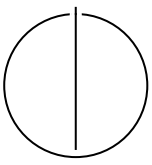
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Design & Implementation of an ARM64 Dynamic Backend for a Hybrid Binary Translator

Theofilos Augoustis



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Design & Implementation of an ARM64 Dynamic Backend for a Hybrid Binary Translator

Entwurf und Implementierung eines dynamischen ARM64-Backends für einen hybriden Binärübersetzer

Author:	Theofilos Augoustis
Supervisor:	Pramod Bhatotia
Advisor:	Sebastian Reimers
Submission Date:	13.03.2025

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 13.03.2025 Theofilos Augoustis

Acknowledgments

I would like to thank a number of people who have guided and supported me in my academic life. First of all, I would like to thank my supervisor, prof. Pramod Bhatotia, who has welcomed me into his chair and supported my academic work from the very start of my Masters. Collaborating with his chair has been the single most positive experience of my studies and I am deeply grateful for this opportunity.

I would also like to thank my research advisor Redha Gouicem, who gave me the chance to work on Arancini. Most importantly, I want to thank my thesis advisor, Sebastian Reimers, who has contributed in more ways than I can mention to my personal and professional development. I have the deepest respect and appreciation for both you.

This Masters is the achievement of more than just one person and I want to thank all those that contributed to it. In particular, I want to thank my wife Katsiaryna for making even the hardest moments during this time happier, while inspiring me to be a better person every day. Furthermore, I want to thank my mother and grandmother, who have worked hard to raise and educate me. It is your efforts that have made this work possible and it is your achievement as much as it mine.

Finally, I would like to acknowledge others who played a major role during this time. I want to thank prof. Doru Todinca for his help in starting my studies in Germany and Andrei Palfi for teaching me how to be an engineer back when I was just starting out. I also want to thank all the friends that I've made at TUM, you all pursue excellence and you have inspired me to do the same.

Abstract

The increasingly widespread use of a diverse set of new computing architectures and the complexity involved in porting existing software to these new platforms has led to increased interest in emulation technology. Emulators promise to support existing programs on different architectures as efficiently as possible.

Modern emulators rely on dynamic binary translation to perform emulation but suffer from *performance* issues. The alternative of static binary translation may alleviate these issues but it is fundamentally incomplete. Preliminary work on the Arancini Hybrid Binary Translator for high-performance emulation has shown that combining the two approaches is both feasible and beneficial. This thesis describes the design and implementation of a Dynamic Binary Translator built from the ground up to integrate with a statically translated program.

The design of the Arm dynamic backed for Arancini is explored in depth as to the techniques needed to successfully blend static and dynamic translations. Furthermore, an evaluation of the dynamic binary translator is provided to highlight its performance and completeness characteristics.

Zusammenfassung

Der zunehmend verbreitete Einsatz einer Vielzahl neuer Computerarchitekturen und die Komplexität, die mit der Portierung bestehender Software auf diese neuen Plattformen verbunden ist, hat das Interesse an der Emulationstechnologie verstärkt. Emulatoren versprechen bestehende Programme auf verschiedenen Architekturen so effizient wie möglich zu unterstützen.

Moderne Emulatoren basieren auf dynamischer Binärübersetzung, um die Emulation durchzuführen, leiden jedoch unter *Leistungsproblemen*. Die Alternative der statischen Binärübersetzung kann diese Probleme lösen, ist aber grundsätzlich unvollständig. Vorläufige Arbeiten zum ARANCINI hybriden Binärübersetzer für Hochleistungsemulation haben gezeigt, dass die Kombination der beiden Ansätze sowohl umsetzbar als auch vorteilhaft ist.

Diese Arbeit beschreibt den Entwurf und die Implementierung eines dynamischen Binärübersetzers, der von Grund auf zur Integration mit einem statisch übersetzten Programm konzipiert ist. Das Design des dynamischen Arm-Backed für ARANCINI wird eingehend auf die Techniken untersucht, die erforderlich sind, um statische und dynamische Übersetzungen erfolgreich zu kombinieren. Außerdem wird eine Bewertung des dynamischen Binärübersetzers vorgenommen, um seine Leistungsfähigkeit und Vollständigkeitseigenschaften hervorzuheben.

Contents

1 Introduction

A major shift is currently underway in the computing architectures used for both personal and server computing from the dominant x86-64 architecture to Arm and RISC-V. This change is motivated by a need for energy efficient and performant computing, as a result of the slowing down of Moore's Law [41, 25]. For the same reason, there is an increasing need for specialization and custom hardware development that is facilitated by the permissive licensing models of those new architectures. For instance, Arm CPUs are now widely deployed in personal computing with the Apple M-series [33], while the AWS Graviton [11], Google Axion [59] and Microsoft Cobalt [34] are steadily being expanded for use in server computing.

It is essential to support applications developed for an existing architecture on these new systems to facilitate user adoption. However, it is not straightforward to directly port existing applications to this diverse set of new architectures. This is because recompilation is required for most programs to execute on a new platform, which necessitates the availability of source code and implicitly assumes that the program makes no assumptions about the underlying system architecture [2]. In many cases, source code is not available, especially for proprietary applications or older programs. In other cases, programs extensively rely on the underlying platform and thus must be partially rewritten to execute on a different system.

Program emulation has been deployed in many existing systems to enable such applications to execute on the architecture without further modification to their code. Such an emulator is used with the Apple M-series chips for executing x86-64 programs on their underlying Arm system [3]. Furthermore, Linux supports transparently executing programs of different architectures on a given host architecture via the binfmt technology [22].

Binary translation is a powerful approach to supporting cross-architecture emulation of unmodified binaries. A *binary translator* is used to generate an equivalent sequence of host ISA instructions from a sequence of guest ISA instructions. This is done either ahead-of-time via Static Binary Translation (SBT) or dynamically via Dynamic Binary Translation (DBT) during program execution. Although widely deployed, state-of-the-art binary translators face difficulty achieving good performance while correctly emulating the semantics of the host ISA. The main challenges arise from differences in provided instructions between ISAs, the use of different memory models and the

comparative lack of program metadata in existing binaries [49, 30].

Handling concurrency, in particular, becomes difficult in view of different memory consistency models supported by different architectures, as well as the differences in the available synchronization primitives. Existing programs are known to rely on the memory model provided by the architecture [38] and thus an emulator may need to implement stronger consistency guarantees than directly provided by the host hardware. However, it is imperative that the emulation of these stronger consistency models be done efficiently to minimize the incurred overhead.

Existing binary translation techniques also make a tradeoff between completeness and performance. The generation of static translations using a Static Binary Translator (SBT) [49, 50, 18] can facilitate extensive optimizations that generate high performance code but SBTs are fundamental limited in the types of program that they can translate. On the other hand, using a Dynamic Binary Translator (DBT) [30, 26, 52] enables any program to be translated at runtime but incurs the overhead of dynamic code generation. Modern systems are exploring hardware-supported Hybrid Binary Translation (HBT) as a viable alternative for maximizing emulation performance by exploiting the strengths of both major types of binary translation.

We have designed the Arancini Hybrid Binary Translator (HBT) to efficiently and correctly perform binary translation from the x86-64 architecture without additional hardware support. This thesis describes the implementation of an Arm DBT subsystem for Arancini, which integrates with the rest of the system to enable end-to-end emulation for executing Arm programs. The goal is to provide completeness with respect to emulation when SBT was not able to statically translate parts of a program, such that a much wider variety of programs can be emulated. The core contributions of this thesis are outlined as follows:

- **Arm DBT**: a complete binary translator to Arm that can translate and execute entire programs.
- **Integration into an HBT**: the integration of a DBT into an SBT system for achieving an HBT execution mode.
- **Optimizations**: the implementation of optimizations on the generated translations to enable efficient DBT and more efficient HBT.
- **Evaluation**: the evaluation of the HBT on the Phoenix benchmark suite directly on Arm hardware, in both dynamic and hybrid execution modes.

Our translator system is available on GitHub at https://github.com/binary-translation/arancini-exploration, integrated within the main Arancini project.

2 Background

In this chapter, we provide a brief overview over the background on which our work is based. We start with a presentation of emulation as a concept and describe its implementation using binary translation. We describe the main forms of binary translators and briefly explain their relative advantages. Finally, we provide an overview of the Arm architecture, since we will heavily rely on its design and on the semantics of its ISA.

2.1 Emulation

Executing a program on an unsupported architecture can be achieved by implementing the semantics of that architecture and exposing them to the program, which then executes as if it were on that architecture. This process is called emulation and may be performed at various levels of the computing stack by a program called an emulator [51]. Its primary goal can be achieved by executing the original program internally and implementing the actions specified by its instructions atop the new architecture.

A common application of emulation is to enable execution of a program designed for a different ISA than the one supported by the system. In such a context, the terms *guest ISA* and *host ISA* refer to the ISA supported by the program and the ISA supported by the system, respectively. This process may be performed at the user-level or the system-level, where a program is executed atop the current OS or the original OS is emulated and the program executes on top of the emulated OS, respectively [51].

Implementing an emulator that performs cross-ISA emulation is possible using either interpretation or binary translation. Both approaches require translating the instructions of the program into actions on the current host system but differ in the mechanisms that they use internally. One similarity that they share, however, is that they both emulate the state of the original program via the facilities provided by the host architecture. This *emulated guest state* consists of the architectural registers defined by the *guest ISA*, as well as the state of the program's virtual address space. An emulator modifies the guest state in accordance with the semantics encoded by the instructions of the guest ISA, such that the overall behaviour of the emulated program matches its behaviour when executing natively on the guest ISA. This particular relation can be seen graphically in figure 2.1.

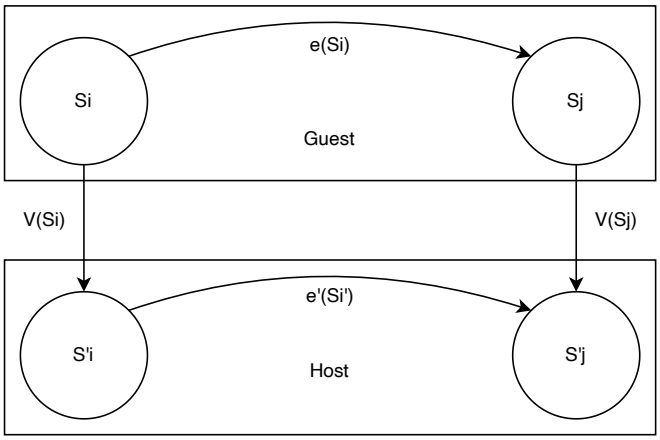


Figure 2.1: The process of emulation visualized as an isomorphism between the states of the original program and those of the emulated program [45].

Interpretation involves iterating over the guest program instructions, while performing the corresponding actions dictated by that instruction on the emulated guest state. The state is modified in the same way as in the original program but each instruction is executed by a number of routines in the interpreter code [51]. These routines perform the specified computation and ultimately update the emulated guest state.

Binary translation is the process of translating the instructions of a binary from one ISA to another ISA, while preserving the execution behaviour of the original binary. This can be achieved iteratively by generating a sequence of host instruction for each guest instruction that emulate its behaviour [51, 49, 50]. Alternatively, it may be achieved by analyzing sequences of instructions and generating a matching sequence for all of them.

Emulators can use binary translation to generate an executable translation that matches the behaviour of the original program. Such a translation would maintain an emulated guest state that it would modify as defined by the instructions of the original program to emulate it on a different architecture. Such an approach would be superficially similar to interpretation but enable much better performance, since equivalent native instructions are executed for each block of the original program instructions by the emulator. Emulators that aim at performant emulation primarily use binary translation over interpretation. Furthermore, they generally implement abstractions developed for compilers to implement extensive optimization [62, 24].

2.2 Binary Translation

Binary translation can be categorized into Static Binary Translation (SBT) and Dynamic Binary Translation (DBT), depending on whether it is performed prior to executing the emulated program or during the execution of the emulated program [51, 50, 30].

Static binary translation translates the complete program code to the guest ISA prior to emulation. The generated program can be directly executed as a native program on the host architecture with no additional dynamic code generation overhead. Furthermore, it can be optimized extensively at translation time using information available at compile time.

Despite the benefits of such an approach, static binary translation is fundamentally limited in the types of programs that can be emulated with it [32, 50, 49]. In particular, the use of register-indirect branches makes program control flow dependent on values computed during program execution. Determining such a value statically is reducible to the halting problem [48, 51], which makes SBT approaches unsuitable as a general emulation solution [49, 62]. Note that these considerations apply to programs written in existing processor ISAs, there do exist programs for other instruction sets where static approaches are not limited.

Additional factors further complicate the static analysis needed to generate static translations. One issue is determining whether an address contains program code or data, which is not possible in the general case. This issue means that it is not possible to determine whether the entirety of a program has been translated, merely that the statically determined paths have been translated. Another problem is that programs can use code generation, which requires dynamic translation of the generated code, effectively mandating the inclusion of a runtime system in statically translated programs. Combining these problems yields another problem, code generation may produce code that executes existing code that was not translated because it was deemed not reachable. The issues limit further the types of programs that can be translated statically or require additional translator complexity to handle [32, 51, 49, 62].

Dynamic binary translators translate program code to a guest ISA during emulation. This process involves parsing part of the guest instructions and generating a translation for them, which is subsequently executed. This enables the emulator to determine the next instructions to be translated for each such block, invoking the DBT for each newly discovered instruction sequence. As a result, dynamic binary translation overcomes the limitations faced by static approaches and can be used to translate any program. For instance, dynamic translation can handle programs that use code generation by retranslating the generated code on an instruction cache flush, while avoiding the issues of differentiating code from data and without requiring the entire code of the program to be discovered ahead of time. Despite these advantages, however, binary translators

must incur the overhead of performing the translation at runtime [51].

We now discuss those two types of binary translation more extensively. We also introduce the notion of Hybrid Binary Translation (HBT) that combines those concepts.

2.2.1 Static Binary Translation

The primary benefit of an SBT approach is that it can perform extensive optimization on the produced translation and can benefit from this optimized translation in all subsequent program executions. In particular, since translations are generated offline, it is generally acceptable to perform expensive static analysis and computationally-intensive optimization passes to produce an optimized translation. Subsequent runs of the translated program will benefit each time from the performed optimizations with no additional overhead [49, 30, 50].

Existing high-performance SBTs exploit this opportunity by *lifting* the instructions of a program to an Intermediate Representation (IR) that is subsequently processed by a compiler, which may perform a variety of optimization passes to generate efficient code. For instance, many systems [16, 14, 62, 50, 30] lift to LLVM IR for this purpose and rely on the extensive optimization capabilities of LLVM [36, 37]. An additional benefit of this approach is that they can utilize the facilities provided for code generation within LLVM, such as the instruction selector, register allocator and assembler. As a result, the focus on designing feasible SBTs has instead shifted to lifting a binary to an IR, designing such an IR or converting a given IR to another IR, while maintaining the correct semantics of a program [62, 50].

However, there still exist key issues in optimizing statically translated code, which often results in lower quality code compared to the original program. The core problem is primarily the lack of information regarding high-level constructs represented by binary code and overall limited metadata, which may have aided a translator to make assumptions about the program code. For instance, the presence of type information, a symbol table and debugging information can enable various optimizations on the translated code. However, the widespread use of stripped binaries means that such information is often not available [30].

2.2.2 Dynamic Binary Translation

The use of DBTs is widespread because they can provide much higher performance than interpretation while supporting a much larger set of programs than SBT approaches. They achieve these properties by operating on sequences of instructions that they parse and subsequently translate to the target ISA.

These instruction sequences are generally organized into *dynamic basic blocks*, which

are defined as a continuous interval of instructions that may be the target of multiple branches but that contain a single branch at the end [51].

This abstraction ensures that control flow is linear for a block of translated instructions, while the next dynamic basic block to be executed can be determined after the execution of a translation at runtime. As such, the entire block can be translated and optimized as a whole to generate efficient machine code, which is typically significantly faster than an interpreter.

However, note that in contrast to the static basic blocks used by compilers that have a single entry point, dynamic basic blocks are defined so that branches to any of their instructions are valid. As such, optimizers cannot eliminate or reorder instructions arbitrarily when optimizing the generated translation, in order to maintain any possible inner branch targets [51].

Additional performance gains primarily result from the use of translation caches that avoid retranslating the same blocks and the optimizations that may be performed on the generated translation [51].

In particular, modern DBTs use extensively *translation caches* that map guest program code addresses to their respective translations. When a branch to the same code is executed, the translation is first looked up in the cache and then subsequently executed if it exists. This works particularly well for loops, which execute the same code multiple times, as long as all the basic blocks of the loop fit in the cache. Note that the use of translations caches is critical for further optimization, since they amortize the cost of performing analysis and optimizations for better code generation over multiple executions of the same translation [51, 61, 19].

Optimizations are the main other driver of higher performance for DBTs, since they can produce significantly better code using information gathered from both static analysis on the source code and dynamic analysis on the emulated guest state. Some basic optimizations include those that can performed using local analysis, such as peephole optimization, dead code elimination, move elimination and so on. Other optimizations can be performed prior to generating machine code, such as vectorization and effective register allocation. There exist optimizations that can be performed on multiple basic blocks, such as the organizations of a sequence of basic blocks into a superblock that executes in one go, aiming to avoid expensive calls back to the DBT runtime. Finally, some optimizations can utilize previously-sampled profiling information to make better optimization decisions, while the emulated guest state can be used to infer properties maintained by the program to enable streamlined code generation [29, 53, 13].

Implementing optimizations is benefited by the use of an appropriate representation for the source instructions, as evidenced by the IRs used by compilers. As such, DBTs also generally include a lifting step to an IR on which they implement their analysis

and optimization logic [24, 13, 49]. However, an IR for a DBT has different design goals from an IR used in a compiler or a SBT. One major difference is that compiler IRs are designed to represent an entire program, whereas a DBT IR must be able to represent smaller portions of a program. Additionally, a DBT IR should generally be able to represent operations rather than guest instructions, allowing them to reuse the same lowering logic for multiple guest instructions that decompose into the same simpler operations.

The actual architecture translated by a DBT plays a large role, since it directly determines the design of an IR and the design of the overall emulator. In particular, Complex Instruction Set Computer (RISC) architectures feature more complex instructions that decompose internally into a sequence of lower-level operations that are combined by the logic of the instruction. They also tend to include fewer registers, such that an IR should facilitate optimizations such as native register mapping to avoid repeated memory accesses. On the other hand, Reduced Instruction Set Computer (RISC) architectures usually contain simpler instructions that decompose to relatively fewer low-level operations or to a single low-level operation and feature a higher number of registers. The IR design for a RISC architecture may omit decomposing into simpler operations and may be able to facilitate one-to-one mappings between guest and target ISAs. An IR that needs to target both architecture types may choose to represent both in terms of low-level operations for consistency, while an IR that is specialized to RISC may prefer to directly map the semantics of the RISC instruction to native instructions [24].

An example of a widely used DBT that uses such compiler techniques is QEMU [26], which translates programs dynamically at the level of a basic block by first lifting it to its own Tiny Code Generator (TCG) IR [24]. The TCG IR represents instructions within a dynamic basic block in terms of its own low-level instructions, similar to assembler instructions. This simplifies optimization, since the operations on the IR are simpler than guest instructions, while also enabling a common layer for supporting multiple guest architectures. In practice, this requires different lifters to the IR but a common optimizer that operates exclusively on that IR. Another important consideration is that the IR represents fence operations in a fine-grained manner to enable QEMU to adapt to different guest-host architecture pairs [30]. As a result, QEMU is able to translate from a wide variety of guest to target architectures[13].

2.2.3 Hybrid Binary Translation

An interesting recent development is the concept of Hybrid Binary Translation (HBT), which combines SBT and DBT into a system that uses both. Generally, it is desirable to use an SBT approach to achieve higher performance than a DBT approach. However,

since SBTs are limited, they can only be used for some programs or even just parts of a program. On the other hand, a purely DBT approach can miss optimizations for programs where significant parts can be analyzed and translated statically. The key implication here is that these capabilities are complementary, which means that a static translator could be designed to be integrated with a DBT to fill the gaps resulting from the limitations of static translation [4, 42, 60].

Such a system, depending on the distribution of static and dynamic translations, would be expected to execute faster than a purely dynamic translation. Furthermore, given sufficiently many static translations in the hot path, it would execute roughly at the speed of a purely statically translated program. On the other hand, it would be able to cope with any program, including those where code is intermixed with data and those that use code generation at runtime, since the DBT would generate dynamic translations for newly discovered code at runtime.

The only major system known to utilize this approach is Apple's Rosetta 2 [3, 4, 35], which is further aided by extensive hardware support for emulation. In particular, the Apple Rosetta 2 system is believed to utilize an SBT for translating parts of a programs code and a DBT for filling in the gaps in produced translations [42]. A core contribution of the Apple Rosetta 2 is that it relies on guarantees made by the hardware with regards to the memory model and the handling of x86-64 architectural features (e.g. flags) to generate extensive static translations. Notably, the Apple Rosetta 2 has shown performance results that are close to native execution for many programs [28] and has motivated our own work about utilizing an HBT approach without relying on additional hardware guarantees.

2.3 Arm

Our DBT work primarily focuses on the Arm architecture, which we describe briefly here. The specifics of the A64 ISA are of major interest to us, since a large part of our system design is focused on lowering to this ISA. As such, we will first briefly explore the Arm architecture and then specifically discuss the A64 ISA.

Arm represents a collection of architectures initially developed by Acorn Computers and now by Arm Holdings. The modern Arm architectures are divided into *profiles*, which are further subdivided into versions of the Arm architecture for that profile. We are primarily interested in the A-profile that powers workstations and server systems. The modern architectures representing this profile are Armv8 and Armv9, both of which support multiple execution state and ISAs [8, 7, 5]. It is important to know that a wide variety of systems are built on these architectures by many companies, since Arm sublicenses its designs and enables select customers to customize these designs [6]. As

such, our description will focus on the specification that Arm provides, rather than on a particular custom Arm system.

A key concept in Arm is the *execution state* of a system, which defines the operation of a system implementing the Armv8 or Armv9 architecture. Each execution state also defines a set of supported ISAs, as well as available extensions to these ISAs. In particular, Arm supports the AArch32 and AArch64 execution states, which roughly correspond to 32-bit and 64-bit computation on Arm. AArch32 is compatible with older versions of the Arm architecture, supporting the A32 and T32 instruction sets that operate on 32-bit word lengths [8]. AArch64, which is our primary focus, supports the A64 ISA that is widely used in modern high-performance systems.

2.3.1 The A64 ISA

The A64 ISA is the primary ISA designed by Arm for 64-bit computing and represents a major rearchitecting of the previous ISAs used on A-profile systems. In particular, many instructions have been modified to support directly computing on 64-bit values, new instructions have been added, some instructions or assembler features have been removed and a variety of extensions have been implemented to provide new instructions [8]. The general design of the A64 ISA will be explored in this subsection, since it will provide the background for our x86-64 to host mapping discussion in the implementation chapter.

The design of A64 is based on the RISC principle from which it adopts a load-store architecture with simple instructions, which operate on a large set of 64-bit general-purpose and floating-point registers, as well as immediate values encoded in the instruction itself [31, 8]. Furthermore, A64 also utilizes a fixed 32-bit length instruction encoding to facilitate quick decoding in hardware [8]. However, in contrast to other RISC architecture, it exposes condition flags for determining branches, supports a variety of complex addressing modes and optimizes particular tasks with more complex instructions [8, 31]. Similarly to many other ISAs, A64 supports multiple pseudo-instructions implemented in terms of a smaller base instruction set [20].

Arm instructions are implemented over a set of 32 of general-purpose 64-bit registers, which can be accessed with their full 64-bit width as *X*<*num*> or as their 32-bit subregister consisting of the lower-order bytes as *W*<*num*>. Additionally, the A64 ISA features support for operations on floating-point numbers and vectors via its NEON ISA, which will be discussed later [8, 10]. A key point, however, for emulation is that the available register count matches other RISC platforms and is double the number of registers provided by x86-64, which enables an emulator to map most of the x86-64 registers to native registers, while still have sufficient registers for other purposes [51].

At its basis, A64 supports data processing instructions, load-store instructions for

accessing main memory, control flow instructions, dedicated instructions for procedure calls and instructions to support system calls [8]. We now briefly describe of the instruction types supported by A64.

Data processing instructions in Arm primarily operate on general-purpose registers for integer arithmetic or logical computation. Typical operations include addition, subtraction, logical AND, logical OR and other such operations. A number of variants are provided, enabling the use of immediates for specifying operands to some operations, the setting of flags based on the operation result, the use of flags to implement operations for larger scalars and the use of shifts as part of an instructions. For instance, the `ADD` operation supports the variants shown in table 2.1.

Analyzing the variants of the `ADD` instruction in table 2.1, as well as those of closely related instructions, we can determine various types of computation that may be performed. Namely, basic additions with or without flags are supported using either 32-bit or 64-bit registers but not with a mix of the two. Furthermore, an addition may include immediates, it may shift a register using a variant of shift operands or it may extend a register to some larger size. These capabilities enable code to effect either 64-bit and 32-bit computation but also use 8-bit or 16-bit operands by extending them to a 32-bit or 64-bit value. Additionally, support is provided for chaining additions via the carry flag, which may enable additions on operands larger than 64-bits, as long as multiple registers are used to store them. Finally, the implementation of both `ADC` and `ADCS` flexibility in choosing whether to effect addition over a chain longer than two registers and whether to set flags [8]. Other data processing instructions largely implement either the same functionarity or a subset of it. As we will see, these capabilities are critical to emulating ArancinIR used in our emulator.

An important point here is that Arm usually supports immediate variants for some instructions, such that smaller values can directly be encoded in the instruction rather than requiring an additional register. This benefits code generation because smaller constants from the ArancinIR can directly be materializes as immediates but it introduces some additional complications. In particular, the supported immediate size depends on the instruction but is generally limited to a 12-bit value. If a large value is used for an operation it must be loaded to a register but loading to a register with the `MOV` instruction is also limited to 12-bits. As such, a sequence of `MOVZ` and `MOVK` instructions are needed to properly load 64-bit immediates in Arm [8]. In contrast, x86-64 supports 64-bit immediates for many instructions and generally does not require such instruction sequences to load immediates into registers.

Load-store instructions facilitate accessing memory at various granularities, including 8-bit, 16-bit, 32-bit and 64-bit in the base A64 ISA without considering NEON. Additional optimized instructions are provided for loading and storing two 32-bit or 64-bit registers at a time, which enables efficient implementation of pushing registers to

Code	Semantics	Flags?
`ADD <Xd|WSP>, <Xn|WSP>, <Wm>, <extend> #<amount>`	Addition with extended 64-bit register	No
`ADD <Wd|WSP>, <Wn|WSP>, <Wm>, <extend> #<amount>`	Addition with extended 32-bit register	No
`ADD <Wd|WSP>, <Wn|WSP>, #<imm>, <shift>`	Addition with immediate and 32-bit register	No
`ADD <Xd|SP>, <Xn|SP>, #<imm>, <shift>`	Addition with immediate and 64-bit register	No
`ADD <Wd>, <Wn>, <Wm>, <shift> #<amount>`	Addition between 32-bit register and shifted 32-bit register	No
`ADD <Xd>, <Xn>, <Xm>, <shift> #<amount>`	Addition between 64-bit register and shifted 64-bit register	No
`ADDS <Wd>, <Wn|WSP>, <Wm>, <extend> #<amount>`	Addition with extended 32-bit register	Yes
`ADDS <Xd>, <Xn|SP>, <R><m>, <extend> #<amount>`	Addition with extended 64-bit register	Yes
`ADDS <Wd>, <Wn|WSP>, #<imm>, <shift>`	Addition with immediate and 32-bit register	Yes
`ADDS <Xd>, <Xn|SP>, #<imm>, <shift>`	Addition with immediate and 32-bit register	Yes
`ADDS <Wd>, <Wn>, <Wm>, <shift> #<amount>`	Addition between 32-bit register and shifted 32-bit register	Yes
`ADDS <Xd>, <Xn>, <Xm>, <shift> #<amount>`	Addition between 64-bit register and shifted 64-bit register	Yes
`ADC <Wd>, <Wn>, <Wm>`	Addition using current carry flag between 32-bit registers	No
`ADC <Xd>, <Xn>, <Xm>`	Addition using current carry flag between 64-bit registers	No
`ADCS <Wd>, <Wn>, <Wm>`	Addition using current carry flag between 32-bit registers	Yes
`ADCS <Xd>, <Xn>, <Xm>`	Addition using current carry flag between 64-bit registers	Yes

Table 2.1: Showcase of the different instruction forms for the ADD instruction and similar instructions [8].

Name	Addressing mode	Explanation
Base register	`[base, #0]`	Use base register as address
Base plus offset	`[base, #imm]`	Use base register + immediate offset
Base plus offset	`[base, Xm, LSL #imm]`	Use base register + shifted register as offset
Base plus offset	`[base, Wm, (S|U)XT(X|W) #imm]`	Use base register + extended register as offset
Pre-indexed	`[base, #imm]!`	Address using base and offset and store address to base
Post-indexed	`[base], #imm`	Access using base and then update base with offset
PC-relative	`label`	Access relative to the current PC

Table 2.2: Showcase of the different addressing modes for load-store operations [8].

the stack as required by the Application Binary Interface (ABI) [8]. In terms of memory operands, the A64 ISA supports a variety of addressing modes that define a 64-bit address for the 64-bit virtual address space defined by the architecture. A summary of the supported addressing modes is shown in table 2.2.

Finally, we will discuss control flow instructions and instructions relating to function calls. Starting with control flow, the A64 ISA supports uncoditional branches via the `B` instruction, conditional branches via multiple instruction variants, such as `BEQ` that implicitly depend on the `NZCV` register that stores flags and conditional branches that also perform comparisons prior to branching via `CBZ` and `CBNZ` [8]. Function calls are handled via the `BLR` instruction which uses a link register to store the return address. All of these instructions, as well as other similar instructions, are widely used in Arancini when generating code.

2.3.2 Memory Model

The Arm ISA uses a weakly-ordered memory model that partially decouples the order of program instructions from the visible order of reads and writes. An important aspect of the memory model is that it depends on observers, which in Arm are defined as *Processing Element (PE)* that usually correspond to cores [8].

In Arm reads and writes may be reordered when they occur on different execution units and even within the same PE when there exist no data dependencies. However, the memory model does ensure that data dependencies are executed in order on the same

$X = Y = 0$ (at initialization)

thread 1	thread 2
$X = 1$	if $(Y == 1)$
$Y = 1$	`assert`$(X == 1)$

Figure 2.2: Example interleavings between threads on weak memory architectures [30].

PE [8, 30]. Furthermore, a particular order is also guaranteed when synchronization instructions are used.

This model is chosen primarily to facilitate optimizations with the explicit goal of increasing performance. This is in part an opportunity presented to the hardware designers but also an additional responsibility for programs to use appropriate synchronization mechanisms when accessing data from multiple threads [8, 12, 49, 30, 58, 31].

An example of the effects of the memory model on a program is shown in figure 2.2, where the possible reorderings performed by the architecture differ significantly from what a programmer may expect. Namely, the assert on the second thread may be triggered on the Arm weak memory model, since writes are not necessarily visible to other PEs ordered in program order. A direct possible optimization arising from this is write reordering in hardware to minimize store latency. A programmer used to the x86-64 Total Store Order (TSO) memory model would not expect such a result and would need to account for such possibilities when programming a weak memory system.

Emulators are particularly concerned with memory model semantics because they must implement the requirements of the guest memory model in terms of the host memory model. This is significantly more difficult when emulating stronger memory models on weaker memory models. For instance, it is significantly harder to emulate the x86-64 memory model on Arm, since the former forbids orderings that may happen on the other in a valid translation. Handling such cases relies on the use of `fence` instructions to ensure that particular reorderings do not occur in the translation [30, 49]. However, it is necessary to include memory model considerations at the level of the IR and the lifting process, such that binary translation can determine when fences must be placed during the lowering of the IR [30].

Besides fences, an important part of code generation affected by the memory model is the use of atomic instructions. The Arm architecture provides support for atomics

but the availability of atomic instructions depends on the extensions to the ISA. At a base level, Arm provides the `LDXR` and `STXR` instructions for atomic loads and stores, which mark an address as exclusive during their operation. In particular, `STXR` enables atomic writes using a status register that denotes whether the write was indeed atomic by verifying whether the address was still marked as exclusive [8].

At a higher-level, Arm also provides via its Large System Extensions (LSE) support for a variety of atomic operations, such as atomic additions, atomic logical AND and atomic Compare-And-Swap (CAS) [8]. The availability of these instructions presents an opportunity to generate optimized atomic code that uses them, but notably all of these operations can be implemented even without LSE support via the `LDXR` and `STXR` instructions. This is particularly important, given the broader range of host Arm systems that our emulator needs to support.

2.3.3 SIMD extensions

The Arm platform includes SIMD support for A-profile and R-profile architectures via the NEON extension, as well as vector processing on the A-profile via the Scalable Vector Extension (SVE) and Scalable Vector Extension 2 (SVE2) [10, 9]. We will briefly discuss them here, since emulation of vectorized programs must take into account existing vector execution facilities on the host for code generation purposes.

We first start with the base facilities for implementing vectors in Arm, which are provided by the existing scalar instructions. Namely, using multiple general-purpose registers, it is possible to implement any vector operation by repeatedly applying the each scalar operation on the registers associated with the vector. This approach enables a large amount of flexibility but does not benefit from the speedup of using actual vector registers, as provided by NEON or SVE. On the other hand, there are operations that are not directly supported by NEON or SVE and implementing them via scalar instructions may be simpler.

The next level of support, included with the A64 ISA, are the NEON extensions to Arm that provide 32 registers that contain 128-bit vectors with elements of 8-bit, 16-bit, 32-bit or 64-bit size. The NEON extensions additionally introduce support for a number of floating-point operations by denoting 32-bit and 64-bit subregisters of the NEON registers that can hold floating-point data. As such, a NEON register may be denoted as `q<num>.<size>`, while registers `d<num>` and `s<num>` are 64-bit and 32-bit subregisters of the `q<num>` vector register. The A64 ISA supports floating-point operations using the floating-point subregisters and vector operations directly using the NEON vector register. It is possible to implement arbitrary vector operations on floating-point values by converting them implicitly to vectors and executing the provided operations on them. The actual operations supported on NEON vectors are largely similar to those

provided on scalar operations [8, 10].

Finally, the SVE and SVE2 extensions provide support for larger vectors and more complex vector operations. A notable feature of SVE and SVE2, compared to other vector ISAs, is that they do not define an architectural vector size and support instead variable-length vectors. The actual vector size supported must include 128-bit vectors but may be larger and it is defined by the platform. Instead the SVE and SVE2 extensions provide mechanisms to query vector sizes at runtime and support operations with variable vector widths for vector computation. The actual vector operations supported include largely the same operations as available on NEON but also additional, more complex, operations that facilitate numeric computation and complex data movement [8, 9]. A key limitation of SVE and SVE2 is that support for them is missing on many platforms, such that they cannot be relied upon to be present and fallback routines must be available for NEON or for scalar code.

3 Overview

In this thesis, we present the design of an Arm DBT that dynamic generates translations for guest programs instructions via the ArancinIR, as part of the wider Arancini HBT. The Arm DBT is implemented to completely support the semantics of ArancinIR and it is integrated to be invoked when there exists no static translation for a part of a program. It is also designed to integrate with a runtime system that can integrate both static and dynamic translations to achieve efficient program emulation.

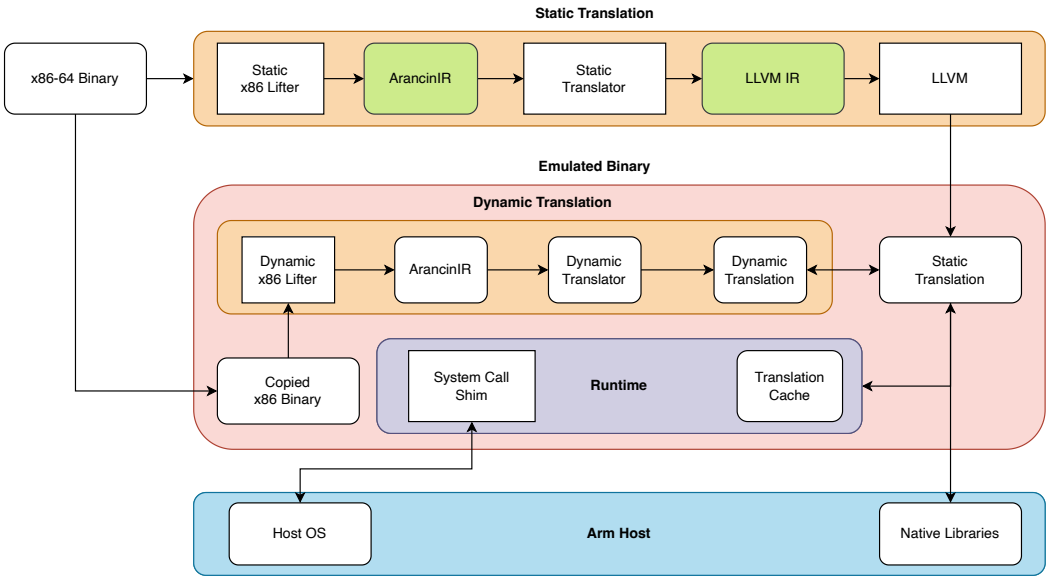


Figure 3.1: Overview of Arancini architecture.

In figure 3.1, we present the high-level view of Arancini. The main components of Arancini include a static translator, a dynamic translator, a runtime and a common lifter from x86-64. We will explore each component separately in the Design chapter of this work.

In figure 3.2, the DBT is presented in terms of its high-level architecture and integration with other components in the wider Arancini. A translation is achieved by lifting

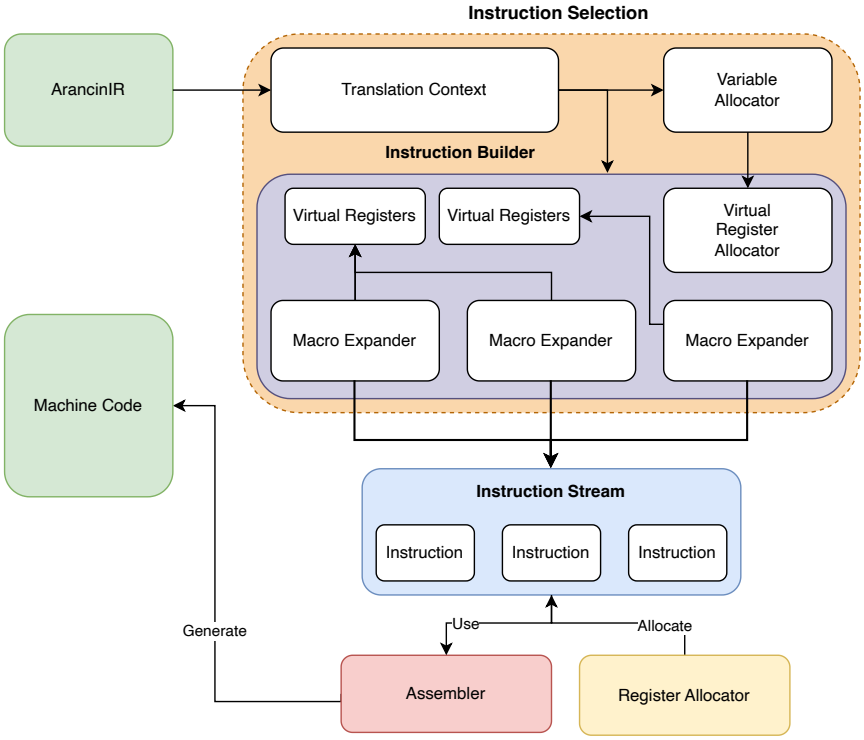


Figure 3.2: Overview of the DBT architecture.

guest instructions into an ArancinIR form, parsing the ArancinIR and then generating code for each node in the ArancinIR. An important consideration is that the ArancinIR forms the contract between the runtime and the DBT, enabling the DBT to generate code from an abstracted view over guest instructions. We implement this process in our work and integrate it with the Arancini runtime.

Additionally, we explore optimizations that can be performed at the level of a DBT and implement them. We also orchestrate the runtime and the DBT to perform optimizations taking multiple translations at once and combining them to produce faster code.

Finally, we analyze and benchmark our dynamic translation logic, as well as the optimizations that we introduced. We also analyze the effectiveness of our integration logic.

4 Design

This chapter explores the design specifics of the Arancini binary translator and the Arm DBT, as well as its integration into the wider HBT system. We initially present the design of ArancinIR, which is the common representation produced by our frontend and the delve into the design of each backend.

4.1 ArancinIR

The key component of our system is ArancinIR that provides the interface between static and binary translation, as well as the lifting target from the instructions of the guest program. The use of a common IR is the main innovation in Arancini that enables the design of multiple backends of different types that can interface together, while being independent of the particular guest architecture that they are emulating.

This means that ArancinIR needs to be able to correctly express the semantics of a program defined by a guest ISA, while mapping efficiently via different mechanisms to a variety of target architectures. Furthermore, ArancinIR needs to be constructed efficiently and facilitate lowering to another representation, while encoding sufficient information to enable efficient code generation.

However, it is generally difficult to achieve these goals with an IR based on Single Static Assignment (SSA) and ArancinIR takes different design decisions that many other existing IRs. In particular, note that to facilitate lowering to LLVM IR for a static backend requires encoding high-level information that enables the LLVM optimizer to generate efficient code. On the other hand, lowering to native code at runtime requires defining low-level operations that can be efficiently mapped to host instructions, while high-level information may provide no benefit. These two scenarios reveal that ArancinIR faces an apparent contradiction in needing to support both high-level and low-level program representation simultaneously.

The design of ArancinIR attempts to overcome this contradiction using a node-based representation of the program instructions that include both a low-level view of their constituent operations and a high-level data flow view of their order in the original program. Specifically, ArancinIR decomposes each instruction into lower-level operations that can be efficiently mapped to host instructions and higher-level graph structures that can be lifted to LLVM IR for static translation.

4.1.1 Structure

ArancinIR is constructed from nodes of different types that representing low-level program operations. Each general type of operation has a distinct node type that includes both the required inputs for that operation and the produced outputs. Additionally, each type corresponds to one of two categories: *value nodes* and *action nodes*. We show a high-level view of ArancinIR in figure 4.1 and proceed to dive deeper into its structure.

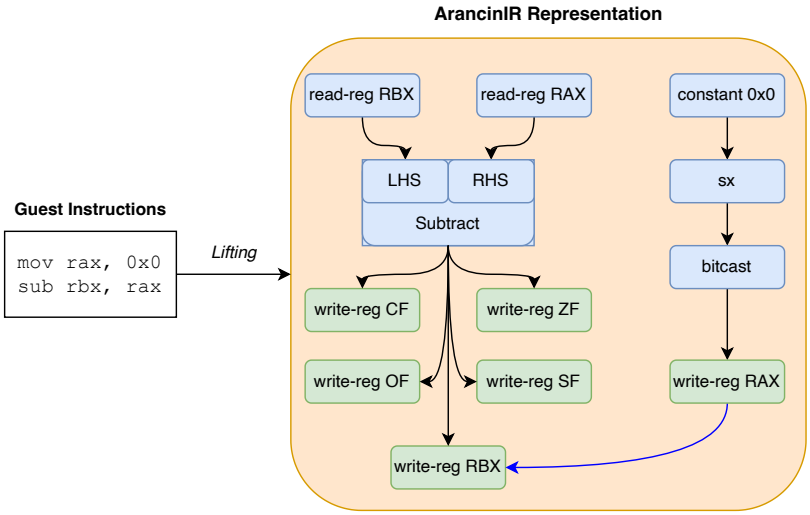


Figure 4.1: High-level view of ArancinIR that represents a single instruction.

Value nodes represent operations that produce one or more values which are a dependency of another value or action node, while themselves relying on zero or more input values. ArancinIR uses value nodes to denote computation performed within an instruction, such as arithmetic or logical operations. On the other hand, value nodes do not update architectural state and their outputs are not directly visible during emulation. In practice, values nodes represent low-level operations, such as binary arithmetic operations, reading memory or performing bit manipulation. Representing only low-level operations in the IR simplifies mapping them efficiently at runtime to native constructs. As such, each guest instruction is decomposed during lifting into its constituent operations primarily represented by value nodes.

Action nodes represent operations that modify the guest architectural state, such that they denote visible side-effects of program emulation. Generally, operations such as register writes to guest registers or memory writes are encoded as action nodes. Since such operations modify the guest state, it is imperative that they be executed in program

order to ensure correct emulation. An important detail here is that write register nodes abstract the particular register referenced in terms of an offset, while also providing the actual register name. This enables a translator to completely abstract guest register handling from a particular register but also provides the means to optimize using the register name, if needed.

Nodes are connected together via edges to form a *packet*, which is a Directed Acyclic Graph (DAG) that represents a single guest instruction. The edges between nodes generally represent data flow dependencies, except for edges between action nodes that represent ordering constraints instead. Each edge is associated with a *port*, which denotes the interpretation of that particular edge. For instance, every value node includes a port that represents the output of the operation described by that node. However, arithmetic nodes may include additional ports for 1-bit flags that represent numeric characteristics of their result. Maintaining such information is critical for lowering, since it provides a structured way to reference dependencies between nodes.

This organization effectively represents a guest instruction with multiple low-level IR operations. Two important properties arise from this organization:

1. *No action node may have an outgoing edge*
2. *Each action node represents a data flow dependency of one or more action nodes*

The first property arises from the definition of action nodes as representing architectural state modifications with no outputs of their own. The second property arises from the fact that action nodes produce no visible side-effects as part of their execution and thus may only exist if some action node depends on them.

Finally, we also define the notion of a *chunk* that represents a linear sequence of packets. The use of chunks as an abstraction enables us to reason about ArancinIR differently depending on our lowering target, since chunks can represent both entire functions or dynamic basic blocks. As such, we do not have to commit to any particular organization of the instruction stream and can offload such decisions to individual backends.

4.1.2 Program Representation

We will proceed to briefly discuss aspects relating to the representation of a program within the general structure of ArancinIR. In particular, we will explore the encoding of the emulated guest state and the encoding of control flow.

The emulated guest state in Arancini is defined as the state of the architectural registers and the state of memory. This means that we include all general-purpose registers (GPRs), all floating-point registers (FPRs), the architectural flag register (`rflags` in

x86-64), the status registers and the vector registers defined by the various extensions to x86-64. In terms of memory, we only include memory that is accessed by the program in any way, including for reading, writing or executing. An important design decision here is that ArancinIR does not impose specific storage or internal handling for this state. Furthermore, ArancinIR is completely decoupled from the actual mapping of this state to the facilities provided by a host architecture, such that it does not depend on the number of available architectural registers and other such host-dependent details. As a result, this mapping and overall handling of this state is left up to each individual translator backend.

The handling of control flow is another important part of program representation, since ArancinIR differs significantly from SSA-based IRs in tracking control flow explicitly. In particular, ArancinIR needs to handle both control flow changes as part of program execution, as well as control flow internal to instructions. We denote control flow determined from control flow instructions in the guest program as *external control flow* and we model it using writes to the emulated guest Program Counter (PC). This means that ArancinIR handles changes in external control flow as simple register writes, while dependencies between instructions remain at the level of the emulated guest state with no additional handling by Arancini. On the other hand, we also distinguish *internal control flow* that denotes control flow within a packet as it results from control flow semantics encoded by a guest instruction. This is important because our guest architecture is x86-64, which supports looping instructions explicitly and defines other instructions as having implicit control flow internally in their implementation. In such cases, the instruction packet is not represented by a DAG anymore but by a graph instead.

The decision to model internal control flow as a graph aids in directly lowering to a target architecture. Specifically, internal control flow can be lowered as a loop, a conditional or any combinations of the two using host instructions that change the control flow. On the other hand, it makes other processing, such as register allocation, more inefficient because dynamic analysis needs to operate on a graph instead of a DAG. However, given that our goal is a complete DBT with a heavy reliance on the use of static code, it is a tradeoff that has optimized simple lowering with acceptable performance.

4.2 Hybrid Translation

The use of a common IR is the first step in enabling hybrid translation in Arancini but a number of other components are needed to make this approach feasible. This section will briefly describe these components, starting with the binary format that

represents a translated program and proceeding to our runtime component that ties static to dynamic translation. We will also discuss issues pertaining to interacting with a host OS and the mechanisms that we use to map the OS interface expected by our guest program to the capabilities provided by our host.

4.2.1 Binary Format

The binary format used in Arancini represents the structure in which the translated code is represented as an executable binary. Such a structure would be similar to the structure of typical programs and libraries on a system but must be adapted to include a copy of the guest program code, as well as the runtime code.

Before delving deeper into the particular format used, we elaborate further on the need for the guest program code. Specifically, we need the original program because we assume that the static translation is incomplete and there may exist portions of the program that must still be translated. Particular cases where this could happen are detailed below:

1. Statically unreachable code that is accessed by self-modifying code
2. A branch in the middle of a discovered instruction
3. Data treated as a code during execution
4. A program may parse its own instructions

Each one of the above cases would require the dynamic translator to access the original program to perform its translation. Since we cannot be certain that all of the program code has been discovered, we cannot include only part of the original program and must include all of it. In particular, we cannot even exclude already translated code portions since a branch to the middle of an instruction would reveal previously unexplored code in the already statically explored code paths.

As such, the key is to determine a binary format that can store multiple code regions of different architectures. Although there exist widely used formats for each architecture and OS, no existing binary format supports doing so directly, such that Arancini must use its own binary format. The resulting binary format for Arancini is the Executable and Linkable Format (ELF) that is widely used on Linux and other UNIX-based OS, adapted for the particular of Arancini.

We will now briefly describe the generation and layout of such a binary with a focus on the modifications made to ELF. First of all, we use the standard sections defined by ELF to include our translation, the runtime and the data. We map each section from the original program, assumed to be encoded as ELF also, into specific sections mirroring those of the guest. Each such guest section is mapped to the same address as in the

original guest program to avoid the need for further mappings when the translation accesses them. This is an important concern for accesses that use absolute addresses, as well as for programs that explicitly parse their own sections, such that we must ensure that they read valid data.

The major addition to the binary format that enables static translation is the use of a *static function map*. The static function map is used to associate each translated symbol to the original guest code that it represents. When an access is made to any symbol, the static function map enables the translation to determine where the original symbol is located in memory. We place this static function map in the `.rodata` section to ensure that we load it in main memory prior to program execution.

Finally, we also include metadata for all the translated symbols, for the relocations needed to handle shared objects and for the newly added ELF sections. The resulting binary is either an executable or a library, depending on the type of the original binary. This is an important aspect that enables Arancini to handle both statically linked and dynamically linked programs.

4.2.2 Runtime

The runtime is the component included in the hybrid binary for providing the interface between the static and dynamic translations, as well as the interaction with the host. In particular, the runtime is tasked with initializing the data structures of the emulator, initializing the emulated guest state and providing runtime support for handling interactions with the host OS, including the system call interface and support for multithreading.

We will briefly discuss the initialization of the guest state prior to the host interface that comprises the core of the runtime. Initializing the guest state involves initializing the guest registers and initializing the memory of the guest. Since initializing the guest registers simply involves either setting them to zero or to some expected value, we will directly discuss guest memory initialization that is more complex.

Initializing guest memory involves setting up a stack, the environment variable, the auxiliary vector, the thread runtime and performing dynamic linking. In Arancini, the guest stack is separate from the stack used by the runtime and the dynamic translator to ensure that the guest instruction accesses relative to the emulated stack pointer remain valid without clobbering from the runtime. This involves allocating the stack as a region of memory and updating the guest stack pointer, while also applying any changes to the stack to the stack region of memory. Note that this means that the guest stack is likely stored at a different address than it would have been when executing natively but programs do not depend on the actual stack base address in practice, so this is not a concern.

Setting up the environment and the auxiliary vector enables the program to access environment variables or OS configuration, while also enabling the dynamic linker to perform its operations. As such, we set it up by allocating memory for them and copying the current environment, while setting up the auxiliary vector to match the guest architecture. Note that the environment may be further manipulated by the program runtime, but this generally is left up to the translated code.

The handling of threads is similar, since it consists in setting up the Thread-Local Storage (TLS) and a thread control block. Further threads are spawned by the translating as part of its execution via a call to the runtime, which then manages each thread spawned by the program.

4.2.3 Interacting with the Host

After discussing the overall setup performed by the runtime, we proceed to discuss the interface that the runtime provides for interacting with the host. The runtime must provide a basic layer to abstract the facilities of the host from the translation, such that it can ensure that they match the semantics of the emulated program's interaction with the OS. However, note that Arancini does not emulate the host OS per se, it merely translates between the interfaces expected by the guest program and the one actually provided on the system.

This means that the runtime must provide support for invoking system calls and delivering signals to programs. The first is provided by a *system call shim* and the later is provided by runtime-registered signal handlers. We will discuss the system call shim layer and then briefly elaborate further on signal handling.

The system call shim layer is a target-specific component that translates systems call codes of the target architecture to actions executed by Arancini. The precise handling depends on the system call but largely two types of handlers exist, those that emulate a system call in terms of one or more system calls on the native system and those that only update the internal emulator state. Generally, system calls such as *openat()* will be invoked on the host architecture by the handler with the arguments provided by the emulated guest state based on the x86-64 SystemV calling convention [40] and the return will be written to the same emulated guest state. Minor changes to these arguments may be needed to guarantee the security of the emulator or a translation of the arguments may be required because their numeric values differ between systems. On the other hand, some system calls cannot be directly executed because they would either return host information to the emulated program or because they would compromise the security of the emulator.

For instance, the *brk()* system call used as part of memory allocation cannot directly proceed because the emulator manages all the memory allocated for the guest internally.

As such, the *brk()* system call simply updates state internal to the emulator and then updates the return value as if the system call had been executed.

We now briefly discuss the handling of signals from the host OS to the guest program. The primary difference here is that the emulated program can only register signal handlers but the actual signal handler is invoked by the OS. We facilitate this via the system call shim by enabling guest programs to register signal handlers for most signals, while our system call shim translates the addresses of these handlers to addresses in the translated code when they exist. In effect, the host OS will directly invoke the translation without further processing when a translating exist, while otherwise invoking untranslated code that will be translated dynamically.

However, note that some signal handlers are not handled in this manner because the guest program cannot be allowed to specify its own signal handling. For instance, the emulator catches `SIGSEG` signals representing a segmentation fault and performs its own processing to display diagnostic information. We currently do not support programs that register such handlers and immediately exit with an error. However, note that it is trivial to extend our emulator to support them by registering guest program signal handlers as being invoked from our own signal handlers.

4.3 Static Translation

A defining component Arancini is the LLVM-based static binary translator that makes us part of hybrid binary translation. This static translator analyzes the guest program to discover guest instructions that it then lifts to ArancinIR, which ultimately is lowered to LLVM IR and finally to machine code.

The processing performed by static translation is aimed at covering the instructions of a guest program as extensively as possible with the goal of translating statically as much as possible. Furthermore, the static translator employs a specialized lowering step that attempts to effectively map ArancinIR to LLVM IR in such a way that LLVM can perform its extensive sets of optimizations.

We will briefly discuss the two main parts of this process: lifting to ArancinIR and lowering to LLVM IR. We will not discuss the actual optimizations performed within LLVM, since we can largely treat them as transparent.

4.3.1 Lifting to ArancinIR

The first step in converting the program to an ArancinIR representation is discovering the instructions of the program. This general process depends on the binary format used and the encoding of the program in terms of that binary format, since the presence of extensive metadata can guide the discovery of program code. In the case of Arancini,

we extensively rely on the symbol table and the relocation sections in the guest ELF to discover program code.

In particular, the static translator iterates over the symbol table and performs a translation for the address referenced by each symbol. Since symbols largely correspond to functions, the static translator is likely to process a large number of functions defined by the guest binary. For external functions, only the declaration is inserted in the relocation section, while the linking support present in Arancini will supply the definition from another translation (or native) binary.

When processing each symbol, we can determine the number of instructions defined by it using its size and we iterate over these instructions to lift them to ArancinIR. Note that when the symbol size is provided as zero in the symbol table, we do not process the symbol, even if it may include instructions. For each instruction, we utilize the XED library to disassemble it and then utilize a frontend lifter to convert it to ArancinIR. The lifted instructions will be processed exclusively as ArancinIR from this point onwards.

Once we have generated an ArancinIR representation, the static translator performs some preliminary optimizations. The primary such optimization is *dead flag elimination*, which determines whether flags generated by various IR operations are overwritten without intervening reads and eliminates all such updates. This particular optimization can be performed using information from the guest architecture and it is an optimization that LLVM is unlikely to make, so it is desirable as a preliminary step.

4.3.2 Lifting to LLVM IR

We will now briefly describe the core of the static translator that deals with lifting ArancinIR to LLVM IR. This particular stage maps ArancinIR ports to LLVM IR values, nodes to LLVM IR instructions and chunks to functions. As part of this mapping, functions are created to load guest registers in their preamble, while the body of the function consists of translated instructions and the epilogue includes saving modified registers back to memory. Additional handling must be included to handle tail calls or jump tables within a function but those are also statically translated.

Once all chunks have been translated, a special `MainLoop` function is created to handle the execution of the statically translated code. The `MainLoop` determines which translated function to invoke at runtime or directly calls into the runtime, which will provide a dynamic translation, when no suitable static translation exists.

A number of other considerations apply to static translation but we avoid further discussion, since the main point of this work is the design of the dynamic translator.

4.4 Dynamic Translation

This section will describe the design of the Arm DBT, which is the main contribution of this work.

The Arm DBT is the component within Arancini that is responsible for handling code that cannot be statically translated by translating dynamically at runtime to native Arm code. As such, its design differs significantly to the Arancini static translator because its primary two goals are completeness and fast translation. Most importantly, it must support hybrid translation in conjunction with the static translator.

First of all, the DBT must be complete to be able to handle any instruction that may arise during dynamic translation. In particular, any instruction that cannot be translated statically or any code that is not cannot be reached via static analysis must be supported by DBT. Furthermore, the DBT must be able to handle programs that generate or modify code at runtime.

Second of all, the DBT must support fast translation to achieve effective emulation. In particular, the DBT operates exclusively at runtime and the speed at which it generates translations constitutes a part of the total execution time of the program.

The Arancini DBT includes two main parts: *lifting* to ArancinIR and *lowering* to host instructions. We explore both parts now in detail.

4.4.1 Lifting to ArancinIR

The Arancini DBT lifts x86-64 instructions to the ArancinIR at runtime in a similar manner to the static lifting performed by the Arancini SBT in Section 4.3.2. Specifically, lifting is performed at the level of a basic block from a starting address, rather than a function from the location of a symbol. Once an untranslated basic block is discovered, Arancini iterates over its instructions until branching control flow or special conditions have been discovered, which denotes the end of that basic block.

This process produces a *chunk* that corresponds to that basic block. The DBT is tasked only with translating this *chunk*, such that the runtime can execute it to determine the next starting address that the program will execute. A chunk may be terminated with a branch or due to a number of special conditions, such as the invocation of a system call.

The important aspect of lifting for the DBT is that the binary translator is invoked at the beginning of a chunk and before each individual instruction in a chunk, as well as the end of each instruction and after the entire chunk has been passed. Arancini only requires that the translation be completed after lifting is completed but lifting can be intermixed with translation, depending on the backend.

In practice, the backend only notes which nodes to materialize for each instruction

and once an instruction is completed, preliminary steps are taken to lower that instruction. However, the Arm translator only produces a complete machine code translation after an entire chunk has been lifted.

4.4.2 Lowering to Arm

We will now describe the steps involved in lowering the lifted ArancinIR to Arm instructions within the Arm DBT. The Arm DBT is internally structured similar to a compiler backend and consists of multiple phases: instruction selection, register allocation and assembly.

Our description will focus on each stage individually, since each plays an important role in lowering to machine code. However, we will underline at a high-level that the Arm DBT lowers ArancinIR directly to generate machine code with no intermediate change to another IR or any such other representation.

4.4.3 Instruction Selection

The Arm backend achieves emulation by interpreting the ArancinIR chunk that it receives to generate sequences of A64 instructions that are semantically equivalent to the operations specified in the IR. The process of determining which instructions must be mapped to an operation from ArancinIR is called *instruction selection*.

Instruction selection in the Arm backend largely follows the design of other DBT systems by relying on *macro expansion* [15]. Macro expansion refers to the process of selecting instructions by matching operations to a number of fixed host instruction patterns. The use of macro expansion is widespread in DBT design due to its simplicity and ability to generate code fast, given a reasonably sized number of patterns. On the other hand, macro expansion is known to generate inefficient code without further optimizations. We will describe the use of macro expansion in the Arm DBT and then proceed to discuss its characteristics applied to ArancinIR.

We implement instruction selection using an *instruction builder* object, which provides a set of high-level operations on variables. These high-level operations correspond to expanding macros to perform some high-level operation to which we map each ArancinIR operation. Internally, each high-level operation appends one or multiple A64 instructions to an instruction stream. An important detail here is that the instruction stream uses virtual registers, register allocation happens only after instruction selection has completed.

Our approach to instruction selection emphasizes using features available on the system to generate optimized code. For instance, we utilize flag-setting operations on Arm to compute the flags in the emulated guest state. Furthermore, we utilize

Data Type	Data Representation
Scalar	variable = scalar(vreg)
Large Scalar	variable = scalar([vreg, vreg, ...])
Vector-backed Vector	variable = scalar(vector-vreg)
Simulated Vector	variable = [scalar([vreg, vreg]), scalar([vreg, vreg]), ...]

Table 4.1: Special registers maintained by the Arm DBT and associated values stored.

shifting and extending variants to effect computation on values smaller than 32-bits. We also liberally utilize 32-bit and 64-bit variants of instructions, while also directly using immediates specified by ArancinIR whenever possible. We generally treat scalar operations, including those on larger scalar values, different from vector operations by handling them in different instruction builder methods. This approach has enabled us to keep the logic of the backend well structured and to clearly define the separation of the various stages in our backend.

Important aspects of our approach to instruction selection relate to our use of variables, which provide a strongly typed mechanism to encode data flow dependencies at a high-level. In particular, variables are variant types capable of representing all units of data that the backend references and mutates. Specifically, they can represent one of the types shown in table 4.1.

In principle, ArancinIR exposes scalars and vectors as data types. However, some scalars may be too large to map directly to the registers provided by the A64 ISA, resulting in a secondary category of large scalars. Similarly, although the A64 ISA supports vectors via the NEON and SVE/SVE2 extensions, there may be no direct mapping for vectors provided by ArancinIR to native vectors, such that we need simulated vectors.

The instructions generated by the Arm backend differs for each type and we divide handling between scalars and vectors as general categories. We find this organization effective because vectors encode SIMD operations that are logically independent, while scalars always must be a single unit of computation even if multiple instructions are needed. Indeed, even operations on large scalars must behave as if the large scalar could have been mapped directly as a register. On the other hand, a real vector is just a special case of a simulated vector and can be generated as an optimization. In particular, we cannot assume support for SVE2 and NEON vectors are limited to 128-bits, which makes simulated vectors a necessity.

It must be noted in our discussions on the supported data types that it is particularly a mismatch between the host and target architectures that require these abstractions. Namely, x86-64 supports much wider vectors in practice and can also express com-

Register Name	Value Stored
x29	Context Block base address register
x30	Link Address register
x0	Return value register

Table 4.2: Special registers maintained by the Arm DBT and associated values stored.

putation larger than 64-bits, whereas the A64 ISA does not directly provide such capabilities.

The design of the instruction builder is focused around manipulating variables, scalars or vectors. We support this by designing these data types to be convertible to and from each other, enabling us to define per data type high-level logic. Each operation on the data type is a high-level operation, such as multiplication, complement-of-2, conditional selections and so forth. Each such operation often maps to a single instruction for scalars and real vectors but typically to multiple instructions for large scalars and simulated vectors. A key design consideration is that we want these operations to exist independently of the IR, such that they can be used further to enable multiple other high-level operations that the IR operations do not facilitate.

Internally, the instruction builder must allocate non-trivial state to correctly track code generation via these operations. We need to maintain a virtual register allocator to enable instructions to refer to as many registers as they need, while delaying register allocation as a separate step. We also store tracking information for labels to facilitate determining liveness as part of register allocation, as well as to determine if any branch label is unmatched by an actual label. We maintain references to physical registers that we use for referencing the x86-64 state, which is stored at a fixed base address in main memory, and for the return value from the emulator. We store the capabilities provided by the system, since we need to know what code we can generator our high-level operations. Finally, we store an assembler object, which provides the ability to convert our assembly instructions in machine code.

An important part of the state of the instruction builder are the references to the physical registers that it maintains. These registers provide the primary interface between the translation and the x86-64 state that we modify, as well as the necessary information to facilitate a transition to and from the runtime. They are described succinctly in figure 4.2.

We describe the use of the registers from figure 4.2 within instruction selection in more detail. First of all, the context block base address register is used for every access to the emulated x86-64 state during instruction selection. The x86-64 state is configured by the runtime at the specified base address and the ArancinIR nodes only provide

```
allocations = {}
for instr in reverse order:
    for op in instr.operands():
        if op.is_def() & !op.is_use():
            if op in allocations:
                allocations[op].pop()
            else:
                kill(instr)
        if op.is_use():
            if op in allocations:
                op = allocations[op]
            else:
                allocations[op] = find_free_register(op.type())
        if instr.is_dead():
            break
```

Figure 4.2: Reverse Linear Scan algorithm (adapted from original [44].)

register offsets that assume a base address as the context block base address. When further optimizations, such as register mapping, are not used, the context block base address register is critical for finding the location of the x86-64 registers and must be provided to the instruction builder. The link address register and the return value register are used exclusively for transition to and from the generated machine code, since they contain the branch location after the execution of a translation and the value returned to the runtime. We will describe these mechanisms further when discussing the integration of our DBT with the Arancini runtime.

4.4.4 Register Allocation

The use of virtual registers significantly simplifies code generation by providing an abstraction over the physical registers provided by Arm. However, in order to convert the generated code to executable machine code, it is necessary to map virtual registers to physical registers. This process is handled by the Arm DBT register allocator, which also implements a number of optimizations as part of its allocation strategy.

The Arm register allocator is designed based on the concept of a linear scan allocator [44], which iterates over instructions to determine liveness ranges and then assigns each virtual register. The primary modification made to this approach for Arancini is performing this process in reverse, which we refer to as *reverse linear scan* register allocation [39, 17]. The reverse linear scan algorithm is shown in figure 4.2. We will explore both the considerations for this approach and its impact on the overall algorithm after we describe its behaviour.

The core concept of the algorithm is that each virtual register used by an instruction must have been defined in some preceding instruction in the generated code. All the references to a given virtual register must be mapped to the same physical register and so must the definition for them [44]. The other major part is that once a definition is found and allocated, a liveness interval has just been discovered and the mapping of the virtual register to the physical register can be freed.

These core ideas, however, must be adapted to the type of code generated. Specifically, we must cope with both 64-bit and 32-bit registers, must be able handle loops and instructions with side-effects, as well as support code that directly specifies physical registers. The handling of loops and instructions with side-effects represent difficult problems for us, since our code directly implements loops via branches and labels, while also referencing memory frequently.

The handling of loops in the Arm register allocator is based on the idea of over-approximating the liveness range of a variable to a liveness interval when a loop is detected. This means that once a loop is detected, the liveness range of all currently live virtual registers is extended into a *liveness interval* until the end of the loop. This liveness interval does not necessarily correspond to the actual liveness range but it significantly simplifies our allocation strategy since we are guaranteed correct allocations without changing our approach to perform further analysis. On the other hand, it causes anything allocated within the loop to remain live for the whole loop.

Implementing this requires us to determine where a loop starts and where it ends. Since ArancinIR explicitly encodes loops via branches and labels, while also supporting both forward and backward branches, approaches developed for handling phi-nodes in SSA-based IRs cannot directly be applied here. Instead, we track which branches refer to which labels and the earliest branch to a label. As such, we consider a branching control flow to start whenever we first a branch or a label during our reverse traversal. We can determine where a loop ends, even if it contains nested loops or generally nested control flow, using the earliest branch to a label for forward branches and by counting the number of seen branches for a label relative to the total number of branches with that label as a target. As such, we can determine any type of control flow and assign correct liveness intervals.

Another key point of handling loops is that this overapproximation using liveness intervals leaves stale mappings even after the loop has completed, since we do not know whether a mapping corresponds to some virtual register defined and used exclusively within the loop or if a mapping describes a dependency on a definition before the loop. Handling this issue is non-trivial and requires further analysis, since leaving it unaddressed causes a large number of physical registers to become unusable. We deal with it by iterating over program instructions in program order until the loop, marking definitions and then deallocating all mappings that do not have corresponding

definitions. This ensures that all the mappings after processing the loop represent values that are defined prior to the loop. Note, however, that this simple scheme means that if a variable is used in the loop but is not defined, the register allocator cannot detect a missing definition and will result in the generated code having undefined behaviour.

The other issue is that of instructions with side-effects, such as those that write to memory or those that set flags. These instructions require special handling since we may not depend on their definitions (if they have any) but nevertheless depend on their side-effects. Indeed, writes to memory should always be executed, while instructions with flags must be kept if there are instructions that depend on these flags. We handle these two broad categories via two different mechanisms, marking instructions as *keep* and tracking implicit reads or writes.

The register allocator treats instructions marked as *keep* differently compared to other instructions. First of all, note that the *keep* attribute is applied to instructions rather than virtual registers, which means the special handling is needed for all possible virtual register referenced by such an instruction. Specifically, such instructions can never be optimized away in any optimization that we implement in the register allocator. Furthermore, each virtual register defined by the instruction via a write must be allocated even if there are no users for the definition. Finally, it is sufficient for our purposes to handle reads to virtual registers without any modification by allocating as if the instruction was not marked as *keep*. This approach ensures that the instruction will be kept in the instructions stream, while its registers will be allocated correctly, irrespectively of further optimizations that may be performed.

The tracking of implicit reads and writes is another major problem that arises in particular due to the design of the Arm ISA. Specifically, considering flags, the A64 ISA supports both setting flags via some instructions and reading flags to determine the behaviour of some instructions. This is critical to the ISA, since conditional branches and other conditional logic is implemented via these flags. Since the Arm backend also maps those Arm flags to the guest flags, we need to ensure that if any such dependency exists between operations and flags, we ensure that it is maintained across optimizations. Note that flags are read and written to implicitly by instructions, meaning that simply analyzing defined registers or used registers is not sufficient. As such, we mark each instruction that either relies on or sets registers implicitly with the exact registers that it reads or writes. During register allocation, we track all live implicit dependencies at any given point in the reverse traversal but we only consider this information when the definitions of an instruction have no users. In such a case, we avoid our typical optimizations that may eliminate the instruction, if we determine that it fulfills any implicit dependency. We ensure this protection from optimizations by setting such as instruction as *keep*.

We also implement a series of optimizations via our register allocator to enable simplified code generation that yields reasonably performant code. In particular, we implement the following the optimizations:

- move elimination
- Dead Code Elimination (DCE)

These optimizations are implemented within the register allocator because the reverse linear scan approach fits naturally with their implementation. Specifically, since we iterate in reverse, we will see every use of a virtual register before every definition. This enables us to determine whether a given definition is actually used, effectively performing liveness analysis during the scan with no additional overhead. If a definition is not used, we can generally eliminate it from the instruction stream. We generally do so, since our approach of macro expansion within the code generator tends to produce generic code patterns that may not be needed in many cases. The only overhead here is that we need to process extra instructions in the register allocator while avoiding a lot of special cases in the code generator. Note that we do not perform DCE in special cases, such as loops or for instructions marked as *keep*.

The use of move elimination is a minor but important optimization for us, since we track dependency transfer via moves. Specifically, we may duplicate values when expanding macros during instruction selection but these duplications are not guaranteed to be necessary. Another case where duplication happens is related to bitcasts, where we use moves to represent the cast. The use of move elimination can save large number of instructions that simply move values between registers and it may optimize code with vectors encoded as GPRs when only part of the vector needs to be actually duplicated.

Eliminating dead code is also very beneficial to the quality of generated since it reduces the overall number of executed instructions. However, it provides significant benefits to the Arm backend specifically due to its use of macro expansion that generates generic code, which may not be needed in many cases. For instance, macro expansion may generate code to prepare flags for writing back into the x86-64 context block, even when there is no such write specified in the ArancinIR. Another example is when extracting from a simulated vectors backed by scalar registers, where only a subset of the registers are needed and many of the vector elements do not have be loaded. The use of DCE in both cases eliminates multiple instructions, providing relatively large reductions in code size, while affecting minimal the actual runtime of the translator. An important point here is that the benefits of DCE are so high that they have avoided register spilling by reducing register pressure significantly for more complex vectorized loops.

The effectiveness of these optimizations will be further explored in the evaluation chapter.

4.4.5 Assembler

We have found it useful to generate assembly code, rather than machine code, because it allows us to use virtual registers for data flow dependencies. However, once we have performed register allocation it is necessary to convert our instructions into machine code that can be executed by our runtime.

There are some non-trivial considerations for assembly that impact DBTs in particular. One such consideration is that machine code generation must be fast to minimize latency. Another consideration is that the generator must support a wide variety of ISA extensions, such that we can generate optimized code for each platform. Another consideration is that must support basic assembler features, such as labels, since Arancini uses such features and expects availability for them.

These issues guided our choice of assembler, since they narrow down the search space significantly. In particular, assemblers such as GAS cannot be used because they require a program to be spawned for each translation, as well as formatting a text file and copying the output to an internal buffer [27, 21]. Note that given our relatively short code for each chunk, the overhead of using such an assembler would be too large. Similarly, we cannot use very low level assemblers because they lack labels and other such features. As such, our choices were between the Keystone assembler and AsmJit [56, 54].

We are currently using Keystone because it integrates directly with our runtime by simply translating in an internal buffer that we can avoid copying, while making no assumptions about the code placement and such. We are evaluating AsmJit for the future, primarily due to its excellent performance and its built-in register allocator.

Given that Keystone simply provided a text interface for assembly, we have built our own assembler interface, complete with operand types and formatting features [56]. In particular, we have designed a strongly-typed interface for our assembler, which has caught a large number of errors at runtime with helpful diagnostics. Finally, given that we have to output ASCII text for Keystone, we have relied on the {fmt} C++ library to facilitate organized formatting of our generated code [23].

4.4.6 Runtime Integration

Our generated machine code can be executed to perform the same operation as ArancinIR but it makes a number of core assumptions that must be handled by the runtime prior to execution. Additionally, the generated code must be stored appropri-

Code	Semantics
0	Normal termination
1	Handle system call
2	Handle interrupt
3	Poisoned instruction

Table 4.3: Return codes recognized by the runtime and their associated semantics.

ately to enable execution and should likely be cached to enable reuse. We discuss these concerns and provide the overall design for the runtimes handling of a translation.

First of all, the generated machine code assumes that it can access most A64 ISA registers for its own use. This causes those registers to be clobbered by the execution of the translation and the translator does not include any mechanism to restore them. This is a design choice that simplifies the design of our translator, since it does not have to reason about ABI constraints and can focus exclusively on translating ArancinIR. The handling of ABI constraints to ensure that clobbered registers are restored is relegated to the runtime, which uses a trampoline mechanism to save and restore registers. We also use the trampoline mechanism to provide the base address of the register context block and the return address to the trampoline to the translation, which references them as needed.

A *trampoline*, as used in the runtime, is a sequence of instructions that save register, execute a translation and subsequently restore the saved host registers. We maintain three distinct trampolines, one for each of our target architectures: x86-64, RISC-V and Arm. Each trampoline is ABI-specific and may differ to support different optimizations to our interface logic. The Arm trampoline at a high-level only saves the target registers that are specified as a callee-saved to the stack, invokes the translation via a branch and the restores the callee-saved registers from the stack. The Arm architecture facilitates this process by providing optimized instructions to load and store two registers at a time.

We briefly discuss the use of the link register and the return value in the context of these trampolines. In particular, the trampoline causes a branch into the translation and must provide a mechanism to enable the translation to return back to the trampoline. This mechanism is provided by the link register, which is directly used by the `RET` instruction that is always generated at the end of each translation. The return value is the mechanism that enables the translation to notify the runtime of special conditions.

In particular, the runtime must be able to handle any of the conditions listed in table 4.3.

We briefly discuss the meaning of each condition. We start with condition code 0

that denotes normal termination, which simply prompts the runtime to proceed to the next chunk based on the updated program counter. Condition codes 1 and 2 require special handling, since they represent that the basic block completed with either the invocation of a system or an interrupt. The handling of system calls and interrupts has been discussed in Section 4.2.2.

Finally, an additional condition code is introduced for poisoned instructions that denote target instructions that cannot be translated by the emulator. The emulator must terminate execution in such cases with proper diagnostics but may also invoke native C++ code to interpret that instruction. The use of such a code is two-fold, one is that it enables support for a subset of x86-64 instructions with others being interpreted and second is that it can clearly denote whether some x86-64 instruction is not supported by the emulator. As a side note, there are instructions that are highly specialized that Arancini does not support, since our primary target are not server deployments of programs using complex x86-64 functionality but rather supporting a large majority of x86-64 programs.

4.4.7 Translation Management

The runtime is also responsible for managing the translations produced by individual translator backends. In general, a runtime may translate each block that it discovers anew every time that it reaches that block. However, this is inefficient due to the presence of loops that execute the same blocks multiple times and other cases where control flow reaches the same blocks repeatedly. As such, the runtime implements a caching mechanism for produced translations, enabling it to repeatedly look them up.

The runtime maintains a hash-table data structure that maps addresses to translations, such that it can be indexed with the current value of the updated program counter after the execution of a translation to find the next translation. Since the overhead of accessing this map is directly in the critical path of executing a translation, it must be low-latency. As such, we use a hash-based approach with $O(1)$ lookup time to optimize for the common case of frequent chunk lookups in a loop.

The translation cache is accessed whenever a new address is encountered that has no static translation or after the execution of a dynamic translation. If the translation cache has a translation, the translation is invoked directly without further lifting or lowering. On the other hand, when no translation is found, the DBT is invoked to generate a translation that will be stored in the translation cache.

A minor complication with this scheme is that a branch to an instruction within a dynamic basic block will cause the entire dynamic basic block starting from that instruction to be translated. This is inefficient, since a suitable translation already exists, however it is simple and avoids having to track the entire range of addresses. In the

future, we plan to extend our lookup logic to take into account all instructions within a dynamic basic block as part of the lookup.

5 Evaluation

In this chapter, we will evaluate our implementation of the Arm DBT for Arancini. We will primarily focus on performance but also analyze secondary concerns, such as code sizes and memory requirements for translation.

5.1 Experimental Setup

We conduct our experiments exclusively on a 64-bit Arm server with the ARMv8 architecture for each PE. Our evaluation machine features a base architecture supporting the A64 ISA, along with NEON vectors, such that it provides a good approximation for our target deployment. Note that we do not use the SVE2 and LSE features present on the system during code generation, since most hosts targeted by Arancini do not support them. The overall system configuration is summarised in figure 5.1.

We use a subset of the Phoenix benchmark suite for evaluating the performance of our dynamic binary translator on Arm [46]. The programs contained in Phoenix focus primarily on numeric computation of number-crunching tasks, such as calculating the k-means or a histogram from an image. However, these tasks are executed within complete programs, which also perform I/O to read their input, print information about their execution and ultimately print their output. As such, although they are small compared to a complex program, they are self-contained and a good approximation of a typical program.

The programs contained in the Phoenix benchmark are detailed in figure 5.2. We note that most benchmarks support specifying input parameters, which directly affect their running time and memory usage. However, some benchmarks hardcode various parameters, such that we cannot modify their execution through them. Finally, note that the matrix multiply benchmark as an exception to other benchmarks does not receive its input via commandline parameters but rather relies on hardcoded filenames that it looks for in its current working directory.

Our evaluation will use the histogram, kmeans, pca and string match benchmarks to illustrate our results. These benchmarks are executed in the configuration shown in table 5.3.

The remaining benchmarks are partially supported currently and still under development, such that they cannot be used in our evaluation yet. Nevertheless, we consider

Resource	System Configuration
Cores	28 cores per NUMA domain, 56 total
Threads	4 threads per core
Superscalar	yes
Architecture	ARMv8-a
Maximum Frequency	33 MHz per core
Per core L1 Instruction Cache	32 KiB
Per core L1 Data Cache	32 KiB
Per core L2 Unified Cache	256 KiB
Total L3 Unified Cache	32 MiB

Table 5.1: System configuration for the Yasmin benchmark sever.

Program Name	Invocation Parameters	Explanation
Histogram	`histogram <image_path>`	Generates a histogram of pixel value frequencies in the RBG channels of the bitmap input
Kmeans	`kmeans -d <vector_dim> -c <num_clusters> -p <num_points> -s <max_value>`	Finds the mean of k groups of n-dimensional data points
Linear Regression	`linear_regression <filename>`	Generates a linear approximation of all points defined by the x-y coordinates from the input file
Matrix Multiply	`matrix_multiply`	Computes the product of two matrices of the same form
PCA	`pca -r <num_rows> -c <num_cols> -s <max_value>`	Computes the mean vector and covariance matrix for a randomly-generated matrix
String Match	`string_match <key_file>`	Finds matching strings to a list of (hardcoded) encrypted strings
Word Count	`word_count <word_file_path>`	Calculates the total number of whitespace-separated words in a file

Table 5.2: The complete set of Phoenix-2.0 benchmark programs, along with their invocation parameters and a brief explanation of their computation [47].

Program Name	Invocation Parameters
Histogram	`small.bmp`
Kmeans	`-d 3 -c 100 -p 100000 -s 1000`
PCA	default configuration
String Match	`key_file_50MB.txt`

Table 5.3: The execution parameters passed to each benchmark.

the benchmarks that we do include as sufficient for our purposes.

5.2 Performance Overhead

We begin with an evaluation of the performance overhead of performing binary translation in Arancini. Since we have developed a hybrid scheme, we evaluate both types of binary translation as to their performance. Specifically, we will run the Phoenix benchmark programs using hybrid translation and fully-dynamic translation to determine their performance. Furthermore, we will analyze both statically-linked and dynamically-linked versions of these benchmarks.

We compare the performance of Arancini to the Risotto state-of-the-art DBT [30] and to native execution with no emulation. We choose to compare to Risotto because it is based on the widely used QEMU DBT and provides additional correctness guarantees, while having the same or better performance. We compare to native execution because it provides the lower-bound in execution time in terms of which we can express our measurements.

The runtimes are shown in figures 5.1 and 5.2 as a bar chart with values relative to native execution. We present the results for both statically and dynamically linked versions of our benchmarks but we will largely not distinguish between them as part of our analysis. We can do so because both result sets observe largely the same patterns and the primary different is the slight overhead associated with using shared libraries that we attribute to the additional overhead of shared libraries in general. Note that both of our results are shown in the log scale to account for the large variations in execution time between the fastest and the slowest benchmarks.

We now proceed to analyse the results of our benchmarks in more depth, starting with the performance of our DBT. We note that our benchmarks show that the Arm DBT is significantly slower than the Risotto DBT. We attribute this fact to two main reasons: *(i)* significantly better instruction selection and *(ii)* much more effective use of translation caching.

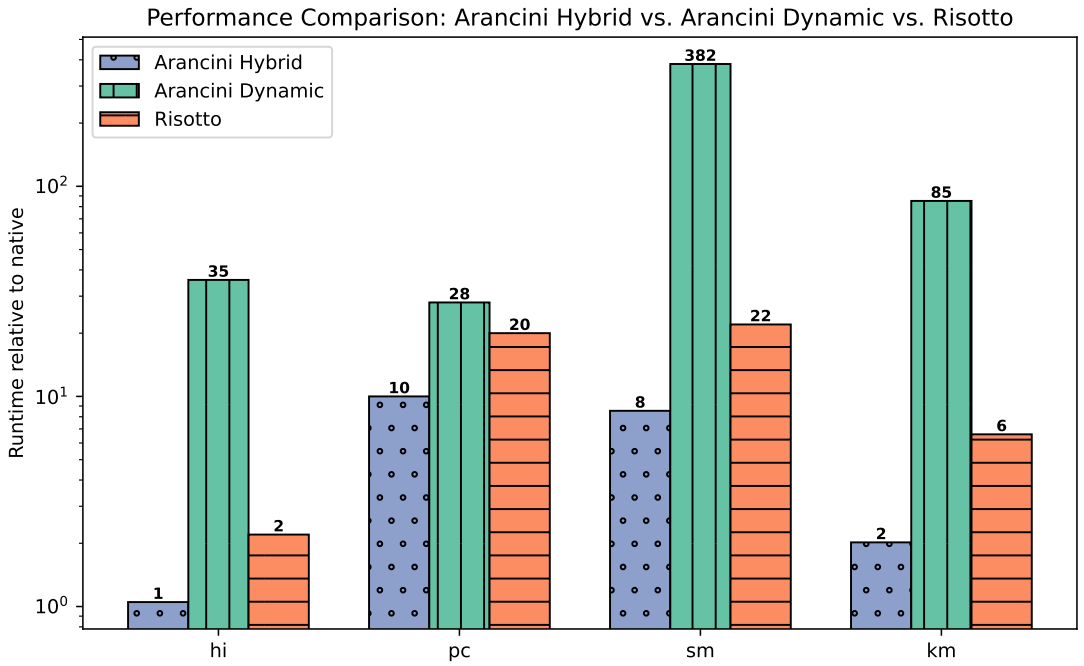


Figure 5.1: Performance overhead evaluation of Arancini compared to Risotto and native execution for the Phoenix benchmarks for statically-linked programs.

In particular, Risotto uses TCG IR with heavily optimized lifting and instruction selection [13, 24], while employing various code cache designs to improve performance. On the other hand, the DBT designed for Arancini were not designed with performance as the primary goal, but rather completeness. Generally, designing an efficient DBT is a daunting task and the QEMU emulator has been extended over more than a decade to achieve such performance.

However, the primary focus of the Arm DBT was to enable the implementation of a hybrid translator and we should evaluate whether its use as part of one achieves any performance improvement of Risotto. As we can observe in figure 5.1, the hybrid translator is at least twice faster than Risotto for every benchmark and significantly faster than the dynamic translator. These results for hybrid translation can be attributed to the following factors:

1. Effective static code generation with very limited dynamic translation
2. The use of the dead flag elimination optimization
3. The lack of any introduced fences

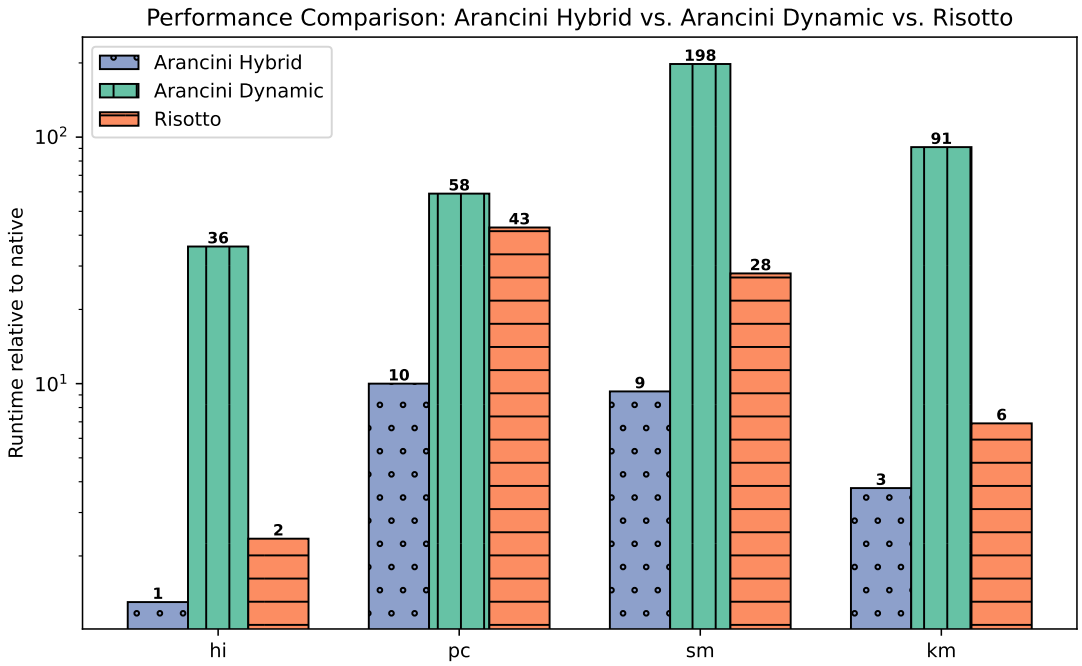


Figure 5.2: Performance overhead evaluation of Arancini compared to Risotto and native execution for the Phoenix benchmarks when using shared libraries.

We have discussed the benefits of static code generation extensively as part of this work and have also discussed the use of dead flag optimization in Section 4.3.1. However, another important optimization here that is specific to our benchmarks is that we do not introduce any additional fences in our static translation to order accesses between threads. This optimization can be performed because our benchmarks are single-threaded and we include it because Risotto also performs this optimization.

The notable point here is that the performance of our hybrid translator cannot be achieved without the Arm DBT, since purely static translation is not sufficient for all programs. As such, we consider the results as further motivating the use of such hybrid translator and as a demonstration of the effectiveness of our DBT.

Arm DBT

We now evaluate the Arm DBT in more depth and analyze the performance of individual components. Specifically, we are interested in the overhead resulting from instruction selection, register allocation, machine code generation and the overhead owed to the rest of the Arancini runtime component.

We will breakdown the runtime of the analyzed Phoenix benchmarks into its individual components, which we will compare to the overall runtime of the benchmark.

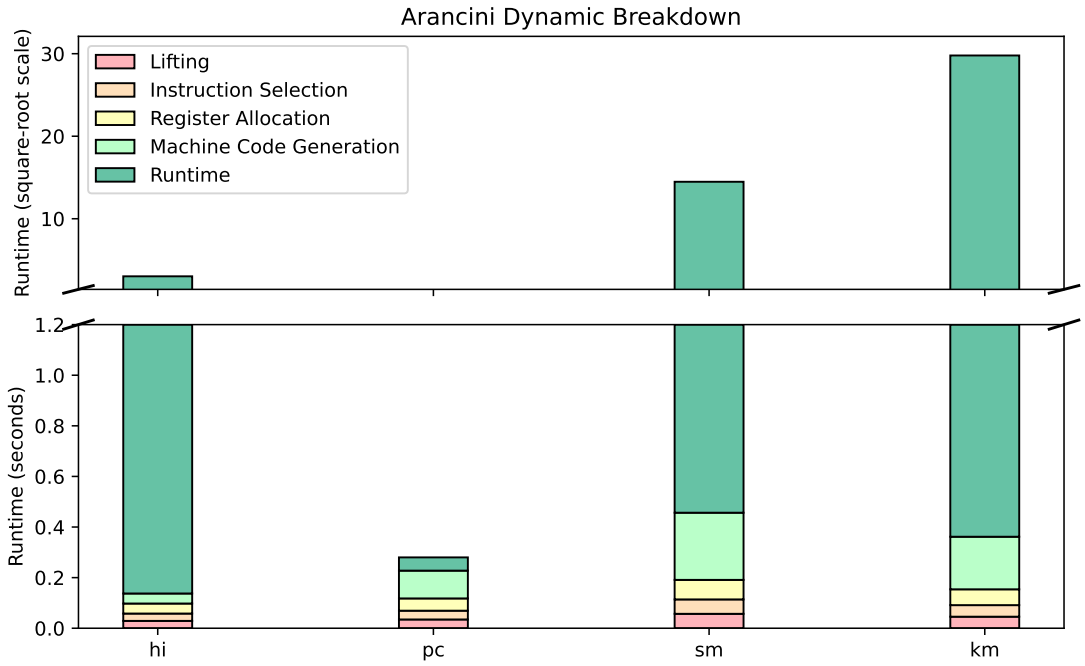


Figure 5.3: Performance overhead of each component of the Arm DBT for each executed benchmark. Note that the we use a linear scale for the lower y-axis and a square root scale for the upper y-axis to make visible the cumulated overheads.

The runtimes for each component are shown in detail in figure 5.3. A notable point is that the percentages associated with the tasks pertaining to Arm dynamic code generation and lifting x86-64 instructions account for a very small proportion of the overall execution of a program when the program executes for a longer duration. On the other hand, programs such as PCA that have short overall runtimes face significant overhead as a result of lifting and instruction selection.

These results show that further optimization opportunities are possible in the Arm DBT, which may increase the code generation latency slightly but could effect major reductions in the overall runtime. However, more optimized code generation should be performed only when there is a high chance of reuse of the generated code. Otherwise, for a shorter program like PCA, we may benefit from even simpler code generation than we perform.

Benchmark	Static Lifting	Dynamic Lifting	Dynamic Translation
histogram	5700	8	162
kmeans	8088	8	162
pca	6176	8	162
string_match	8239	8	162

Table 5.4: Comparison of statically and dynamically-lifted instruction counts for the statically-linked benchmarks.

We note that the primary reason why the observed overhead is so low is due to the use of the translation cache, since we perform most of our code generation logic towards the beginning of program execution. At that stage, the translation cache is empty and each newly discovered chunk must be translated. However, as we fill this cache with translated chunks, we eliminate possible retranslations and thus minimize significantly the overhead associated with code generation. This results is expected, as other emulators also use code caches and observe similar benefits [13, 61, 19, 51].

Another important point is that the percentages for lifting and instruction selection are both significantly larger than the rest of the code generation logic. Since most of our optimizations on the generated code occur in the register allocation section, this confirms that those optimizations introduce relatively low overhead. In fact, we can claim that the design decision to use virtual registers and a register allocator component over directly generating using physical registers was indeed correct, since we have achieve major reductions in code size with very low runtime overhead.

5.3 Translation Evaluation

We proceed to evaluate the actual code generated by our translation with regards to other characteristics, such as its size and form. This evaluation will show results for both statically-linked and dynamically-linked benchmarks but focus primarily on the statically-linked benchmarks, since the implementation of dynamic linking is not the primary goal of this work.

5.3.1 Code Size

We begin with an overview of the code size generated by our various components, such as the instruction selector and register allocator. We will particular focus on the reductions that we can achieve using our included optimizations.

We will initially describe the overall distribution of code between hybrid and dynamic

Benchmark	Guest code (x86-64)	Translation (Arm)	Native
histogram	1274	13324	1877
kmeans	1983	38437	3313
pca	1376	16131	1807
string_match	2427	26159	3458

Table 5.5: Comparison of the instruction counts of the guest code, the generated translation code and the natively-compiled benchmark program for a run of each statically-linked benchmark in fully-dynamic mode.

Benchmark	Guest code (x86-64)	Translation (Arm)	Native
histogram	1854	21036	6209
kmeans	3387	48719	6826
pca	2722	39852	6289
string_match	3565	49955	6702

Table 5.6: Comparison of the instruction counts of the guest code, the generated translation code and the natively-compiled benchmark program for a run of each dynamically-linked benchmark in fully-dynamic mode.

translation and then delve deeper into the dynamically translated code for fully-dynamic execution. We can observe the distribution of code for lifting and lowering the statically-linked benchmarks in figure 5.4. In particular, we notice that a large number of program instructions are lifted statically, while a small number of instructions cannot be lifted. An important result here is that the static translation may lift significantly more instructions than a fully-dynamic execution because it always lifts greedily as many instructions as it can find. A fully-dynamic translation generally only translates instructions encountered at runtime, which may consist in just part of the overall program.

However, it is important to note that our benchmarks do not use self-modifying code or other code patterns that may require more extensive use of the dynamic translator. A benchmark using features such as self-modifying code would require the dynamic binary translator to generate translations for the generated code and thus would lead to a much higher number of dynamically lifted instructions.

We will proceed to discuss only the dynamic backend in fully-dynamic program executions. In particular, we will analyze the code size and memory size to better understand the characteristics of the Arm DBT.

We show the code size for each benchmark that we evaluate in table 5.5 for the statically-linked and in table 5.6 for the dynamically-linked benchmarks, respectively.

The guest code and generated translation code sizes were determined using ARANCINI, while code size for native executing was determined using the instrgrind plugin for valgrind [43, 1].

We observe minor differences in code sizes between the statically and dynamically-linked versions, primarily results from the additional code that is used to handle the dynamic linking. However, since we observe the same overall performance characteristics, we will discuss these patterns further without differentiating between the two benchmarks.

We note that our comparison with the guest code here can be misleading because both the encoding of code and the overall structure of the ISA differ between our platforms. Namely, a single x86-64 instruction can range from 1 to 15 bytes, while Arm instructions are always 4 bytes long [8]. Furthermore, as a result of these encoding differences, the logic implemented within a x86-64 instruction often requires multiple Arm instructions. We will further expand on this by comparing the actual memory requirements needed to store the code.

We analyze the total amount of memory needed in our instruction cache for storing the entirety of the translation for each program run. The instruction cache for our evaluation purposes is not emptied during the entire runtime of a benchmark, such that its total size at the end of a program represents the entire dynamic translation code.

Benchmark	Guest Code	Translated Code	Native Code
histogram	5 KiB	52 KiB	7.7 KiB
kmeans	7.4 KiB	190 KiB	12.9 KiB
pca	5.3 KiB	63 KiB	7.1 KiB
string_match	9 KiB	100 KiB	13.5 KiB

Table 5.7: Comparison of the memory requirements for storing the guest code and the translated code in the statically-linked benchmarks. Note that the values are rounded to the nearest value.

Benchmark	Guest Code	Translated Code	Native Code
histogram	6.5 KiB	82 KiB	24.2 KiB
kmeans	12.4 KiB	82.4 KiB	26.6 KiB
pca	10.1 KiB	155 KiB	24.2 KiB
string_match	12.9 KiB	195 KiB	26.1 KiB

Table 5.8: Comparison of the memory requirements for storing the guest code and the translated code in the dynamically-linked benchmarks. Note that the values are rounded to the nearest value.

We show this result in table 5.7, where we compare the memory requirements for the guest code and the translation code. As we noticed from both table 5.5 and 5.7, we observe that our translations are roughly 10-times larger than the original code. The same pattern occurs in the dynamically-linked benchmarks, which we observe in table 5.8. We attribute this increase in both instructions, as observed previously, and memory sizes to the following factors:

1. Loading the emulated guest state when accessing it from memory
2. Writing back the emulated guest state to memory
3. Handling of wide immediates
4. CISC instructions typically map to multiple RISC instructions
5. Inefficient conversions specified by ArancinIR

Specifically, we observe that a single CISC instruction in x86-64 may simultaneously load memory and process data within a single instruction. We also observe the CISC instructions can encode large immediates within their variable-size format. These capabilities are not directly available on Arm, which requires explicit loads/stores and does not support large immediates. An instruction such as `IMUL RAX, qword ptr [RBX], 0x1234` would cause the translator to generate a sequence of `MOVZ` and `MOVK` instructions to load the 32-bit immediate, loads from the register context block to load `RAX`, instructions for executing the multiplication and stores to write back the result. Note that `IMUL` can set flags, so we need to ensure that they are computed correctly and written to the emulated flag register, which results in at least four additional instructions because each flag is treated as a separate register in ArancinIR.

As in the case of flags, code bloat may result directly from the implementation of ArancinIR that we use. For instance, ArancinIR introduces bitcasts and other conversion operations which may lead to bloated code even when the conversion is effectively a no-op. At their core, these issues appear because ArancinIR is not particularly optimized

Benchmark	Instruction Selection	Register Allocation
histogram	23908	13324
kmeans	38437	21109
pca	29910	16131
string_match	47068	26159

Table 5.9: Comparison of code sizes generated at each pass of the Arm DBT for the translation of each statically-linked benchmark.

Benchmark	Instruction Selection	Register Allocation
histogram	36420	21036
kmeans	100281	48719
pca	83996	39852
string_match	102258	49955

Table 5.10: Comparison of code sizes generated at each pass of the Arm DBT for the translation of each dynamically-linked benchmark.

for lowering to Arm. This is a design decision to facilitate lowering to a diverse set of target architecture, while requiring individual backends to optimize to their target architecture. Such optimizations can be implemented effectively using a pass over the IR that regularizes it for Arm before lowering to the target architecture. However, we do not perform such passes currently but plan to in the future.

As mentioned when discussing the hybrid format in section 4.2.1, Arancini needs to store the guest code and also caches translations. As such, the total size that is maintained in memory for each benchmark is the sum of the two columns.

We now analyze the number of instructions that remain in our instruction stream after the execution of each pass. We note that the majority of our move elimination and DCE logic is performed as part of register allocation, as mentioned in section 4.4.4. As such, we expect a significant reduction is code size.

We show this result in tables 5.9 and 5.10, where we observe the halving of the generated code size. In particular, we note that this is largely a demonstration of the inefficiency of using macro expansion as an instruction selection mechanism.

We also note the effectiveness of our simple optimizations at reducing code size with such an instruction selection scheme. It is very important that these optimizations are directly performed as part of our register allocation scheme, meaning that we get these large reductions with negligible bookkeeping overhead during our register allocation.

6 Conclusion

In this thesis, we present the following contributions: *(1)* a DBT system capable of translating x86-64 to Arm code via an intermediate conversion to ArancinIR, *(2a)* the integration of such a DBT into a wider runtime system to enable HBT for efficient and complete emulation of x86-64 code on Arm, *(2b)* the implementation of a variety of optimizations on for dynamic generated code and *(3)* an evaluation of the DBT with regards to its completeness and efficiency performed directly on Arm.

We expect the use of emulators to increase further in the future and this makes our work particularly timely. Furthermore, we consider that the use of hybrid translation, although limited nowadays, will be a major boost to the performance of emulators in the future. However, we note that although we have been able to implement a dynamic binary translator in this work, significant development effort was spent both implementing it and debugging it. We believe future research should address more the topic of effective dynamic binary translator design for hybrid translators.

6.1 Future Work

6.1.1 Testing & Verification

Our implementation and evaluation rely on an ad-hoc testing infrastructure that can execute functional tests and verify their output. This approach has enabled us to discover various problems, since we were able to test multiple possible invocation of Arancini. However, this approach is not systematic nor comprehensive in regards to the verifying correctness. Furthermore, it is not capable of providing precise diagnostics as to the cause of an issue, only logs detailing the behaviour of our translator.

This has motivated us to design a comprehensive verification framework for Arancini that can verify the correctness of a given dynamic translation, as well as pinpoint the exact issue. In particular, since Arancini maintains an emulated guest state, we can compare its values with the actual state maintained by the original program. These values are not guaranteed to be numerically equal but they should be semantically equivalent. Techniques such as symbolic execution enable us to extract the semantics of the original program, such that we can determine such equivalence relations. This provides sufficient information to enable automatic trace analysis as a verification

mechanism, which can also directly determine what state mutations are incorrectly performed by the translated code. This work is currently undergoing development, as we seek to further enhance Arancini.

6.1.2 Debuggability

Since Arancini operates at a very low-level, just above the host OS, we have spent extensive time debugging a large variety of issues. A major limitation is the lack of a debugger to interactively analyze the state of the translator. Although multiple mature debuggers and standard interfaces for them exist, non-trivial engineering work is required to integrate them into a given binary translator. We are exploring techniques for integrating a GDB stub within Arancini, in a similar way that QEMU does [55, 57]. The challenge here is to determine how to integrate such a debugger with our static translations, since they do not necessarily maintain consistent guest state after every host instruction.

6.1.3 Using Dynamic Translations to offline patching of Static Translations

Our current implementation stores generated dynamic translation in a hash map data structure to facilitate fast lookup when the same code must be executed. However, those translations are lost upon program termination or when a predetermined limit on the size of the stored translations is exceeded. This represents a wasted opportunity to use the dynamically generated code to fill the gaps of static translations after program execution, enabling an even larger ratio of static translations relative to dynamic translations.

An interesting benefit of such an approach would be that after the initial run of the program, most of the gaps in static translations are expected to be filled. This would mean that after sufficiently many runs, the DBT would remain unused since the entire code would be statically translated, thus eliminating any overhead incurred from dynamic translation. Note that the DBT would still be required, since it is not possible to statically determine when the entirety of the programs code has been discovered, but it would be used.

6.1.4 Parallel Optimization of Dynamic Translations

There exists a tradeoff in dynamic translation between generating code quickly and generating optimized code, since optimizations rely on dynamic analysis that can be time consuming. In particular, the time spent optimizing generated code may increase the latency more than the decrease in latency due to faster execution [51]. A key

consideration in this tradeoff is that the time savings related to optimization increase as a function of the frequency of executing the optimized code.

A possible optimization for Arancini would be to utilize the DBT to generate code very quickly for every block, while adding tracking logic into the runtime that analyzes the frequency of executing a given translation. Once a threshold is reached, a background thread could be spawned that retranslates the block via LLVM with all available optimizations, while the unoptimized translation would continue to be used until an optimized version is available. Once an optimized version is available, the system could replaced the unoptimized translation with the optimized one.

The utility of such an optimization is difficult to ascertain, but it would be useful for programs where static translation is not largely infeasible, since it would produce faster dynamic translations over time.

6.1.5 Evaluating Arancini on Apple Silicon

Our evaluation was carried out an Arm system that is different from the Apple M-series processors but our approach is heavily inspired by Apple's Rosetta 2 software [3]. It is generally believed that Apple systems feature hardware support for emulating x86-64 code, such that they can provide significantly higher performance. We are planning to evaluate Arancini on an Apple M-series system and compare it to the performance of Rosetta 2 with regards to the execution time of x86-64 binaries.

List of Figures

List of Tables

Bibliography

[1] Andrei Rimsa Álvares. *Instrgrind plugin for Valgrind*. https://github.com/rimsa/instrgrind. 2025.

[2] Apple. *Porting Your macOS Apps to Apple Silicon*. https://developer.apple.com/documentation/apple-silicon/porting-your-macos-apps-to-apple-silicon. 2024.

[3] Apple. *Rosetta 2 on a Mac with Apple silicon*. https://support.apple.com/fr-fr/guide/security/secebb113be1/web. 2021.

[4] Apple-Silicon. *Addressing Architectural Differences in Your macOS Code*. Available at https://developer.apple.com/documentation/apple-silicon/addressing-architectural-differences-in-your-macos-code. 2020.

[5] ARM. *ARM Cortex-A Series Programmer's Guide for ARMv8-A*. https://developer.arm.com/documentation/den0024/a/. 2015.

[6] ARM. *Technology Licensing and Subscriptions*. https://www.arm.com/products/licensing. 2025.

[7] ARM. *The Official History of Arm*. https://newsroom.arm.com/blog/arm-official-history. 2023.

[8] Arm Limited. *ARM Architecture Reference Manual for A-profile architecture*. https://www.arm.com/architecture/cpu. 2025.

[9] Arm Limited. *Introduction to SVE2*. https://developer.arm.com/documentation/102340/0100/Introducing-SVE2. 2021.

[10] Arm Limited. *Neon Programmer Guide for Armv8-A 4.0*. https://documentation-service.arm.com/static/6530e5163f12c06bc0f740ef. 2023.

[11] Amazon AWS. *AWS Graviton Processor*. https://aws.amazon.com/ec2/graviton.

[12] Martin Beck et al. "AtoMig: Automatically Migrating Millions Lines of Code from TSO to WMM." In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ASPLOS 2023. Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 61–73. ISBN: 9781450399166. DOI: 10.1145/3575693.3579849.

[13] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator." In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, p. 41.

[14] Lifting Bits. *Framework for lifting x86, amd64, and aarch64 program binaries to LLVM bitcode*. https://github.com/lifting-bits/mcsema. 2022.

[15] Gabriel Hjort Blindell. *Instruction Selection: Principles, Methods, and Applications*. 1st. Springer Publishing Company, Incorporated, 2016. ISBN: 3319340174.

[16] Ahmed Bougacha. *Binary Translator to LLVM IR*. https://github.com/repzret/dagger. 2022.

[17] brrt to the future. *Reverse Linear Scan Allocation is probably a good idea*. https://brrt-to-the-future.blogspot.com/2019/03/reverse-linear-scan-allocation-is.html. 2023.

[18] Jiunn-Yeu Chen and Wuu Yang. *A Static Binary Translator for Efficient Migration of ARM based Applications*. 2008.

[19] Wei Chen et al. "A Light-weight Code Cache Design for Dynamic Binary Translation." In: *2009 15th International Conference on Parallel and Distributed Systems*. 2009, pp. 120–125. DOI: 10.1109/ICPADS.2009.134.

[20] David Chisnall. "How to Design an ISA: The popularity of RISC-V has led many to try designing instruction sets." In: *Queue* 21.6 (Jan. 2024), pp. 27–46. ISSN: 1542-7730. DOI: 10.1145/3639445. URL: https://doi.org/10.1145/3639445.

[21] Chris Lattner. *Intro to the LLVM MC Project*. https://blog.llvm.org/2010/04/intro-to-llvm-mc-project.html. 2010.

[22] The kernel development community. *Kernel Support for miscellaneous Binary Formats*. Available at https://www.kernel.org/doc/html/latest/admin-guide/binfmt-misc.html.

[23] The {fmt} developers. *{fmt}: A modern formatting library*. https://fmt.dev/latest/index.html. 2025.

[24] The QEMU Project Developers. *TCG Intermediate Representation*. Available at https://www.qemu.org/docs/master/devel/tcg-ops.html.

[25] Chris Edwards. "Moore's Law: what comes next?" In: *Commun. ACM* 64.2 (Jan. 2021), pp. 12–14. ISSN: 0001-0782. DOI: 10.1145/3440992. URL: https://doi.org/10.1145/3440992.

[26] Fabrice Bellard. *QEMU: the FAST! processor emulator*. https://www.qemu.org/.

[27] Free Software Foundation. *The GNU Assembler*. https://sourceware.org/binutils/docs/as/. 2008.

[28] Andrei Frumusanu. *The 2020 Mac Mini Unleashed: Putting Apple Silicon M1 To The Test – Anandtech*. https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested/6. 2020.

[29] Sheng-Yu Fu et al. "Efficient and retargetable SIMD translation in a dynamic binary translator." In: *Software: Practice and Experience* 48.6 (2018), pp. 1312–1330. DOI: https://doi.org/10.1002/spe.2573.

[30] Redha Gouicem et al. "Risotto: A Dynamic Binary Translator for Weak Memory Model Architectures." In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. ASPLOS 2023. Vancouver, BC, Canada: Association for Computing Machinery, 2022, pp. 107–122. ISBN: 9781450399159. DOI: 10.1145/3567955.3567962. URL: https://doi.org/10.1145/3567955.3567962.

[31] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*, Waltham, MA: Morgan Kaufmann, 2012. ISBN: 9780123838728.

[32] R. N. Horspool and N. Marovac. "An Approach to the Problem of Detranslation of Computer Programs." In: *The Computer Journal* 23.3 (Aug. 1980), pp. 223–229. ISSN: 0010-4620. DOI: 10.1093/comjnl/23.3.223. eprint: https://academic.oup.com/comjnl/article-pdf/23/3/223/1066131/23-3-223.pdf. URL: https://doi.org/10.1093/comjnl/23.3.223.

[33] Apple Inc. *Apple unveils M3, M3 Pro, and M3 Max, the most advanced chips for a personal computer*. Available at https://www.apple.com/newsroom/2023/10/apple-unveils-m3-m3-pro-and-m3-max-the-most-advanced-chips-for-a-personal-computer/. 2023.

[34] Arun Kishan. *Azure Cobalt 100-based Virtual Machines are now generally available*. Available at https://azure.microsoft.com/en-us/blog/azure-cobalt-100-based-virtual-machines-are-now-generally-available/. 2024.

[35] Koh M. Nakagawa, FFRI Security. *Project Champolion: Reverse-engineering Rosetta 2*. https://ffri.github.io/ProjectChampollion/. 2021.

[36] C. Lattner and V. Adve. "LLVM: a compilation framework for lifelong program analysis & transformation." In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* San Jose, CA, USA: IEEE, 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[37] LLVM Team. *The LLVM Compiler Infrastructure*. https://llvm.org/.

[38] Daniel Lustig et al. "ArMOR: defending against memory consistency model mismatches in heterogeneous architectures." In: *SIGARCH Comput. Archit. News* 43.3S (June 2015), pp. 388–400. ISSN: 0163-5964. DOI: 10.1145/2872887.2750378. URL: https://doi.org/10.1145/2872887.2750378.

[39] Matt Keeter. *The Solid-State Register Allocator*. https://www.mattkeeter.com/blog/2022-10-04-ssra/. 2022.

[40] Michael Matz et al. *System V Application Binary Interface*. Version 0.99.6. 2013. URL: https://refspecs.linuxfoundation.org/elf/x86_64-abi-0.99.pdf.

[41] Gordon E. Moore. "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff." In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (2006), pp. 33–35. DOI: 10.1109/N-SSC.2006.4785860.

[42] Koh M. Nakagawa. *Project Champollion: Reverse engineering Rosetta 2*. Version 0.1.0. 2021. URL: https://github.com/FFRI/ProjectChampollion.

[43] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation." In: *SIGPLAN Not.* 42.6 (June 2007), pp. 89–100. ISSN: 0362-1340. DOI: 10.1145/1273442.1250746. URL: https://doi.org/10.1145/1273442.1250746.

[44] Massimiliano Poletto and Vivek Sarkar. "Linear scan register allocation." In: *ACM Trans. Program. Lang. Syst.* 21.5 (Sept. 1999), pp. 895–913. ISSN: 0164-0925. DOI: 10.1145/330249.330250. URL: https://doi.org/10.1145/330249.330250.

[45] Gerald J. Popek and Robert P. Goldberg. "Formal requirements for virtualizable third generation architectures." In: *Commun. ACM* 17.7 (July 1974), pp. 412–421. ISSN: 0001-0782. DOI: 10.1145/361011.361073. URL: https://doi.org/10.1145/361011.361073.

[46] Colby Ranger et al. "Evaluating MapReduce for Multi-core and Multiprocessor Systems." In: *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. 2007, pp. 13–24. DOI: 10.1109/HPCA.2007.346181.

[47] Colby Ranger et al. *Phoenix*. https://github.com/kozyraki/phoenix. 2007.

[48] H. G. Rice. "Classes of Recursively Enumerable Sets and Their Decision Problems." In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366. ISSN: 00029947, 10886850. URL: http://www.jstor.org/stable/1990888 (visited on 10/30/2024).

[49] Rodrigo C. O. Rocha et al. "Lasagne: A Static Binary Translator for Weak Memory Model Architectures." In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 888–902. ISBN: 9781450392655. DOI: 10.1145/3519939.3523719. URL: https://doi.org/10.1145/3519939.3523719.

[50] Bor-Yeh Shen et al. "LLBT: an LLVM-based static binary translator." In: *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. CASES '12. Tampere, Finland: Association for Computing Machinery, 2012, pp. 51–60. ISBN: 9781450314244. DOI: 10.1145/2380403.2380419.

[51] James E. Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. The Morgan Kaufmann Series in Computer Architecture and Design. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN: 1558609105. URL: http://portal.acm.org/citation.cfm?id=1204009&coll=GUIDE&dl=GUIDE&CFID=36297498&CFTOKEN=48195634.

[52] Tom Spink, Harry Wagstaff, and Björn Franke. "A Retargetable System-Level DBT Hypervisor." In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 505–520. ISBN: 978-1-939133-03-8. DOI: 10.1145/3302516.3307357. URL: https://www.usenix.org/conference/atc19/presentation/spink.

[53] Tom Spink et al. "Efficient code generation in a region-based dynamic binary translator." In: *Proceedings of the 2014 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*. 2014, pp. 3–12.

[54] The AsmJit Developers. *AsmJit: Low-Latency Machine Code Generation*. https://asmjit.com/. 2025.

[55] the GDB developers. *GDB: The GNU Project Debugger*. https://sourceware.org/gdb/documentation/.

[56] The Keystone Developers. *Keystone: The Ultimate Assembler*. https://www.keystone-engine.org/. 2025.

[57] The QEMU developers. *QEMU GDB Documentation*. https://www.qemu.org/docs/master/system/gdb.html.

[58] Viktor Vafeiadis et al. "Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it." In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '15. Mumbai, India: Association for Computing Machinery, 2015, pp. 209–220. ISBN: 9781450333009. DOI: 10.1145/2676726.2676995.

[59] Amin Vahdat. *Introducing Google Axion Processors, our new Arm-based CPUs*. Available at https://cloud.google.com/blog/products/compute/introducing-googles-new-arm-based-cpu. 2024.

[60] Jun Wang et al. "Dynamic Translation Optimization Method Based on Static Pre-Translation." In: *IEEE Access* 7 (2019), pp. 21491–21501. DOI: 10.1109/ACCESS.2019.2897611.

[61] Wenwen Wang et al. "A general persistent code caching framework for dynamic binary translation (DBT)." In: *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '16. Denver, CO, USA: USENIX Association, 2016, pp. 591–603. ISBN: 9781931971300.

[62] S. Bharadwaj Yadavalli and Aaron Smith. "Raising binaries to LLVM IR with MCTOLL (WIP paper)." In: *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 213–218. ISBN: 9781450367240. DOI: 10.1145/3316482.3326354.