of data fetched in bits can be made. It may also be of interest to analyse the resulting impact on instruction cache utilisation which can be done using the Spike simulator.

Other metrics such as performance and code-size are important to analyse to get a more holistic overview of the effects the custom instruction set has. Static code-size is straightforward to measure from the binary files generated after compiling programs. Performance can be approximated by setting each instruction to cost one cycle except for more intensive actions like memory access, as in Andrew Waterman's work [4].

4 Extensions

This project acts as a proof of concept. Some possible extensions include:

- Formalising the common-pattern searching techniques that might be good candidates for custom instructions. Whilst domain-knowledge and a deep understanding of the language implementation are useful it may be interesting to explore how this process could be automated.
- Through instruction stream analysis of the most frequently used instructions, develop a method by which to compress these using custom instructions.
- Develop RISC-V oriented compiler optimisations aimed at lower total power consumed such as *cold scheduling* in which instructions are scheduled in such a way as to decrease the total hamming distance between successive instructions [3].

Each of these extensions seeks to broaden the current hypothesis within the OCaml ecosystem and so suitable evaluation methods are those used in the main part of the project: the proxies for power-consumption, compiled binary sizes and overall performance.

[4]`https://github.com/riscv/riscv-isa-sim`

5 Work Plan

10th October - 23rd October

Start of term, finalising project ideas, reading relevant papers and making the project proposal document.

Goal: *Project proposal completed and submitted.*

24th October - 6th November **Friday 25th October: PROPOSAL DEADLINE**

Continue to learn more about the OCaml language and improve my understanding of the OCaml compiler pipeline with a main focus on the intermediate representations (e.g. Lambda) and the final assembly code emission stage. Read more about CPU architectures and the RISC-V ISA.

Goal: *Detailed description of OCaml compiler backend and increased familiarity of RISC-V.*

7th November - 20th November

Investigate the proposed OCaml data representation optimisation through increased understanding between the OCaml code and the emitted RISC-V instructions.

Goal: *Documentation of the relationship between the two and prototype RISC-V custom instructions.*

21st November - 4th December

Integrating the custom instruction emission into the OCaml compiler pipeline and adding the instruction to the Spike RISC-V ISA simulator.

Goal: *Compiling well-established code using the modified compiler and running tests.*

5th December - 18th December **Friday 6th December: LAST DAY OF MICHAELMAS**

Perform initial testing of code and running the instruction stream analysis for various programs as well as noting binary code-sizes and impact on performance. More time may be devoted here to integrating with the OCaml compiler.

Goal: *Have a working, modified compiler for producing custom instructions.*

19th December - 1st January

Set aside time to go over the courses from Michaelmas term, this time is contingency if there is some delay in completing the goals from the previous weeks.

2nd January - 15th January

Begin drafting the progress report. Perform more rigorous testing of the custom instruction optimisation using large code-bases such as *MirageOS*[5]. Produce interesting data visualisations for findings.

Goal: *Produce the meaningful data for the evaluation of the core goals of the project.*

[5]`https://github.com/mirage/mirage`

16th January - 29th January **Thursday 16th January: FIRST DAY OF LENT**

Finish the progress report and produce the slides for the presentation - practice the delivery of it. With time permitting, choose a possible extension and begin sketching a plan to implement it.

Goal: *Progress report and presentation completed*

30th January - 12th February **Friday 31st January: PROGRESS REPORT**

Begin writing the draft preparation and implementation chapters of the project. With time permitting, continue work on extensions.

Goal: *Draft versions of the preparation and implementation chapters, have the project in a state of being completed with respect to the success criteria in the proposal document.*

13st February - 26th February

Discuss whether the core goals of the project have been achieved and produce a draft introduction and evaluation chapter of the project. With time permitting, continue work on extensions.

Goal: *Draft versions of the introduction and evaluation chapters.*

27th February - 11th March

Write the conclusion chapter of the project. With time permitting, continue the work on the extensions of the project and make a decision of how to evaluate these meaningfully and an esimated amount of work for these to be completed.

Goal: *A finished draft of the conclusion - submit the dissertation as a first draft.*

12th March - 25th March **Friday 13th March: LAST DAY OF LENT**

Take a break from the writing process, focus on revising Lent term subjects and possibly work on the extensions if these need some more work, ideally they will be completed.

26th March - 8th April

Begin polishing the dissertation using the feedback from anybody that has seen the initial draft.

Goal: *A good second draft of the dissertation - submit for further feedback.*

9th April - 22nd April

Final draft of the dissertation completed.

Goal: *Dissertation has been submitted.*

23rd April - 6th May **Friday 8th May: PROJECT DEADLINE**

Contingency for unseen circumstances up to this point. Ideally be revising for exams.

6 Resource Declaration

Personal Laptop

I will be writing my dissertation, modifying the OCaml compiler and performing evaluations on my laptop (*MacBook Pro 2017, Intel Core i5 2.3GHz, 8GB RAM*). In the unfortunate circumstance of laptop failure, I can use the facilities provided in the Intel lab (MCS) as these will have the tools necessary to complete my project.

To mitigate against SSD failure all work will be version-controlled using git and pushed to a private, remote GitHub repository as well as being backed-up to a cloud storage facility and copied to external USB drive. Using git will also mitigate against accidentally recursively removing all files.

References

[1] Y. Minsky, A. Madhavapeddy, and J. Hickey, *Real World OCaml*, O'Reilly, 1 ed., 2014.

[2] D. Patterson and A. Waterman, *The RISC-V Reader: An Open Architecture Atlas*, Strawberry Canyon LLC, 1 ed., 2017.

[3] C.-L. Su, C.-Y. Tsui, and A. M. Despain, *Low power architecture design and compilation techniques for high-performance processors*, Proceedings of the IEEE COMPCON, (1994).

[4] A. Waterman, *Improving energy efficiency and reducing code size with risc-v compressed*, tech. rep., University of California at Berkeley, 2011.