Imperial College
London

Imperial College London

Department of Electrical and Electronics Engineering

JITBoy - Exploring JIT Binary Translation as an Accelerated Means of Emulation

Author:
Yusuf Ismail
CID: 01345818

Supervisor:
Dr John Wickerson

Second Marker:
Dr James J. Davis

June 23, 2021

Final Report Plagiarism Statement

I affirm that I have submitted, or will submit, an electronic copy of my final year project report to the provided EEE link.

I affirm that I have provided explicit references for all the material in my Final Report that is not authored by me, but is represented as my own work.

Abstract

Emulation of a binary program written for a foreign instruction set architecture (ISA) is of great importance for academic, industrial, and recreational usage. Among many uses, emulation allows the continued usage of legacy software on new ISAs. Traditional emulation using interpretation does not always allow for emulation at an acceptable speed, especially for complex or real-time programs.

This project first develops a traditional interpreter to emulate foreign binary by translating and executing each instruction individually. The project then develops an emulator to explore the utilisation of just in time (JIT) compilation to accelerate the emulation through dynamic binary translation (DBT). The project also explores hybridising the interpreter and JIT techniques to develop a concurrent hybrid emulator that exploits the multiple logical processors available on all modern consumer CPUs to accelerate performance. The three emulators were written in C++ to emulate foreign programs written for the MIPS-1 ISA on a modern x86 CPU.

The developed JIT emulator was able to emulate the provided benchmarks at speeds far exceeding the interpreter. In some cases the JIT emulator was able to emulate programs at over 2000 mega instructions per second (mips) while the interpreter peaked at less than 60 mips. The hybrid was able to simultaneously beat the interpreter for heavy workloads and the JIT for light workloads.

Acknowledgements

I would like to thank my supervisor Dr John Wickerson for his continued advice and support throughout this project. I would like to state my appreciation to my friends and family for their support and companionship throughout my studies.

Contents

1 Introduction

1.1 Motivation

Emulation of software written for a different instruction set architecture (ISA) is frequently required throughout the industry both by developers and non-technical end users. This ability to emulate software both correctly and efficiently is greatly prevalent in recent times. Emulation allows developers and researchers to develop, debug and utilise software for a foreign ISA without the need for additional hardware, which can significantly reduce costs, but only if the emulation is adequately fast to substitute for the real hardware. Popular examples of such include QEMU [48].

Another use of emulation is the ability to emulate legacy software that cannot be feasibly rewritten. Many industries continue to use old software as it is too expensive or difficult to rebuild, and in some cases, such as those using IBM mainframes [71], can end up locked into restrictive and/or old hardware choices due to their programs being written for ISAs that are no longer popular. Recently, this has become important as ever for many average consumers. With recent industry events, notably including Apple moving their desktop line to use their in-house ARM-based Apple Silicon CPUs [8, 10], many consumers are now finding themselves needing the ability to emulate old, foreign software on their new hardware. It is in the industry's utmost interest that this emulation is fast and accurate, otherwise the user experience of their new hardware offerings will be severely affected. Apple developed Rosetta 2 [8] to solve this problem.

Another large motivation for efficient emulation of foreign code is in the enthusiast space; many enthusiasts want to run deprecated software such as old video games written for hardware that is no longer manufactured, and thus must resort to emulation. Popular examples of such emulators include Dolphin [25], an emulator for the Nintendo GameCube and Wii systems, and PCSX2 [42], an emulator for the Sony PlayStation 2 system. In this case performance is key due to the real-time nature of video games. Emulation of old games has also been used in the industry by 1st parties as a way to sell older games without rebuilding them. Examples of this include Nintendo's release of *Super Mario 3D All Stars* [21] and the Microsoft Xbox One's ability to play select Xbox 360 games [37].

1.2 Why a JIT?

Binary translation, and in general, execution of any program, can be handled with one of three fundamental techniques:

- Interpretation
- Ahead of Time (AOT) Compilation
- Just in Time (JIT) Compilation

Binary translators are typically implemented as interpreters. This is a widespread technique for system emulation as interpreting is fairly easy and largely agnostic of the host system. However, it has a large per-instruction overhead as the interpretation needs to be executed every time the instruction is emulated.

AOT compilation can be the fastest technique of execution, as the program is fully translated to the native host machine code before execution. Since compilation is only needed once, much more time can be spent on aggressive optimisations to improve the performance of the generated code. On the other hand, the lack of runtime knowledge prevents the AOT compiler from making any host specific optimisations. AOT compilers are limited by the fact that all decisions must be made statically, and none can be deferred to execution time. In practice, it is impossible to statically and comprehensively determine which contents of a program correspond to instructions and what corresponds to data, especially with the possibility of self modifying code. Given this, an AOT compiler alone is not a solution that could be extended to commercial-grade emulators as its emulation correctness is fundamentally limited to what it can statically determine to be executable code.

JIT compilation strikes a balance between the two. Some unit of source code, which we will refer to as a source block, is compiled just before it is needed for execution. This is typically not as fast as a full AOT compiler because the full compilation needs to occur at runtime, adding additional runtime costs; it can, however, have a much lower per-instruction execution overhead than an interpreter, if implemented well. It provides the following advantages over an AOT compiler:

- Self modifying programs can be supported as the compiler exists and is present at the time of execution.
- The latency of execution, or the time to first execution, is lower because the entire program does not need to be compiled before execution can begin.
- The machine code can be generated specifically for the host system, allowing system-specific optimisations to be made such as the use of modern ISA extensions, e.g. SIMD.
- Tiered compilation techniques can be supported as the execution can be analysed for hotpaths, which can then be recompiled with greater optimisations.
- Highly dynamic language features, such as those involving runtime code generation, can be supported. An example of this is allowing a regex to be JIT compiled to an extremely fast finite state machine (FSM).

Since compilation is only required once per source block, the eventual performance of the JIT emulator should be very high, in some cases close to native. This is a stark contrast with the interpreter which must perform all the work associated with a given instruction every single time it is executed. This limits the performance of the interpreter due to the constant need to decode and emulate the foreign instructions. On the other hand, the JIT compiler can in theory reach very high performance for program fragments such as hot loops, where the one time compilation overhead becomes irrelevant to the continued execution overhead.

With everything considered, a JIT compiler is the best option to achieve greater performance over an interpreter without fundamentally limiting correctness like an AOT compiler based emulator would.

1.3 Hybrid Emulation

While the JIT emulation should perform better than the interpreter in most cases due to its far reduced per-instruction overhead, as explored in subsection 1.2, this won't always be the case. Since JIT compilation involves a far larger overhead due to the translation required, we would expect the JIT emulator to have worse responsiveness than the interpreter and perform poorer for light workloads.

In an attempt to mitigate the weakness of either technique we propose the JIT-Interpreter hybrid emulator. Conceptually, the hybrid would asynchronously JIT compile source blocks off of the main thread whilst executing code on the main thread. In the case that a block isn't compiled yet, it would use the interpreter as a fallback.

In theory, this would avoid the large overhead associated with JIT compiling a source block; the compilation would still occur, but by deferring it from the main thread the emulator can simultaneously use the interpreter for its low overhead. At the same time, the emulator should be able to reach the same peak performance as the JIT emulator as it will eventually compile all blocks and no longer use the interpreter fallback.

Furthermore, neither the interpreter or the JIT emulator can trivially use multiple CPU cores. This leaves much to be desired as all modern desktop systems contain multiple logical processors. The hybrid emulator can potentially make better utilisation of these logical processors by concurrently JIT compiling multiple source blocks at once on several threads. In theory, this should allow for higher performance than the JIT emulator as it is able to perform more work at once.

1.4 Aims and Objectives

The project aims to accelerate binary emulation beyond the speeds possible with traditional interpretation based emulators. The specific criteria by which this is judged will be explored further in subsection 3.3. This will be achieved by completing the following objectives:

1. A traditional interpreter will be developed as a performance and functionality baseline.
2. The JIT compiler based emulator will be developed in an effort to outperform the interpreter, demonstrating the viability of JIT compilation as technique for accelerating binary translation.
3. The concurrent JIT-Intepreter hybrid emulator will be developed in an effort to mitigate the weakness of the JIT emulator and better utilise the multiple logical processors on modern systems.

1.5 Report Structure

The report first explores the current background and literature on binary emulation in section 2.

Next, the project is scoped. Specific details about the functionality and extent of the project are determined and the research questions used to describe the success of the project are proposed. This is covered in section 3.

The implementation of the project is then covered in section 4. This begins with the technology stack used and the high level architectural design of the key components such as the various emulators. It then dives into the more interesting implementation details.

With implementation complete, section 5 focuses on testing. This section describes the methodology behind the testing and the framework design. It describes the different metrics and criteria developed, but will not focus on any individual tests or results. This is covered by section 6 which shows and evaluates the test results and aims to answer the research questions proposed in section 3.

Finally, section 7 concludes the report and summarises the evaluation performed in section 6. This section covers the implications of the results before exploring further work suitable for the project.

Additionally, section 8 contains a brief user guide for building and running the finished project. section 9 clarifies any potential issues regarding the legality of the project.

2 Background

This section will explore the existing work in the field of JIT accelerated binary emulation. subsection 2.1 will focus on academic work and research whereas subsection 2.2 will look at the application of these techniques in industry, both in commercial and open source projects.

2.1 Academia

Several papers have investigated utilizing JIT compilation to accelerate binary emulation, otherwise known as dynamic binary translation (DBT). Mark Probst [46] and others [65] investigate a JIT-Interpreter hybrid to emulate foreign binary programs. Unlike the hybrid model proposed, I will be exploring a concurrent architecture to exploit the multiple cores present on modern CPUs.

Typically, a well written JIT system's bottleneck will be the compilation latency as opposed to the execution time. Several papers [28, 32] explored a hybrid model to mitigate this in which the execution and compilation are decoupled from the main thread. In this hybrid system one or more worker threads are responsible for compiling blocks in the background, with a fallback emulator used on the main thread when compiled blocks are not ready. This is a technique that will be exploring with my JIT-Interpreter hybrid. Böhm [13] explores optimisations to a concurrent system including how to determine when and what to compile; my project does not initially aim to explore this in depth and is left as further work. Unlike the mentioned papers, this project will emulate the MIPS-1 ISA under an x86 CPU: two fundementally different ISAs.

FX!32 [17] developed by Digital allows Alpha platforms to emulate x86 Windows applications, but takes a different approach to optimising performance. Initial emulation runs purely under an interpreter, which builds an execution profile of the program. Using this, FX!32 uses a background optimizer to generate efficient native code that is used in subsequent runs. Whilst this provided good results, it fundamentally relies on inter-run improvements, something out of the scope of this project. It does not perform any JIT acceleration for new programs.

2.2 Industry

Apple developed Rosetta [51] to aid in the transition from their earlier PowerPC-based Macs to the, at the time, new Intel-based Macs. This software was thus a binary translator from PowerPC to x86, based off of QuickTransit by Transitive Corporation [64]. Rosetta was unable to emulate programs that utilised AltiVec instructions (a SIMD ISA extension). Rosetta was shown to have poor performance when emulating computationally heavy programs [53]. Unlike Rosetta, which is proprietary and commercial, my work will be open source and free to use.

Microsoft developed an x86 to ARM emulator as part of Windows 10 for ARM [29], utilising the WOW64 [55] layer of Windows 10. Recently, this emulation layer was extended to support x64 binaries running under Windows 10 for ARM [31]. The emulator utilises a JIT compiler to convert blocks of x86 to native ARM code, which are cached by a background service [29, 62]. This artefact caching allows subsequent runs to benefit from the compilation incurred in initial runs, helping to amortise the initial compilation cost and improving the performance of subsequent executions. Despite this, the emulation was shown to have very poor performance in comparison to native applications [72, 73]. My emulator will employ a JIT compiler with an in process artefact cache, however I will not be exploring the technique of persisting the JIT artefacts between runs.

Recently, Apple developed Rosetta 2 [8] in house to aid in the transition from the Intel based Macs to their new ARM based Apple Silicon Macs [8, 10]. Rosetta 2 reportedly employs an AOT translation technique, in which portions of the program are converted before it is first executed [35, 52]. This technique involves statically analysing portions of code in the binary and translating them ahead of time which are stored in an artefact cache; any new code encountered at runtime falls back to a JIT. This is a similar technique to that employed by Microsoft, but with the addition of the static AOT lookahead. Whilst this technique causes longer than desirable initial program boot times [38], Rosetta 2 has been shown to have much higher emulation performance than either the original Rosetta or Microsoft's ARM emulation [54]. The Apple M1 SOC reportedly includes hardware support for the x86 memory consistency model, which may attribute to Rosetta 2's performance success [30]. Just like the original Rosetta, Rosetta 2 is unable to emulate programs utilising new SIMD ISA extensions such as AVX, AVX2, and AVX512 [8]. Unlike Rosetta 2, I will not be exploring any AOT techniques as they cannot be employed in isolation and will focus on the JIT compilation that Rosetta 2 uses as a fallback. Furthermore, my work will not be tied to any custom hardware requirements and will operate purely in software.

QEMU 'is a generic and open source machine emulator and virtualizer' [48]. QEMU utilises a decoupled architecture in which the front end converts source instructions into tiny code generator (TCG) ops which are then converted into source instructions by the backend [24]. The TCG ops are an abstraction layer frequently referred to as an intermediate representation (IR); it brings a large benefit in modularity and decoupling but sacrifices some performance by introducing additional overhead. I do not initially plan on exploring this IR technique due to this overhead. QEMU's TCG is powered by a JIT compiler, but they also employ a tiny code interpreter (TCI) to provide host agnostic emulation of TCG ops. QEMU provides a multithreaded option for the TCG, however this is to allow multithreaded emulation of a multithreaded system [26]. I will not be exploring this as my project emulates a single threaded system, however I will explore utilising multiple worker threads to accelerate a JIT-Interpreter hybrid, something that QEMU does not support. QEMU also offers a kernel based virtual machine (KVM) that allows it to bypass the TCG/TCI and only emulate the kernel when the host and source architecture match, allowing for increased performance. I will not be exploring this technique as my project is focused on binary translation and not system virtualization.

3 Requirements Capture

3.1 Project Specification

The project will consist of developing the following emulators:

- **Interpreter**

  The interpreter will emulate the source instructions by inspecting each instruction as required and directly executing the equivalent behaviour. Only a single thread will be used and no translation to the host architecture will be performed. This will largely serve as a performance baseline.

- **JIT Runtime**

  The JIT runtime will translate each source block from the foreign ISA to the host ISA which will then be executed to emulate the foreign program. Only a single thread will be used and no interpretation will be performed.

- **Hybrid Runtime**

  The hybrid runtime will utilise both the interpreter and JIT components in tandem with the aim to maximize the performance. At its discretion, it will utilise additional worker threads to JIT translate foreign source blocks to native host machine code blocks. The main thread will execute the translated blocks and will fall back to the interpreter if they are unavailable.

The following criterion have been defined for the emulators:

- **Source Architecture**: MIPS-1

  MIPS-1 has been selected as the ISA for the source architecture as it is a simple ISA. The MIPS-1 CPU will be simulated to the register and memory level.

  No operating system features will initially be emulated and the project will serve to emulate MIPS-1 programs running on bare metal. Methods to explore abstracting the OS layer in the emulator is left for future work.

  None of the co-processors will be supported as they increase the complexity of the implementation without providing increased insight of value. This means no floating point operations will be supported.

- **Host Architecture**: x86

  x86 has been selected as the host ISA as it is the most frequently used ISA on modern desktop computers. This allows the project to demonstrate its potential use in a commercial product designed for x86 systems. Furthermore, x86 is a complex instruction set computer (CISC) architecture in contrast to MIPS's reduced instruction set computer (RISC) architecture. This difference allows us to explore more demanding emulation as more abstraction will be required due to the different natures of the ISAs.

  x64 is not considered or supported at this time as it provides extra implementation complexity without providing evaluation results of extra interest.

- **Host System**: Windows 10

  Due to project details such as calling conventions and executable memory allocation, the project is not cross-platform out of the box without further work. For this reason only a single host system will be supported for the initial implementation to keep the scope manageable. Windows 10 has been selected due to it being the majority OS on desktops in consumer use [22, 23].

- **Emulator Input**: MIPS-1 binaries, MIPS-1 assembly

  MIPS-1 binaries will be supported as this allows for compatibility with built binaries from various sources and tools. Direct support for assembly makes development and testing much easier as the emulator can then directly consume the source code. Pseudoinstructions such as `mov` will be supported for convenience.

- **Microarchitecture Simulation**: None

  Microarchitecture simulation can be of great use in research for simulating different CPU designs and investigating their performance and power characteristics. Having said this, it does not aid in the emulation of a foreign ISA and will introduce a large and unnecessary overhead and complexity. Given this, no microarchitecture simulation will be considered for this project.

3.2 Functionality

The emulators will be written with full functional correctness with respect to the subset of MIPS-1 defined earlier and with functional parity to the interpreter. This will be verified through comprehensive manual testing.

Test suites will be written for every instruction in the MIPS-1 ISA to ensure that the behaviour of the emulators are consistent with the behaviour described in the ISA. Furthermore, additional tests will be written to verify the behaviour of more complex constructs when specified by the ISA such as the branch delay slot.

3.3 Performance

The performance of the JIT and hybrid emulators will need to be evaluated in comparison to the interpreter to analyse and conclude if they do indeed accelerate emulation. Given that different programs will give different performance characteristics, a range of program sizes and structures will need to be explored. These include:

- **Short Simple Programs**

  In short and simple programs, especially those without many jumps, the fixed overhead costs of the emulator such as initialization time will become more significant. Many test suites will be written for this including basic single instruction suites, such as the `xor` suite, to investigate the performance characteristics.

- **Long Complex Programs**

  Conversely, in longer running tests initialization time and fixed overheads become amortized. These tests are of particular importance as the performance of the emulator becomes more important for longer tests. Test suites will be written with long running times and large source codes to investigate the performance characteristics.

- **Memory Intensive**

  Memory instructions can be difficult to accelerate due to the abstraction required to fully emulate the entire memory space whilst running in the same process as the emulator. If this is the case, the JIT emulator may see little speedup over the interpreter. Test suites such as `memcpy(n)` will be written to investigate the performance characteristics.

- **Arithmetic Intensive**

  Conversely, arithmetic operations can be executed very close to the metal with minimal abstraction, providing the JIT emulator with an opportunity for a large speedup over the interpreter. Test suites such as `primal(n)` will be written to investigate the performance characteristics.

- **Iteration**

  Programs and functions with a high degree of iteration, such as those with hot loops, tend to invoke the same blocks of code over and over again. This will be interesting to investigate as it's expected that the JIT emulator's relative performance to the interpreter will increase. To test this, test suites designed with hot loops, such as `primal(n)` will be designed. The suite will cover the same loops being executed from a few times to very many times, allowing us to see how the performance characteristics of emulators evolve as the loops get more frequently executed, and at which point, if any, the JIT emulator surpasses the interpreter.

- **Recursion**

  Similar to iteration, heavily recursive functions cause the same blocks to be executed many times, however they have a higher load on the memory instructions due to the stack operations required. Recursive test suites such as `fibonacci(n)` and `factorial(n)` will be created to investigate how the performance characteristics of recursive programs differ to that of iterative programs.

4 Implementation

This section will cover the implementation of the project. It will first begin by inspecting the technical requirements of the project and determining the technology stack that will be used throughout. Next, it will discuss the design and high level architecture of key components of the project such as the emulator architectures. Finally, the section will heavily explore select implementation details that are of particular interest.

4.1 Technology Stack

The technology stack used to develop the project was the first major decision required before further work could take place; the component of the highest importance was the primary programming language. The following list details a set of requirements that the language must meet in order to be a good fit for the project:

- **Windows 10 and x86 Support**

  Cross-platform support and portability is highly desirable for the language of choice; given the project scope is restricted to Windows 10 and x86 these are the only strict requirements.

- **Strong Typing**

  Due to the size, complexity and length of this project, I have included strong typing as a requirement. Strong typing greatly increases the readability by self documenting the codebase and the reliability improvements due to compile time type errors are paramount.

- **No Garbage Collection**

  Garbage collection (GC) is an automatic memory management scheme that does not require the user to manually free resources. While this is very flexible and easy to use, the unreliable performance implications of a GC are unacceptable for this project.

- **Unmanaged Function Pointers**

  The ability to create an unmanaged pointer to a function allows us to obtain the address of the native machine code for the function. This allows the JIT to compile direct function calls to runtime functions written in the programming language of choice.

- **Executable Memory Allocation**

  Executable memory cannot be allocated through the typical means that languages and programmers use for general memory allocation such as `malloc`. Special OS APIs are required and thus the language must be able to use these APIs. A native call will give the highest performance however an interop call with marshalling would still be viable.

- **Reinterpret Cast**

  The compiled block of x86 code will need to be reinterpreted as a function pointer and executed as a function in order to execute the output of the JIT compiler.

Table 1 contains a table of many programming languages and which requirements they meet, including both languages that I have previous project experience in and popular [59] but unfamiliar languages.

	Windows 10 and x86 Support	Strong Typing	No Garbage Collection	Unmanaged Function Pointers	Executable Memory Allocation	Reinterpret Cast
x86 Assembly	✓	✗	✓	✓	✓	✓
C	✓	✓	✓	✓	✓	✓
C++	✓	✓	✓	✓	✓	✓
C#	✓	✓	✗	✓	✓	✓
Java	✓	✓	✗	✗	✗	✗
JavaScript	✓	✗	✗	✗	✗	✗
TypeScript	✓	✓	✗	✗	✗	✗
Python	✓	✗	✗	✗	✗	✗
Rust	✓	✓	✓	✓	✓	✓
Haskell	✓	✓	✗	✓	✓	✓
F#	✓	✓	✗	✓	✓	✓
Go	✓	✓	✗	✗	✓	✓
Swift	✗	✓	✗	✓	✓	✓
Objective-C	✗	✓	✓	✓	✓	✓

Table 1: Various potential candidates for the language of implementation and the requirements they meet.

The only languages that met all of the requirements were Rust, C, and C++. Rust provides a very safe yet fast memory model via its ownership model [70] and has also recently become a community favourite [59]. This said, it is a very new language with little industrial presence [59]. C provides little in the way of encapsulation making it difficult to write predictable and mantainable code.

This leaves us with C++, a language that provides very good performance with relatively high safety if used correctly. While the underlying memory model is just as unsafe as C's, many modern constructs such as smart pointers [57] and RAII [49] help avoid typical issues like memory leaks while maintaning high performance. Due to these reasons, and my current experience with the language, I believed it was the best fit for this project.

As the project is scoped for Windows 10, MSVC & Visual Studio 2019 was used as the C++ compiler toolchain.

Functional languages like Haskell and F# were considered due to their natural affinity for developing components such as the parser or intermediate representation (IR) based optimisers; while they would have been a good fit for such components, they would have been less viable for core components such as the compiler due to reasons discussed previously. A multi-process solution was not desirable so they were not considered further.

Python was utilised for the analysis and visualisation of the test results. Python is a very powerful scripting language with strong community support making it an ideal choice for smaller scripts that do not require high performance. The package matplotlib was used for visualisation due to its power and ease of use.

MIPS assembly was used for extensive testing as discussed in detail by section 5.

The usage of each language is detailed in Table 2.

Language	Usage	Files	Lines of Code
C++	Core Project	108	8031
Python	Analysis	14	791
DOS Batch	Automation	1	4
MIPS Assembly	Testing	567	123319

Table 2: Languages used in the project and the number of lines of code and file count for each language. Lines of code are inclusive of comments and blank lines. 3rd party libraries are not included.

4.2 Interpreter

The interpreter operates by inspecting the source instruction at the current program counter (PC) and executing the appropriate C++ code to emulate the behaviour of the source instruction. It then increments the PC and repeats until program execution is complete. This architecture is outlined in Figure 1.

The interpreter is composed of the following components:

- **Interpreter Core**

  The core emulator component of the interpreter. It does not control the execution flow but is responsible for interpreting individual instructions. This was separated into its own modular component, so it could be reused in further emulators such as the hybrid emulator.

- **Register File**

  The register file models the 32 general purpose registers found in MIPS-1 in addition to the special `$hi` and `$lo` registers. The register values are stored contiguously for

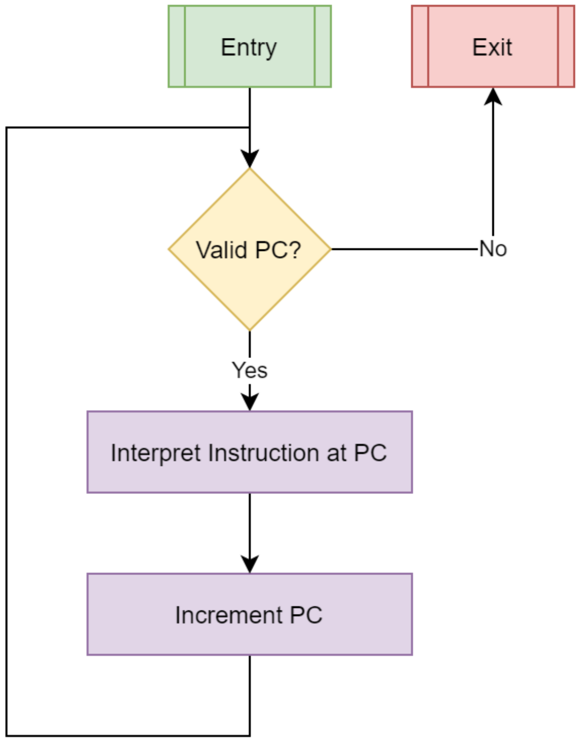

Figure 1: Top level architecture of the interpreter emulator.

better cache performance. All writes to `$0` are ignored, preserving its constant zero value.

- **Memory Map**

  The memory map uses an associative model powered by an unordered hash table. This allows for the entire 32-bit address space to be emulated without requiring it to be pre-allocated, something that might prove difficult in a 32 bit process. The current implementation allows for $\mathcal{O}(1)$ read and write times. Despite this, the performance is substandard compared to a raw array due to the extra overhead and being less cache friendly.

  After profiling, the `std::unordered_map` originally used was found to be a bottleneck and thus was replaced with a 3rd party implementation Tessil/robin-map [12, 63].

4.3 JIT Emulator

The JIT emulator functions by partitioning the program, as it is executed, into source blocks. These blocks are compiled into corresponding x86 host blocks and stored in the translation cache. Each host block terminates by returning an address, corresponding to the exiting MIPS program counter (PC). For blocks without a terminating jump, this will be the subsequent instruction. The system will then execute the block at the desired PC, compiling a new host block if required. This architecture is outlined in Figure 2.

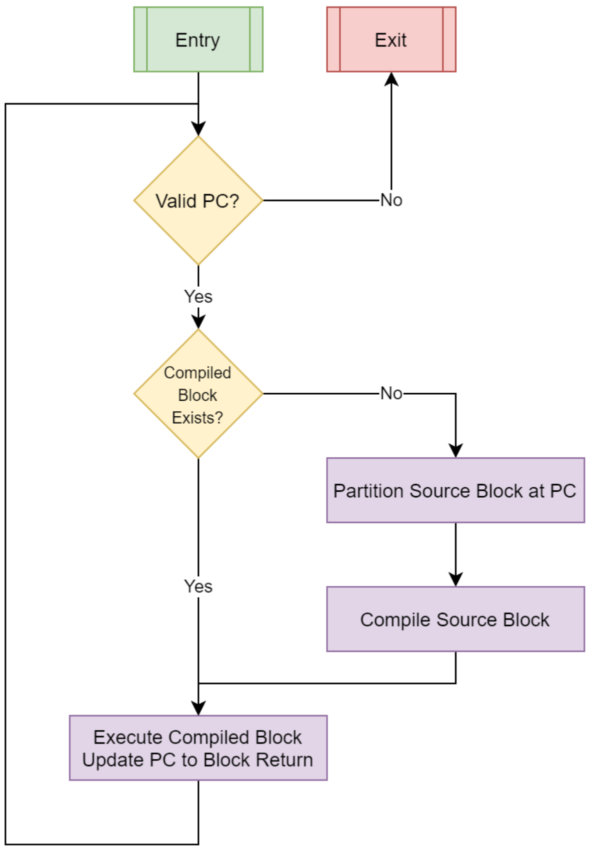

Figure 2: Top level architecture of the JIT emulator.

This architecture was chosen as compiled host blocks cannot have a jump compiled directly to the desired PC, as the destination may not be compiled yet.

The JIT emulator is composed of the following components:

- **Runtime**

  The runtime is the top level JIT emulation system and ties together all the other JIT components. When provided with a MIPS program the runtime will then emulate it by partitioning it into blocks and compiling them with the compiler. These compiled host blocks are stored in a cache and re-used on subsequent executions.

- **Compiler**

  The compiler is used to convert source MIPS-1 blocks into host x86 blocks.

- **Assembler**

  The assembler is responsible for encoding the desired `x86` instructions into an internal buffer, which can then be copied into the executable memory buffer once compilation of a block is complete. It makes heavy usage of templates and `constexpr` evaluation to offload as much work as possible to `C++` compile time, increasing the runtime performance. It is currently designed to support 8-bit, 16-bit and 32-bit `x86` instructions.

- **Block Partitioner**

  The block partitioner is responsible for partitioning the program into a source block for a given address. Source blocks are terminated when a jump instruction is encountered, however the following instruction is also included in order to support the MIPS delay slot. Other partitioning schemes are possible such as a single instruction, however shorter blocks lead to more compilation overhead.

- **Register File**

  The register file described in subsection 4.2 provides the runtime and compiler with a means of emulating the MIPS register file. The current implementation uses an in memory model, where a C++ array represents the MIPS register bank. Translated x86 instructions load and store this array when they need to emulate a MIPS register modification. The special `hi` and `lo` registers in MIPS are mapped internal to `$32` and `$33` respectively, so the same compilation architecture can be used. This memory model provides for fast compile times but subpar execution times, as the contents must be written back after every instruction, causing in some cases large overhead.

  Other solutions can be explored such as direct register mapping or a hybrid model [46]; these should yield increase execution performance, but potentially at the cost of worse compiler performance. These will also increase the complexity of the solution.

  The compiler omits any instructions that would cause a write to `$0`, preserving its constant zero value.

- **Memory Map**

  Instead of attempting to generate native x86 code to model the MIPS memory space, the JIT emulator re-uses the same `MemoryMap` component created for the interpreter runtime described in subsection 4.2 to emulate the memory space. The x86 code is then compiled with calls to the existing C++ implementations.

4.4 Hybrid Emulator

4.4.1 Architecture

The blocks compiled by the JIT compiler have superior execution performance to the interpreter, however the overhead associated with the compilation is relatively high; this means in cases where a block is not executed very many times, the increased execution performance does not have time to pay off. This makes the interpreter a better choice for less ‘hot’ programs. Furthermore, neither the interpreter nor the JIT emulator utilise multiple threads to accelerate emulation; given that most modern systems have multi-core CPUs this leaves something to be desired.

The hybrid emulator aims to achieve the best of both worlds by combining the JIT compiler and the interpreter core into a single runtime. The emulator aims to exploit the multiple cores present in modern CPUs to yield better average performance than either system alone.

The general execution flow and architecture of the hybrid emulator is illustrated in Figure 3. The hybrid begins operation as a traditional interpreter, using the same interpreter core that the standalone interpreter utilises. Whenever the PC 'jumps' (typically as the result of executing a branch instruction) the hybrid will then inspect the translation table for compiled x86 blocks, like the JIT emulator would. If a block is found, then it will execute it natively and operate like the JIT emulator; if the block isn't found, then it will schedule it for compilation on a worker thread before continuing like a normal interpreter.

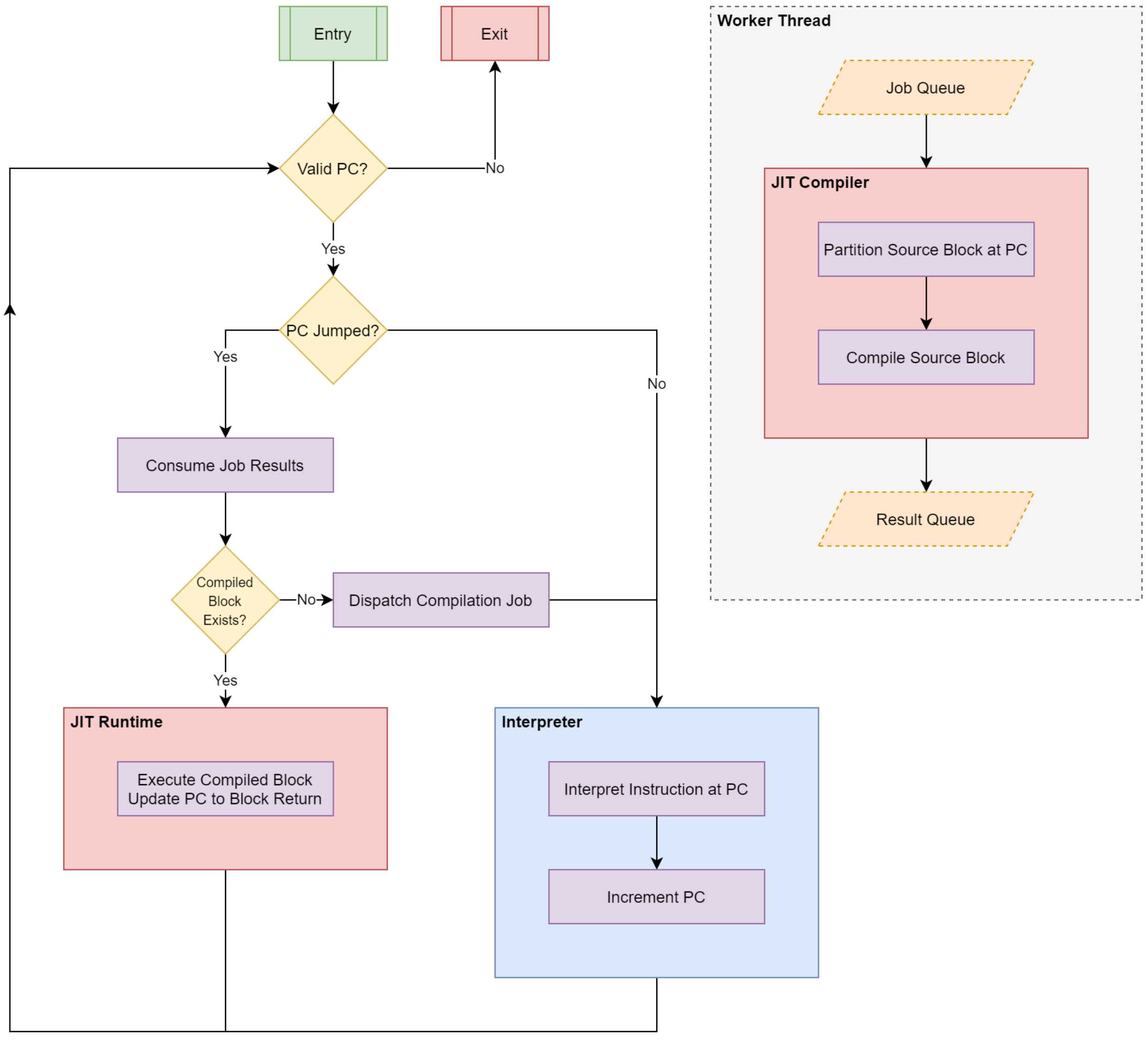

Figure 3: Top level architecture of the hybrid emulator.

This combination allows it to defer the compilation of blocks to asynchronous worker threads, leaving the main thread free to emulate the program either via the JIT compiled blocks or the interpreter fallback. This minimizes the latency associated with JIT compilation as it is deferred into a non-blocking task. At the same time, the high execution performance of the JIT is *eventually* present. This system can be expanded to multiple worker threads, allowing full utilization of the systems CPU for maximum compilation throughput.

The emulator contains a single state (memory map, register file etc.) that is shared between the interpreter and JIT components ensuring consistent behaviour no matter which means of emulation is used for any given block.

In some programs the compiler threads may be overloaded with compilation jobs being queued faster than they can be processed. Whilst not a disaster, this is non-ideal as very frequently executed blocks may be left in the queue for a long time due to the queue being filled with unimportant and infrequently used blocks. This would result in worse performance as the interpreter will be used more than necessary. To mitigate this, the compilation threshold denoted by `-T` is introduced. This threshold specifies how many times a block must be requested before it is actually compiled, ensuring that only sufficiently hot blocks are scheduled for compilation.

4.4.2 Job System

For asynchronous compilation to be possible, the hybrid emulator requires multiple threads in some capacity. Spawning a new thread for every compilation would be a particularly bad idea for several reasons, including but not limited to:

- The overhead for creating and destroying threads is particularly high
- Creating more threads than the system has logical processors will lead to unnecessary contention and performance degradation

For these reasons, a thread pool was developed, as outlined in Figure 4. This provides a highly flexible and efficient API for deferring work, packaged as jobs, to worker threads. The caller (in this case the hybrid emulator) gives the thread pool jobs which are then stored in a concurrent job queue; the worker threads will then fetch jobs from these queue, process them, then emit the results of the job to a second concurrent results queue. The main thread can then consume results from the queue at a time that is appropriate for it.

This job system approach has a variety of benefits. It helps decouple the asynchronous job from the main thread that requires the work; minimising the overlap of state between threads is key in concurrent programming to help avoid the array of concurrency bugs that are typical of the field. By using the results queue, instead of having the worker threads themselves act on the results, we can delegate the stateful changes to the main thread, localising the state to a single thread.

Listing 1 shows the typical caller code for scheduling a job on a thread pool. The job queue is automatically managed by the thread pool whereas the results queue is managed by the caller, as shown in Listing 2.

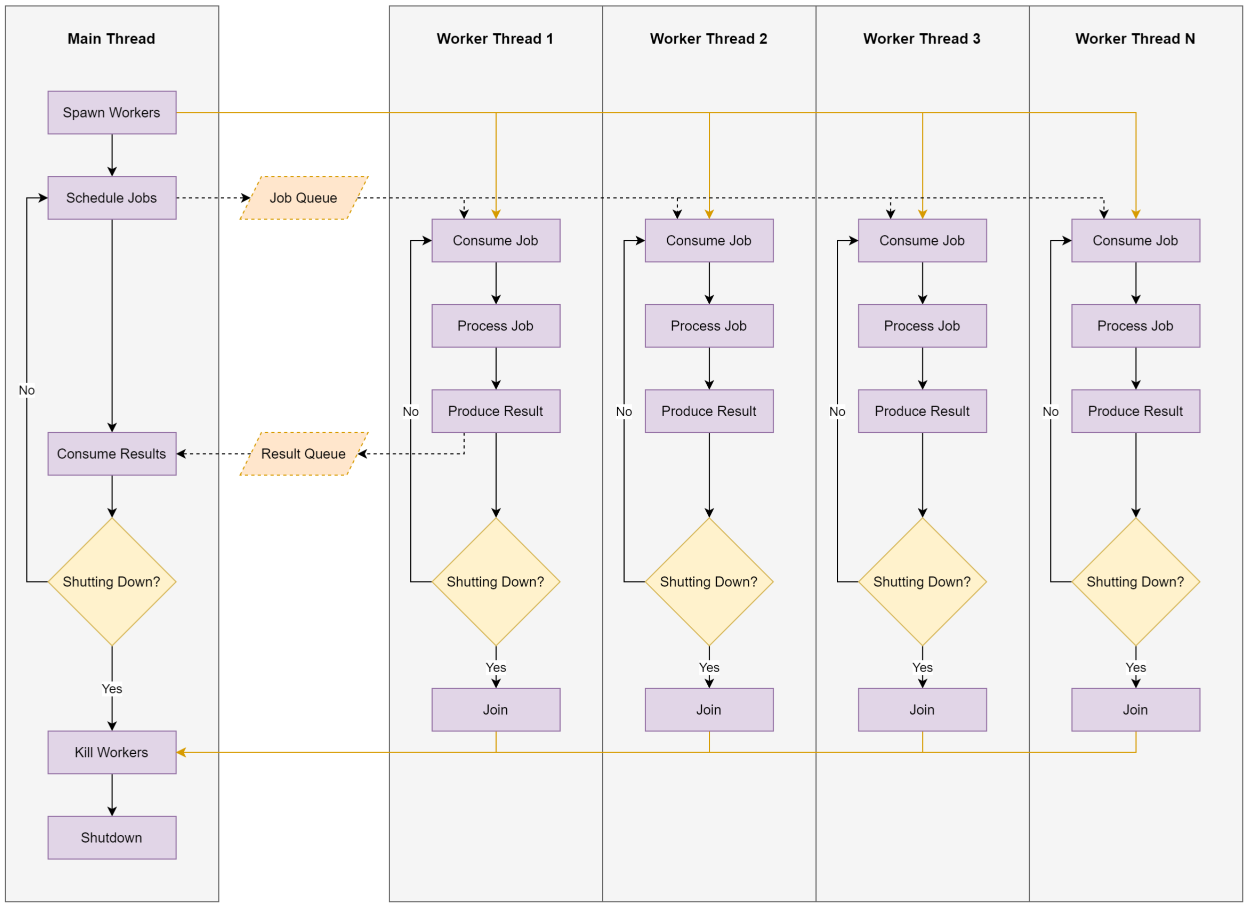

Figure 4: The job system developed for offloading work from the main thread to many worker threads.

```cpp
auto& thread_pool = common::Environment::get().thread_pool();
thread_pool.schedule_job(threading::Job([&]
{
    // do work
    _result_queue.enqueue(/* results */);
}));
```

Listing 1: Typical caller code for scheduling a job on the thread pool.

```cpp
Result result;
while (_result_queue.try_dequeue(result))
{
    // process result
}
```

Listing 2: Typical caller code for consuming results produced by thread pool.

The thread pool will lazily initialize more worker threads if the number of pending jobs exceeds the number of free workers, up to a maximum of `N-1` workers, where `N` is the number of logical processors on the system. The lazy initialization helps avoid the overhead of thread creation in workloads that do not require extra workers, and the maximum worker count avoids excessive thread contention.

The thread pool should be created once and used globally for the entire processes' lifetime. This avoids contention between multiple thread pools and minimises the cost associated with creating and killing threads. This was achieved by having a global `Environment` which was encapsulated through a lazy-initialised singleton pattern; this `Environment` then contains the thread pool and any other global instances required. The singleton pattern provides a mechanism to initialize a single, globally accessible instance of the environment, whilst avoiding the static initialization order fiasco [60].

The 3rd party implementation `moodycamel::ConcurrentQueue` [16] was used for the concurrent queues due to its high performance [7]. Despite the queue's high performance, it is clear that it was designed with high throughput performance in mind. One might expect that attempting to dequeue from an empty queue would be trivial, however, both profiling and inspection of the source code revealed that this was not the case.

This is problematic as the emulator attempts to consume results every time it requests a block for execution; if attempting to dequeue an empty queue is slow, then we waste a large amount of time checking for results degrading the overall performance. To remedy this, a results queue abstraction was developed around the concurrent queue. Internally, the results queue uses a shared atomic counter [61] that is modified every time an item is enqueued or dequeued from the inner concurrent queue. The emulator is then able to check if any results are 'expected' before attempting the dequeue.

The expected count of items in the queue does does not strictly follow the true contents of the inner concurrent queue due to the specific thread timings involved, however, this is not a problem. Any code that relied on the precise size for functional correctness would ultimately be subject to race conditions and would fail. If the expected count of the results queue was zero while the 'true' count was non-zero, then the emulator would simply miss its chance to consume results; this could happen anyway due to the specific thread timings involved and thus is not problematic. The real utility of the expected count is that it usually allows the hybrid emulator to skip the slow empty dequeue process, improving the overall performance.

4.4.3 Worker Pool

Generally, jobs should avoid causing any stateful changes to the system; this is typically avoided with the results queue by moving the state changes to the main thread. In some cases, however, state modification is not completely avoidable.

In the hybrid emulator's case, the scheduled jobs utilise the `Compiler` to generate the required x86 blocks. Not only does the compilation process utilise internal buffers for high performance, but it also needs to allocate executable memory (which isn't thread safe with a single allocator) for the final compiled block. Both of these reasons mean it is impossible to use a single compiler across all jobs.

Since a compiler instance cannot be shared between concurrent jobs, one might reason that the job itself could be responsible for creating its own compiler. In some scenarios, where the stateful object is fast to construct, this could be a viable option, even if not the most performant. For the compiler however, this would this have very poor performance, largely in part due to the non-trivial overhead associated with constructing the executable allocator. Performance aside, the approach in question is still unviable due to lifetimes. The lifetime of the compiled x86 block is tied to the lifetime of the compiler, and thus the compilers would need to be kept alive until the program execution is complete.

Moreover, if the compiler or the results queue are destructed during the execution of a job then we can expect catastrophic behaviour. This is somewhat problematic as asynchronous jobs can take an unknown amount of time to complete, and the program emulation may complete while there are still currently executing or pending jobs.

The worker pool is a layer of abstraction that helps avoid all of these issues by lazily creating worker objects as required, which are then provided to the executing jobs and recycled for future jobs. Listing 3 details an example of how the hybrid emulator uses the worker pool to schedule a worker job; this is a special type of asynchronous job that is provided with the worker as an input. Unlike the thread pool, which is global, the worker pool should be created per *user*. The worker pool will internally use a provided thread pool for job scheduling.

```cpp
const auto job = [this, addr](Compiler& compiler)
{
    // compile block with provided worker

    _result_queue.enqueue(Result{
        .addr = addr,
        .block = std::move(block),
    });
};

_worker_pool.schedule_job(threading::WorkerJob<Compiler>(job));
```

Listing 3: Hybrid runtime caller code for scheduling a compilation job on its worker pool.

Concurrent jobs are guaranteed to have distinct workers, yet an excess of workers will never be produced, maximising the performance by minimising the amount of wasteful worker constructions required. These workers will be kept alive until the worker pool is shutdown, ensuring that there are no unpredictable lifetime issues.

Furthermore, the shutdown process will ensure that all pending jobs are discarded and that and that any current jobs are complete before the shutdown completes. This allows us to guarantee that no jobs are executing as the hybrid emulator shuts down, avoiding the race conditions. The inter-thread synchronisation required to implement the safe shutdown utilises condition variables to maximise performance.

4.5 Code Generation

Code generation is the core aspect of JIT compilation responsible for converting the source MIPS-1 code into the executable x86 code. This section will explore and discuss details of this process that are of particular interest.

4.5.1 Architecture

The architecture for code generation was split into multiple components to create a compilation pipeline as outlined in Figure 5.

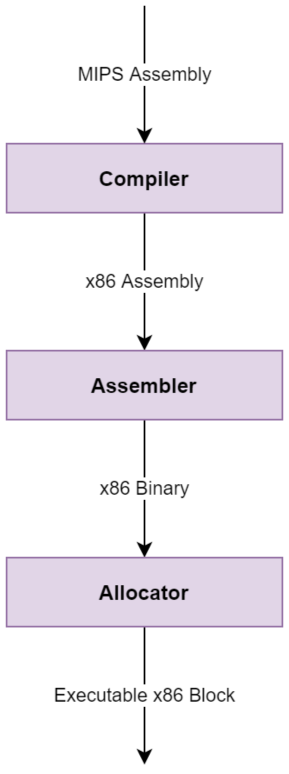

Figure 5: Compilation pipeline and architecture for JIT compiling MIPS to x86.

The `Compiler` first translates the MIPS assembly to the appropriate x86 assembly. This is then converted to x86 binary by the `Assembler`. Finally, the `Allocator` allocates the appropriate executable memory for the compiled binary, producing an executable block of x86 code.

For increased performance, the `Compiler` does not actually produce an intermediate stream of x86 assembly; instead it utilises the `Assembler` to directly produce the correct x86 binary through template functions, largely minimising the runtime overhead.

4.5.2 Executable Memory Allocation

Once the code has been generated, it needs memory allocated to hold the final executable `x86`. Typical memory allocation techniques such as `new` or `malloc` are unsuitable for these purposes as the memory allocated by them is non-executable; instead, they only have `read` and `write` permissions. This is done for security purposes, as it prevents many instances of arbitrary code execution exploits by restricting execution on general purpose memory.

In order to work around this, the executable memory allocator developed uses `Win32` APIs to change the permissions of the allocated memory; in specific, it uses `VirtualProtect` [68] to change the permissions to `PAGE_EXECUTE_READWRITE` [36]. This allows the contents of the memory to be executed directly without the operating system throwing an exception. Upon destruction, the allocator restores the permission of the touched memory to `PAGE_READWRITE`.

Typically, we allocate as much memory as needed when required; if the generated code required `x` bytes of memory, then we would simply allocate `x` bytes on demand. In the case of executable memory, however, it is not so simple. The memory protection status discussed earlier does not have byte level granularity, but instead is at the page level. This means we must make page aligned allocations in order to not disrupt the permissions of surrounding allocations. The `Win32` API `VirtualQuery` [69] is used to determine the page boundary, allowing the allocator to guarantee it only changes permissions on pages that it has full ownership over. The C++11 specifier `alignas` [9] is used to force page alignment on automatically managed allocations (such as stack allocations).

In order to guarantee full ownership over the page, the allocation must span over the entire page (inclusive of alignments). Allocating an entire page for every compiled block would be extremely wasteful as pages in Windows are `4096` bytes large whereas the generated code for a given block rarely exceeds `100` bytes. To work around this, the allocator internally allocates a buffer of memory spanning a whole number of pages and sets the permissions appropriately; when the compiler then requests an allocation of size `x`, the allocator simply hands out an `x` byte portion of the pre-allocated memory, and no further allocations are performed.

4.5.3 Calling Conventions

The calling convention of a function specifies both how arguments are passed to it from the caller and how it returns its values to the caller. When the stack is used for argument passing or return values, it also dictates which function out of the caller and callee is responsible for the stack setup/cleanup.

Typically, this detail is not of a concern to the programmer and is left as a compiler implementation detail, but this is not true in our case. The caller and callee must match their calling convention or else the function invocation will be catastrophic. Normally, this is guaranteed by the compiler, but since we are hand generating x86 code that needs to communicate with compiler generated C++ code, we are not afforded this luxury. Due to this, we need to pay close attention to the calling conventions used whenever we encounter one of these JIT - C++ interop barriers.

Fundamentally, there are two distinct cases that are of importance:

1. The runtime executing blocks of JIT code

   In this case we are generating the callee code, and since our compiled blocks do not take any arguments, we only need to take care that return values are handled appropriately. When reinterpreting the generated block of memory as a function pointer, we can specify the calling convention, causing the compiler to generate the appropriate caller code that will allow it to call our function correctly.

2. The JIT code making a call into the runtime

   This is required as some aspects of the emulation may be provided by the runtime written in C++, as opposed to being replicated natively in the generated x86. In this case we are generating the caller code, and thus need to take care that the arguments passed to the callee are done correctly, as well as any `prolog-epilog` code. We can specify the calling convention on the C++ function, causing the compiler to generate function with the appropriate callee code, which we can then design our JIT caller code against appropriately.

Calling Convention	Stack Cleanup	Parameter Passing
`__cdecl` [1]	Caller	Pushed on stack in reverse order
`__clrcall` [2]	N/A	Pushed on CLR expression stack
`__stdcall` [4]	Callee	Pushed on stack in reverse order
`__fastcall` [3]	Callee	`ECX`, `EDX`, then pushed on stack in reverse order
`__thiscall` [5]	Callee	Pushed on stack in reverse order, `ECX = this ptr`
`__vectorcall` [6]	Callee	Registers, then pushed on stack in reverse order

Table 3: Breakdown of calling conventions supported by MSVC [11]

There are some unique points of interest concerning the different calling conventions listed in Table 3:

- `__cdecl` is required for `vararg` as it uses caller stack cleanup
- `__clrcall` is only usable for managed function calls and thus is not a consideration
- `__stdcall` is required for Win32 API calls
- `__thiscall` is only usable for member function calls
- `__vectorcall` has restricted argument types and is typically only useful for floating point and SIMD. For integer arguments, it is equivalent to `__fastcall`

Outside previously aforementioned unique cases and properties, the calling convention used was of little importance, as long as both the caller and callee are consistent with one another. All calling conventions available under MSVC utilised `EAX` as the return register. A set of macros were introduced in the config that would define a symbol, `CALLING_CONV`, to match the selected calling convention. This symbol was then used throughout the project, whenever a JIT - C++ interop barrier occurred, allowing the calling convention to be globally changed with ease if required.

Furthermore, these calling conventions are specifically for x86, and are ignored when compiling for x64. x64 uses a `__fastcall` like calling convention [74] and thus generated caller code will need to be modified if built for x64 platforms.

Another related challenge is trying to call member functions from our JIT code. This is required as the runtime component of the emulator, which we may need to call into from our JIT code, is a class instance. Unfortunately, member function pointers cannot be reinterpreted in C++; this is due to the fact that their structure and representation is implementation defined, and may not be that of a typical pointer [44]. Furthermore, the calling convention would be less trivial to develop against, as we would be forced into `__thiscall`. Fortunately we can work around this using templates.

```cpp
template <typename...Args>
struct instance_proxy
{
    template <typename T, typename R, R(T::* F)(Args...)>
    static R CALLING_CONV call(T* obj, Args...args)
    {
        return (obj->*F)(args...);
    }
};
```

Listing 4: `instance_proxy` template

The usage of the struct in Listing 4 is to avoid a limitation of C++ templates. Variadic parameter packs, such as `template...Args`, must be the last argument in the parameter list, yet `R(T::* F)(Args...)` requires that parameter pack to already exist. For this reason, the struct is required so that `Args...` can be defined as the final argument of the template, which can then later be used in the inner function template by `R(T::* F)(Args...)`, circumventing the limitation.

This template allows us, at compile time, to make a non member function that proxies us to the invocation of the actual member function. By utilising this, our JIT generated code only needs to invoke a free function as normal, providing a pointer to the instance as the first argument. It is decorated with `CALLING_CONV` such that it has a known strict calling convention for our JIT to generate against.

For example, if we had the following member function `int Runtime::func(float x);`

we can generate the corresponding proxy function as follows

`instance_proxy<float>::call<Runtime, int, &Runtime::func>`

which we can then link against and compile a call to from our JIT code, passing along a `Runtime*` and `int` as dictated by `CALLING_CONV`.

Initially, I decided to use the `__fastcall` calling convention; by utilising registers over the stack when possible the generated caller code can be simpler and faster. To make a more informed decision, I analysed how fast `__fastcall` would be compared to `__stdcall`. `__stdcall` was considered over `__cdecl` as the callee cleans the stack, reducing the amount of code generation required in the caller. I analysed the cost of calling a 3 argument function, such as `memory_map::write<T>(this, addr, value)`, which was used to implement several MIPS memory instructions.

`__fastcall`	`__stdcall`

`MOV ECX this`
`MOV EDX addr`
`PUSH value`
`CALL f`
	`PUSH value`
	`PUSH addr`
	`PUSH this`
	`CALL f`

Table 4: Comparison of generated caller code for different calling conventions

From Table 4, it appears that `__fastcall` would be faster. Both cause the JIT to emit 4 instructions, however the `__fastcall` ones would be faster to execute. This is because only the last `PUSH` requires a corresponding `POP` on the callee code, resulting in 5 instructions executed as opposed to the 7 required by `__stdcall`. Furthermore, `__stdcall` incurs more memory operations which tend to be slower than ALU operations.

It might appear that this analysis is incorrect, as it does not account for the fact that `__fastcall` writes to `ECX` and `EDX` which may be managed by the runtime for other use cases, such as the register file pointer. In this case extra instructions would be needed to restore `ECX` and `EDX`. The analysis however, is correct, as `EAX`, `ECX`, and `EDX` are volatile [66] and may be overwritten by the callee: given this, they will need to be restored regardless of the calling convention used.

This analysis assumes that the `args` are ready and can be moved or pushed directly, however in some cases, such as offset loading/storing, the `arg` first needs to be computed. If the destination is a register, as in the case of the first 2 `__fastcall` arguments, then the move can potentially be avoided by using the destination register as the accumulator for the calculation. In the case of a stack destination, such as in `__stdcall`, this optimisation cannot be made.

4.5.4 Branching

As mentioned previously, a block terminates by returning the target MIPS address of the next block to execute; the exact code generated for this is different based on the type of branch involved. Due to the branch delay slot in MIPS, the instruction following the branch instruction is usually ‘peeked’ and pre-emptively compiled.

Under the `__fastcall` calling convention, as described in subsubsection 4.5.3, the return register is `EAX`.

The simplest branches are the unconditional immediate jump instructions `J` and `JAL`. They have a single fixed compile time destination. The x86 generated for these instructions is detailed in Listing 5.

```
compile delay slot

MOV EAX, target
RET
```

Listing 5: Psuedo-x86 generated for the MIPS instruction `J`.

Following this are the jump register instructions `JR` and `JALR`. These use the contents of a specified MIPS register as the destination of the jump. This can be problematic when the delay slot is introduced, as the instruction in the delay slot could change the same register used for this jump.

To work around this, the desired register value can be pushed onto the x86 stack before executing the delay slot, before popping it from the stack and performing the jump. This stack operation guarantees functional correctness but is wasteful if the delay slot does not change the contents of the register. The code generated with stack operations is detailed in Listing 6.

A utility was introduced to determine if a given MIPS instruction writes to a given register; using this, the stack operations can only be emitted when the delay slot writes to the jump register, leading to better performance in most cases whilst guaranteeing correct functionality. This optimised code is shown in Listing 7.

```
MOV  EAX, [jump_register]
PUSH EAX

compile delay slot

POP  EAX
RET
```

Listing 6: Psuedo-x86 generated for the MIPS instruction `JR` when the delay slot **does** overwrite `jump_register`.

```
compile delay slot

MOV ECX, [jump_register]
RET
```

Listing 7: Psuedo-x86 generated for the MIPS instruction `JR` when the delay slot **does not** overwrite `jump_register`.

The most complex family of branch instructions to emulate are the conditional branches such as `BEQ` and `BNE`. Since they are conditionally executed branches, the condition may evaluate to false and the branch should be skipped.

Conditional execution in x86 can be performed by first using `CMP` [19] to compute the condition status stored in `EFLAGS`. A following conditional instruction using some condition code `cc` will then be predicated by the contents of `EFLAGS`.

Unfortunately, the `RET` instruction does not have any conditional variants [50]. This means that the termination of the block itself cannot be trivially predicated.

We turn our attention to `CMOVcc`, which allows us to conditionally move a value from one register to another [18]. We can then compute two target destinations, `target_true` for the actual destination of the branch instruction, and `target_false` for the expected destination address after no branch is executed. `CMOVcc` can then be used to return the correct target based on the outcome of the comparison. The generated code for this is detailed in Listing 8.

In some cases, the delay slot instruction may cause `EFLAGS` to change, typically those that call other functions or use `CMP`. In these cases, the `CMOVcc` will then act on an incorrect condition status producing unpredictable behaviour. To work around this, the instructions `PUSHF` and `POPF` can be used to push and pop `EFLAGS` to the stack [45, 47]. The variant that preserves `EFLAGS` is shown in Listing 9.

```
CMP [mips_reg_1], [mips_reg_2]

compile delay slot

MOV    EDX, target_true
MOV    EAX, target_false
CMOVcc EAX, EDX
RET
```

Listing 8: Psuedo-x86 generated for the conditional branch MIPS instructions. `cc` denotes the selected x86 condition code.

```
CMP [mips_reg_1], [mips_reg_2]

PUSHF
compile delay slot
POPF

MOV    EDX, target_true
MOV    EAX, target_false
CMOVcc EAX, EDX
RET
```

Listing 9: Psuedo-x86 generated for the conditional branch MIPS instructions when `EFLAGS` must be preserved. `cc` denotes the selected x86 condition code.

4.5.5 Function Calls

The x86 instruction `CALL` [15] is used for generating function calls from the JIT compiled x86 code to the C++ written runtime. This is primarily used to call into the memory map functions to emulate the MIPS instructions `LW`, `LB`, `LBU`, `LH`, `LHU`, `LWL`, `LWR`, `SW`, `SB`, `SH`.

Function arguments are handled as described in subsubsection 4.5.3. Since the memory map is non-static, we cannot directly call its member functions from the x86 code and must use the proxy call mechanism described in Listing 4.

The `CALL` instruction comes in many different forms, some of which use a register or pointer for the target destination. The encoding used has an immediate based destination; this is ideal as the destination is fixed and it avoids the extra instructions that would be required for loading the destination into the register/memory. This encoding does however have a caveat as the destination is relative and not absolute.

This may not seem like a problem, but it means that the code generated when CALL (and most jump functions) is included is not position independant code (PIC). This means that the code generated will be different based on where that code will be loaded; given that the executable memory is allocated *after* all the code is generated, this is slightly problematic.

In order to get around this the `Linker` component was developed. When assembling the `CALL` instruction, the linker can be provided with a symbolic destination address; the linker will then proceed to track the location in the block that should resolve to the provided symbol. After allocating the executable memory and committing the generated binary to it, the linker can then resolve all symbolic references. This allows it to rewrite the placeholders with correct relative offsets to the symbol.

4.6 Optimisations

Some features added to the emulators allow for improved performance under certain circumstances. This section will explore the configurable optimisations developed for the emulators. These were left configurable as they do not yield universal speed improvements and are to some degree more situational.

4.6.1 Direct Linking (`-L`)

As described in subsection 4.3, each compiled host block terminates by returning the target MIPS address; this implementation is required for supporting branching as the desired MIPS address may not yet have a corresponding compiled x86 block. By first returning to the runtime the desired block can first be compiled if necessary.

While this provides flexibility, it does lead to non-ideal performance once the necessary blocks have been compiled. Figure 6 illustrates the execution flow of the JIT emulator when all the relevant blocks have been compiled. In some cases, such as loops in the source code, this leads to an undesirably large overhead in the emulation. This is because for every iteration of the loop, the control flow must return from the compiled x86 block to the runtime which checks if the destination block has been compiled then returns control flow back to the compiled x86 block; while these dispatch overheads seem trivial, profiling has shown that they can become significant bottlenecks for hot loops. In some cases extreme cases, up to 40% of all execution time was spent finding the compiled blocks in the translation cache.

The direct linking feature, denoted by `-L`, was developed to remedy this. It is available for both the JIT and hybrid emulators. Direct linking allows the emulator to retroactively patch the end of the block to have a native jump to its target x86 block instead of returning to the runtime. This allows the emulated program to eventually become a native x86 program that does not require the runtime for dispatching, but is still able to use the runtime to compile and dispatch to unseen blocks.

In order to support direct linking, all unresolved jumps must be stored for each x86 block compiled; the x86 address of the terminal is stored so that it can be patched with a direct jump, and the desired MIPS target address is stored so that the corresponding x86 block can be found. When a new x86 block is compiled, all existing x86 blocks will be tested for relinking; if any unresolved jumps are found with a target MIPS address that now has an

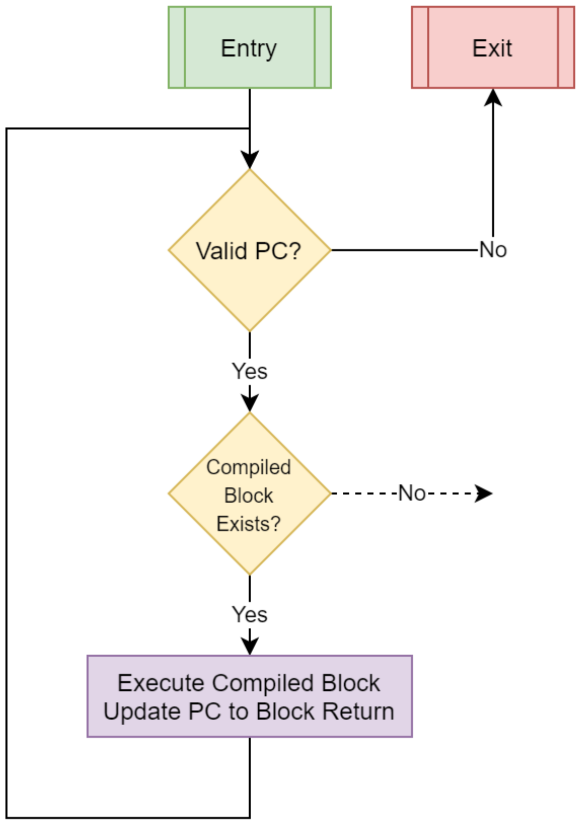

Figure 6: The execution flow of the JIT emulator when all source blocks have been translated.

entry in the translation table, the terminal will be overwritten with a direct jump to the corresponding x86 block.

This should allow for much higher execution performance at the cost of additional overheads as extra work needs to be performed to track the unresolved jumps and perform the relinking. On the surface, it may appear that relinking every block when a new block is compiled is wasteful; we have to check more blocks for relinking and may perform relinking on a block that is never executed again. It might then be reasonable to assume that instead the runtime should only relink the block that is about to be executed: Figure 7 illustrates a scenario which demonstrates why this is not the case.

In the following scenario, when only relinking the current block, `block_1` is directly linked to `block_2` when `block_1` is executed. Executing `block_1` now causes a direct jump to `block_2`, which then terminates and uses the runtime to dispatch to `block_3`. In this case, `block_2` can no longer be executed directly from the runtime and is only ever executed as a direct jump from `block_1`; this means that the runtime never performs relinking on `block_2`, stopping `block_2` from directly jumping to `block_3`. This is not ideal as it greatly reduces the peak performance of the emulated program.

Unconditional jumps, such as `J` and `JAL` are the simplest to relink. They contain a single target MIPS address and a single x86 source address. If the MIPS target is present in the

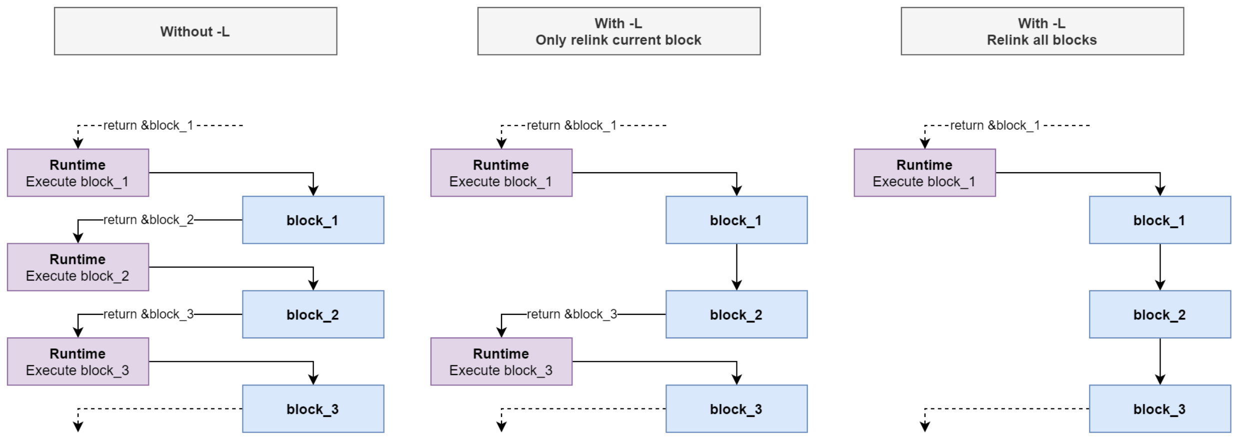

Figure 7: The effects of different relinking schemes on an example program fragment consisting of 3 blocks.

translation table, then the source x86 address (pointing to the current terminal) will be overwritten with a jump to the x86 block corresponding to the target MIPS address.

Conditional jumps, on the other hand, have two target MIPS addresses. This is because conditional jumps are implemented by having a *true* jump to the desired address in the MIPS source code and a *false* jump to the following instruction. Due to this a simple implementation would require that both targets are present in the translation table before relinking can proceed.

```
 1  init:
 2      addi $1, $0, 0
 3      addi $2, $0, 0
 4
 5  loop_start:
 6      add  $2, $2, $1
 7      addi $1, $1, -1
 8      bne  $1, $0, loop_start
 9      nop
10
11  loop_end:
12      add  $1, $2, $0
```

Listing 10: An example MIPS program containing a simple loop.

Listing 10 details an example MIPS program with a typical loop, with the `BNE` on line `8` being the branch of interest. After the first iteration of the loop, the *true* case jumps to `loop_start` which has a corresponding x86 block compiled, yet the *false* case jumps to `loop_end`, which does not yet have a corresponding x86 block; in fact, `loop_end` would not be translated until the loop has finished execution. This means the branch cannot be relinked until the loop has terminated, somewhat defeating the purpose of relinking.

By performing partial relinking this can be remedied. For conditional branches, if either the *true* or *false* cases are present in the translation table, then that half of the branch can be relinked whilst the other case is left as a dispatch to the runtime. A partially relinked branch can then be fully relinked and resolved later once both destinations are present in the translation table.

The instructions `JR` and `JALR` cannot be relinked; this is because their target MIPS address is determined by the contents of a register, and thus the desired target of the jump cannot be known at (JIT) compile time. The MIPS address must be resolved to a corresponding x86 block dynamically by dispatching to the runtime. In practice, this means function returns cannot be relinked.

4.6.2 Speculative Compilation (`-S`)

The hybrid architecture described in subsubsection 4.4.1 allows the system to use extra worker threads to asynchronously JIT compile source blocks, deferring the work away from the main thread; whilst this is helpful, the just in time compilation could in some cases be seen as just too late. Blocks are scheduled for JIT compilation when they are needed, meaning they are not ready at that time and the interpreter fallback must be used.

Speculative compilation, denoted by `-S`, can help remedy this by building off of the work provided by direct linking. Whereas `-L` aims to relink all unresolved jumps present in the compiled blocks, speculative compilation instead attempts to schedule the destination blocks for JIT compilation whilst they remain unresolved; this allows the system to speculatively compile blocks that *might* be executed from the blocks currently in the translation table. The current speculation algorithm attempts to schedule the destinations of all unresolved jumps for compilation; a more complex speculation criteria could yield better results.

This would allow the system to translate the blocks ahead of time, potentially before they are needed, reducing the amount of execution that must be delegated to the interpreter fallback. On the other hand, this could fill the compilation queue with low priority or completely wasteful blocks; increased contention could increase the time before important blocks are compiled, degrading the overall performance.

Speculative compilation is compatible with direct linking, and is only available for the hybrid emulator. It would serve no benefit on the standard JIT emulator as without asynchronous compilation there is nothing to gain by compiling blocks early.

4.7 Assembly Parser

The parser for the MIPS assembly could easily become unmanageable and messy if not handled with proper care. While it is a very simple language syntactically and there is no real difficulty with parsing any given instruction, it is eliminating redundancy that poses the biggest challenge. Regex expressions can quickly become verbose and unwieldy and many of the instruction parsers would require the same implementation for their inner parsers, such as to parse a register.

To preserve the elegance and readability of the code-base, a semi-automatic approach to writing parsers was developed that preserved full type safety. Listing 11 shows an example of this system in action for parsing canonical R type MIPS instructions such as '`ADD $1, $2, $3`'.

```cpp
static thread_local const auto parser =
    generate_parser<Register, Register, Register>
        (R"(\w+ ??, ??, ??)");

const auto [dst, src1, src2] = parser.evaluate(instr);

return InstructionR
{
    .op = opcode,
    .rd = dst,
    .rs = src1,
    .rt = src2,
    .sa = 0
};
```

Listing 11: Parser function responsible for parsing basic R type instructions such as `ADD` and `SUB`. Parsing the opcode is handled before this function, as the opcode determines the instruction format.

To begin, the parser is generated: this is only performed once per thread as denoted by `static thread_local` to improve performance. The function `generate_parser` is first supplied with the types of the components to parse from the expression as a variadic parameter pack. By utilising templates the parser object created is fully typed. Next, the regex expression for the parser is supplied, however this is not a typical regex expression.

First, the expression contains these `??` sequences which are not part of the standard regex specification. These have been used to denote the 'component' that is being parsed. When generating the parser, the `??` will be replaced by a regex pattern in a capturing group based on the types provided. Table 5 shows the type substitutions provided to the parser system by the MIPS parser. In this example, all three `??`s correspond to `Register`s as determined by the type signature and thus are substituted with `\$[A-Za-z0-9]+`. It is important to note that at this stage, the substituted regex is not responsible for ensuring that the format is correct and the parse may still fail; the job of the regex is to ensure that the correct substring is provided to the inner parser at each stage.

Type	Substitution
`std::string`	`\S+`
`Register`	`\$[A-Za-z0-9]+`
`uint32_t`	`[-+]?[A-Za-z0-9]+`
`uint16_t`	`[-+]?[A-Za-z0-9]+`
`uint8_t`	`[-+]?[A-Za-z0-9]+`
`int32_t`	`[-+]?[A-Za-z0-9]+`
`int16_t`	`[-+]?[A-Za-z0-9]+`
`int8_t`	`[-+]?[A-Za-z0-9]+`

Table 5: Automatic pattern substitutions supported by the MIPS assembly parser.

The next stage in generating the final regex string is further type substitutions. In MIPS assembly, it is valid to have any number of spaces following a comma, and hence we should support that. To do that we would need to write the appropriate regex each time, which becomes overly verbose. Instead, the parser generator has a set of expansion substitutions that it can detect instances of, which it will then replace with the full comprehensive regex pattern. These are detailed in Table 6. This allows the parser to replace instances of a general pattern with the pattern itself, and in our case the '`, `' becomes '`,\s*`'.

Substitution
`,\s*`
`\s+`

Table 6: Automatic expansion substitutions supported by the MIPS assembly parser.

Finally, some extra boilerplate is added to the regex pattern to handle leading and trailling whitespace. In the end, the simple and human-readable pattern `\w+ ??, ??, ??` is automatically converted to the final but completely unreadable regex pattern shown below:

```
^\s*\w+\s+\($[A-Za-z0-9]+),\s*\($[A-Za-z0-9]+),\s*\($[A-Za-z0-9]+)\s*$
```

With the automatic pattern generation covered, we can now return to the example in Listing 11. With the parser constructed, we can now evaluate the input string on the parser; this returns a strongly typed tuple, where each element corresponds to the pattern and type signature used to generate the parser. The parser first uses the regex pattern with its capturing groups to split the input into several substrings, each of which corresponds to the types provided; in this case, each of them is a `Register`. The inner parser converts each substring into the desired type, which is then packaged together into a tuple by the regex parser.

How is the parser able to dispatch to the correct inner parser functions for each type? When generating the parser, it is provided with an `InnerParser` as one of the type parameters; this type must contain a template function of the signature shown in Listing 12.

```cpp
template <typename T>
T parse(const std::string& raw)
```

Listing 12: `parse` function of the `InnerParser` type required by the regex parser.

This function can then be implemented for each supported type (such as `Register`) through template specialization. With this, the regex parser is able to parse each of the individual components automatically without sacrificing static typing at any stage. Furthermore, the parser uses several caches internally to accelerate performance at no extra work of the user.

Finally, the example in Listing 11 is able to use the tuple components to construct the instruction object.

5 Testing

5.1 Methodology

A single test framework to verify the functionality of both the JIT system and the interpreter system, otherwise known as the systems under test (SUT). The test coverage aims to cover three key areas:

- **Functionality**

  Tests are written to verify that the functionality of every instruction is as defined by the MIPS-1 ISA [40]. All interesting edge cases have specific tests written to improve the functional coverage of the test suite. Some tests are also designed to verify edge cases of the SUT. This includes tests containing MIPS code that would be optimised or handled uniquely by the SUT, to verify that the optimisations do not cause functional defects.

- **Performance**

  Tests are written to observe the performance characteristics of the SUT. These allow us to evaluate the performance of an SUT and make analytical conclusions about a given SUT; a necessity, given that this project is about accelerating emulation.

- **Regression**

  If any defects are detected, tests are written specifically to induce the incorrect behaviour. This is an invaluable tool for detecting regressions should the defects reappear.

5.2 Framework

In order to evaluate the functional correctness of the emulators a directive based test framework was developed that allows us to create MIPS assembly files with special test directives included in the initial comment block. The following directives are supported:

- `name:` *`value`*

  Sets the name of the test to *value*. If no `name` directive is provided the file name without extension will be used instead. `name` may only appear once per test.

- `desc:` *`value`*

  Sets the description of the test to *value*. `desc` may only appear once per test.

- `init:` *`x`* `=` *`y`*

  Initializes the register x to the 32-bit constant literal y before executing the MIPS program.

- `assert:` x `==` y

  Asserts that the register x is equal to the 32-bit constant literal y after the MIPS program has terminated.

If the emulator throws an exception during program execution, then the test will *fault*, and the exception will be printed in addition to the resultant register file and any logs produced by the emulator. If any assertions fail, then the test will *fail*, and the failing assertions will be printed in addition to the resultant register file and any logs produced by the emulator.

This testing framework has been highly flexible and invaluable in writing a large amount of functionality tests to provide us with proper regression testing for both the JIT and interpreter. Tests are automatically found and loaded from a test directory, meaning the program does not need to be rebuilt to add new tests to the suite. Given that the test runner is templated by the SUT, the same framework can be easily used for any future emulators designed. By creating the directive system I have allowed myself to bring the setup and assertion code out of the MIPS program itself, simplifying and improving the robustness of the MIPS tests. Further directives will be developed as necessary.

Listing 13 contains an example of a test under this framework.

```
# desc: adds values into a zero initialized register
#
# init: $1 = 0
# init: $2 = 10
# init: $3 = 1
# assert: $1 == 11

add $1, $2, $3
```

Listing 13: A simple MIPS test verifying functionality of the `add` instruction.

5.3 Generators

In some cases, test suites may be difficult and time-consuming to write and maintain by hand, such as when multiple variations are required of the same test. One example of this is the `fibonnaci(n)` test suite, which contains tests `fibonnaci(0)` to `fibonnaci(20)`.

To automate this, test generators were written in Python where applicable. These programmatically build the tests, usually using a predefined test template with marked substitution points. This is much more maintainable as the entire suite can be modified at once.

In other cases a completely programmatic approach is taken to generating the tests. This is the case for the `unroll` test suite.

5.4 Output

The test runner aggregates the results and statistics of all the tests run on an SUT and outputs the aggregate to a csv for further processing and analysis. Some statistics are only present on specific SUTs (such as the JIT system). The collected statistics are:

- **name**

  The name of the test.

- **status** (S)

  The termination status of the test, taking on either `pass`, `fail` or `faulted`.

- **time** ($t_{\mu s}$)

  The execution time of the test in microseconds. The test is re-run in batches to avoid timer errors. The cumulative average execution time is calculated and the benchmarking is terminated once average execution time stops deviating by a defined precision, for a defined number of batches. This is to improve the statistical robustness of the execution time. Further work will develop this to yield quantifiable statistical certainties and error margins for the results. Only populated for passing tests.

- **blocks** (B_c)

  The number of blocks compiled. Only populated by the JIT.

- **blocks executed** (B_e)

  The number of times blocks are executed. Only populated by the JIT.

- **host instructions** (H_c)

  The number of host instructions compiled. Only populated by the JIT.

- **source instructions** (S_c)

  The number of source instructions touched. Only populated by the JIT.

- **host instructions executed** (H_e)

  The number of times host instructions are executed. Only populated by the JIT.

- **source instructions emulated** (S_e)

  The number of times source instructions are emulated.

Table 7 shows an example output from the test runner. Further statistics will be collected as necessary for future developments.

name	S	$t_{\mu s}$	B_c	B_e	H_c	S_c	H_e	S_e
addiu_0	passed	7.577	1	1	4	1	4	1
addiu_1	passed	10.533	1	1	8	5	8	5
addiu_2	passed	7.837	1	1	10	3	10	3
addiu_3	passed	7.099	1	1	4	1	4	1
addiu_4	passed	7.203	1	1	6	1	6	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
fibonacci(0)	passed	21.392	5	5	39	13	39	13
fibonacci(1)	passed	21.324	5	5	39	13	39	13
fibonacci(2)	passed	32.634	8	12	87	29	135	43
fibonacci(3)	passed	32.711	8	19	87	29	231	73
fibonacci(4)	passed	33.607	8	33	87	29	423	133
fibonacci(5)	passed	34.217	8	54	87	29	711	223
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
unroll(512/128)	passed	12.588	2	129	14	6	1411	641
unroll(512/256)	passed	12.985	2	257	12	4	2307	769
unroll(512/512)	passed	15.251	2	513	11	3	4099	1025

Table 7: Subset of the output statistics for the JIT generated by the test runner.

5.5 Configuration

The test runner has been made configurable with the following options:

- **timing**

  Various timing configurations used for performance benchmarking.

  - **precision**

    The precision used when measuring benchmark time. Benchmarks will be performed until the proportional error deviation is smaller the `precision`.

  - **batch_size**

    The number of times to run the test in a single benchmark. Used for reducing timing inaccuracies.

  - **threshold**

    The number of benchmarks that must occur, with the precision met, before the performance collection is complete. Used to reduce the probability of a random low error causing early termination.

Further options will be developed as required.

5.6 Analysis

The aggregated test statistics are then consumed by the analysis framework written in Python. In addition to generating all the graphs and analyses, it also calculates the following composite statistics:

- **mips** $= \frac{\textbf{source instructions emulated}}{\textbf{time}}$

  Mega instructions per second (mips), not to be confused with MIPS! Defines the performance of a given test run in a quantity invariant of the duration of the test itself.

- **hotness** $= \frac{\textbf{blocks executed}}{\textbf{blocks}}$

  How frequently the compiled host blocks are executed on average, or how hot they are.

- **host block size** $= \frac{\textbf{host instructions}}{\textbf{blocks}}$

  Average block size in host instructions.

- **source block size** $= \frac{\textbf{source instructions}}{\textbf{blocks}}$

  Average number of source instructions represented by a single host block.

- **compilation inefficiency** $= \frac{\textbf{host instructions}}{\textbf{source instructions}}$

  Average number of host instructions required to translate a single source instruction.

- **execution inefficiency** $= \frac{\textbf{host instructions executed}}{\textbf{source instructions emulated}}$

  Average number of host instruction executions required to emulate a single source instruction.

6 Results and Evaluation

6.1 Configuration

Table 8 details the machine used to produce the test results.

OS	Windows 10 Home
CPU	i7-4790k @ 4.60 GHz, 4C/8T
RAM	16 GB DDR3 @ 1867 MHz
GPU	GTX 970
Storage	512 GB SSD

Table 8: The machine used to produce the test results.

Table 9 details the build used to produce the results.

Commit Hash	`3408bef`
Configuration	Release
Visual Studio 19	16.9.5

Table 9: The build used to produce the test results.

The program was invoked with the `-timing=final` flag resulting in the test runner configuration detailed in Table 10.

Item	Value
`timing.precision`	0.0001
`timing.batch_size`	100
`timing.threshold`	10

Table 10: The configuration used for the test runner.

6.2 Functionality

All variants of the interpreter, JIT and hybrid emulators passed all 557 included tests. The functional tests were designed as described in subsection 3.2. The full testbench can be found in Appendix A; the full results can be found in Appendix B.

6.3 Performance

This section will explore the performance characteristics shown by all of the SUTs and will answer the research questions proposed in subsection 3.3. Furthermore, the optimisations introduced in subsection 4.6 will be explored to determine which configurations are optimal and when.

This investigation will be conducted in two steps. First, the generalised performance across all tests will be explored for each emulator; this can be done by observing the relationship between different test statistics (such as performance vs hotness). From this the ideal configuration for each emulator will be determined. This will be covered in Sections 6.3.1 to 6.3.3.

Next, in Sections 6.3.4 to 6.3.6, the performance will be investigated more thoroughly on the bespoke set of test suites. Unlike the functionality tests, these test suites are specifically designed to stress the SUTs in different ways and have been designed to answer the various performance research questions. The test suites of interest are as follows:

- `fibonacci(n)`

  Recursive program that computes `fibonacci(n)` designed to explore the performance of highly recursive program. Uses a mixture of ALU and memory operations. The computational complexity is $\mathcal{O}(2^n)$ and thus we can expect to see very long and intensive tests for larger n.

- `fibonacci_rep[k](n)`

  The same as `fibonacci(n)`, however k unique instantions of the `fibonacci` function are generated. This helps investigate how performance scales with more unique code. The values of k used were 10 and 100.

- `factorial(n)`

  Recursive program that computes `factorial(n)`. Since the complexity is $\mathcal{O}(n)$ and $n!$ grows exponentially with respect to n, this test suite will not lead to long intensive tests.

- `memcpy(n)`

  Iterative program that copies n bytes from one location in memory to another. The very high proportion of memory instructions touching a large space of memory will stress the memory map.

- `memcpyw(n)`

  The same as `memcpy(n)` however words will be touched and copied instead of individual bytes.

- `primal(n)`

  An iterative program that determines if n is prime. With no memory instructions present this should result in a very hot ALU heavy loop. Since the complexity is $\mathcal{O}(n)$, the values of n supplied to the test suite will grow on the order of 2^n; all values of n used will be prime.

- `unroll(n/m)`

  A program loop of n iterations that has been unrolled into n/m chunks such that the unrolled program has m iterations. Designed to investigate how the trade-off of unrolling varies with each SUT. All values of n and m used were powers of 2 where $m \leq n$.

6.3.1 Interpreter

The performance of the interpreter with regard to program hotness is shown in Figure 8. From this plot, it is very clear that the interpreter exhibits quite a 'flat' performance profile; increasing hotness past a point yields no change in performance. It can be seen however that extremely cold programs do have a lower performance.

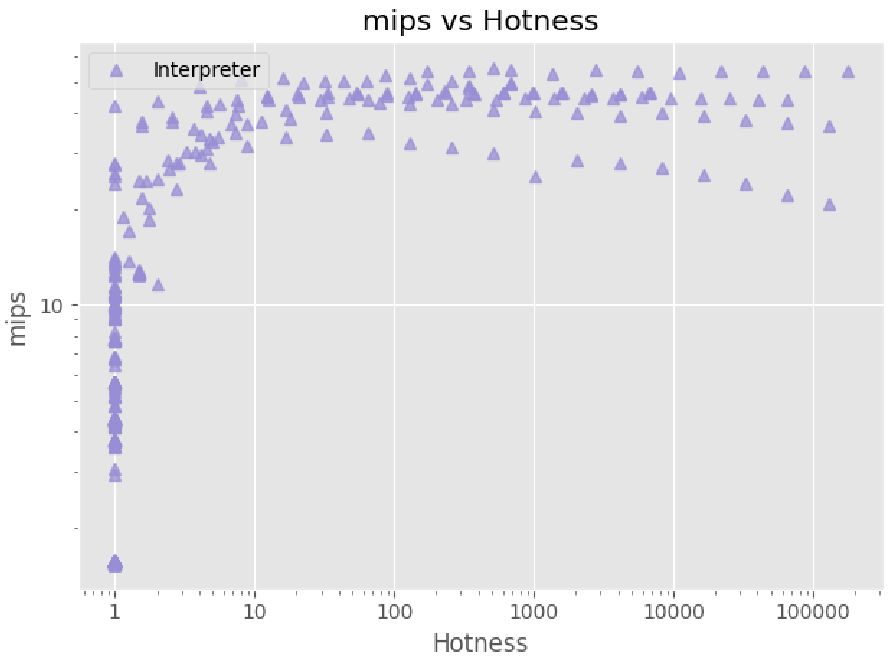

Figure 8: Performance (mips) vs hotness for all tests run on the interpreter.

These results are to be expected; the interpreter is unable to explicitly take advantage of hot blocks as all the work required to emulate a block once is still required for further emulation. Despite this, there are still some savings to be made by executing hotter programs as the (small) initial overheads are amortised. Furthermore, the CPU's branch predictor and cache performance will both increase as the same instructions are emulated more frequently; both of these factors can be attributed to the initial increase in performance as hotness increases.

Figure 9 shows the same relationship between performance and hotness for the interpreter, JIT and hybrid emulators. It can clearly be seen that the JIT and hybrid's performance rapidly increase with program hotness in a manner that the interpreter is simply incapable of.

We see that the interpreter is unable to reach or exceed 60 mips under any circumstances. The performance of the interpreter is very consistent and is not as adversely affected by cold programs as the JIT emulator is, however it is also unable to reach a particularly high performance and has a low performance ceiling.

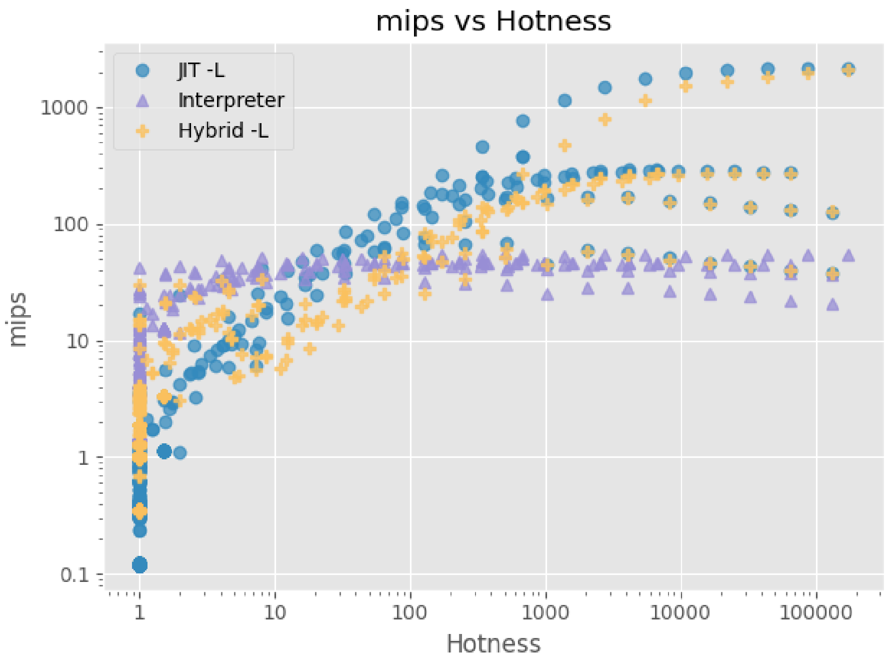

Figure 9: Performance (mips) vs hotness for all tests.

The interpreter does not contain any configuration options and thus no investigation was required.

6.3.2 JIT Emulator

Figure 10 shows the execution time of all tests for the JIT emulator vs the interpreter. It can be seen that, in general, the JIT emulator is able to significantly outperform the interpreter for longer tests, yet performs poorly on shorter tests. More specifically, the execution time appears to have a 'floor' of $\sim$ 10µs for the JIT emulator. For very short tests, the execution time for the interpreter falls while the JIT is not able to overcome this barrier. This property will be explored further in more depth.

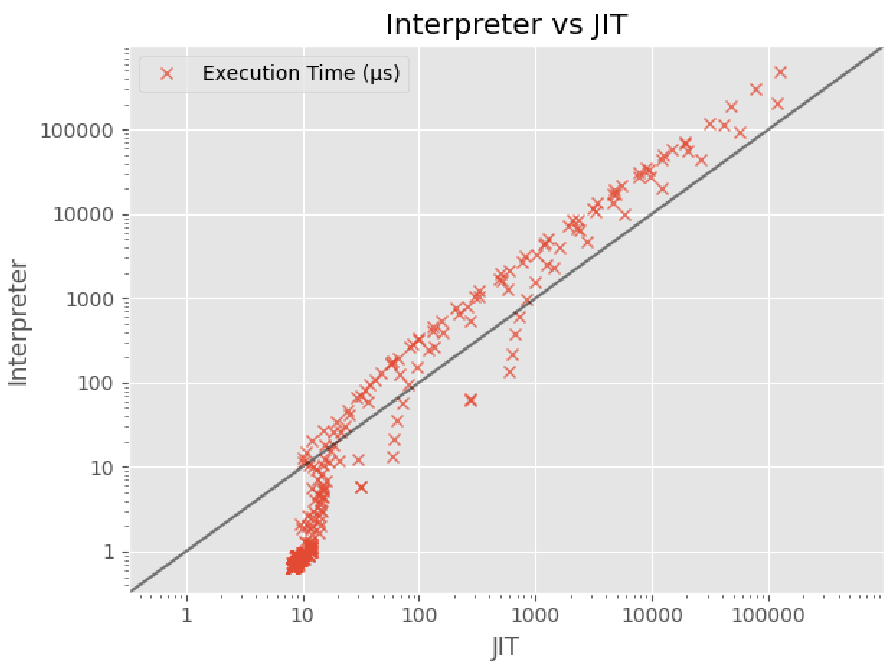

Figure 10: Execution time of all tests for the JIT emulator vs the interpreter.

To investigate the performance characteristics of the JIT emulator, one factor of interest is the hotness of the tests; the performance vs hotness is shown in Figure 11.

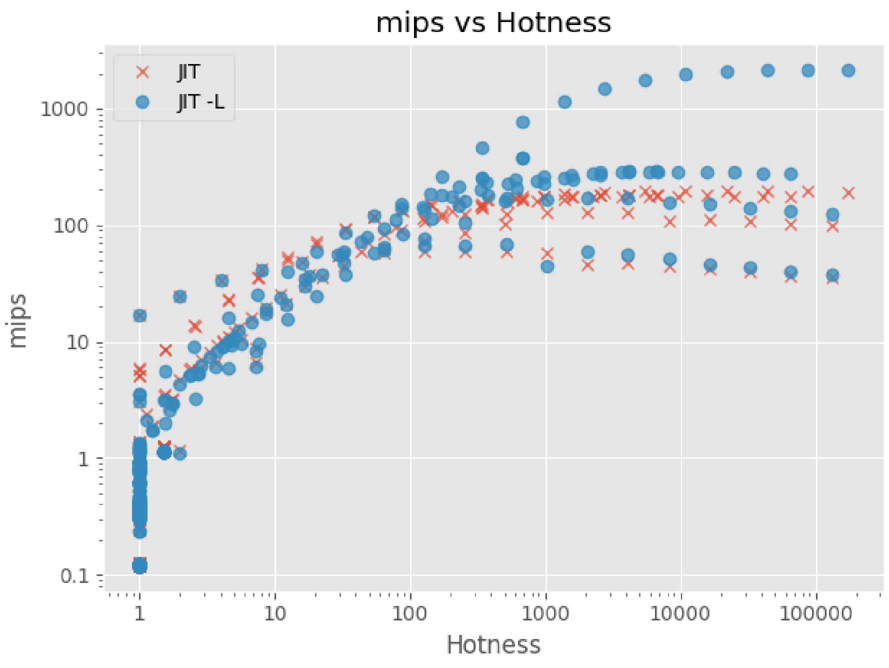

Figure 11: Performance (mips) vs hotness for all tests run on the JIT emulator.

A very clear positive relationship between the hotness of a test and the performance under the JIT can be seen. As the hotness increases the performance also tends to increase, up until a point at which the performance 'saturates'. This behaviour occurs because hotter blocks are run more frequently, and thus the initial compilation cost becomes proportionally smaller. After a point, however, this initial cost becomes irrelevant and the execution time of the block becomes the dominating factor; since being hotter doesn't reduce this (excluding caching benefits) the performance increase saturates. It can be seen that `JIT -L` saturates later at a much higher performance.

Figure 29 shows the performance of the `primal(n)` test suite on the JIT emulators. The test suite consists primarily of a single hot loop, and thus we should expect to see similar performance characteristics.

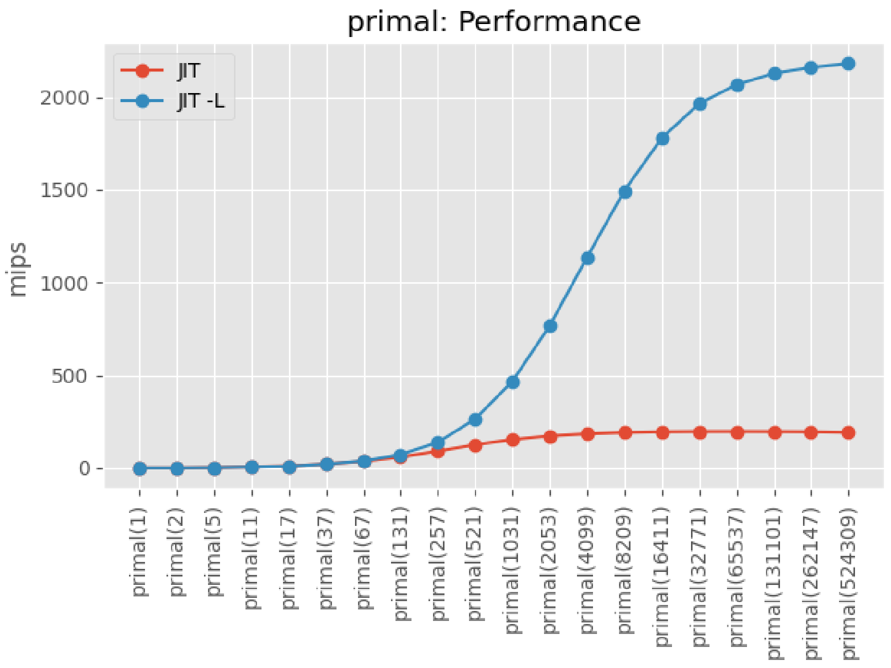

Figure 12: Performance in mips of the primal test suite run on the JIT emulator.

It is evident that as `n` increases the performance of both SUTs increases significantly, however `-L` saturates later at a much higher performance, roughly $\sim 10\times$ higher. `primal(n)` is a best case scenario for `JIT -L` as it is a very hot loop with no memory operations; this allows it the emulator to operate extremely fast once all the overheads have been amortized.

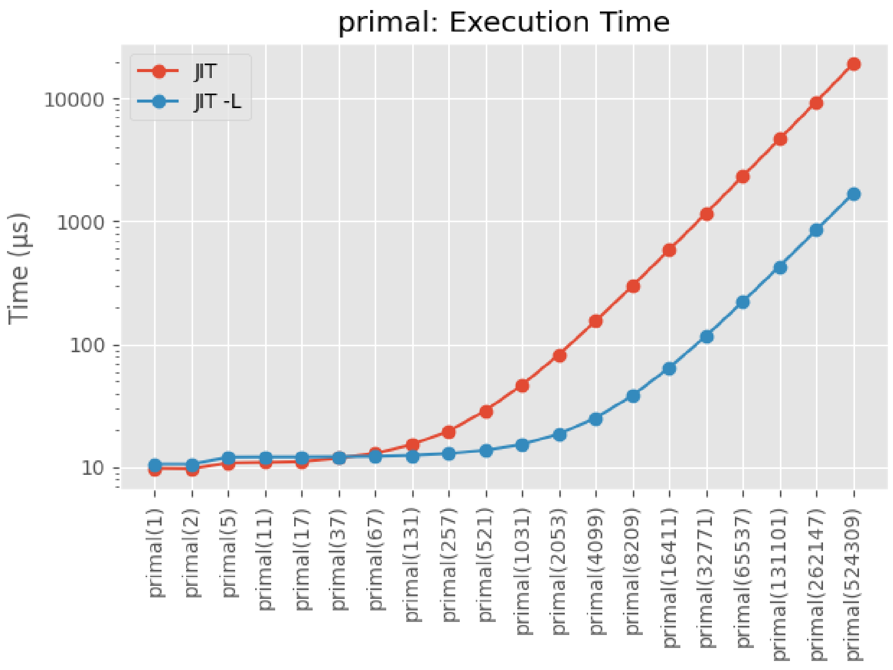

Figure 13: Execution time of the primal test suite run on the JIT emulator.

The effects of the fixed costs and initial overheads is much more present in the execution times as illustrated by Figure 13. It can clearly be seen that for small `n` the execution time remains relatively constant at ∼ 10µs. This would indicate that, under the circumstances tested, the JIT had a fixed cost of ∼ 10µs. As `n` increases the execution time becomes more significant than the overhead, and we start to see the total execution time rise significantly.

While `-L` clearly gives superior results for hotter and more performance intensive tasks, we would expect lighter workloads to suffer; `-L` increases the overheads due to the relinking process, and the increased execution performance likely won't have time to make up for it.

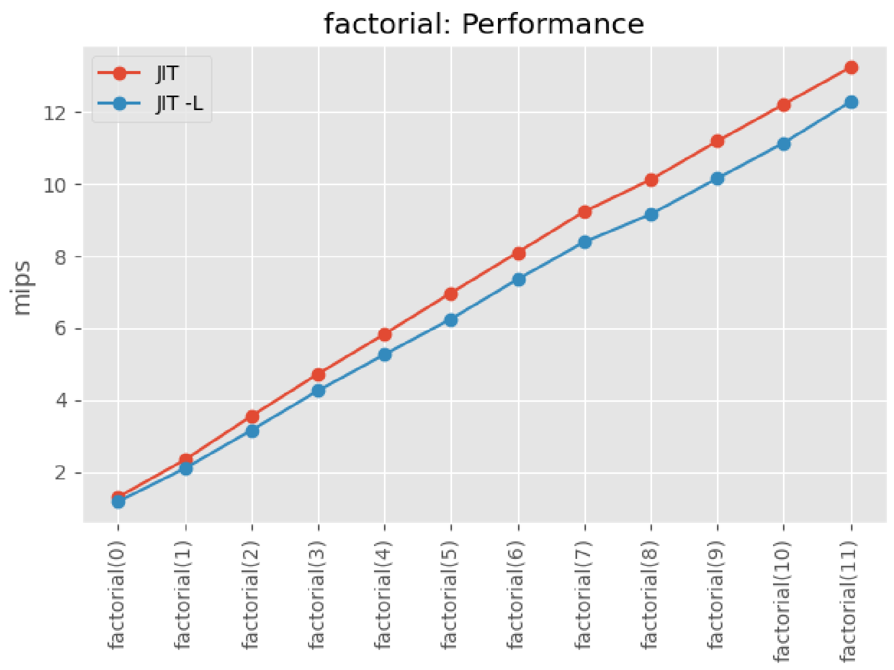

Figure 14: Performance in mips of the factorial test suite run on the JIT emulator.

The test suite `factorial(n)` can only be used for small `n` due to how explosively $n!$ increases with n; the performance for this test suite is shown in Figure 14. It can be seen that `-L` causes a slight, but consistent drop in performance compared to the vanilla `JIT` configuration. The performance in these scenarios is very poor even without `-L`, so this is not a compelling reason to disable `-L` as the high performance in heavier workloads more than makes up for it.

A more general picture can be seen by comparing the execution time of all tests between `JIT -L` and the standard `JIT` as shown by Figure 15. It can be seen that most tests perform the same or better with `-L` enabled, with some tests performing orders of magnitude better.

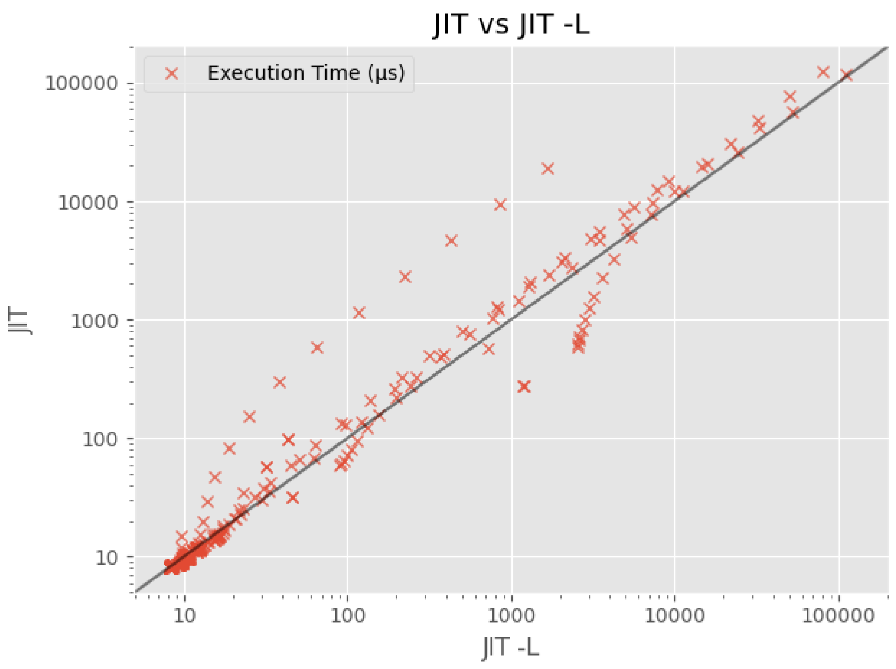

Figure 15: Execution time of all tests for `JIT -L` vs `JIT`.

We can then compare `JIT -L` to the interpreter as shown in Figure 16. For short tests, `JIT -L` loses to the interpreter just as the standard `JIT` does (as shown by Figure 10). The clear difference with `-L` enabled is that the tests that already perform well without `-L` now perform even better, in some cases orders of magnitude better. This is further reason to believe that `-L` is a beneficial feature despite not globally improving performance.

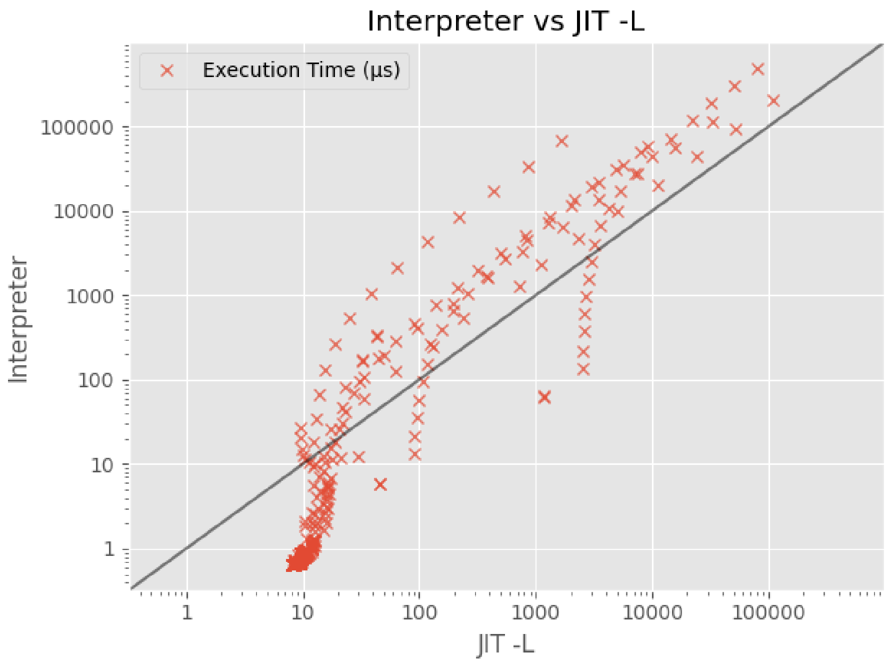

Figure 16: Execution time of all tests for `JIT -L` vs the interpreter.

Another metric of interest is the compilation inefficiency. A high compilation inefficiency would indicate that many host instructions are compiled to emulate a few source instructions; generally speaking, the fewer instructions used the better. Not only should they be faster to compile, as generating instructions has an associated overhead, but execution performance should also increase due to better cache utilisation. Having said this, there are many other factors at play such as the instructions themselves and how often they are executed, so we should not expect a strict relationship between compilation inefficiency and performance.

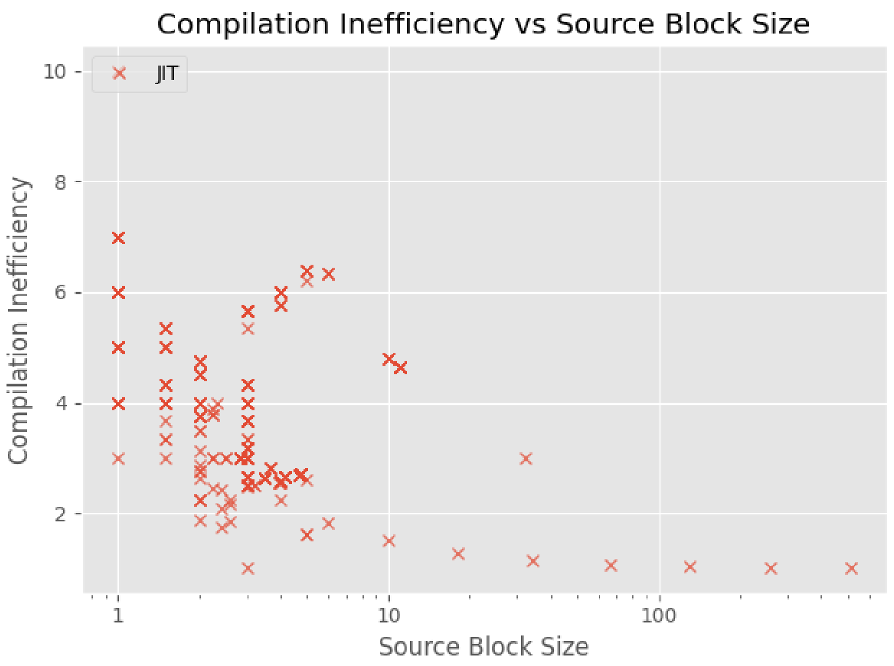

Figure 17: Compilation inefficiency vs average source block size for all tests.

Figure 17 shows the relationship between the average source block size and the compilation inefficiency. From this graph it is very clear that as the source block size increases, the compilation inefficiency falls. This is to be expected as each compiled block includes some setup and teardown instructions, and thus the bigger the block the smaller effect these have. Due to this, it is ideal that the block partitioner produces the largest source blocks possible.

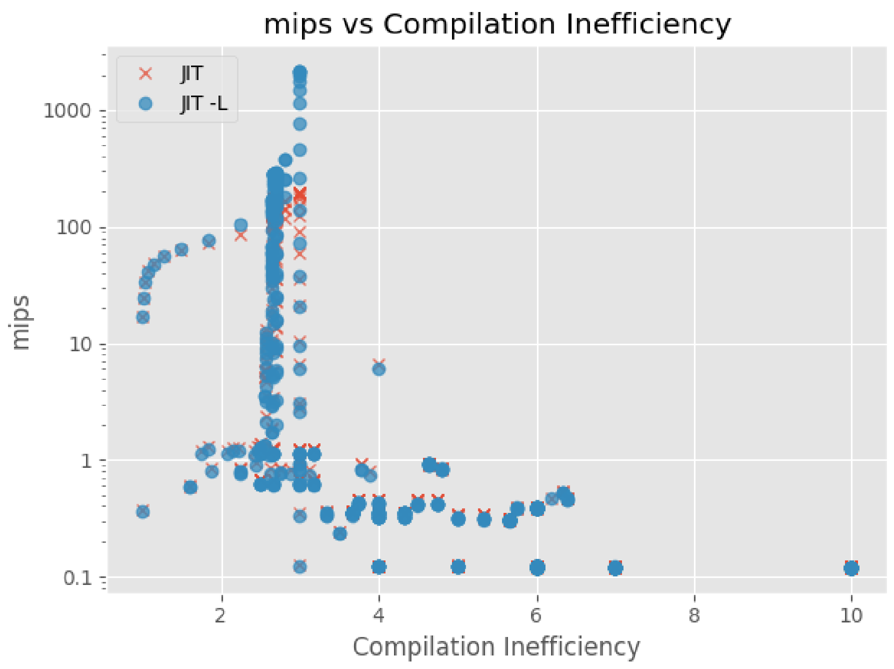

Figure 18: Performance (mips) vs compilation inefficiency for all tests.

More importantly, however, is the relationship between compilation inefficiency and performance. Figure 18 shows that there is quite a clear reciprocal relationship between the performance and the compilation inefficiency; as compilation inefficiency rises the performance drops drastically for both `JIT` and `JIT -L`.

The ideal configuration found for the JIT emulator is detailed in Table 11

Option	Value
`-L`	✓

Table 11: Optimal configuration found for the JIT emulator.

6.3.3 Hybrid Emulator

Figure 19 shows how the performance of the hybrid emulator changes with relationship to the hotness of the programs; unlike the JIT, which has a monotonic relationship, the hybrid shows a more complex relationship.

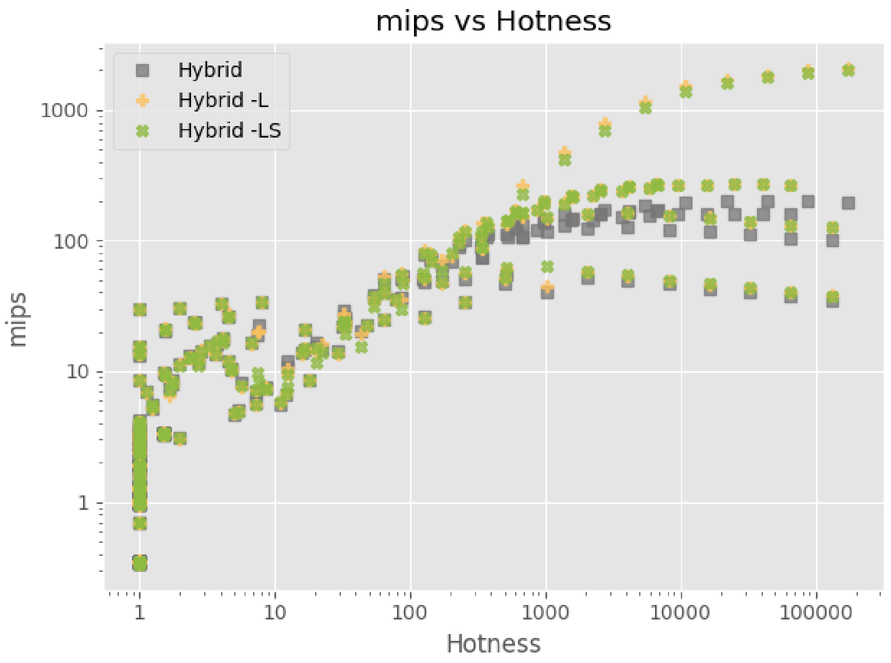

Figure 19: Performance (mips) vs hotness for all tests on the hybrid emulators.

This is because hybrid behaves like an interpreter for sufficiently cold tests, but like a JIT for sufficiently hot tests. This causes it to exhibit the 'flat' performance profile of the interpreter for low hotness, before dropping in performance then rising as the JIT would. The dip in performance or the inflection point is where the hybrid begins to act like a JIT, and thus takes on all the associated overheads of a JIT, without having time make them worthwhile. This region therefore performs worse than both the interpreter or the JIT.

Naturally, we should expect the location of this inflection point to change with `-T`. Since `-T` determines how many times a block must be executed before it is compiled, it roughly corresponds to how hot a block should be before the hybrid begins emulating like a JIT; given this, we would expect the inflection to occur at higher hotness values for higher `-T`.

The ideal value for `-T` is not something that can be found through intuition; larger values will avoid more unnecessary compilation, but smaller values will cause better utilisation of the worker threads and the JIT compiler's increased execution performance.

One might logically think that the minimum value of `-T1` would lead to the highest performance on heavy test cases, but this was not found to be the case. Figure 20 shows how the different values of `-T` perform on the `primal(n)` test suite.

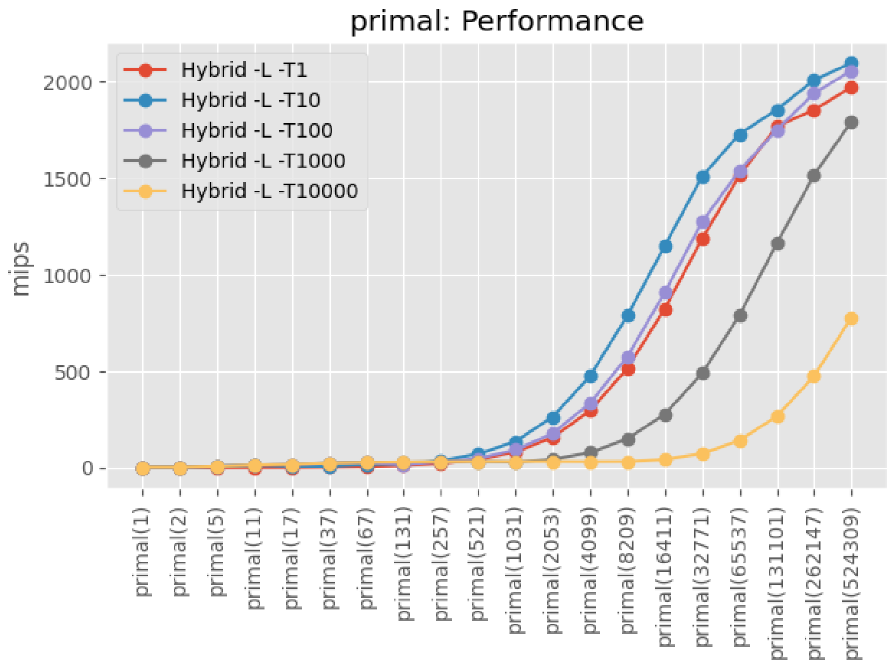

Figure 20: Performance in mips of the primal test suite run on the hybrid emulator with varying values for `-T`.

Interestingly, we observe that `-T10` has the highest performance for the intensive `primal(n)` tests; we might expect that `-T1` would hold this position as that would cause the most aggressive compilation which would eventually lead to the best performance. The most likely explanation is that `-T1` wastes too much time compiling cold blocks that are executed very infrequently and have no time to pay off (such as the entry point) causing the worker pool to be busy for longer. This would delay the compilation of the hot blocks causing the interpreter fallback to activate more frequently, despite the fact that more blocks are translated overall. We observe that larger values of `-T` cause steadily worse performance for the more intensive tests, as we might expect.

Figure 21 shows the execution times for the same test suite, which helps paint a far clearer picture for the performance characteristics with small `n`. We observe that `-T1` is unable to benefit from the low overhead characteristic of the emulator and begins with a relatively high execution time. Furthermore, we can see that the performance at low `n` is improved as `-T` is increased. We can see that each curve behaves characteristically like the interpreter, showing a direct proportionality between the performance and `n` up until an inflection point caused by the hybrid acting more like the JIT emulator. This inflection point is similar to the one observed when investigating the relationship with hotness. Once this point is hit, the execution time becomes fairly constant before eventually curving back into a line just like with the JIT emulator. We can also see that whilst `-T10` generally performs the best, this is not universal, and it has a sour spot where it has begun compiling more blocks without having any time for it to pay off.

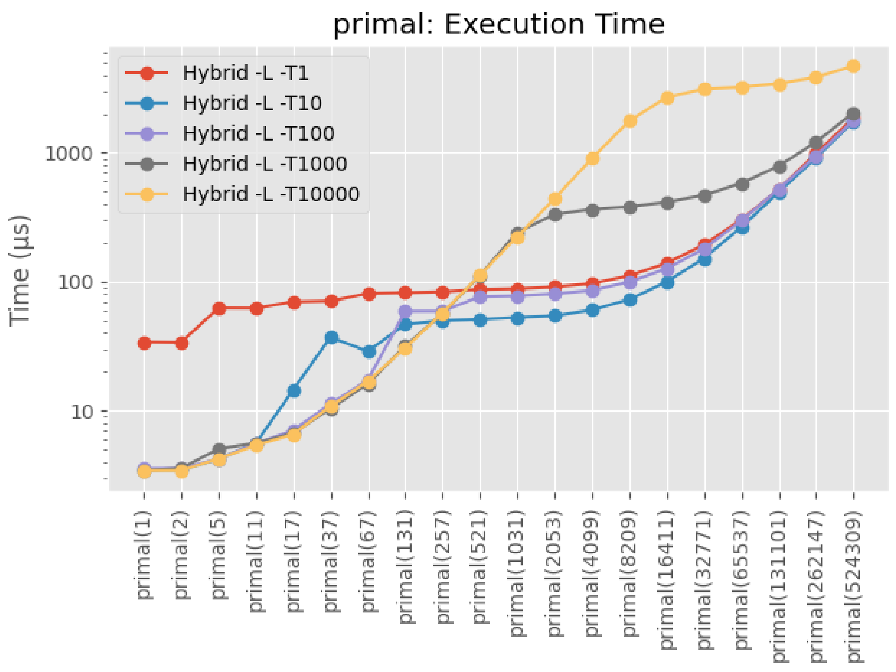

Figure 21: Execution time of the primal test suite run on the hybrid emulator with varying values for `-T`.

This general effect of `-T` on the performance characteristics was observed for most other test suites including `fibonacci(n)` and `memcpy(n)` as shown in Figures 22 and 23

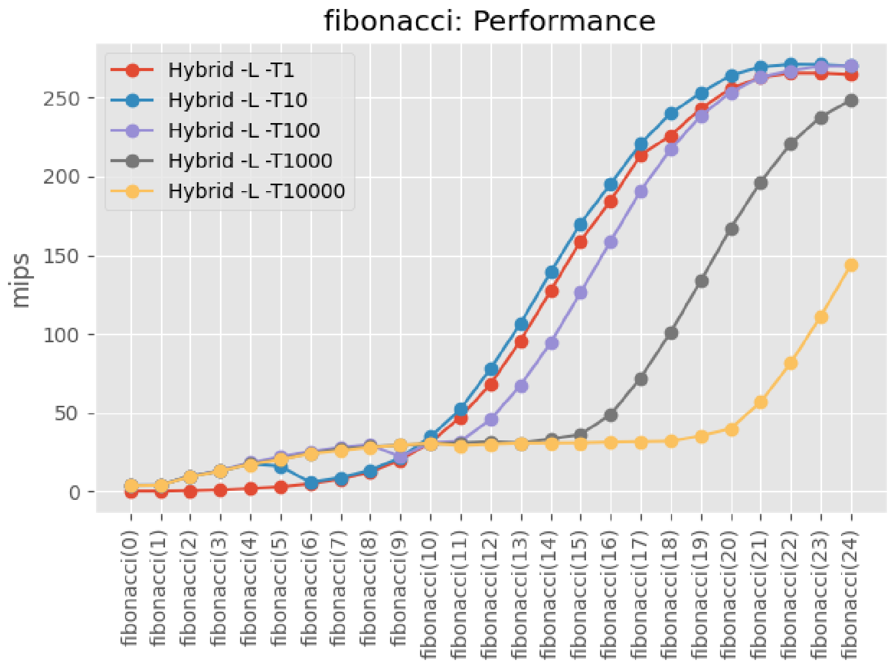

Figure 22: Performance in mips of the fibonacci test suite run on the hybrid emulator with varying values for `-T`.

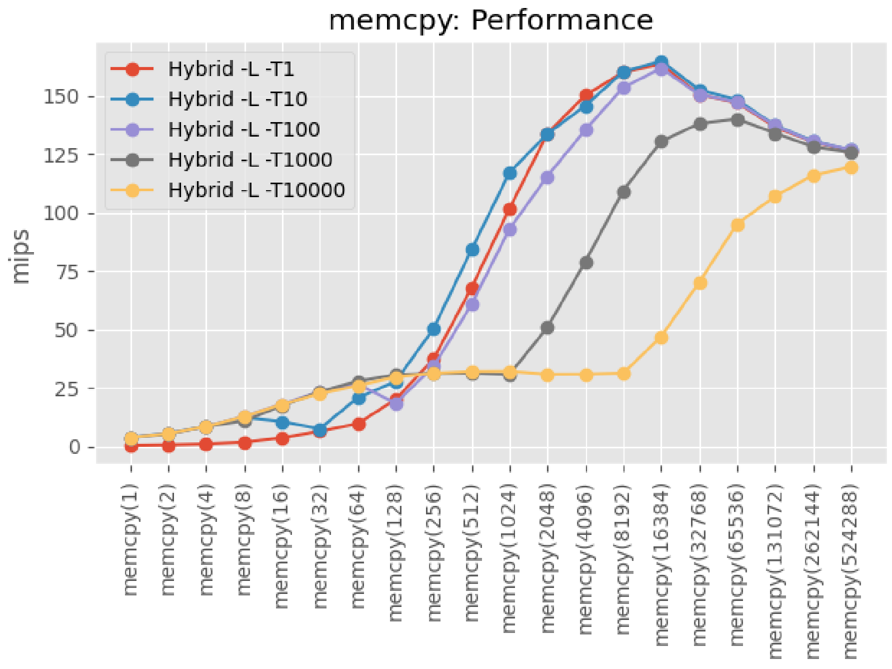

Figure 23: Performance in mips of the memcpy test suite run on the hybrid emulator with varying values for `-T`.

The optimisation `-L` was enabled for the hybrid emulator as it produced the same favourable results as with the JIT emulator, as explored in subsubsection 6.3.2, for reasons already discussed. The majority of tests saw performance improvements with `-L` enabled with hotter tests seeing improvements by orders of magnitude whereas shorter tests saw very slight performance degradation.

The next aspect to investigate is the performance characteristics of the hybrid emulator with speculative compilation (`-S`) enabled. One would expect that it might improve the performance in heavy workloads, as the hybrid will be capable of speculatively compiling blocks before they are first required.

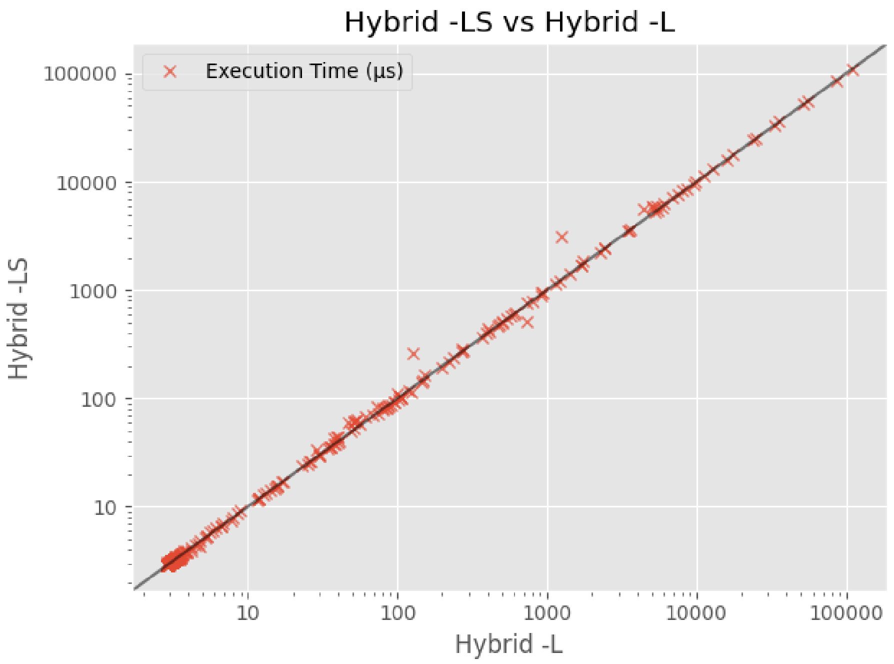

Figure 24: Execution time of all tests for `Hybrid -L` vs `Hybrid -LS`.

The execution time of all tests run on the hybrid emulator with and without `-S` is shown in Figure 24; it is unclear from this what difference, if any, `-S` makes to the performance. In some instances, we see noticeably worse performance with `-S` enabled, however in the majority of cases there is no significant difference. To obtain a clearer picture, we turn our attention to individual test suites.

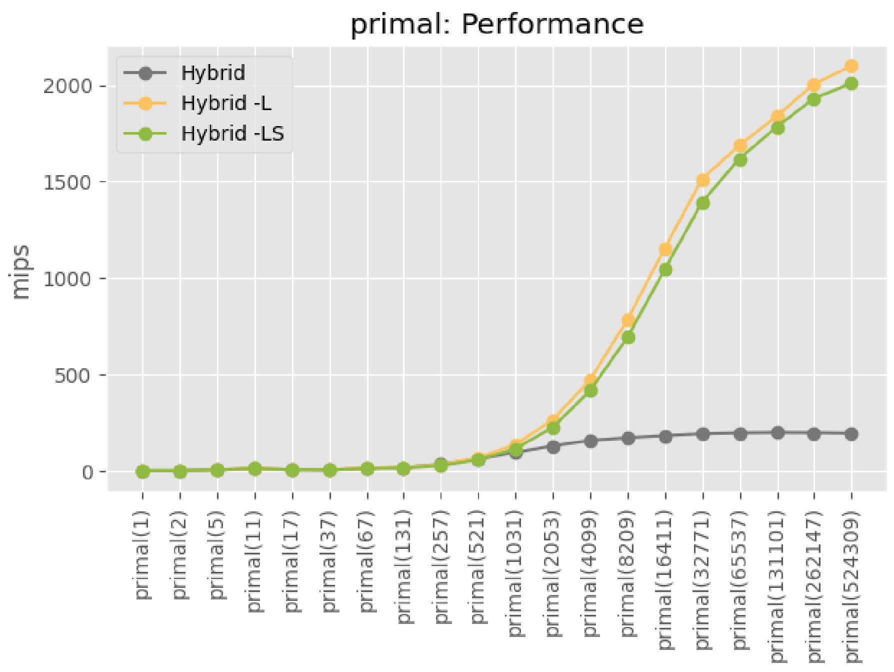

Figure 25: Performance in mips of the primal test suite run on the hybrid emulator with different optimisations enabled.

Figure 25 shows the performance of the `primal(n)` test suite for different configurations of the hybrid emulator. From this, it is clearer that the effect of `-S` is detrimental; this may seem counterintuitive, but it is likely for the same reason that `-T10` was better than `-T1` even for hot workloads, as discussed previously. `-S` is likely overloading the worker pool with unimportant blocks (even more so, since they may never be executed at all) which delays the compilation of the important hot blocks, despite increasing the total number of compiled blocks.

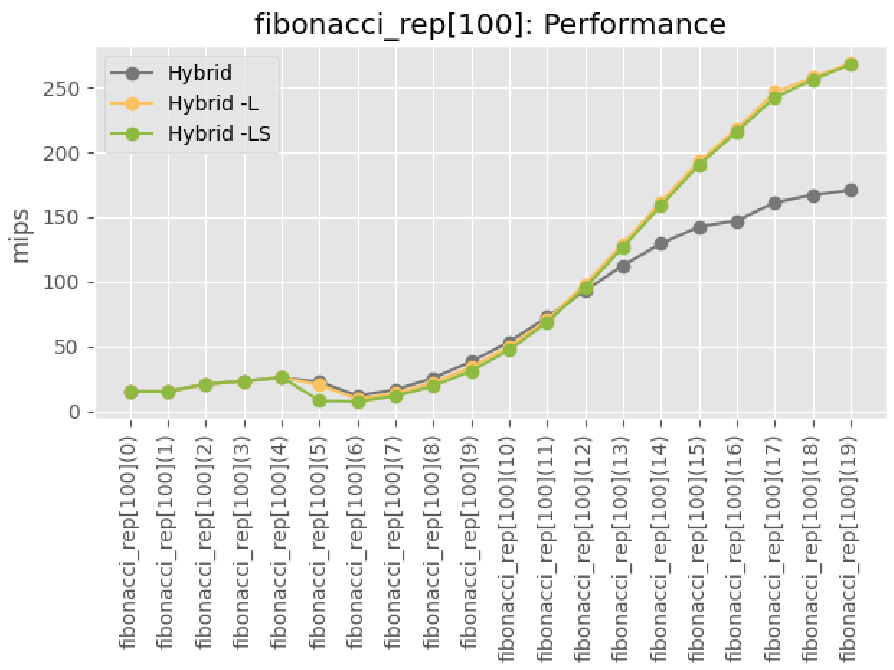

Figure 26: Performance in mips of the fibonacci_rep[100] test suite run on the hybrid emulator with different optimisations enabled.

In order to determine if `-S` shows benefit in larger programs we observe the results of the `fibonacci_rep[100](n)` test suite shown by Figure 26. Despite the much larger program size, `-S` appears to either make no difference or degrade the overall performance. For these reasons, `-S` should be considered detrimental in its current state and disabled.

With the configuration investigated, we can now move onto comparing the performance of the hybrid emulator with that of the JIT emulator; Figure 27 shows the performance of all tests on the hybrid emulator vs the JIT emulator, both with `-L` enabled.

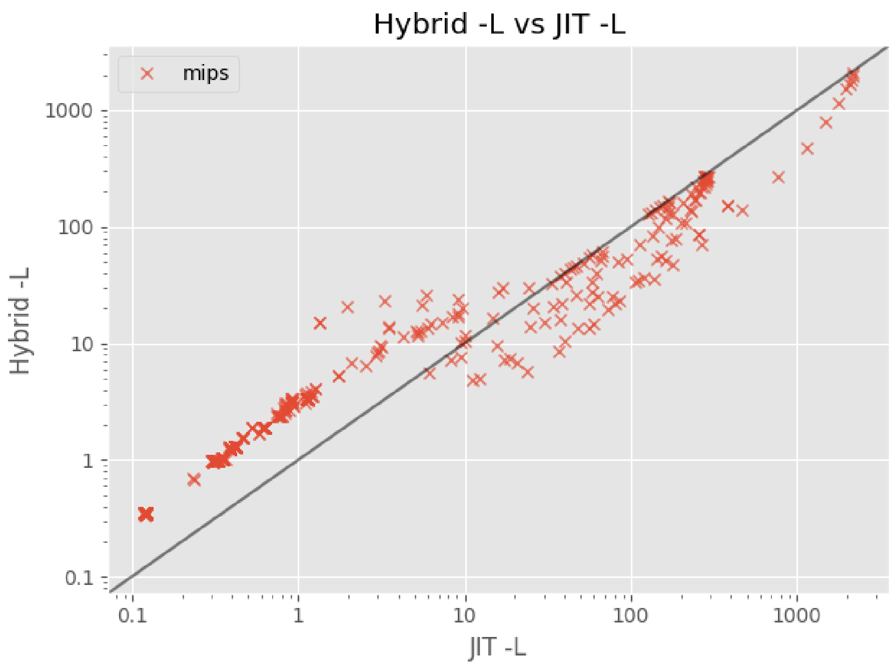

Figure 27: Performance in mips of all tests for `Hybrid -L` vs `JIT -L`.

The interesting trend we see in the plot is caused by the same inflection point we observed when comparing the hybrid emulator's performance to program hotness. Programs with lower performance on both the JIT and hybrid emulators tend to have lower program hotness, causing the hybrid to behave more like an interpreter and outperform the JIT emulator. Likewise, programs with higher performance tend to also have higher hotness, and thus the hybrid acts more like a JIT emulator again. This stops the JIT emulator from beating the hybrid by orders of magnitude like it does to the interpreter.

Despite acting like a JIT emulator for hotter tests, we see that the hybrid still performs worse than the JIT emulator (but not nearly as bad as the interpreter); this due to the added overheads and complexities associated with the hybrid emulator. It appears that the utilisation of the worker thread to asynchronously JIT compile blocks whilst executing the program is not able to make up for the increased overheads.

Despite this, the hybrid emulator's performance characteristic is quite desirable compared to the interpreter; when it loses to the JIT emulator it is never by more than an order of magnitude, unlike the interpreter. While the hybrid sacrifices peak performance over the JIT emulator at the high end, it arguably makes up for it by considerably improving the performance at the low end, resulting in more consistent performance across the board.

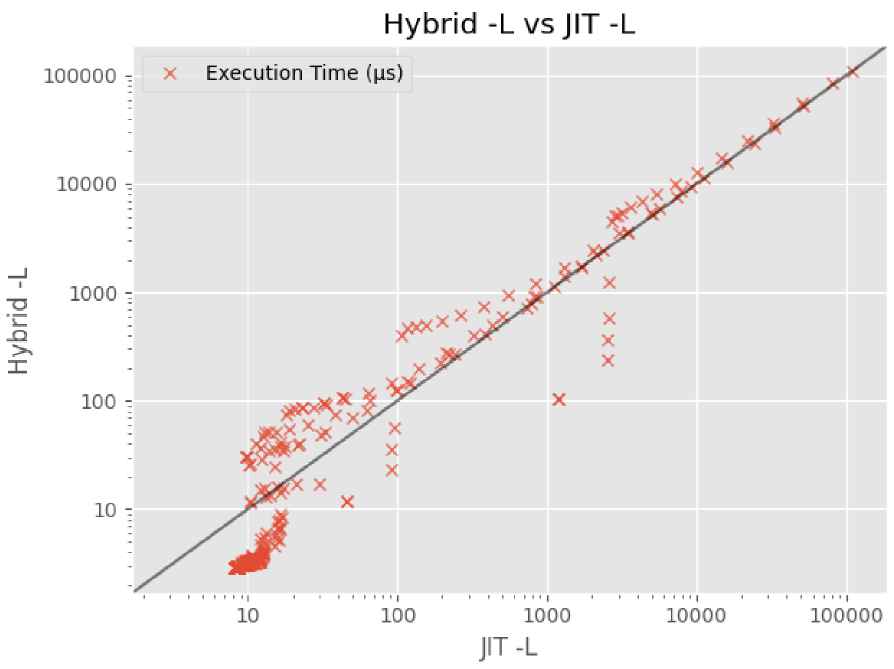

Figure 28: Execution time of all tests for `Hybrid -L` vs `JIT -L`.

Figure 28 shows the execution time of all tests for both the hybrid and JIT emulators and shows a similar picture. Shorter tests tend to run faster on the hybrid emulator, especially at the low end, whereas longer tests tend to perform better on the JIT emulator; in the majority of cases the performance of both emulators is on the same order of magnitude.

The ideal configuration found for the hybrid emulator is detailed in Table 12

Option	Value
`-L`	✓
`-S`	✗
`-T`	10

Table 12: Optimal configuration found for the hybrid emulator.

6.3.4 Iteration

The first relevant test suite for investigating the performance regarding iteration is the `primal(n)` test suite, the performance of which is shown in Figure 29.

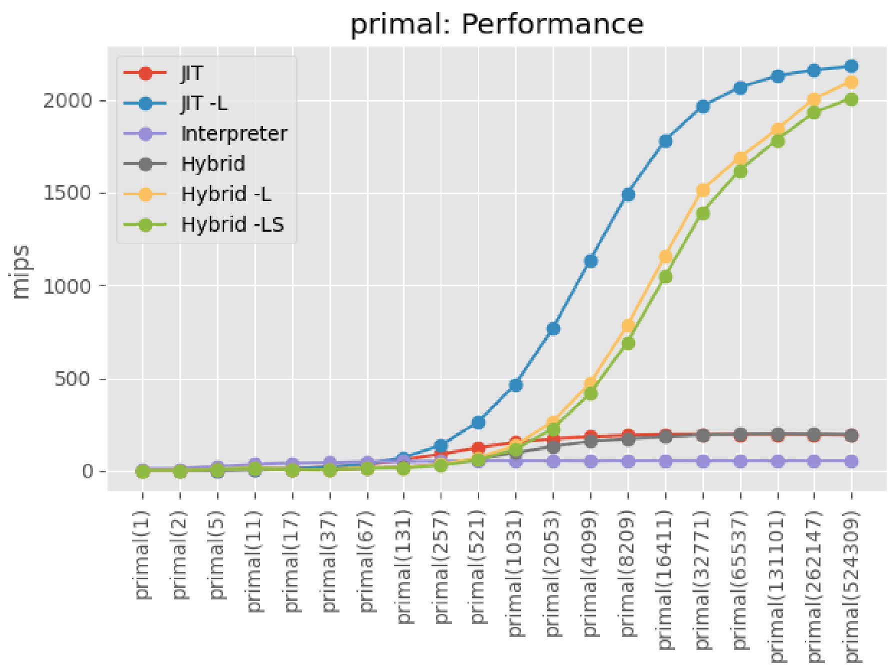

Figure 29: Performance in mips of the primal test suite.

Despite the relatively low performance across the board for low `n`, we see that the performance sky rockets for the JIT and hybrid emulators with `-L` enabled as `n` increases, before plateuing above a staggering 2000 mips. Without `-L` enabled, the peak performance is an order of magnitude higher, yet they still far exceed the performance of the interpreter, which appears to see little to no improvement as `n` increases.

Figure 30 shows the execution time of the same test suite, yielding us a different perspective on the performance characteristics. We see that the interpreter is able to achieve extremely low execution times at low `n`, as low as $\sim$ 1µs. As `n` increases, the exeuction time is directly proportional to `n`. This shows that the performance of the interpreter is relatively constant, seeing little change with `n`; the extremely low execution time floor is indicative of the interpreter's minimal overheads and thus suitability to small workloads.

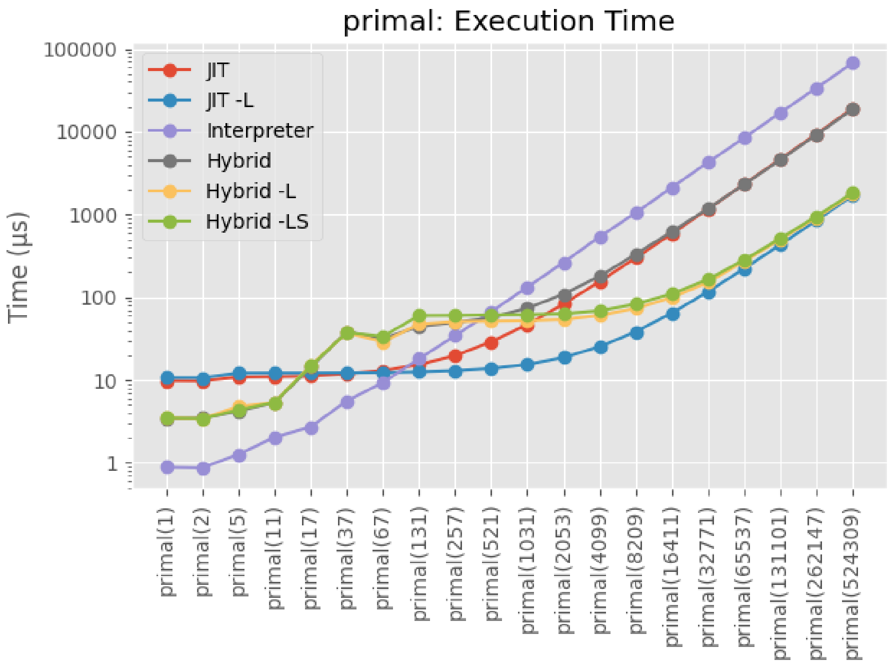

Figure 30: Execution time of the primal test suite.

The JIT emulator shows quite a different curve. It remains constant at $\sim$ 10µs for about half of the test suite, indicative of its high overhead and low initial performance. For these values of `n`, the added work is insignificant to the initial overhead and thus we see continually increasing performance. As `n` becomes sufficiently large, we see the execution time of the JIT emulator become proportional to `n`, showing that the overhead is no longer the dominating factor. We can see that both the hybrid and JIT emulator's converge to the same curve, regardless of `-L` being enabled or not, suggesting that both emulators have the same 'steady state' peak performance under this workload.

The hybrid emulator displays its signature behaviour of acting like the interpreter for low `n`, after which it acts more like the JIT emulator.

The underlying reason for these performance characteristics is clear; higher `n` leads to more intensive iteration, which in turn leads to hotter programs. The results observed here closely match with how the emulators respond to hotness, as explored in Sections 6.3.1 to 6.3.3. Furthermore, the `primal(n)` test suite is extremely arithmetic heavy with no memory operations, allowing the JIT and hybrid emulators to achieve extremely high performance with `-L` enabled. It is clear that the JIT significantly outperforms the interpreter for arithmetic and iteration heavy workloads.

The next test suite to investigate is the `unroll(n/m)`; unlike the other test suites explored, the amount of functional work in all tests is fixed. The test contains a loop with `n` iterations, unrolled into chunks of size `m`. In other words, when $m = n$, no unrolling is performed and we have a single instance of the loop, iterating `m` times. Likewise, when $m = 1$, the loop is fully unrolled into a single instance with no iterations. As `m` increases, the program becomes more iterative, whilst keeping the amount of actual work relatively fixed.

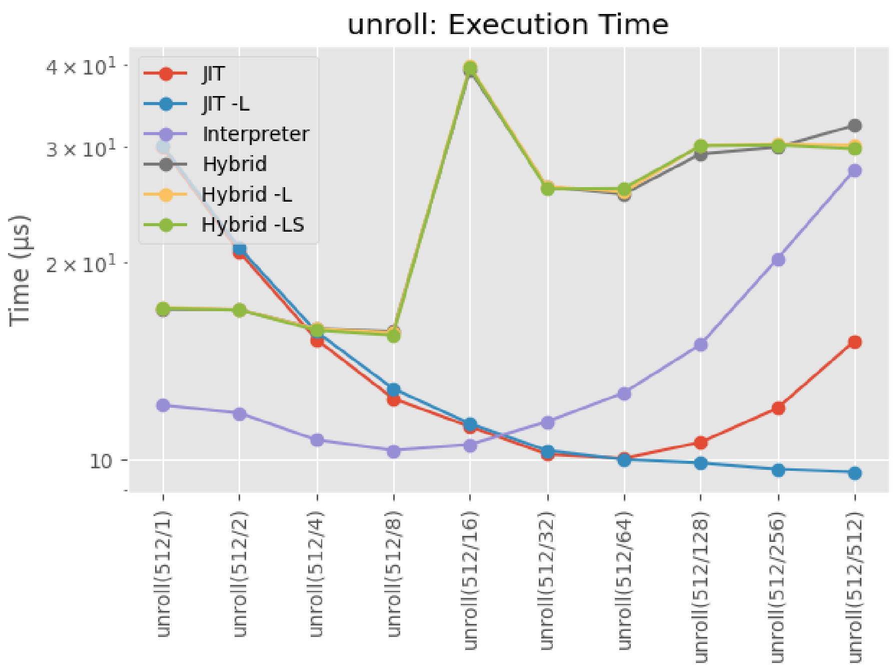

Figure 31: Execution time of the unroll test suite.

The execution times of the test suite is shown in Figure 31; unlike other test suites, the execution time is not monotonic in all cases.

We will begin with the interpreter. As `n` increases, the unroll factor decreases and the program becomes more iterative. We can see that the execution time generally increases; the less unrolled the loop, the more branch instructions evaluated and thus the more work required by the interpreter. This also reduces the proportion of time spent by the interpreter on fast arithmetic instructions. In this sense, the interpreter behaves quite typical of how a real CPU would with an unrolled program: more unrolling generally gives higher performance.

On the other hand, the JIT emulator experiences a very different characteristic. The JIT emulator has the highest execution time when the loop is fully unrolled, and increases in performance as the loop becomes more iterative. As mentioned earlier, less unrolling leads to slightly higher total work required, as more branches need evaluating. Despite this, the JIT emulator performs better the less unrolled the program is for a simple reason; less unrolling means the source block is smaller, and less instructions are compiled. Less unrolling leads to a hotter program with a higher compilation efficiency, both of which were shown in subsubsection 6.3.2 to increase the performance of the JIT emulator.

One peculiarity is how execution time falls monotonically with `-L` enabled, but is actually parabolic with it disabled. This is because, without `-L` enabled, a less unrolled loop would also cause a higher overhead from the runner dispatching to the host blocks each iteration, similar to how the interpreter suffered from the increased branch frequency. This becomes a trade-off with the reduced compilation overhead, and hence a sweetspot appears where the combined effect of the two opposing factors is minimised, resulting in the parabola. With `-L` enabled, the blocks are relinked and no significant overhead is introduced by the more frequent branches, and hence the sweetspot disappears.

6.3.5 Recursion

The next question to explore is the performance characteristics of the SUTs when executing a recursion heavy program. To begin, we will look at the `fibonacci(n)` test suite. At first glance, this will seem very similar to the `primal(n)` test suite: as `n` increases the program hotness will increase. The fact that `fibonacci(n)` is recursive has some important implications.

First the proportion of memory instructions will be much higher. As memory operations are required for the stack, the program will have a lower arithmetic intensity. Second, the recursive function must use `JR` to unwind the stack, a variable jump that cannot be relinked by `-L` as explained in subsubsection 4.6.1. These factors will help us determine which aspects of the performance characteristics observed previously were due to program hotness, and what was due to arithmetic intensity.

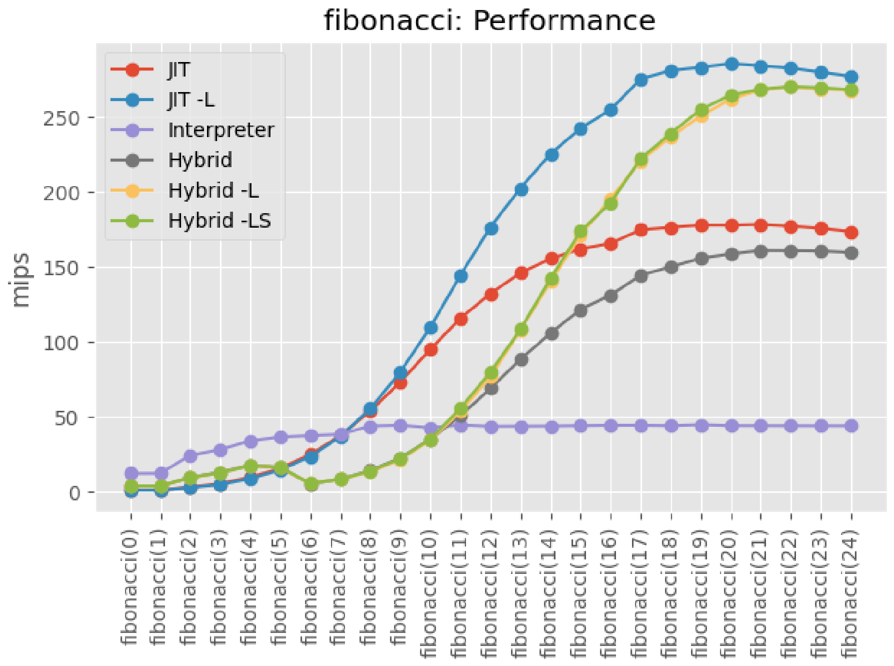

Figure 32: Performance in mips of the fibonacci test suite.

The performance of the `fibonacci(n)` test suite is shown in Figure 32. The pecking order of the different SUTs' peak performance remains the same as for the `primal(n)` test suite explored in subsubsection 6.3.4 and is as shown below:

`JIT -L` > `Hybrid -L` > `JIT` > `Hybrid` > Interpreter

While we can see that the general shape is the same for each SUT, the magnitude of the performance achieved is drastically different in some cases. As `n` increases, the performance of the JIT and hybrid emulators is able to rise significantly due to the increased program hotness, however it peaks at an order of magnitude lower than for `primal(n)` due to the significantly lower arithmetic intensity of the recursive test suite. This helps confirm that while hotness is the underlying factor giving rise the shape of the performance curve in both cases, the high arithmetic intensity of the `primal(n)` test suite significantly amplifies the potential peak performance of the JIT and hybrid emulators, particularly with `-L` enabled.

Under all SUTs, the performance of the recursive test suite is lower than for iterative test suite. This is because, no matter the emulator used, the memory instruction emulation is slow due to the comparatively slow memory map powering the emulators. Emulators with higher execution performance, such as `JIT -L`, suffer from this the most. The interpreter is relatively unaffected due to its generally poor performance.

One peculiarity of interest is how the performance characteristics change for the highest values of `n`; unlike with `primal(n)`, the performance is not actually monotonic, and instead of plateauing it begins to slightly drop. This is due to the memory map and will be explored in subsubsection 6.3.6.

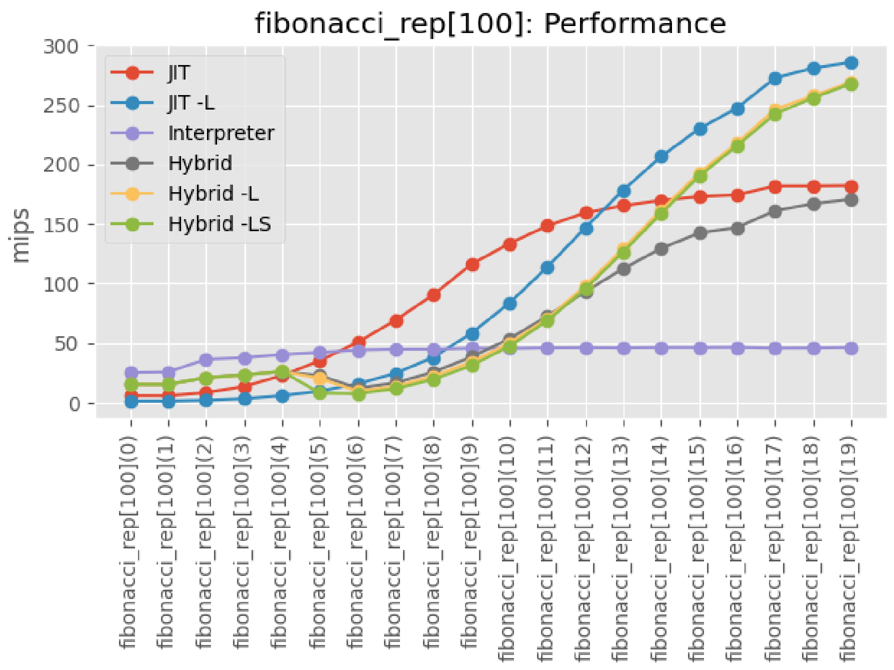

Figure 33: Performance in mips of the fibonacci_rep[100] test suite.

The performance of the sister test suite `fibonacci_rep[100](n)` is shown in Figure 33 and we can see that the performance is much the same with no significant differences of interest. Since `fibonacci_rep[100](n)` contains 100× more unique source code than `fibonacci(n)`, we can conclude that previous findings hold true for larger programs.

We can confidently conclude that the JIT and hybrid emulator have a significant advantage over the interpreter for recursion heavy workloads, just not as large of a lead as with heavily iterative workloads.

The next recursive test suite of interest is `factorial(n)`. While it is recursive, it is not a heavy workload and is very light compared to `primal(n)` and `factorial(n)`.

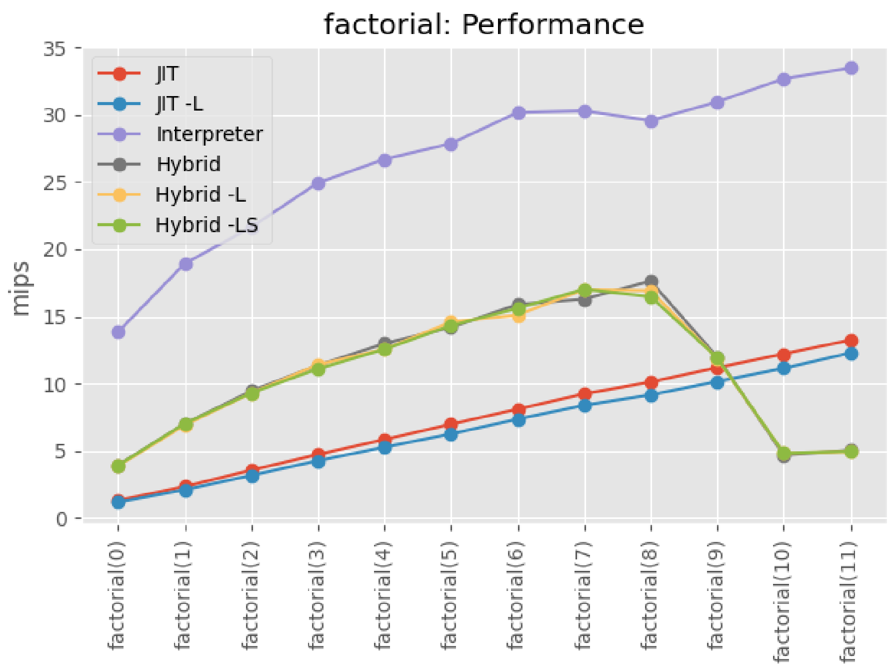

Figure 34: Performance in mips of the factorial test suite.

The performance for the test suite is shown in Figure 34. We see a rather clear reversal in the pecking order of the various SUTs. The program is simply not hot enough for the JIT emulator to overcome its overheads and outperform the interpreter, which remains the best performer for all `n`. Theoretically, if we were able to increase `n` exponentially, we would see the same curves as with `fibonacci(n)`, but the explosively larger values of `n!` prevent us from doing so with a 32-bit result.

The performance behaviour of the hybrid emulator is more interesting. We can clearly see the point at which it starts compiling blocks, causing its performance to plummet lower than that of the JIT emulator; again, if we could increase `n` we would expect the hybrid to eventually climb back up in performance and outperform the interpreter.

6.3.6 Memory Intensive

Last but not least, we will examine the performance of the SUTs when executing memory intensive program; the test suite `memcpyw(n)` copies `n` words (each of which is 4 bytes) from one memory location to another. Not only does this involve very frequent memory instructions, it also causes large amounts of memory to be touched in the memory map for large `n`.

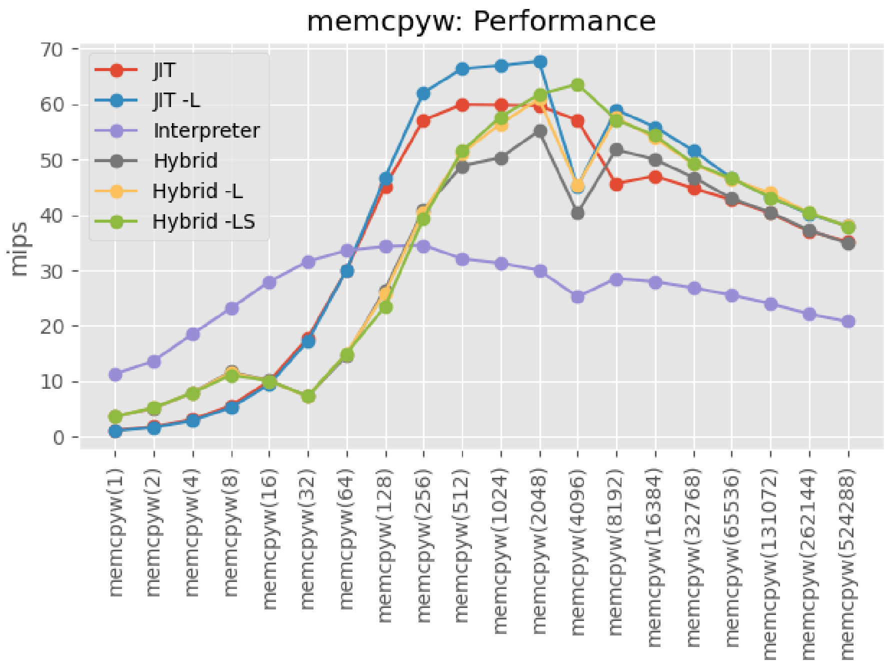

Figure 35: Performance in mips of the memcpyw test suite.

The performance for the test suite is shown in Figure 35; unlike most test suites which show monotonic performance with `n`, all SUTs experienced a parabolic relationship with `n` where the performance rapidly rose to a peak before falling again. This may seem confusing, as higher `n` results in hotter blocks, which we have established results in higher performance. The answer becomes clear however when we acknowledge that increased `n` also causes a larger footprint of emulated memory.

This is because the more memory copied by the test, the more insertions and accesses are made to the internal hash table powering the memory map. Insertion and access time for the map implementation used becomes slower the more items present in the map [12]. This linear slowdown is true for all other major C++ unordered map implementations [12]. This slowdown affects both SUTs as they both internally use the same memory map.

Since large `n` increased performance due to hotness, but decreases performance due to memory map congestion, the two factors oppose each other, and we are left with a performance sweet spot where the combined overhead is minimised. We can see that this sweet spot moves further to right the more the emulator benefits from program hotness by comparing the curve to `JIT -L` to the interpreter: in the case of `JIT -L`, the sweet spot occurs at $n \approx 2048$, yet for the interpreter it is much lower at only $n \approx 256$.

Moreover, the peak performance of all emulators is significantly lower in this test suite than the other test suites covered previously; regardless of program hotness or the emulator used, memory instructions are slow to emulate due to the memory map implementation and result in universally poor performance. This said, the JIT and hybrid emulators are still able to outperform the interpreter and the typical performance pecking order is maintained.

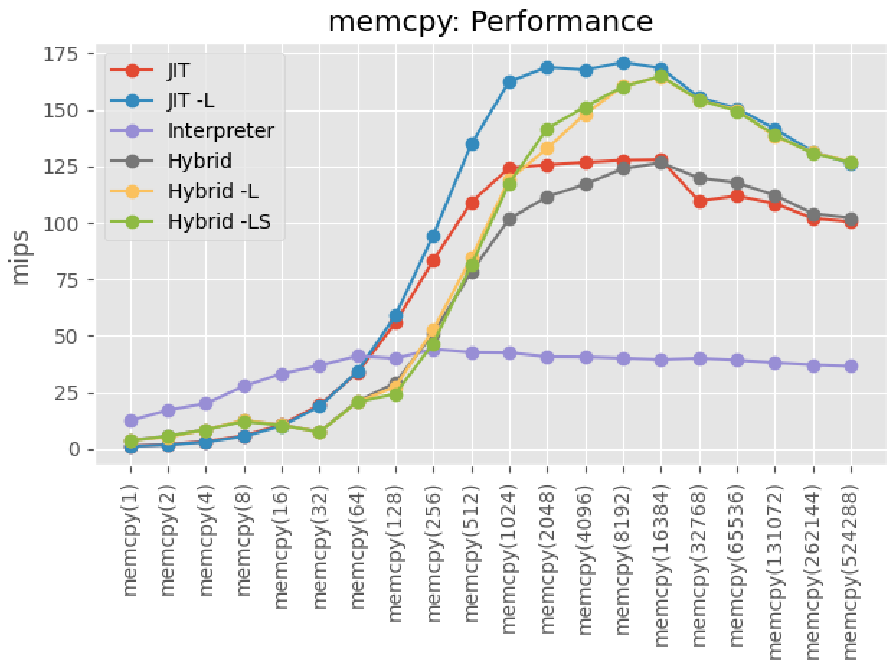

Figure 36: Performance in mips of the memcpy test suite.

Another test suite of interest is `memcpy(n)`, the performance of which is shown in Figure 36. `memcpy(n)` copies `n` bytes of memory instead of `n` words.

On a real MIPS system, we might expect that `memcpy(n)` would perform worse. This is because the reads and writes are no longer word aligned which typically results in lower performance. As is obvious, this was not the case under the emulation. Despite the added complexity and overhead required with non-word-aligned, sub-word memory accesses, the fact that the emulated memory footprint is 4× smaller made up for it. Since the memory map stores entries at the word level, 4 contiguous bytes are stored in a single entry. This reduced congestion increases the performance of the hash table allowing `memcpy(n)` to outperform `memcpyw(n)`.

To conclude, despite being able to outperform the interpreter in memory intensive workloads, neither the JIT nor hybrid are able to perform particularly well due to the slow memory map.

7 Conclusions and Further Work

7.1 Conclusions

The interpreter, JIT and hybrid emulators were successfully written in C++20. The emulators were able to pass all included tests and showed functional parity with one another. The JIT and hybrid emulators were able to significantly outperform the interpreter in many scenarios, demonstrating the viability of JIT compilation as a technique for accelerated binary translation in both industrial and academic applications.

It was shown to outperform the interpreter in many benchmarks as shown in subsection 6.3; in some cases, the JIT emulator was able to achieve speeds as high as 2000 mips whereas the interpreter was unable to even exceed 60 mips. Despite the impressive performance, the JIT emulator was not a universal improvement over the interpreter.

More specifically, the JIT emulator was only able to beat the performance of the interpreter when the emulated blocks were sufficiently hot. If the blocks are not executed frequently enough then the increased execution speed of the compiled blocks is not enough to outweigh the high overhead associated with the compilation of the blocks; in these cases, the slow, but consistent performance of the interpreter is higher. As blocks get hotter however, the performance of the JIT emulator is able to far exceed that of the interpreter — especially with direct linking (`-L`) enabled, resulting in performance improvements by orders of magnitude.

The JIT emulator was able to see significant speedup from arithmetic heavy workloads; in the case of the interpreter, the overhead associated with executing an arithmetic instruction becomes significant relative to the near native performance of the JIT emulator's translated blocks. The same cannot be said for memory intensive programs. Since both the interpreter and the JIT emulator use the same (and relatively slow) memory map, the execution overheads associated with either emulation technique becomes less relevant. In these cases the JIT emulator is only able to slightly outperform the interpreter. The JIT emulator was able to significantly outperform the interpreter for highly recursive workloads, due to the increased program hotness; programs containing hot loops saw even higher performance improvements as no stack operations are required, avoiding the slow memory instructions.

These results are enough to demonstrate the utility and viability of JIT compilation as a technique for accelerated binary emulation; while it performs poorly for simple and short programs, it outperforms the traditional interpreter in heavier workloads, where performance is typically of a greater priority.

Furthermore, the hybrid emulator was successful in capturing desirable properties from both the JIT and interpreter emulators. Whilst it was unable to outperform the JIT emulator in heavy workloads, it significantly outperformed the interpreter in these workloads whilst simultaneously able to outperform the JIT for light workloads; this combination allowed it to minimise the weakness of either technique by sacrificing the maximum performance, greatly improving its viability for widespread general use.

Despite having access to additional logical processors, the hybrid emulator was unable to outperform the JIT emulator; this is likely due to the simple compilation threshold used to determine if blocks should be translated with JIT compilation — subsubsection 7.3.3 explores alternative techniques that could make better utilisation of the worker threads and ultimately push the peak performance of the hybrid emulator beyond that of the JIT emulator.

7.2 Challenges and Reflection

Despite the success of the project, it was not without a great deal of challenges and problems encountered on the way. The following outlines a few of the biggest challenges encountered during the development of the project:

- **Micro-optimisations**

  In most cases, only algorithmic changes have a significant impact on the final performance and micro-optimisations are generally advised against; in this project, that couldn't be further from the case. Operations that would be considered trivial under normal circumstances (such as an extra hash table lookup) would cause significant performance degradation due to the extremely small scale of execution time involved.

  One such example that was a surprise was that `std` implementations of containers such as `std::unordered_map` were relatively slow compared to 3rd party implementations, as mentioned in subsection 4.2.

  By combining profiling, manual inspection of the code, experimentation, and a strong understanding of the language I was able to shave off microseconds here and there resulting in significant performance improvements.

- **Lock-free Multithreading**

  While multithreading sounds great on paper, making extra utilisation of the existing hardware, it isn't without its share of issues. Many painful and transient race conditions were encountered during development which conveniently were not reproducable with logging or debugging enabled.

  Furthermore, the difficulty of multithreading is amplified when everything is made lock-free, something that is required for high performance. For macro scale tasks, such as IO or long jobs, locks are an acceptable means of safe synchronisation between threads. Given that the jobs in the hybrid emulator took only microseconds to execute, the overhead of locking would defeat the point of distributing the work to multiple threads, as the lock overhead would be on a similar order of magnitude to the job itself. Examples of lock-free synchronisation include the results queue optimisation mentioned in subsubsection 4.4.2.

  Effectively optimising was unfortunately not a trivial task as profiling becomes far less viable when analysing a multithreaded system on such a small scale.

- **x86 Assembly**

  Despite the popularity of x86 as an ISA, it is not nearly as well designed as one might expect. As it has evolved heavily from the 8-bit era whilst maintaining a high level of backwards compatibility, it has resulted in some *interesting* instruction encodings and peculiarities. While the official Intel x86 guide *is* comprehensive, it is over 2000 pages long and not particularly digestible. This made it quite difficult to find and extract the relevant information required to build the assembler in an optimal fashion.

On reflection, one of the biggest achievements of the project was how well the JIT and hybrid emulators were able to perform, especially for light workloads. This may seem like a strange takeaway, given the JIT and hybrid emulators far superior performance for heavy workloads, yet relatively poor performance for light workloads compared to the interpreter.

Stopping to consider the sheer amount of work that the JIT emulator must perform to translate a block of MIPS to x86, compared to the trivial emulation required by the interpreter, one would expect the JIT emulator to perform significantly worse than it actually does. Very tight optimisation and design was required at every stage of the process, in addition to the high level algorithm and architectural design in order to achieve the performance it is capable of.

7.3 Further Work

Whilst the project succeeded at its original goals and demonstrated the utility of JIT compilation in binary emulation, there remains room for improvement and expansion. This section will explore some high value additions that can be made to the project with minimal rewriting that improve, optimise, or extend the project.

7.3.1 Cross-platform Support

To keep the scope limited, the current project was only developed to build for Windows x86; despite this, the vast majority of the codebase is still portable to other operating systems (such as macOS and Linux) and x86_64. Other than the exceptions that will be stated otherwise, all code written is portable and compliant to the C++20 standard.

- **Calling Conventions**

  As detailed by subsubsection 4.5.3, the project uses the Windows calling convention `__fastcall`. In order to port it to UNIX like operating systems, the compiler will need to be able to generate code that respects a supported calling convention such as `cdecl` [14].

- **Executable Memory Allocation**

  Executable memory allocation is not possible through the C++ standard library and thus is currently performed by utilising the `Win32` APIs such as `VirtualProtect` [68]; this can be replaced on UNIX like operating systems by using `mmap` [41].

- **Pointers**

All pointers are currently treated by the compiler as 32-bit and thus 32-bit registers and instructions are used by the assembler for pointers such as the register file location and function pointers. In a x86_64 build this would not be true and pointers would be 64-bit large; modifications will be required to handle this appropriately.

7.3.2 Tiered Compilation

JIT compilers provide an interesting conundrum that AOT compilers don't have to consider; more optimised code will run faster, but optimisations take time to perform. In a JIT compiler, the optimisations are happening at runtime and thus even if the compiled code is faster, it might not have been worth to spend the extra time performing those optimisations.

This creates a trade-off where any extra time spent on the compilation process in turn reduces the overall performance unless the improvements are significant enough (or have enough time) to outweigh the increased overhead. This means that many traditional optimisations which take significant time to perform are either difficult or unviable to implement in a JIT compiler.

Intuitively, it is clear that the JIT compiler should spend the time to aggressively optimise blocks that are used frequently; if they are executed many times, then the increased overhead of performing the optimisations becomes less significant and the increased execution performance becomes worth it. Unfortunately, we don't if a block will be executed frequently when it requires compilation.

This can be circumvented through tiered compilation. Each compilation tier can have a different set of optimisations and code generation techniques enabled such that higher tiers spend more time generating code but produce faster code; any optimisations that do not increase compilation times should be included in all tiers. The JIT emulator can then recompile an existing block to a higher tier once some criteria has been met (such as a minimum execution count). This allows the JIT compiler to minimise the overhead associated with compilation, but for the few blocks that are executed enough to warrant more aggressive optimisations, it is still possible for the JIT compiler to eventually produce more optimal code. This can provide significant performance improvements for heavy workloads with frequently executed hot blocks.

7.3.3 Improved Compilation Heuristics

Due to the limited number of worker threads (constrained by the number of logical processors on the system), we must be selective when determining which blocks should be compiled. If we are too aggressive and compile everything, performance is degraded due to overloading the worker pool, as shown in subsubsection 6.3.3.

The current compilation heuristic is a simple threshold controlled by `-T`: if a block has been requested more times than the threshold, it is scheduled for compilation. A sufficiently large value of `-T` will ensure that only important blocks are eventually compiled and unimportant blocks are ignored, but at a cost; the larger the value of `-T`, the longer it will take to compile the correct blocks and the interpreter fallback will be required more frequently.

In a perfect world, the hybrid emulator would immediately know the importance of every block as well as how soon each one would be needed, allowing it to prioritise the compilation so that higher priority blocks are scheduled before moving onto lower priority blocks. This would maximise the performance as we would maximise the utilisation of the worker pool whilst simultaneously avoiding the issue of overloading the system with unimportant jobs. Sadly, we do not live in a perfect world.

In order to improve the hybrid emulator a myriad of techniques can be explored to provide a more intelligent compilation heuristic, many of which could be adopted from cache prefetching and branch prediction. Whilst they aren't exactly the same, they aim to solve the same fundamental problem of working out what the system needs *before* it actually needs it; solving the same problem for the hybrid emulator's compilation heuristic would allow it to schedule the correct blocks for compilation before they are required, maximising performance.

One such technique that could be adopted from branch prediction is a simple static branch predictor: if the branch is forwards, assume it is not taken, and if it is backwards, assume it is taken. This technique may seem simplistic to the point of uselessness, and indeed modern branch predictors adopt far more complex techniques, but the fundamental strategy still has merit; backwards branches are commonly indicative of a loop, in which case it will be taken many times, whereas forwards branches can indicate a loop exit, which will be taken far less frequently. This could be adapted and used as a compilation heuristic; if the requested block is at a lower PC than the current PC, schedule it for compilation more aggressively (such as with a lower threshold).

7.3.4 Binary Object Loader

The current binary loader assumes that the binary file is composed entirely of instructions and decodes them as such, yielding a contiguous stream of instructions. An enhanced loader would understand the 'Execution and Linking Format' (ELF) [39] used by the MIPS object file format for a given operating system. This would allow it to properly load the different sections present in a binary, such as `.text` and `.data` sections.

This would allow the emulator to directly consume a prebuilt executable program; not only does this allow for far better testing, as now prebuilt executable benchmarks/tests can be used in addition to the output of common compilers such as GCC, but the utility of the project to an end user would be greatly increased.

7.3.5 Automatic Testing and Verification

Despite the size and coverage of the current test suite, it is fundamentally limited to what someone could write by hand. For a more comprehensive view of the SUTs' performance and functionality characteristics, much more test cases would be optimal, specifically those with different sizes, structures and complexities. Writing hundreds of thousands of tests by hand would quickly become infeasible, even with generators.

Csmith [20] is a tool that can generate an unlimited number of valid C programs. This combined with GCC [27] cross compiled for MIPS [34] can be used to generate an unlimited number of valid MIPS assembly/binary files. These tests can then be added to the test directory and consumed by the existing testing framework.

Currently, the test framework is not directly compatible with the output of GCC for a few reasons:

- **Compiler Directives**

  The parser can be upgraded to consume the compiler directives either ignoring them or actioning them if applicable.

- **No Entry Point**

  The generated assembly does not have a compiled entry point and instead the `.ent` directive is used to indicate the main function as being an entry point. A simple post processor script can be used to auto generate entry point assembly at the beginning of the file that calls the main function appropriately.

The output of Csmith is not guaranteed to terminate [75]. This would cause the test runner to stall indefinitely whilst trying to measure the performance. To remedy this, a post-processing step can be added to the test generation process. Once a test has been completely generated by the previous steps, it can be invoked directly on the emulator with a timeout; if the timeout is exceeded then the test will be discarded from the suite and re-generated.

With this we would be able to greatly improve the coverage of the performance data. Unfortunately, automatic test generation alone is unable to improve the functional aspect of the testbench. Without an oracle, it is impossible to determine what results the tests should produce. If we guarantee that all generated tests are deterministic and free from undefined behaviour, we can exploit this property to create a derived oracle. Given this, a single test should produce the exact same results no matter what SUT it is run on, given that the SUT is correct.

This can be used for a form of automatic verification. All tests can be run on all SUTs, and the results can be cross compared. Any tests that do not exhibit consistent behaviour on all SUTs will be flagged for manual testing to detect the issue. This can help detect defects in the test themselves (such as containing undefined behaviour) or as a basic form of functionality testing for tests without any assertions defined; this can be useful for the output of Csmith.

8 User Guide

This section will briefly detail how to build and run the JiTBoy project for those interested in doing so. It is not intended as an end user product, but the full source can be found at Appendix A.

8.1 Building JiTBoy

The JiTBoy project can be built by building the solution `JiTBoy.sln` either through Visual Studio 2019 or with MSBuild.

JiTBoy has two different build configurations that can be used:

- **Release**

  Release builds give the highest performance but take longer to build; furthermore, no logs are produced from release builds. Use Release for running mass tests or collecting performance data.

- **Debug**

  Debug builds enable debugging in addition to producing more detailed logs of the emulator execution, but the performance is significantly worse. Use Debug builds for running single tests or investigating test failures and errors.

8.2 Running JiTBoy

JiTBoy can be run by executing `JiTBoy.exe` in a command line interface (CLI) such as `cmd`. The executable will be in either `Release` or `Debug` depending on which configuration the project was built with.

By default, JiTBoy will run all tests on all emulator configurations with no performance data collected. To collect performance data, provide one of the following `--timing` arguments in Table 13.

Argument	Configuration	
`--timing=none`	`batch_size`	1
	`threshold`	0
	`precision`	1
`--timing=fast`	`batch_size`	1
	`threshold`	2
	`precision`	0.1
`--timing=accurate`	`batch_size`	10
	`threshold`	10
	`precision`	0.01
`--timing=final`	`batch_size`	100
	`threshold`	10
	`precision`	0.0001

Table 13: Supported timing configurations for the test runner CLI.

JiTBoy can also be run on a single test with an emulator config of choice by using `--single`; it is recommended to build with the `Debug` configuration for improved logging. To use `--single`, invoke JiTBoy with the following format:

```
JiTBoy.exe --single test_path emulator_config
```

`test_path` must be a relative or absolute path to the test file desired. If a normal assembly file is provided, then no test validation will be performed. Emulator config must be one of the following emulators in Table 14. The emulator config is case insensitive.

Emulator	Argument Options	
`JIT`	`-L`	Enables direct linking
`Hybrid`	`-L`	Enables direct linking
	`-S`	Enables speculative compilation
	`-Tx`	Sets the compilation threshold to `x`
`Interpreter`	None	

Table 14: Supported emulator configurations for JiTBoy's `--single` mode.

For example, the following would invoke the test `long_mul.s` under the JIT emulator with direct linking enabled:

```
JiTBoy.exe --single tests/mips/func/long_mul.s JIT -L
```

9 Ethical, Legal and Safety Plan

At times emulation finds itself in a gray area legally, usually due to two reasons:

- **Unlawful redistribution of protected Intellectual Property (IP)**

  Some emulators require protected pieces of IP in order to function correctly. One example of this is PCSX2 requiring a copy of the PlayStation 2 BIOS in order to operate [43]; distribution of this BIOS would be illegal and thus PCSX2 does not come with the BIOS included. Emulators for recreational purposes are also believed to be illegal by some as they allow for piracy of protected IP. This is a misconception as it is the unlawful distribution of the protected IP that is illegal and not the ability to emulate them [33, 56, 58]. Neither of these are a concern in my case as both the emulator and the test programs are written myself and/or sourced under sufficiently permissive licenses.

- **Utilising protected IP to build the emulator**

  In some cases protected IP, such as code, design or otherwise protected documents, are utilised when building an emulator. This poses an issue as is unlicensed usage of the protected IP [67]. In my case all documentation and resources used to build the emulator are legally publicly available and thus this is not a concern.

Given this discussion, there are no identifiable ethical, legal or safety risks associated with the project.

References

[1] *__cdecl | Microsoft Docs*. URL: https://docs.microsoft.com/en-us/cpp/cpp/cdecl. (accessed: 08.11.2020).

[2] *__clrcall | Microsoft Docs*. URL: https://docs.microsoft.com/en-us/cpp/cpp/clrcall. (accessed: 08.11.2020).

[3] *__fastcall | Microsoft Docs*. URL: https://docs.microsoft.com/en-us/cpp/cpp/fastcall. (accessed: 08.11.2020).

[4] *__stdcall | Microsoft Docs*. URL: https://docs.microsoft.com/en-us/cpp/cpp/stdcall. (accessed: 08.11.2020).

[5] *__thiscall | Microsoft Docs*. URL: https://docs.microsoft.com/en-us/cpp/cpp/thiscall. (accessed: 08.11.2020).

[6] *__vectorcall | Microsoft Docs*. URL: https://docs.microsoft.com/en-us/cpp/cpp/vectorcall. (accessed: 08.11.2020).

[7] *A Fast General Purpose Lock-Free Queue for C++*. URL: https://moodycamel.com/blog/2014/a-fast-general-purpose-lock-free-queue-for-c++. (accessed: 21.5.2021).

[8] *About the Rosetta Translation Environment | Apple Developer Documentation*. URL: https://developer.apple.com/documentation/apple_silicon/about_the_rosetta_translation_environment. (accessed: 17.1.2021).

[9] *alignas specifier (since C++11) - cppreference.com*. URL: https://en.cppreference.com/w/cpp/language/alignas. (accessed: 20.6.2021).

[10] *Apple announces Mac transition to Apple silicon*. URL: https://www.apple.com/uk/newsroom/2020/06/apple-announces-mac-transition-to-apple-silicon/. (accessed: 16.1.2021).

[11] *Argument Passing and Naming Conventions | Microsoft Docs*. URL: https://docs.microsoft.com/en-us/cpp/cpp/argument-passing-and-naming-conventions. (accessed: 08.11.2020).

[12] *Benchmark of major hash maps implementations*. URL: https://tessil.github.io/2016/08/29/benchmark-hopscotch-map.html. (accessed: 30.1.2021).

[13] Igor Böhm. "Speeding up dynamic compilation: concurrent and parallel dynamic compilation". PhD thesis. June 2013.

[14] *C/C++ Calling Conventions*. URL: https://scc.ustc.edu.cn/zlsc/sugon/intel/compiler_c/main_cls/bldaps_cls/common/bldaps_calling_conv.htm. (accessed: 17.6.2021).

[15] *CALL: Call Procedure (x86 Instruction Set Reference)*. URL: https://c9x.me/x86/html/file_module_x86_id_26.html. (accessed: 27.5.2021).

[16] *cameron314/concurrentqueue: A fast multi-producer, multi-consumer lock-free concurrent queue for C++11*. URL: https://github.com/cameron314/concurrentqueue. (accessed: 21.5.2021).

[17] A. Chernoff et al. "FX!32 a profile-directed binary translator". In: *IEEE Micro* 18.2 (1998), pp. 56–64. DOI: 10.1109/40.671403.

[18] *CMOVcc: Conditional Move (x86 Instruction Set Reference)*. URL: https://c9x.me/x86/html/file_module_x86_id_34.html. (accessed: 30.5.2021).

[19] *CMP: Compare Two Operands (x86 Instruction Set Reference)*. URL: https://c9x.me/x86/html/file_module_x86_id_35.html. (accessed: 30.5.2021).

[20] *Csmith*. URL: https://embed.cs.utah.edu/csmith/. (accessed: 25.1.2021).

[21] *Dataminer Shares New Details About The Emulation In Super Mario 3D All-Stars - Nintendo Life*. URL: https://www.nintendolife.com/news/2020/09/dataminer_shares_new_details_about_the_emulation_in_super_mario_3d_all-stars. (accessed: 17.1.2021).

[22] *Desktop Operating System Market Share Worldwide | StatCounter Global Stats*. URL: https://gs.statcounter.com/os-market-share/desktop/worldwide. (accessed: 26.1.2021).

[23] *Desktop Windows Version Market Share Worldwide | StatCounter Global Stats*. URL: https://gs.statcounter.com/os-version-market-share/windows/desktop/worldwide. (accessed: 26.1.2021).

[24] *Documentation/TCG - QEMU*. URL: https://wiki.qemu.org/Documentation/TCG. (accessed: 20.1.2021).

[25] *Dolphin Emulator | GameCube/Wii games on PC*. URL: https://dolphin-emu.org. (accessed: 16.1.2021).

[26] *Features/tcg-multithread*. URL: https://wiki.qemu.org/Features/tcg-multithread. (accessed: 20.1.2021).

[27] *GCC, the GNU Compiler Collection Free Software Foundation (FSF)*. URL: https://gcc.gnu.org. (accessed: 25.1.2021).

[28] J. Ha et al. "A Concurrent Tracebased JustInTime Compiler for Singlethreaded JavaScript". In: *Workshop on Parallel Execution of Sequential Programs on Multicore Architectures*. PESPMA 01, 2009.

[29] *How x86 emulation works on ARM | Microsoft Docs*. URL: https://docs.microsoft.com/en-us/windows/uwp/porting/apps-on-arm-x86-emulation. (accessed: 19.1.2021).

[30] *How x86 to arm64 Translation Works in Rosetta 2*. URL: https://www.infoq.com/news/2020/11/rosetta-2-translation/. (accessed: 19.1.2021).

[31] *Introducing x64 emulation in preview for Windows 10 on ARM PCs to the Windows Insider Program | Windows Insider Blog*. URL: https://blogs.windows.com/windows-insider/2020/12/10/introducing-x64-emulation-in-preview-for-windows-10-on-arm-pcs-to-the-windows-insider-program/. (accessed: 19.1.2021).

[32] Prasad Kulkarni, Matthew Arnold, and Michael Hind. "Dynamic Compilation: The Benefits of Early Investing". In: *Proceedings of the 3rd International Conference on Virtual Execution Environments*. VEE '07. San Diego, California, USA: Association for Computing Machinery, 2007, pp. 94–104. ISBN: 9781595936301. DOI: 10.1145/1254810.1254824. URL: https://doi.org/10.1145/1254810.1254824.

[33] *Lewis Galoob Toys, Inc., Plaintiffappellee, v. Nintendo of America, Inc., Defendant-appellant.nintendo of America, Inc., Plaintiffappellant, v. Lewis Galoob Toys, Inc., Defendantappellee, 964 F.2d 965 (9th Cir. 1992).* URL: https://law.justia.com/cases/federal/appellate-courts/F2/964/965/341457/. (accessed: 19.1.2021).

[34] *Linux Toolchain MIPS*. URL: https://www.mips.com/develop/tools/compilers/linux-toolchain/. (accessed: 25.1.2021).

[35] *macOS 11.0 Big Sur: The Ars Technica review | Ars Technica*. URL: https://arstechnica.com/gadgets/2020/11/macos-11-0-big-sur-the-ars-technica-review/10/. (accessed: 19.1.2021).

[36] *Memory Protection Constants (WinNT.h) - Win32 apps | Microsoft Docs*. URL: https://docs.microsoft.com/en-us/windows/win32/memory/memory-protection-constants. (accessed: 20.6.2021).

[37] *Microsoft Compares Sony's Exclusive Line-up With Theirs, Comments On Backwards Compatability & More*. URL: https://gamingbolt.com/microsoft-compares-sonys-exclusive-line-up-with-theirs-comments-on-backwards-compatability-more. (accessed: 17.1.2021).

[38] *Microsoft: Office will be about 20 seconds slower initially on Apple Silicon, Rosetta 2 | ZDNet*. URL: https://www.zdnet.com/article/microsoft-office-will-be-about-20-second-slower-initially-on-apple-silicon-rosetta-2/. (accessed: 19.1.2021).

[39] *MIPS Assembly Language Programmer's Guide*. URL: http://www.cs.unibo.it/~solmi/teaching/arch_2002-2003/AssemblyLanguageProgDoc.pdf. (accessed: 15.6.2021).

[40] *MIPS32™ Architecture For Programmers Volume II: The MIPS32™ Instruction Set*. URL: https://www.cs.cornell.edu/courses/cs3410/2008fa/MIPS_Vol2.pdf. (accessed: 22.1.2021).

[41] *mmap(2) - Linux manual page*. URL: https://man7.org/linux/man-pages/man2/mmap.2.html. (accessed: 17.6.2021).

[42] *PCSX2 The Playstation 2 emulator*. URL: https://pcsx2.net. (accessed: 16.1.2021).

[43] *PCSX2 The Playstation 2 emulator Getting Started*. URL: https://pcsx2.net/getting-started.html. (accessed: 18.1.2021).

[44] *Pointers to Member Functions, C++ FAQ*. URL: https://isocpp.org/wiki/faq/pointers-to-members. (accessed: 23.12.2020).

[45] *POPF/POPFD: Pop Stack into EFLAGS Register (x86 Instruction Set Reference)*. URL: https://c9x.me/x86/html/file_module_x86_id_250.html. (accessed: 30.5.2021).

[46] Mark Probst. “Dynamic Binary Translation”. In: *Linux Developer’s Conference*. UKUUG, 2002.

[47] *PUSHF/PUSHFD: Push EFLAGS Register onto the Stack (x86 Instruction Set Reference)*. URL: `https://c9x.me/x86/html/file_module_x86_id_271.html`. (accessed: 30.5.2021).

[48] *QEMU*. URL: `https://www.qemu.org`. (accessed: 17.1.2021).

[49] *RAII - cppreference.com*. URL: `https://en.cppreference.com/w/cpp/language/raii`. (accessed: 25.5.2021).

[50] *RET: Return from Procedure (x86 Instruction Set Reference)*. URL: `https://c9x.me/x86/html/file_module_x86_id_280.html`. (accessed: 30.5.2021).

[51] *Rosetta | Apple*. URL: `https://web.archive.org/web/20101116094453/http://www.apple.com/asia/rosetta/`. (accessed: 19.1.2021).

[52] *Rosetta 2 is Apple’s key to making the ARM transition less painful | The Verge*. URL: `https://www.theverge.com/21304182/apple-arm-mac-rosetta-2-emulation-app-converter-explainer`. (accessed: 19.1.2021).

[53] *Rosetta Performance (or lack thereof) Apple’s Mac Pro - A True PowerMac Successor*. URL: `https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested/6`. (accessed: 19.1.2021).

[54] *Rosetta2: x86-64 Translation Performance The 2020 Mac Mini Unleashed: Putting Apple Silicon M1 To The Test*. URL: `https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested/6`. (accessed: 19.1.2021).

[55] *Running 32-bit Applications | Microsoft Docs*. URL: `https://docs.microsoft.com/en-us/windows/win32/winprog64/running-32-bit-applications`. (accessed: 19.1.2021).

[56] *Sega Enters. Ltd. v. Accolade, Inc., 977 F.2d 1510 (9th Cir. 1992)*. URL: `https://www.copyright.gov/fair-use/summaries/segaenters-accolade-9thcir1992.pdf`. (accessed: 19.1.2021).

[57] *Smart pointers (Modern C++) | Microsoft Docs*. URL: `https://docs.microsoft.com/en-us/cpp/cpp/smart-pointers-modern-cpp`. (accessed: 25.5.2021).

[58] *Sony Computer Entertainment, Inc. v. Connectix Corporation 203 F.3d 596 (2000)*. URL: `https://casetext.com/case/sony-computer-entertainment-v-connectix-corp-2`. (accessed: 19.1.2021).

[59] *Stack Overflow Developer Survey 2020*. URL: `https://insights.stackoverflow.com/survey/2020`. (accessed: 25.5.2021).

[60] *Static Initialization Order Fiasco - cppreference.com*. URL: `https://en.cppreference.com/w/cpp/language/siof`. (accessed: 22.5.2021).

[61] *std::atomic - cppreference.com*. URL: `https://en.cppreference.com/w/cpp/atomic/atomic`. (accessed: 15.6.2021).

[62] *Teardown: Windows 10 on ARM x86 Emulation*. URL: `https://blogs.blackberry.com/en/2019/09/teardown-windows-10-on-arm-x86-emulation`. (accessed: 19.1.2021).

[63] *Tessil/robin-map: C++ implementation of a fast hash map and hash set using robin hood hashing*. URL: `https://github.com/Tessil/robin-map`. (accessed: 30.1.2021).

[64] *The brains behind Apple's Rosetta: Transitive CNET*. URL: `https://www.cnet.com/news/the-brains-behind-apples-rosetta-transitive/`. (accessed: 19.1.2021).

[65] Nigel Topham and Daniel Jones. "High speed CPU simulation using JIT binary translation". In: (Jan. 2021).

[66] *Using and Preserving Registers in Inline Assembly | Microsoft Docs*. URL: `https://docs.microsoft.com/en-us/cpp/assembler/inline/using-and-preserving-registers-in-inline-assembly`. (accessed: 22.12.2020).

[67] *Using Leaked Nintendo Source Code Poses Serious Legal Risk to Emulators*. URL: `https://www.vice.com/en/article/g5pxjx/using-leaked-nintendo-source-code-poses-serious-legal-risk-to-emulators`. (accessed: 19.1.2021).

[68] *VirtualProtect function (memoryapi.h) | Microsoft Docs*. URL: `https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualprotect`. (accessed: 08.11.2020).

[69] *VirtualQuery function (memoryapi.h) | Microsoft Docs*. URL: `https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualquery`. (accessed: 20.06.2020).

[70] *What Is Ownership? - The Rust Programming Language*. URL: `https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html`. (accessed: 25.5.2021).

[71] *Who uses mainframes and why do they do it?* URL: `https://www.ibm.com/support/knowledgecenter/zosbasics/com.ibm.zos.zmainframe/zconc_whousesmf.htm`. (accessed: 17.1.2021).

[72] *Windows 10 on ARM extensively benchmarked natively and with x86 emulation - MSPoweruser*. URL: `https://mspoweruser.com/windows-10-on-arm-extensively-benchmarked-natively-and-with-x86-emulation/`. (accessed: 19.1.2021).

[73] *Windows on ARM Benchmarked x86 Emulation Performance*. URL: `https://www.techspot.com/review/1599-windows-on-arm-performance/page2.html`. (accessed: 19.1.2021).

[74] *x64 calling convention | Microsoft Docs*. URL: `https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention`. (accessed: 23.12.2020).

[75] Xuejun Yang et al. "Finding and Understanding Bugs in C Compilers". In: *SIGPLAN Not.* 46.6 (June 2011), pp. 283–294. ISSN: 0362-1340. DOI: `10.1145/1993316.1993532`. URL: `https://doi.org/10.1145/1993316.1993532`.

Appendices

A Code Repository

GitHub: `https://github.com/QFSW/JiTBoy`

B Results Repository

GitHub: `https://github.com/QFSW/JiTBoy-Results`