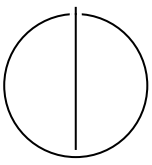
SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Design and Implementation of a Binary Translator from AArch64 to a Custom Intermediate Representation

Konstantin Garbers



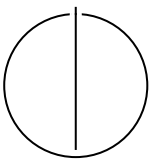
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Design and Implementation of a Binary Translator from AArch64 to a Custom Intermediate Representation

Author:	Konstantin Garbers
Advisor:	Martin Fink
Examiner:	Pramod Bhatotia
Submission Date:	28th February 2025



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

Munich, 28th February 2025 Konstantin Garbers

Acknowledgments

I would like to express my sincere gratitude to my advisor, Martin Fink, not only for providing this opportunity but also for his exceptional patience and kindness throughout this journey. His guidance, mentorship, and valuable insights have been instrumental to the development of this work. I am also deeply grateful to Professor Pramod Bhatotia for the opportunity to pursue this research. Special thanks to Veronika and Pera for their help and feedback to my thesis.

Abstract

This thesis presents the design and implementation of a binary translator, or "lifter," which converts AArch64 machine code into a custom Intermediate Representation (IR) called Assembly Intermediate Representation (AIR). Designed with formal verification in mind, the AIR instruction set is minimal, with each instruction performing a single, well-defined operation. The lifter is a key component of the TrustNoJit (TNJ) project, which seeks to ensure the correctness of Just-In-Time (JIT) compiled code through the Proof-Carrying Code (PCC) approach. By translating complex AArch64 instructions into a minimal, verification-focused IR, this work simplifies the verification process, enabling more efficient and accurate analysis of machine code.

The lifter achieves support for more than 99.7 % of instructions of the Sightglass benchmarking suite. We provide the source code in a publicly accessible Github repository [1].

[1] https://github.com/TUM-DSE/aarch64-air-lifter

Zusammenfassung

Diese Arbeit präsentiert das Design und die Implementierung eines Binärübersetzers, oder „Lifters“, der AArch64-Maschinencode in eine benutzerdefinierte Zwischendarstellung (Intermediate Representation, IR) namens Assembly Intermediate Representation (AIR) umwandelt. Die AIR-Befehlssatzarchitektur wurde mit Blick auf formale Verifikation entworfen und ist minimal gehalten, wobei jede Instruktion eine einzelne, klar definierte Operation ausführt.

Der Lifter ist eine zentrale Komponente des TrustNoJit (TNJ)-Projekts, das die Korrektheit von Just-In-Time (JIT)-kompiliertem Code durch den Proof-Carrying Code (PCC)-Ansatz sicherstellen möchte. Durch die Übersetzung komplexer AArch64-Instruktionen in eine minimalistische, auf Verifikation ausgerichtete IR vereinfacht diese Arbeit den Verifikationsprozess und ermöglicht eine effizientere und präzisere Analyse von Maschinencode.

Der Lifter unterstützt mehr als 99.7 % aller in der Sightglass-Benchmarking-Suite enthaltenen Instruktionen. Der Quellcode ist in einem öffentlich zugänglichen Github-Repository verfügbar [1].

[1] https://github.com/TUM-DSE/aarch64-air-lifter

Contents

1 Introduction

Computers execute programs using a set of fundamental operations known as an instruction set. The instruction set defines the basic commands a processor can understand, such as loading data from memory, performing arithmetic, and controlling program flow. Different processor architectures have their own instruction sets, influencing how software is written and optimized for them.

The AArch64 instruction set, a 64-bit extension of the ARM architecture, is designed as part of the Reduced Instruction Set Computer (RISC) family. As such, the instruction set prioritizes a smaller set of simpler instructions rather than many complex ones, aiming for efficiency and streamlined execution. However, AArch64 introduces considerable complexity through implementation-defined behaviors, diverse addressing modes, an extensive set of over 300 base instructions, and more. These intricacies pose significant challenges for program verification and analysis. Developers and researchers must navigate complex control flow paths, account for implicit instruction side effects, and design verification policies that accurately reflect AArch64's nuanced instruction semantics. Understanding and addressing these challenges is crucial for ensuring software correctness and security in AArch64-based systems.

To address this, we propose a verification-focused Intermediate Representation (IR), Assembly Intermediate Representation (AIR), and corresponding lifter that translates AArch64 machine code into this simplified representation. While alternative lifters exist, our solution integrates well into the TrustNoJit (TNJ) project, which focuses on machine code verification. This custom approach enables precise adaptation to verification requirements.

Our solution implements a two-pass lifter in Rust that transforms AArch64 machine code into AIR. We test our lifter using binaries we compiled to AArch64. Despite current limitations with specific instruction types (compare-and-swap, authentication, dynamic jumps), the lifter successfully processes more than 99.7 % of all instructions within the Sightglass benchmarking suite.

The primary contributions include:

- A verification-oriented intermediate representation for AArch64
- An efficient machine code lifter integrated with TNJ
- Demonstrated real-world applicability through benchmark testing

2 Background

2.1 AArch64

AArch64 instructions are uniformly 32 bits long and can access 31 general-purpose 64-bit registers [1]. Consider this AArch64 assembly example in listing 1.

```
ADD X1, X1, X0 // Add X0 to X1, store in X1
LDR W10, [X1] // Load 32-bit value from address in X1
```

Listing 1: AArch64 Example

Register names start with 'X' for 64-bit or 'W' for 32-bit operations, followed by a number (0-30) identifying one of the general-purpose registers. AArch64 implements operations involving 'X' and 'W' on the same 31 general-purpose registers. Operations using 'W' registers utilize the lower 32 bytes of a register and usually clear the upper 32 unused bytes of a register. An instruction in AArch64 follows the same structure: The first operand specifies the destination register. Memory addresses are specified within square brackets, as shown in the load instruction above. As shown subsequently, we can address memory in a multitude of ways.

Memory Addressing

AArch64 supports five different types of memory addressing modes.

1. Register addressing: `R0, [R1]`. The processor uses the register value `R1` as the address.
2. Pre-index addressing: `R0, [R1, R2]`. The processor uses the sum of register `R1` and `R2` as the address.
3. Pre-index addressing with a shift operation: `R0, [R1, R2, LSL #2]`. The operator first applies the specified shift operation to register `R2`, which is, in this case, a logical left shift by two. To retrieve the memory address, the processor adds the result of the shift operation to our first operand, register `R1`.

4. Pre-indexed addressing with write-back: `R0, [R1, #32]!`. The operator calculates the address by adding the offset `32` to the register value `R1`. After accessing the memory location, the processor updates the register `R1` by adding the offset value `32`.

5. Post-indexed addressing with write-back: `R0, [R1], #32`. The processor uses the register value `R1` as the memory address. After access, the processor updates the value of `R1` by adding the offset to the register `R1`.

Flags

The AArch64 architecture includes four flag registers:

- **N** (Negative) – Set to 1 if the result of an operation is negative.
- **Z** (Zero) – Set to 1 if the result of an operation is zero.
- **C** (Carry) – Set to 1 if the operation produces a carry.
- **V** (Overflow) – Set to 1 if the operation results in an overflow.

AArch64 uses flag registers to enable conditional execution, as in instructions like `B.EQ`. Additionally, AArch64 employs flag registers in certain arithmetical operations such as `ADC`, where the processor adds the carry flag to the result of the sum of two registers. Flag registers are set during run-time using specific comparison operations and using the result of certain operations like `ANDS` or `SUBS`. However, accessing flag registers directly in AArch64 is impossible without workarounds.

2.2 AIR

An IR is a program representation between source and target languages that serves to simplify complex instructions into basic operations, provide architecture independence for analysis passes, enable program optimization and transformation, and, as in our case to establish a foundation for formal verification of code properties [14].

AIR is an IR specifically designed for assembly code verification. It uses the Static Single Assignment (SSA) form, where each variable has exactly one definition point. This property simplifies analysis by making data flow relationships explicit and enabling more powerful optimizations [21]. AIR is designed to be as minimal as possible: Each instruction performs a single operation (e.g., no complex addressing modes, flags set explicitly, no high-level control flow, no function calls . . .).

2.2.1 Design

Even though AIR is a RISC-like instruction set, choosing what instructions AIR includes, is a challenge: A larger AIR instruction set will increase the effort in proving correctness. If we choose a smaller instruction set, we must translate instructions in the source language using a lot of AIR-instructions that might affect performance. In addition, a small AIR-instruction set may worsen code comprehension. The challenge, thus, is to find the right balance.

The AIR instruction set contains 34 instructions as listed in table 2.1. AIR supports the following types:

- Integers of size 8-bit, 16-bit, 32-bit, 64-bit, 128-bit denoted as I8, I16, I32, I64, I128. We refer to all integer types using `INT` as an abbreviation. We exclude certain integer types from `INT` using set notation as in `INT\{I128}`.
- Boolean values which we refer to as `BOOL`
- BasicBlocks (BBs) that we use to translate conditional or jump-instructions. A BB is a sequence of instructions that executes linearly from a single entry point to a single exit point, with no branching or jumping between its instructions.
- Comparison types as `CMPTY` when specifying what kind of comparison we want to execute.

AIR also contains an `opaque` function, which creates an opaque value to specify an unknown value. We use this opaque value to mark certain registers as unknown in certain operations. This opaque value can assume all integer and boolean values.

Figure 2.1 showcases an example of how translated AIR code looks like.

```
ldr x0, [x1, #0x80]
```

(a) A64 code

```
v0 = i64.read_reg "x1"
v1 = i64.add v0, 0x80
v2 = i64.load v1
i64.write_reg v2, "x0"
```

(b) Equivalent AIR-code

Figure 2.1: Comparison of AArch64 to AIR code

AIR Instruction	Description	Input Types	Output Types
opaque	Create an opaque value	-	INT
ret	Return from a function	-	-
trap	Unconditionally trap	-	-
trapif	Trap if condition is true	BOOL	-
invalidate_regs	Write an opaque value into all registers	-	-
icmp	Compare two operands	INT, INT, CMPTY	BOOL
jump	Jump unconditionally	BB	-
jumpif	Jump conditionally	BB, BB, BOOL	-
dynamic_jump	Jump to an address	I64	-
read_reg	Read from a register	REG	INT
write_reg	Write to a register	INT, REG	-
load	Load from memory	I64	INT
store	Store to memory	INT, I64	
zext	Zero-extend value	INT/{I128}	INT/{I8}
sext	Sign-extend value	INT/{I128}	INT/{I8}
bitwise_not	Bitwise NOT	INT	INT
highest_set_bit	Find the highest set bit in a value	INT	INT
reverse_bits	Reverse bits	INT	INT
add	Add two integers	INT, INT	INT
sub	Subtract two integers	INT, INT	INT
lshl	Logical shift left	INT, INT	INT
lshr	Logical shift right	INT, INT	INT
ashl	Arithmetic shift left	INT, INT	INT
ashr	Arithmetic shift right	INT, INT	INT
ror	Rotate a value	INT, INT	INT
and	Logical AND	INT, INT	INT
or	Logical OR	INT, INT	INT
xor	Logical XOR	INT, INT	INT
imul	Signed multiply	INT, INT	INT
umul	Unsigned multiply	INT, INT	INT
idiv	Signed divide	INT, INT	INT
udiv	Unsigned divide	INT, INT	INT
modulo	Signed modulo	INT, INT	INT

Table 2.1: Overview of AIR instructions, their descriptions, input and output types.

3 Motivation

When we browse the web, our browser runs JavaScript code that the Just-In-Time (JIT) compiler compiles on the fly. A JIT compiler functions like an interpreter but with a key advantage: it detects frequently accessed code, or "hot spots," and translates them into machine code. This method provides fast execution in dynamic environments where ahead-of-time compilation is not an option while ensuring a quick startup time.

Even though this allows for a faster execution, this comes at a cost to security: In 2023, 9 out of 19 in-the-wild zero-day bugs attacking browsers were in JS engines that all employ JIT-compilation [15]. From 2019 to 2021, 45% of Google Chrome's JavaScript engine V8's exploits were JIT exploits [17].

One approach to make JIT-compilers safer utilizes the Proof-Carrying Code (PCC)-approach [16]. As demonstrated in fig. 3.1, PCC forces code producers to submit a formal safety proof ensuring compliance with a security policy defined by the code consumer in addition to the produced code. A verifier verifies the produced code using the submitted proof and defined security policy. We aim to apply the PCC-approach to JIT-compilers in a framework we named TNJ. This thesis focuses on implementing a module within TNJ: the 'lifter'. The lifter simplifies the proof-generation process in TNJ by translating AArch64 assembly code into a custom IR, which we call AIR. We refer to this translation process as 'lifting'. The alternative to lifting AArch64 machine code into AIR is submitting machine code to the verifier, which requires the verifier to process machine code of different architectures. Implementing the lifter thus provides the following advantages for TNJ:

- **Simplification of TNJ's verifier:** To support different architectures, the verifier only needs to support AIR because lifters first lift machine code into AIR before passing it to the verifier.
- **Simplification of TNJ's policy-writing process** Policy writers do not need to know platform-specific details such as special instructions, conventions, or addressing modes of source languages to create policies. Instead, policy writers can define rules on AIR, designed to be simple and platform agnostic.

We depict the final workflow of TNJ in fig. 3.2. The lifter receives machine code from a JIT-compiler. The lifter then translates the machine code into AIR-code and submits

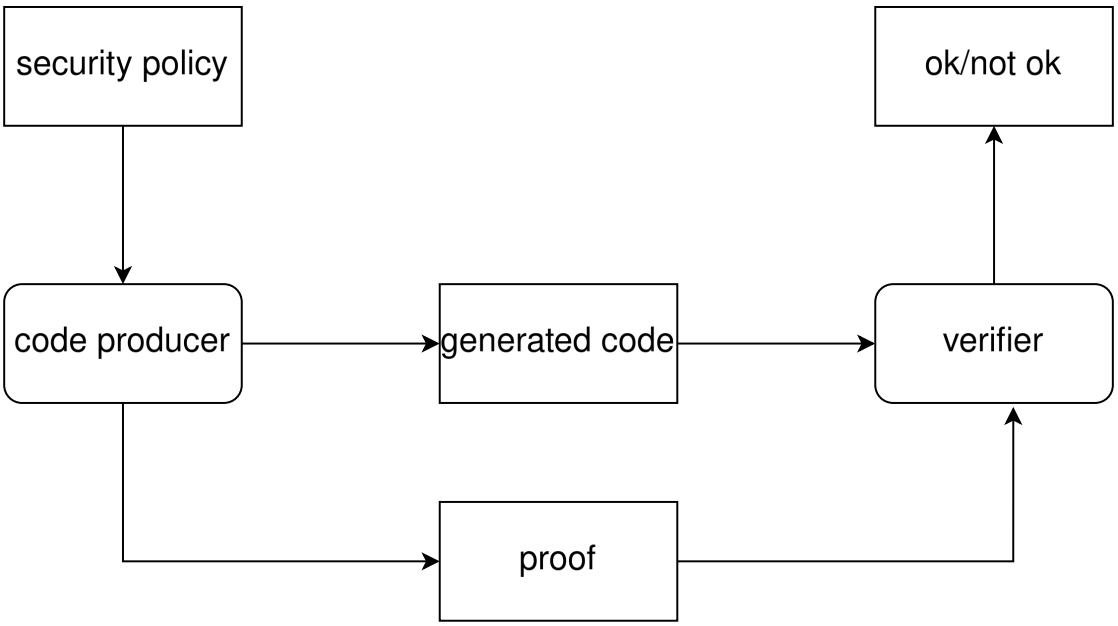


Figure 3.1: Workflow of PCC

it to the verifier module. The verifier evaluates the proof against user-defined policies to determine whether it complies with the security requirements and rejects or accepts accordingly.

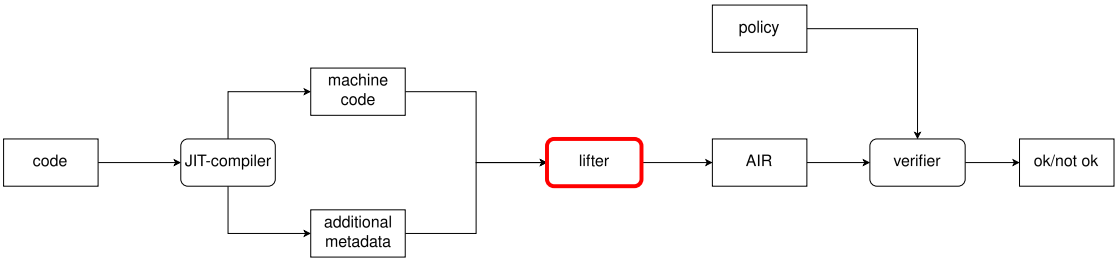


Figure 3.2: The workflow of TNJ

4 Design

This chapter discusses what design alternatives we considered and how we implemented the chosen design.

4.1 Lifter Implementation Design Choices

We considered two implementation strategies for our binary lifter from AArch64 assembly to AIR:

1. manually translating each AArch64 instruction
2. using ARM's Machine Readable Architecture Specification (MRAS) for AArch64 [18]

Manual Translation The first strategy involves directly translating individual AArch64 instructions into the appropriate instructions within the AIR instruction set. This direct approach offers several advantages: it provides a more straightforward implementation path and allows for precise handling of edge cases that might arise during translation. In addition, it is a well-studied approach that offers much quicker prototyping.

Translation using the Machine Readable Architecture Specification The alternative approach leverages ARM's Machine Readable Architecture Specification (MRAS). The MRAS system provides a detailed, machine-readable pseudocode that precisely describes the functionality of each ARM instruction in terms of lower-level primitive operations. With this approach, the lifter would be generated automatically using the MRAS specifications by a 'transpiler'. The advantages are:

1. Enhanced reliability through formally verified MRAS definitions
2. Potential reusability for Instruction Set Architecture (ISA)s of other ARM architectures
3. Support for automated build processes responding to MRAS updates, though less critical given AArch64's maturity since its 2011 release

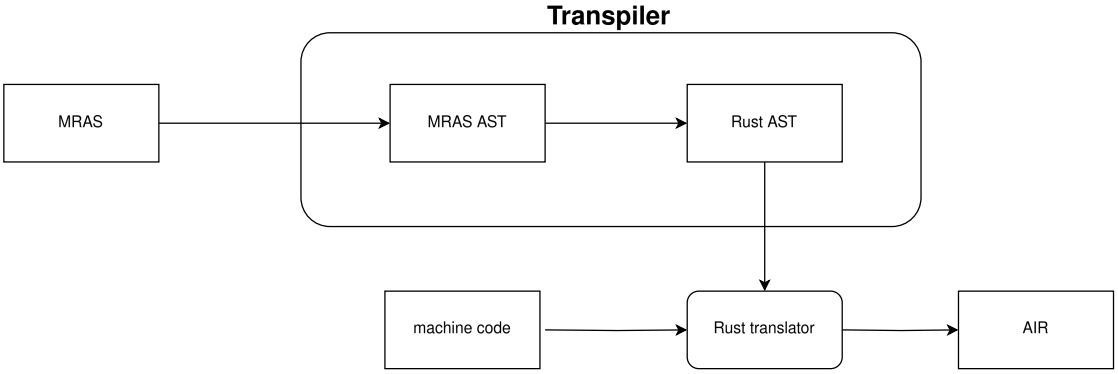


Figure 4.1: MRAS-transpiler-design approach

A transpiler, which translates between source-code languages, follows compiler design principles. Creating this transpiler requires three key steps: parsing the MRAS into an Abstract Syntax Tree (AST), converting this AST into a Rust AST, and finally generating a Rust-lifter from the Rust-AST [13]. We depict the overall workflow in fig. 4.1. While tools exist for the initial MRAS to MRAS-AST conversion, implementing the remaining steps demands substantial engineering effort that exceeds manual translation requirements. In addition, successful completion within project time constraints cannot be guaranteed.

Lastly, a fundamental limitation arises: the AIR-code generated by the transpiled MRAS would necessarily mirror the MRAS structure for each feature, potentially producing inefficient and unreadable code. The following example illustrates this challenge with the "BitCount" function, used to define AArch64-instructions like CNT:

```
integer BitCount(bits(N) x)
    integer result = 0;
    for i = 0 to N-1
        if x<i> == '1' then
        result = result + 1;
    return result;
```

Listing 4.1: MRAS code that is challenging to translate

Automated translation of this code would generate excessive AIR instructions since AIR lacks native loop support. It requires implementing iterations through unconditional jumps and additional basic blocks. In contrast, manual translation enables direct semantic capture by extending the AIR instruction set with a specialized `count_bits` operation. Afterward, we can use this operation in function definitions requiring the `BitCount` functionality. These efficiency constraints and the ability to optimize through

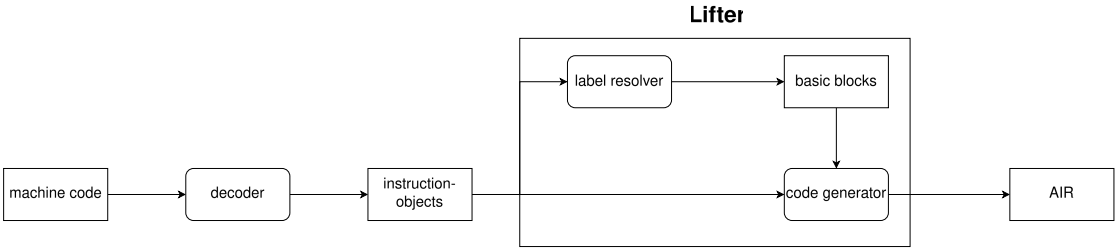


Figure 4.2: AArch64 to AIR Translation Pipeline Design

manual semantic mapping ultimately justified choosing a manual translation approach over automation.

4.2 Manual Lifter Design

We divide our lifter into disassembly, label resolution, and code generation as seen in fig. 4.2. During disassembly, we parse machine code into instruction objects in a higher-level language for further processing. Similar to a two-pass compiler, we then scan through all instruction objects twice. The first pass focuses on preprocessing and mapping out jumps. The second pass is the actual AIR-generation step, where we translate instruction objects into equivalent AIR-instructions.

4.2.1 Disassembly

For our first step, we must convert machine code into higher-level-language objects. We do not require this step for correctness purposes but because it simplifies AIR-code generation during label resolution and code generation. It does not cost additional development time because we can leverage a third-party library that already implements machine-code parsing [11].

4.2.2 Label Resolution

Code structure falls into different categories based on the management of control flow:

- **Unstructured code** (e.g., assembly) lacks enforced organization. Execution flows freely, with jumps capable of targeting any instruction, making control flow unpredictable and harder to analyze.

- **Semi-structured code** (e.g., AIR) imposes some restrictions while retaining flexibility. It groups instructions into BBs and limits jumps to BB entry points, ensuring a more controlled flow.

This distinction has two key implications for AArch64 translation:

- **AIR code is structured into BBs,** where each block executes sequentially, unlike assembly's loose sequence of instructions.
- **AIR restricts jumps to BB entry points,** preventing arbitrary branches and making control flow more manageable than in assembly.

Due to this change in code structure, the lifter must handle three types of instructions in AArch64 machine code differently:

Sequential Instructions Unlike jumps or branches, sequential instructions do not alter the control flow. The lifter can translate these instructions without relying on conditional structures like if-statements. Examples include basic operations like `add` and `mov`.

Jump Instructions As the name suggests, jump instructions alter the control flow in the original AArch64 code. Examples of jump instructions include `b` and `tbz`. To correctly translate jump instructions in AArch64, we must: *(i)* determine their target locations, and *(ii)* ensure that every jump destination aligns with the start of a BB.

Conditional Instructions Conditional instructions are instructions that contain an internal if-statement. An example of this is the `ccmp` instruction. The `ccmp` instruction checks condition flags and subsequently updates the condition flags either to an immediate value or to the result of a comparison. To implement this in AIR, we create two distinct BBs, each corresponding to one of the possible outcomes of the conditional check. Control flow must resume at the next translated AArch64 instruction regardless of what the if-condition evaluates. In other words, the last AIR-instruction in both BBs is an unconditional jump to a BB containing the next instruction. To achieve this, we must ensure that the AArch64 instruction following `ccmp` aligns with the start of a BB.

Implications As previously mentioned, the lifter groups all AIR instructions into BBs, which AIR jump instructions can target. Suppose the lifter processes the AArch64 machine code in a single pass. In that case, the lifter cannot determine during translation whether another AArch64 instruction branches to the current one, especially since jumps

can be backward. However, this information is crucial for deciding to which BB the lifter assigns individual AIR instructions.

Because of this, the lifter first performs an initial pass over the machine code to identify the target locations of all jump instructions. This information allows the lifter to generate BBs for the second pass.

For further details on the implementation of jump handling, refer to section 5.1.

4.2.3 Code Generation

During the last step of the lifting process, we create AIR-code. We iterate through all instruction objects and map these to equivalent AIR-instructions and BBs that we created during label resolution. Because AIR-instructions are simple, the lifter generally translates AArch64 instructions into multiple AIR-instructions.

5 Implementation

We implement the lifter in Rust to ensure seamless integration with TNJ. We develop the project using `rustc` 1.84.0-nightly. As of the date of writing (February 12, 2025), the lifter comprises 11,029 Lines of Code (LOC) of Rust code and has seven dependencies. [1] The lifter currently supports 129 base instructions from the AArch64 core instruction set. Additionally, we have implemented a workaround for the lifting of all SIMD and floating-point instructions that minimizes impact on memory verification when generating AIR code as described in section 5.3.

During the implementation of the lifter, we encountered several challenges, which we discuss in detail below.

5.1 Jump Instructions

When a jump instruction targets an address in the original code, we must ensure that the corresponding jump in AIR correctly lands at the beginning of the translated instruction sequence for that address. To maintain this precise mapping between original machine code addresses to their corresponding AIR instructions, we construct BBs according to two rules. BB are containers for instructions that instructions in AIR can jump to. We create a new BB:

1. After every jump instruction
2. At every address that serves as a potential jump target

For clarity and maintainability, we assign each BB a name based on the address of its first instruction. For example, if the first translated instruction of a BB corresponds to address `4` in the original code, we name that BB `block_4`. Table 5.1 illustrates this translation process, emphasizing how the lifter handles jumping instructions. To aid readability, we slightly shortened the AIR-code.

Thus, the lifter performs two passes through machine code, similar to a two-pass compiler architecture. The first pass identifies and creates the necessary BBs for jump instructions, while the second pass performs the actual instruction translation.

[1] `https://github.com/TUM-DSE/aarch64-air-lifter`

AArch64 code	Equivalent AIR code

`_start:`
`add x1, x1, x1`
	`[...]`
	`v37 = i64.read_reg "x1"`
	`v38 = i64.read_reg "x1"`
	`v39 = i64.add v37, v38`
	`i64.write_reg v39, "x1"`
	`jump block_4`

`_label:`
`add x2, x2, x2`
	`block_4:`
	`v40 = i64.read_reg "x2"`
	`v41 = i64.read_reg "x2"`
	`v42 = i64.add v40, v41`
	`i64.write_reg v42, "x2"`

`b _start`	`jump block_4`

`add x1, x1, x1`	`block_12:`
	`v43 = i64.read_reg "x1"`
	`v44 = i64.read_reg "x1"`
	`v45 = i64.add v43, v44`
	`i64.write_reg v45, "x1"`

Table 5.1: Comparison of AArch64 instructions and their corresponding translated AIR code, highlighting jump instruction handling.

This two-pass approach ensures that all required BB structures are available during translation.

5.2 Register Sizes

Although AArch64 is a 64-bit architecture, some instructions operate on 128-bit values. One example is `SMULH`, as shown in table 5.2, which multiplies two 64-bit values, producing a 128-bit result. Since the result is truncated and stored in a 64-bit register, we extended AIR to support 128-bit values to accurately represent such operations.

AArch64 code	**Translated AIR code**

`smulh x1, x2, x3`	`v0 = i64.read_reg "x2"`
	`v1 = i64.read_reg "x3"`
	`v2 = i64.imul v0, v1`
	`v3 = i128.ashr v2, 0x40`
	`i64.write_reg v3, "x1"`

Table 5.2: Comparison of `SMULH` in AArch64 and AIR

5.3 SIMD and FP Instructions

Supporting all SIMD and FP operations in AIR would require substantial effort on two fronts: modifying the lifter to handle vector registers and translate all AArch64 SIMD and FP operations into AIR instructions, while also extending AIR itself—potentially complicating verification procedures. A more practical approach is to simply bypass instructions that operate exclusively on vector registers, avoiding these complexities altogether.

This approach is feasible because TNJ's primary goal is to verify memory safety. Values stored in vector registers are generally irrelevant in this context, as they cannot be used as memory addresses. Instead, we only need to ensure that when a value from a vector register is written to a general-purpose register, it is treated as an unknown value. To achieve this, we generate an `opaque` value and assign it to the destination register. For example, the `ADDV` instruction, which sums all elements of a vector and stores the result in a general-purpose register, is handled by marking the general-purpose destination register as unknown.

5.4 Processor Flag Implementation

A key design decision in our lifter concerns the representation of processor flags, as they are essential for preserving control flow semantics and instruction dependencies during the transformation to AIR. We evaluate two distinct approaches:

1. A consolidated approach using an 8-byte value, where we encode the Null, Zero, Carry and Overflow (NZCV) flags within the first 4 bits. This design offers:
   - Atomic updates for all flags simultaneously
   - Reduced AIR instruction count when implementing flag register writes
   - More compact representation in memory
2. Individual AIR registers dedicated to each flag, providing:
   - Direct access to individual flags without masking operations
   - Improved code readability and maintenance
   - Simplified debugging and verification processes

While the consolidated approach (Option 1) offers potential performance benefits through atomic operations and a reduced instruction count, we ultimately implement the individual register approach (Option 2). This decision prioritizes code clarity and maintainability over minor performance gains. The key factor is eliminating the need for complex bit masking operations when accessing individual flags, which would be necessary with the consolidated approach. This simplification makes the code more readable and reduces the potential for errors in flag manipulation operations.

The individual register approach also aligns better with our goal of creating verifiable code, as it makes the state of each flag explicitly visible and independently trackable throughout program execution.

5.5 Call Instructions

AArch64 provides two main types of branching instructions: *(i) static branching*, where the instruction itself encodes the target address within its 4-byte format and *(ii) dynamic branching*, where a general-purpose register holds the target address.

Handling static branches is straightforward. Since our lifter operates via static code analysis, it can determine jump destinations at translation time, ensuring accurate code conversion. The lifter achieves this using a first 'label resolution' pass through the code where the lifter detects jump targets and creates BBs. However, dynamic branches pose

a challenge, as we lack access to runtime register values and the execution environment of the assembly code.

This issue is particularly problematic for the `blr` instruction. Similar to a function call, `blr` jumps to a subroutine whose address is stored in a register while also saving the return address (`PC + 4`) in `X30`. Since our lifter cannot access runtime register values, we cannot resolve the jump destination. Moreover, register values may have changed after returning, making it essential to account for potential state modifications. Beyond `blr`, hypervisor, supervisor, and system calls also introduce uncertainty, as they may modify registers unpredictably.

To address this, we introduce the `invalidate_regs` instruction in AIR to explicitly mark registers as potentially modified. Whenever the lifter encounters a function that might modify AArch64 registers to an unknown extent, the lifter will add the invalidate_registers to the translated AIR code. `invalidate_regs` is equivalent to assigning an `opaque` value to all general-purpose and flag registers. However, writing opaque values to all registers one by one generates excessive AIR instructions. This may cause a significant slowdown, so the lifter uses `invalidate_regs` instead.

6 Evaluation

6.1 Research Questions

The goal of this thesis is to create a usable lifter. As such, we must answer two research questions to evaluate the success.

RQ1: Is the lifter usable on real-world-applications? Initially, TNJ's main focus will be WebAssembly (WASM)-applications compiled to AArch64. Thus, we analyze to what degree the lifter can successfully lift these binaries. In addition to that, we analyze what kind of applications the lifter is not able to translate.

RQ2: Is the lifter fast enough for real-world-applications? We conduct performance tests on benchmark AArch64 binaries. We compare the lifter's performance with another popular tool used on binaries, `objdump`, to assess the viability of the lifter. Lastly, we analyze the factors influencing the lifter's performance.

6.2 Methodology

6.2.1 Experimental Setup

We conduct all benchmark tests on Ubuntu 22.04.05 using a system equipped with an AMD Ryzen 5 5500U CPU and 16 GiB of RAM. We execute all benchmark tests with five warmup runs and 20 measured test runs to ensure accurate and reliable results.

For our benchmark tests, we used WASM applications from the Sightglass benchmark suite [22] that we compiled at optimization level O2 using cranelift on `aarch64-apple-darwin`. We use the default cranelift settings for compilation on AArch64 MacOS, as outlined below:

```
has_lse
has_pauth
sign_return_address
sign_return_address_with_bkey
```

6.3 Benchmarking

6.3.1 Instructions

We measure the lifter using three key metrics for each instruction:

1. Execution time: Duration required to lift a code segment containing only one instance of the benchmarked instruction (see fig. 6.4).
2. AIR instruction count: Number of AIR instructions generated per AArch64 instruction (see fig. 6.1).
3. Block count: Number of blocks (containers containing instructions) generated during translation (see fig. 6.2).

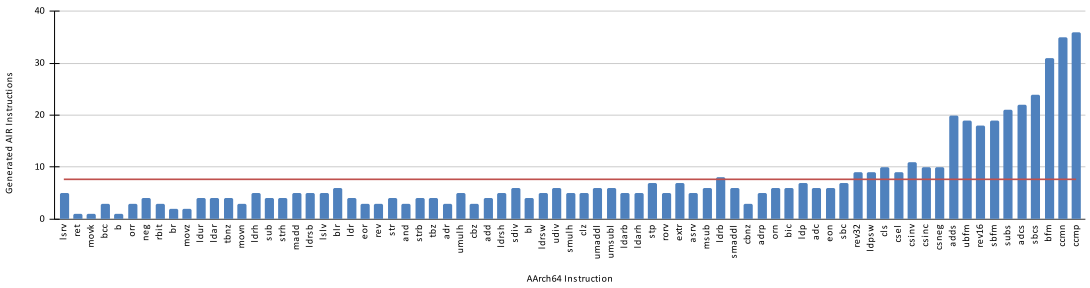


Figure 6.1: Generated AIR instructions per AArch64 instruction

We detect a general trend: If the lifter needs to generate more instructions, the lifter also needs longer for the entire translation process. The correlation coefficient or r-value between generated AIR instructions and execution time supports this, which reaches as high as 0.95 on a scale from -1 to 1. A 0.95 correlation coefficient suggests a strong positive linear relationship, where -1 represents a perfect negative correlation, and 1 denotes a perfect positive correlation.

Two categories of instructions seem to exhibit higher execution times consistently:

- Complex instructions (e.g., `bfm`, `sbfm` and similar instructions)
- Flag-setting instructions (e.g., `adcs`, `ccmp` and similar instructions)

The longer execution times for complex operations align with expectations, as they require translation into multiple AIR instructions. Flag-setting instructions are particularly time-intensive because they must explicitly set each of the four flag registers, resulting in multiple AIR instructions per flag update.

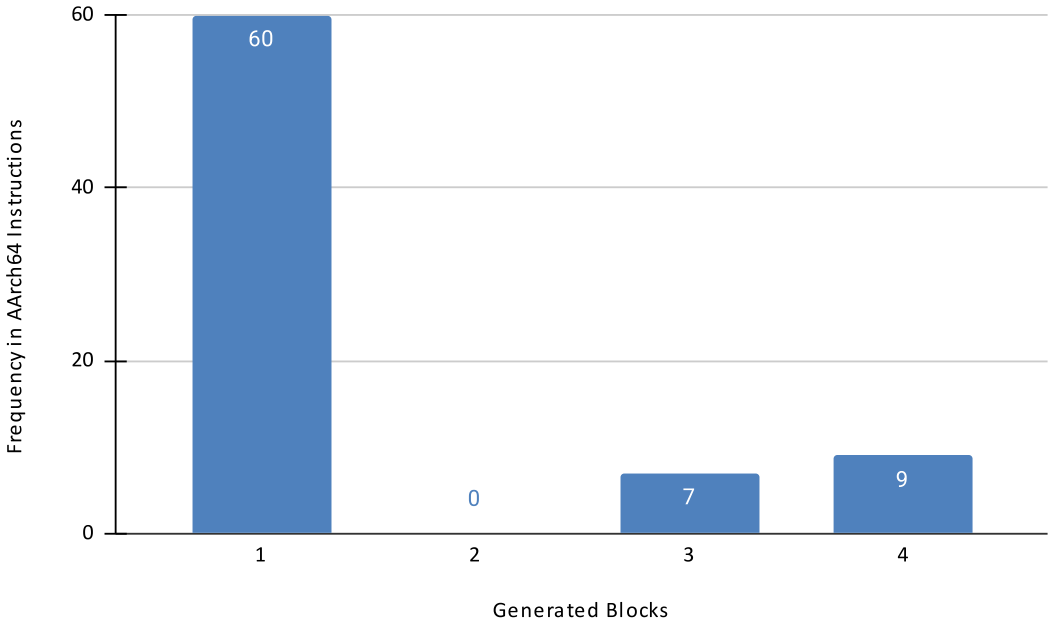


Figure 6.2: Frequency of generated BB per AArch64 instruction

6.3.2 WASM-Applications

To test the lifter in a real-world environment, we lift the `.text` segment of all WASM-applications of the Sightglass benchmark suite [22]. We lifted 9 out of 31 tested WASM applications without any errors, achieving a success rate of approximately 29 %. For the remaining 21 applications, we encountered decoding issues due to limitations in the `yaxpeax` decoding package we rely on, which is still under development and does not yet support all instructions in the machine code. For example, `yaxpeax` does not support all instructions from ARM's Scalable Vector Extension (SVE) or Scalable Matrix Extension (SME). To understand the lifter's performance on real-world applications better, we assess the lifter using additional metrics across all nine WASM applications from the Sightglass benchmark that the lifter successfully processed without errors: `blake3-scalar`, `blind-sig`, `noop`, `pulldown`, `regex`, `shootout-ed25519`, `shootout-gimli`, `shootout-keccak`, and `shootout-minicsv`. We measure two key metrics for each application: (1) the time required to lift the `.text` segment of the WASM application and (2) the time taken to execute `objdump` on the same application.

We compare the lifter with `objdump` because it is a widely used tool for examining Executable and Linkable Format (ELF) files and disassembly, making it a familiar reference point. Since many developers have prior experience with `objdump`, this comparison offers a more intuitive understanding of performance differences. However, unlike `objdump`, which disassembles and prints the code, the lifter performs additional processing. It not only disassembles the code but also analyzes it in two passes. We present the results in fig. 6.3.

6.3.3 Performance Analysis

Across all applications, the lifter took approximately 13 times longer than `objdump`. Nevertheless, as demonstrated in fig. 6.3, the lifter's performance varies significantly across applications compared to `objdump` 's execution time. The lifting process takes between 2 and 63 times longer than `objdump` 's disassembly of the same application. Given this thesis's objective of developing a fast lifter, we analyze the code to identify potential bottlenecks and their underlying causes.

Segment Sizes A WASM-application comprises various sections: type, data, function, and others. Since only the code section is relevant to the lifter's operation, we exclusively process this segment of all WASM-applications. While `objdump` 's performance correlates with the total WASM-application size, the lifter's performance correlates only with the code segment's size. A substantial difference between code segment and total size could explain performance variations. However, comparing lifted assembly code

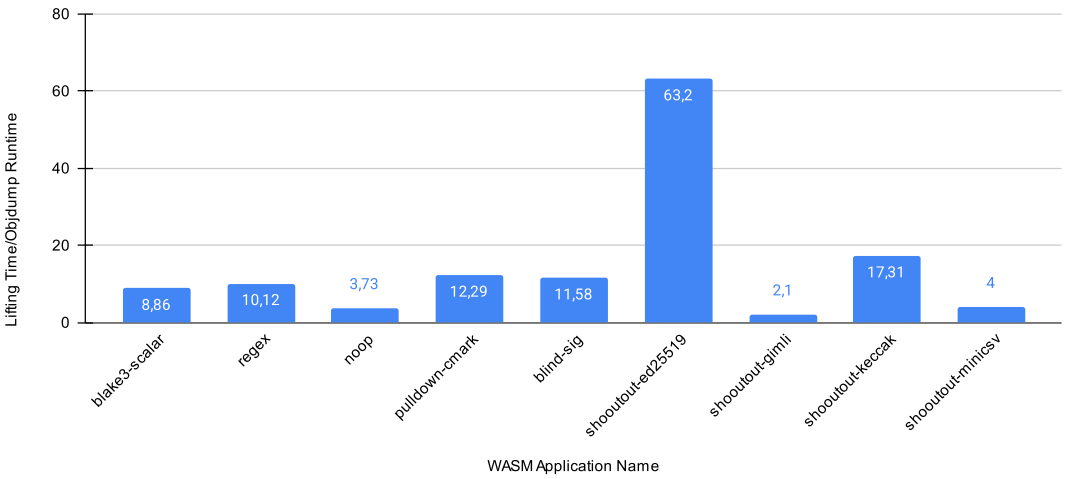


Figure 6.3: Comparison of Lifting to `objdump` Execution Time

lines to total assembly code lines revealed that the code segment size was nearly identical to the total application size in most cases. The ratio of code-segment instructions to total instructions exceeded 99.6 % across all applications, suggesting this factor had minimal impact on performance.

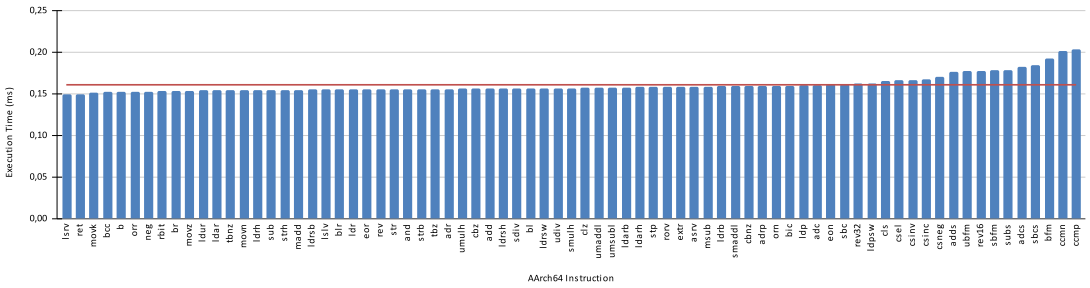


Figure 6.4: Execution Time per AArch64 instruction

Instruction Frequency As illustrated in fig. 6.4, execution time varies among different instructions. When lifting machine code containing single instances of each instruction, the fastest instruction executes in less than 75 % of the time required for the slowest instruction. The measured execution time includes the lifter's startup time, which begins when we execute the test command. This startup time remains constant,

independent of individual instruction lifting time. After subtracting this startup time from all execution times, the fastest instruction executes in 5 % of the time required for the slowest instruction unlike 75 % as before. We applied these insights to analyze the execution time of our nine successfully lifted WASM-applications.

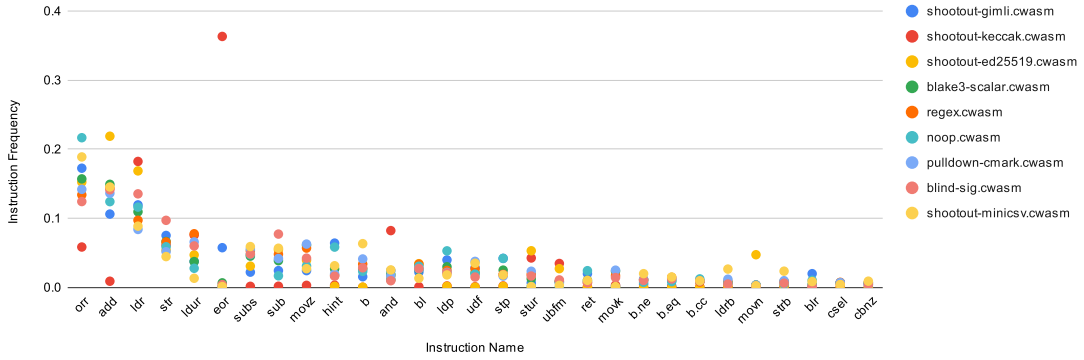


Figure 6.5: Instruction frequency of the 30 most popular instructions in WASM-applications

Figure 6.5 illustrates the frequency distribution of instructions within each application, as instruction type significantly influences execution time. Instructions such as `mov`, `ldr`, `add`, and `str` appear frequently across all WASM-applications. However, certain instructions show application-specific prevalence: `eor` in `shootout-keccak.cwasm`, `add` in `shoout-ed25519.cwasm`, and `and` in `shootout-keccak.cwasm`. As discussed in section 6.3.1, an AIR-instruction's execution time strongly correlates with the number of generated AIR instructions. Thus, the average number of generated AIR-instructions per original machine code instruction should explain the significant variations in application lifting times.

Figure 6.6 largely confirms this correlation: `shooutout-ed25519.cwasm`, which performed worst compared to `objdump`, generates the most AIR-instructions per AArch64 instruction. Conversely, `shootout-gimli.cwasm` and `noop.cwasm`, which performed best compared to `objdump`, generated the fewest AIR instructions per AArch64 instruction. One anomaly exists: despite generating relatively few AIR-instructions per AArch64-instruction, `shootout-keccak.csv` performs poorly compared to `objdump`. We attribute this to the average number of generated BBs per AArch64-instruction, shown in fig. 6.7. `shootout-keccak.csv` generates the fewest BBs among all applications, resulting in larger individual BBs. Since we implement AIR-instruction addition to BBs using Rust-vectors, larger BBs may reduce overall execution speed. While Rust vectors typically guarantee $\mathcal{O}(1)$ for pushes, the worst-case scenario requires $\mathcal{O}(capacity)$ time

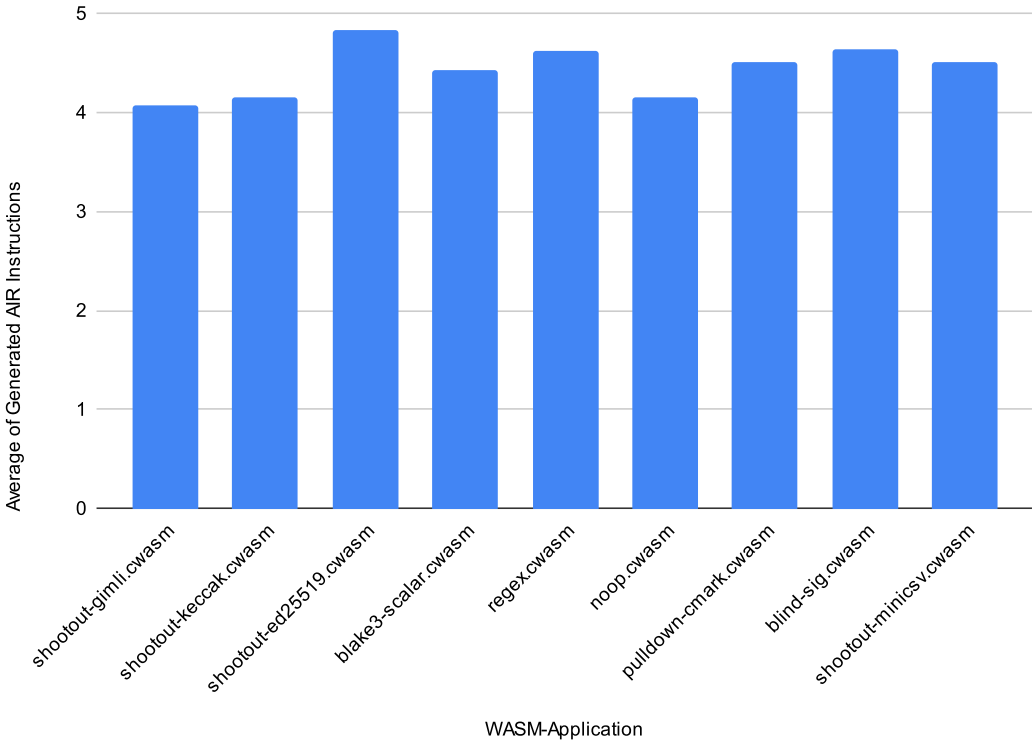


Figure 6.6: Average of generated AIR-instructions per AArch64 instruction

when vector growth occurs. Even though the amortized push complexity is $\mathcal{O}(1)$, large BB might have impacted performance. Additionally, the superior performance of `shootout-gimli.cwasm` and `noop.cwasm` likely benefits from their substantially smaller LOC compared to other applications, resulting in more compact data structures.

6.4 Research Question Evaluation

RQ1: Is the lifter usable for real-world applications? The lifter currently supports all non-feature-flag base instructions (129 out of 301) as well as SIMD and FP instructions. These instructions account for more than 99 % of every application within the Sightglass benchmark. On average, the lifter lifts more than 99.7 % of every application in the benchmarking suite.

However, the in-development `yaxpeax` decoding package the lifter uses, which does not yet support SME or SVE instructions, constrains its capabilities. This limitation causes the lifter to lift only 9 out of 31 applications in the Sightglass benchmark suite. However, all code segments successfully decoded the lifter lifts without errors.

RQ2: Is the lifter fast enough for real-world applications? As previously demonstrated, the lifter's performance varies significantly depending on the instruction types. Our results show that disassembling applications with `objdump` is generally 10 to 15 times faster than lifting the code with our tool. This discrepancy is unsurprising, as `objdump` merely disassembles and prints machine code while the lifter performs two passes over the code and executes additional computations.

On average, lifting a single instruction takes approximately 20 µs, with the worst-case scenario (`shootout-ed25519`) reaching 80 µs. Given these low translation times, the lifter is unlikely to become a bottleneck when used in conjunction with the TNJ JIT machine code verification program.

Since JIT compilers process significantly fewer LOC than the complete applications tested here, the lifter can likely operate efficiently in parallel with the JIT compiler without impacting overall performance. However, we can only verify this once TNJ is finished.

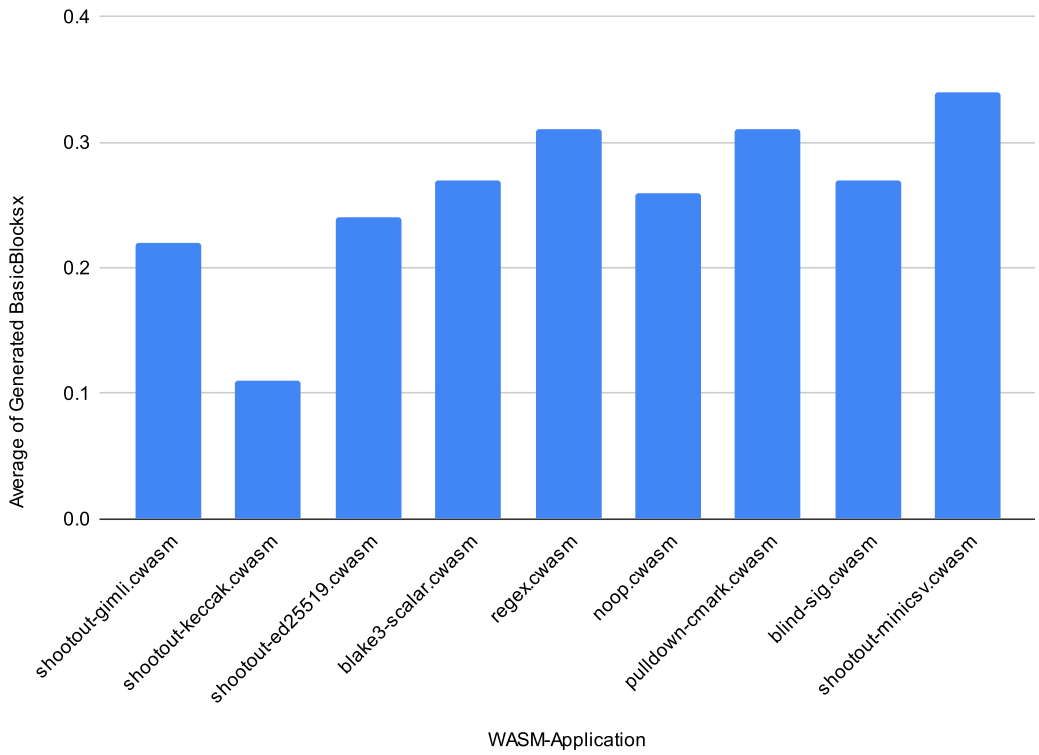


Figure 6.7: Average of generated BB per AArch64 instruction

7 Related Work

Binary lifters are essential to binary translation frameworks, enabling machine code conversion from one architecture to another. These lifters serve as an intermediary step, translating code into an intermediate representation (IR) before generating the target architecture's machine code.

Several lifters target established IRs such as **LLVM**'s [12] IR [8, 5, 7, 19, 20] and **QEMU**'s IR[1] [23, 10]. Other frameworks define their own custom IRs, such as **BinRec**, **BAP**, and **Alive2** [3, 6, 2]. Notably, some tools, such as **Xbyak**, bypass an IR entirely, performing direct instruction translation [9].

Multiple lifters support AArch64 code: **Forklift** lifts code from `x86`, **ARM**, and **RISC-V** architectures into LLVM IR using a token-level encoder-decoder transformer. While Forklift supports AArch64 and offers broad compatibility, it does not guarantee semantic equivalence and has limitations in translation size [5].

Other lifters, such as **Remill** [7] and **Instrew** [8], also translate AArch64 code into LLVM IR. However, these lifters are suboptimal for our purpose due to limitations inherent to LLVM IR. Specifically, the size and complexity of LLVM IR make verification challenging.

While workarounds exist, like in the case of **Alive2**, where the lifter lifts LLVM IR into an IR designed for verification, chaining multiple dependencies reduces reliability and increases complexity.

Beyond LLVM-based solutions, several tools lift AArch64 code into IRs explicitly designed for verification:

- **angr** lifts AArch64 into the VEX IR [4].
- **Miasm** lifts AArch64 into its own Miasm IR.
- **BAP** lifts AArch64 into its own BAP IR.

While these tools provide robust and mature solutions, we opted against their use for two main reasons:

1. **Integration with TNJ**: Our lifter is specifically designed for TNJ. A smooth integration into TNJ will allow for a seamless verification pipeline.

[1] `https://www.qemu.org/docs/master/devel/index-tcg.html`

2. **Control and Customization**: The design of existing IRs encompasses a broad range of use cases, even in the case of IRs designed for verification. By developing a custom lifter and IR, we can tailor the system to specific needs for JIT-compiled code.

8 Summary

In this thesis, we developed a minimal Intermediate Representation, called Assembly Intermediate Representation (AIR), designed for representing assembly code. This work is part of the larger TNJ project, which aims to verify the correctness of JIT-compiled code. In addition to that, we developed a binary lifter that translates AArch64 machine code into AIR.

We evaluated the lifter using the Sightglass benchmarking suite, a collection of applications designed to test the capabilities of binary analysis tools. On average, the lifter supports more than 99.7 % of all instructions across the entire suite and more than 99 % of instructions for any individual application within the suite.

However, the lifter successfully lifts only 9 out of the 31 applications. The lifter cannot fully lift the remaining 22 applications because the `yaxpeax` decoding package, which is still under development, has limitations. Nevertheless, the lifter successfully lifts every decodable WASM application.

To promote transparency and collaboration, we have made the source code for the lifter publicly available on GitHub: `https://github.com/TUM-DSE/aarch64-air-lifter`. We encourage researchers and developers to explore, use, and contribute to the project.

9 Future Work

Our analysis has identified three key areas for improvement in the lifter's development:

Lifting Performance Optimization As demonstrated in chapter 2, the lifter's performance primarily depends on the type of processed instruction. AArch64 instructions that require extensive translation—those generating numerous AIR instructions—significantly impact lifting speed. To enhance performance, we propose two strategies: First, expanding AIR 's instruction set could reduce the total number of AIR instructions needed for translation. Second, our analysis revealed that a low BB count in the final AIR code can degrade performance due to the current data structure implementation. Possible solutions are redesigning the underlying data structures or implementing algorithms that generate more balanced BB distributions.

Extended Instruction Set Support The lifter currently achieves complete translation of WASM-applications containing only base instructions. However, it may fail to fully translate applications utilizing advanced instruction sets. Future development should focus on implementing support for:

- Memory-ordering operations
- Authentication instructions
- Additional specialized feature sets
- Indirect jumps

Additionally, further development on the yaxpeax decoding package will improve the lifter's functionality and applicability.

Verification While we have conducted thorough testing, we cannot guarantee absolute correctness due to the manual verification process. Therefore, additional testing and verification would further strengthen the lifter's reliability.

Abbreviations

AIR Assembly Intermediate Representation

IR Intermediate Representation

JIT Just-In-Time

PCC Proof-Carrying Code

TNJ TrustNoJit

BB BasicBlock

MRAS Machine Readable Architecture Specification

NZCV Null, Zero, Carry and Overflow

SSA Static Single Assignment

AST Abstract Syntax Tree

ISA Instruction Set Architecture

WASM WebAssembly

SIMD Single Instruction, Multiple Data

ELF Executable and Linkable Format

LOC Lines of Code

RISC Reduced Instruction Set Computer

SVE Scalable Vector Extension

SME Scalable Matrix Extension

FP Floating Point

List of Figures

List of Tables

Bibliography

[1] *A64 Instruction Set Architecture*. ARM. URL: https://developer.arm.com/Architectures/A64%20Instruction%20Set%20Architecture.

[2] AliveToolkit. *alive2*. 2025. URL: https://github.com/AliveToolkit/alive2.

[3] A. Altinay, J. Nash, T. Kroes, P. Rajasekaran, D. Zhou, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, C. Giuffrida, H. Bos, and M. Franz. "BinRec: dynamic binary lifting and recompilation." In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387550. URL: https://doi.org/10.1145/3342195.3387550.

[4] *angr*. 2025. URL: https://github.com/angr/angr.

[5] J. Armengol-Estapé, R. C. Rocha, J. Woodruff, P. Minervini, and M. F. O'Boyle. "Forklift: An Extensible Neural Lifter." In: *arXiv preprint arXiv:2404.16041* (2024).

[6] BinaryAnalysisPlatform. *bap*. 2025. URL: https://github.com/BinaryAnalysisPlatform/bap.

[7] T. of Bits. *remill*. 2025. URL: https://github.com/lifting-bits/remill.

[8] A. Engelke, D. Okwieka, and M. Schulz. "Efficient LLVM-based dynamic binary translation." In: *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE 2021. Virtual, USA: Association for Computing Machinery, 2021, pp. 165–171. ISBN: 9781450383943. DOI: 10.1145/3453933.3454022. URL: https://doi.org/10.1145/3453933.3454022.

[9] Fujitsu. *xbyak_translator_aarch64*. 2022. URL: https://github.com/fujitsu/xbyak_translator_aarch64.

[10] R. Gouicem, D. Sprokholt, J. Ruehl, R. C. O. Rocha, T. Spink, S. Chakraborty, and P. Bhatotia. "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.

[11] iximeow. URL: https://github.com/iximeow/yaxpeax-x86.

[12] C. A. Lattner. "LLVM: An infrastructure for multi-stage optimization." In: (2002). URL: https://llvm.org/.

[13] *Compiler Architecture*. Loyola Marymount University. URL: https://cs.lmu.edu/~ray/notes/compilerarchitecture/.

[14] *Intermediate Representations*. Loyola Marymount University. URL: https://cs.lmu.edu/~ray/notes/ir/.

[15] J. S. Maddie Stone and J. Sadowski. *We're All in this Together*. Tech. rep. Google, 2024.

[16] G. C. Necula. "Proof-carrying code." In: *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1997, pp. 106–119.

[17] J. Norman. *Super Duper Secure Mode*. Tech. rep. Microsoft, 2021.

[18] A. Reid. "Trustworthy specifications of ARM® v8-A and v8-M system level architecture." In: *2016 Formal Methods in Computer-Aided Design (FMCAD)*. 2016, pp. 161–168. DOI: 10.1109/FMCAD.2016.7886675.

[19] repzret. *Dagger*. 2017. URL: https://github.com/repzret/dagger.

[20] R. C. O. Rocha, D. Sprokholt, M. Fink, R. Gouicem, T. Spink, S. Chakraborty, and P. Bhatotia. "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.

[21] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. "Global value numbers and redundant computations." In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '88. San Diego, California, USA: Association for Computing Machinery, 1988, pp. 12–27. ISBN: 0897912527. DOI: 10.1145/73560.73562. URL: https://doi.org/10.1145/73560.73562.

[22] ByteCodeAlliance. URL: https://github.com/bytecodealliance/sightglass.

[23] *QEMU*. 2025. URL: https://www.qemu.org/docs/master/devel/tcg.html.