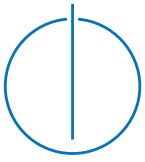
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Translating x86 Binaries to LLVM Intermediate Representation

Martin Fink



DEPARTMENT OF INFORMATICS

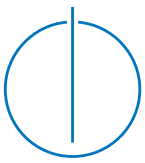
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Translating x86 Binaries to LLVM Intermediate Representation

Übersetzen von x86-Binärdateien in LLVM-Zwischendarstellung

Author:	Martin Fink
Supervisor:	Prof. Dr. Pramod Bhatotia
Advisor:	Dr. Rodrigo Rocha, Dr. Redha Gouicem
Submission Date:	November 15, 2021



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

Munich, November 15, 2021 Martin Fink

Acknowledgments

I would like to thank the following people for helping me with this thesis:
My supervisor Prof. Pramod Bhatotia as well as my advisors Dr. Rodrigo Rocha and Dr. Redha Gouicem for the possibility to work on this interesting project and their support and guidance throughout it. S. Bharadwaj Yadavalli and Aaron Smith for taking their time to review my pull requests and contributions to MCTOLL. My friends Andreas, David, and Simon, as well as my sister Lena for proof-reading my work.

Abstract

With new CPU architectures such as ARM or RISC-V gaining popularity, achieving application support is a problem hindering their adaption. Existing applications need to be adapted, compiled, and distributed for the new architecture, which is not always possible. Static binary translation is a concept that can help port binaries to new architectures without introducing significant runtime overhead. These translators act similar to a compiler in the sense that they process and translate binaries ahead of time. In this thesis, we try to improve and extend the capabilities of the existing binary lifter MCTOLL. It gradually recovers abstraction levels from binaries and produces LLVM bitcode, which can then be compiled to all architectures supported by LLVM.

To test our implementation, we compare native programs to their translated counterparts by measuring runtime performance and code size of the phoenix-2.0 benchmark.

The results show that static binary translators are able to generate very efficient code with the help of optimizers such as LLVM. However, they do have some limitations and cannot process binaries with certain properties, which limits their use cases.

Zusammenfassung

Mit dem zunehmenden Aufschwung von neuen Prozessorarchitekturen wie ARM oder RISC-V stellt die mangelnde Kompatibilität bestehender Anwendungen mit diesen Architekturen ein Problem dar. Programme müssen an die neue Architektur angepasst, neu kompiliert und an die Benutzer verteilt werden, was nicht immer möglich ist. Statische Übersetzung von Binärdateien ist ein Ansatz, Programme auf neue Architekturen zu portieren, ohne erheblich an Laufzeit einzubüßen. Diese Übersetzer verhalten sich ähnlich wie klassische Compiler indem sie Programme verarbeiten, bevor sie ausgeführt werden. In dieser Arbeit versuchen wir die Fähigkeiten des bestehenden Übsersetzers MCTOLL zu verbessern und erweitern. MCTOLL versucht schrittweise Abstraktionsebenen des ursprünglichen Programmcodes zurückzugewinnen und erzeugt LLVM Bitcode, welcher von LLVM dann für alle unterstützten Zielplattformen kompiliert werden kann.

Um die Ergebnisse unserer Arbeit zu testen, vergleichen wir native Programme mit den übersetzten und portierten Gegenstücken. Wir messen die Laufzeit und Programmgröße des phoenix-2.0 Benchmarks. Die Ergebnisse zeigen, dass statische Binärübersetzer mit der Hilfe von Codeoptimierern wie LLVM in der Lage sind, sehr effizienten Code zu erzeugen. Es gibt jedoch Einschränkungen in der Anwendbarkeit, da diese Typen von Binärübersetzern nicht jede Art von Programm verarbeiten können.

Contents

1 Introduction

During this thesis we have worked on expanding the capabilities of the MCTOLL (see Section 2.5) static binary translator, mainly on the x86 (see Section 2.3) frontend. In Chapter 2 we will describe MCTOLL, in Chapter 3 similiar projects, their capabilities and limitations. Chapter 4 describes our contributions to the project, in Chapter 5 we will discuss the impact and performance of our implementation.

1.1 Motivation

Today's CPU architecture is not a homogeneous field. While x86 still remains the most common instruction set architecture (ISA) in use in desktop computers, laptops, and servers, new architectures such as ARM and RISC-V are continuously being developed and are gaining market share. ARM historically was mainly used in the mobile computing world, seeing widespread adoption in smartphones and low-powered devices such as Raspberry Pis. In recent years, manufacturers in the desktop and laptop computing world have been shifting their focus, as ARM-based CPUs potentially offer more performance using less power than comparable x86 chips from Intel or AMD.

The transistion from one architecture to another results in compatibility issues with existing binaries where re-compilation is not possible. If the source code for an application is available, the preferred solution is to compile it to the target architecture directly, as this will produce the most efficient result, both in runtime and binary size. If the source code is not available, or it has been developed with specific assumptions about the processor architecture in mind, binary translation offers a possibility to port legacy binaries to a new architecture.

Another potential use case we discovered during this project is re-optimization. This poses the possibility to improve performance for legacy binaries by leveraging new processor extensions and improved compiler passes or optimize binaries for different goals. Legacy binaries might not make use of vector extensions on modern CPUs if the program has been written and compiled before such extensions were widely available.

1.2 Binary Translation Approaches

There are two possible approaches to binary translation, both of which we'll discuss below.

1.2.1 Dynamic Binary Translation

Dynamic binary translation is the approach which is widely used today when programs compiled for a different ISA need to be run and which is used by projects such as QEMU [Bel05] and Apple's Rosetta 2. They usually work by translating code on demand, often on a per-basic block basis. This method has the possibility to run a wide range of programs. Issues like indirect jumps, that pose problems for static binary translation, are solved at runtime. The downside which dynamic binary translation suffers from is performance. Translation needs to be done at runtime and there is not much possibility to optimize the translated code, as only the current basic block is available.

1.2.2 Static Binary Translation

Static binary translation performs the translation ahead of time, raising and recompiling programs before they are executed on the target machine. This has the advantage of not needing to translate a program everytime it is run, increasing performance similar to how compiled languages typically have a lower runtime overhead compared to interpreted languages. Another benefit is that static binary translators raise whole functions and reconstruct the program's control flow graph, offering the possibility to apply existing compiler optimizations. These optimizations typically get rid of unused expressions generated while raising a CPU instruction (e.g. calculation of CPU flags that will not be used), and apply optimizations that are only possible when the control flow graph is available.

However, static binary translation is not able to raise indirect branches as the jump targets are not known before running the program.

2 Background

In this chapter we will discuss the technical background required for the later sections. We will describe the architecture of MCTOLL in Section 2.5 and its limitations in Section 2.5.5.

2.1 Control Flow Graph

A control flow graph (CFG) [All70] is a graph representation of a program where the nodes represent basic blocks, while edges represent jumps.

Basic Blocks A basic block is a segment of code without any jumps or jump targets where jump targets start a block while jumps end a block. Figure 2.1 is an example CFG with an entry node (1), an exit node (6) and a loop (nodes 2, 3, 4, 5).

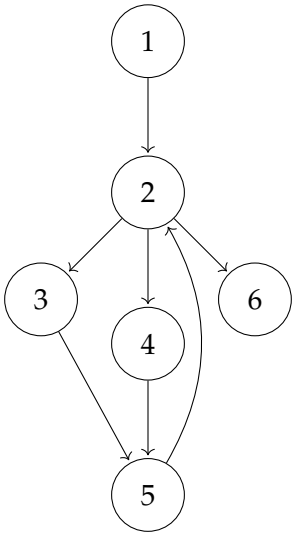


Figure 2.1: Example of a control flow graph

2.1.1 Dominator Trees

In a CFG a node n dominates a node p if every path from the entry node to p must go through n [Pro59].

Additionally, the following definitions hold:

- A node n *strictly dominates* a node p if n dominates p and $n \neq p$.
- A node n *immediately dominates* a node p if n strictly dominates p but there exists no node n' that strictly dominates p.
- A dominator tree is a directed graph where the children of a node n are the nodes that n immediately dominates.

Dominator trees can be derived from CFGs as shown by Lowry and Medlock in [LM69].

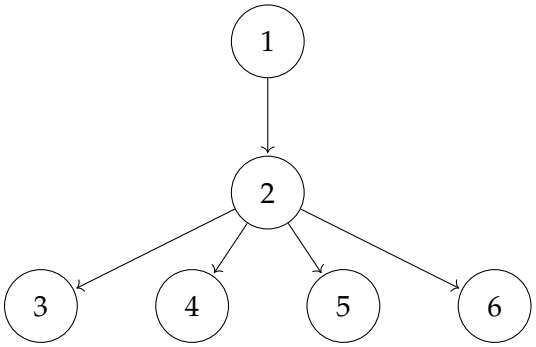


Figure 2.2: Dominator tree from the CFG in Figure 2.1

2.2 LLVM

LLVM is a set of compiler and toolchain technologies, designed around an intermediate representation (IR) [LLV]. LLVM was started as a research project at the University of Illinois. The IR serves as a portable high-level assembly language, which can be targeted by compiler frontends such as clang[1], rustc[2], swiftc[3], and others. LLVM transforms and optimizes the intermediate representation with a set of compiler passes before passing it on to LLVM's codegen, which compiles the IR to the target's machine code [LA04].

[1] `https://clang.llvm.org`
[2] `https://www.rust-lang.org`
[3] `https://swift.org`

$$\mathbb{P}_{\mathsf{C}} \xrightarrow{\mathsf{clang}} \mathbb{P}_{\mathsf{IR}} \xrightarrow{\mathsf{LLVM.opt}} \mathbb{P}'_{\mathsf{IR}} \xrightarrow{\mathsf{LLVM.codegen}} \mathbb{P}_{\mathsf{target}} \tag{2.1}$$

Figure 2.3: Workflow of clang and LLVM

2.2.1 LLVM Intermediate Representation

LLVM IR is in single static assignment (SSA) form, used by many intermediate languages to simplify compiler optimizations. SSA is a property of a language which requires that each variable is assigned exactly once and defined before it is used by subsequent instructions. Constant propagation, dead code elimination, and register allocation are some compiler optimizations that are enabled by the SSA form [RWZ88].

In LLVM IR, each expression needs to be type-annotated. LLVM natively supports arbitrary-sized integers, pointers, tuples, single- and double-precision floating-point numbers, and vectors. These type annotations allow LLVM to perform some optimizations that would not be possible with a traditional three-address code.

```
%0 = i32 5
%1 = add i32 %0, i32 2
```

Listing 2.1: Example of LLVM IR addding 5 + 2

2.2.2 Intrinsic Functions

Intrinsic functions (or built-in functions) are functions that are available for use by the programmer but implemented in the compiler itself. Depending on the target, the compiler may insert a series of instructions in place of the function call or call a function in some library. This behaviour is opaque to the user.

In LLVM some essential operations such as addition with overflow or square roots are implemented as intrinsic functions. On platforms such as x86, these are then compiled to a single CPU instruction, while they might be implemented in software on other platforms.

2.3 x86_64

x86_64 is an extension to x86 designed by AMD and used by the main desktop chip manufacturers Intel and AMD. x86_64 is a complex instruction set computer (CISC) architecture that builds on x86 in a backwards-compatible way, supporting all 32 bit instructions while adding an array of features. The most important ones are the

expansion of general purpose registers to 64 bits (see Figure 2.11), the addition of new general purpose registers and the possibility to address an extended 64 bit address space.

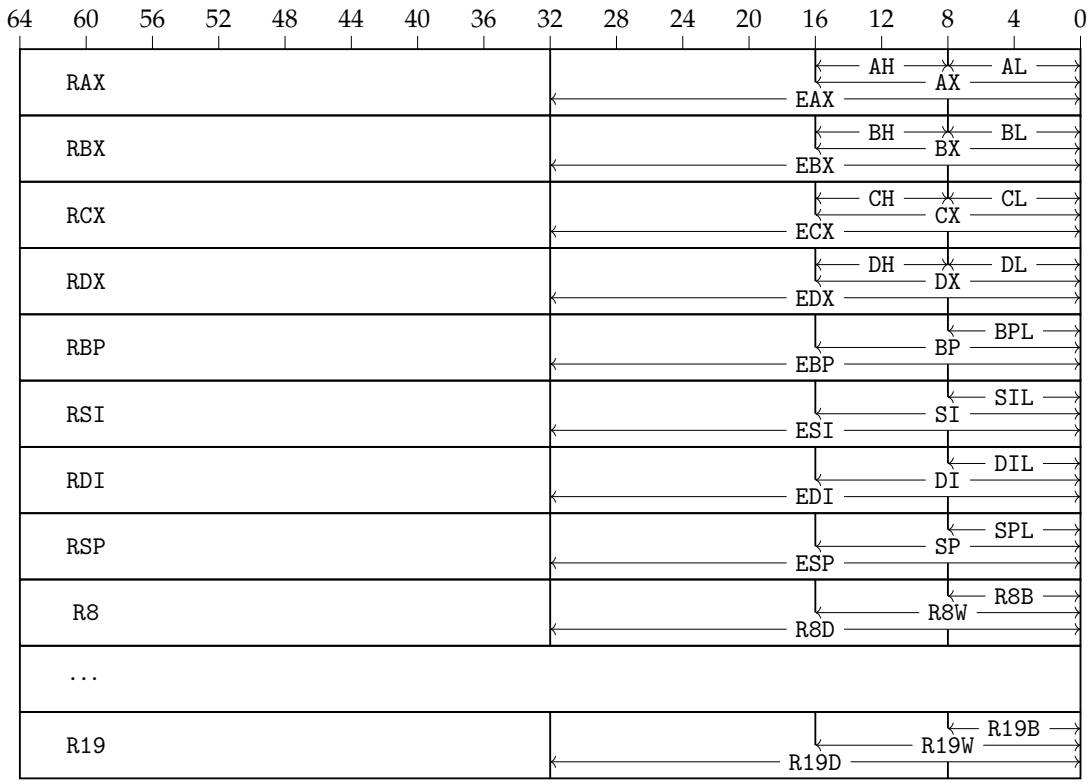


Figure 2.4: x86_64 general purpose registers

2.3.1 SSE

Most modern CPUs feature some kind of single instruction multiple data (SIMD) instructions and registers. The most widely used SIMD extension for x86 is SSE (Streaming SIMD Extensions). Additionally, SSE is also utilized to perform floating-point operations. It uses the independent `XMM` registers (see Figure 2.5).

2.4 System-V ABI

System-V is an application binary interface (ABI) containing specifications for calling conventions, object file formats, executable file formats, dynamic linking semantics,

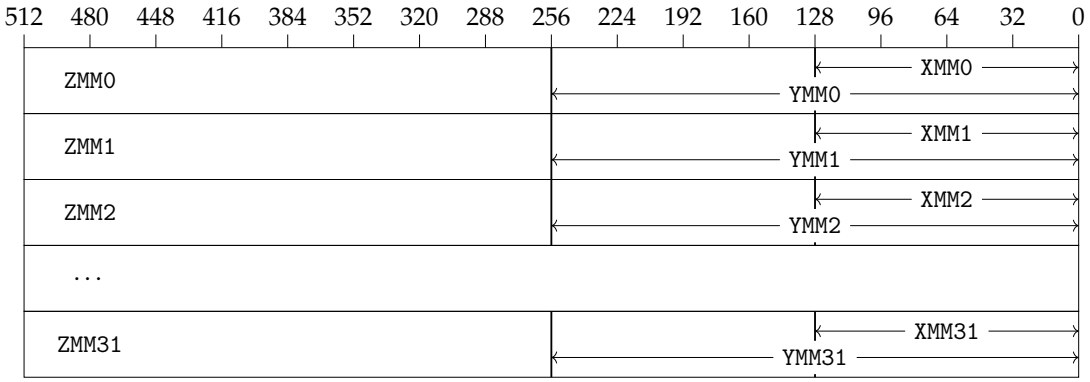


Figure 2.5: Vector registers introduced by SIMD extensions

and more for the Intel i386 and x86_64 architectures [Lu+]. System-V is the default ABI used by Unix-like operating systems such as Linux, BSD distributions, macOS[4], and others. In this thesis, we will focus on the x86_64 part of the specification with programs running in 64 bit mode.

2.4.1 System-V Calling Convention

The System-V calling convention describes how applications should call functions. This includes how parameters are passed, values are returned, which registers should be callee- or caller-saved, the layout of stack frames, and more. System-V groups parameters into different classes, depending on their type. We will only describe the most important ones for this project:

Integer consists of integral types that fit into one of the general purpose registers. This includes both pointers and integers.

SSE consists of types that fit into an SSE vector register. It includes floating-point values (`float` and `double` in C) and 128 bit vectors (packed `int`s, packed `float`s, ...).

Parameter Registers

System-V allows for up to six Integer and eight SSE parameters to be passed via registers, while additional ones will be spilled onto the stack. Table 2.1 lists all registers for each parameter.

[4]While macOS binaries follow the System-V calling convention, the system does not use other System-V features, such as the ELF format for executables, object code, and shared libraries. Instead, Apple's format is Mach-O [App].

Register Allocation The registers in Table 2.1 get assigned from left to right according to the following algorithm:

1. If the class is Integer, use the next available register of `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9`.
2. If the class is SSE, use the next available register of `XMM0-XMM7`.
3. If there are more arguments than argument registers, push the remaining arguments to the stack in reverse order.

Integer types				
#	i64	i32	i16	i8
1	RDI	EDI	DI	DIL
2	RSI	ESI	SI	SIL
3	RDX	EDX	DX	DL
4	RCX	ECX	CX	CL
5	R8	R8D	R8W	R8B
6	R9	R9D	R9W	R9B

#	SSE types
1	XMM0
2	XMM1
3	XMM2
4	XMM3
5	XMM4
6	XMM5
7	XMM6
8	XMM7

Table 2.1: System-V parameter registers

Vararg Functions

In C, a vararg function is a function with a variable amount of arguments. This is useful for functions such as `printf`, where the user is able to format a string with a number of parameters. The C specification does not provide any information about the number of variable parameters passed to a function, which is why this needs to be handled manually. `printf` does this by parsing the format string and determining the number of arguments according to that (see Figure 2.6). This also means that passing less parameters than required by the format string will introduce undefined behaviour into the program. The standard also defines no way to declare types of passed arguments. For `printf`, this is again done by parsing the format string and analyzing it.

```
int printf(char *format, ...);
printf("%d␣+␣%f␣=␣%f", 1, 2.5, 3.5);
```

Figure 2.6: Example of a vararg function

System-V requires a caller of a vararg function to specify the number of SSE registers used in a hidden argument passed in `AL`. This is shown in Figure 2.9.

2.4.2 Return Registers

While System-V supports up to two Integer and two SSE return values, we will focus on the more common single return value of a function. `RAX` and its subregisters are used as the return register for Integer types, while `XMM0` is used for SSE types.

Type	Register
`i8`	`AL`
`i16`	`AX`
`i32`	`EAX`
`i64`	`RAX`
`float`	`XMM0` (lower 32 bits only)
`double`	`XMM0` (lower 64 bits only)
Vector	`XMM0`

Table 2.2: System-V return registers

2.4.3 Examples of System-V Calls

Following are examples demonstrating how programs pass arguments using the System-V calling convention. Figures 2.7 to 2.8 show function calls with both Integer and SSE arguments, while Figure 2.9 shows a call to a vararg function.

```
long long add(int a, long long b);
add(2, 5ll);
```

(a) Function definition and call

Register	Value
`EDI`	2
`RSI`	5
`RAX`	Return value

(b) Arguments and return register used for the function call

Figure 2.7: Integer-only method

2.5 MCTOLL

MCTOLL is a static binary translator implemented as an LLVM tool. It leverages existing LLVM infrastructure and acts as a binary lifter, raising machine code to a higher abstraction level. Its role is similar to a compiler frontend, but instead of processing

```
double test(int a, float b,
  double c, long long d);
test(1, 1.5f, 3.0, 2ll);
```

(a) Function definition and call

Register	Value
EDI	1
RSI	2
XMM0	1.5
XMM1	3.0
XMM0	Return value

(b) Arguments and return register used for the function call

Figure 2.8: Mixed arguments

```
int printf(const char *format, ...);
printf("%f + %d = %f\n", 1.5, 1, 3.5);
```

(a) Function definition and call

Register	Value
AL	2
RDI	Pointer to string
EDI	1
XMM0	1.5
XMM1	3.5
EAX	Return value

(b) Arguments and return register used for the function call

Figure 2.9: Vararg arguments

high-level source code, MCTOLL recovers abstraction from low-level machine code. By producing LLVM bitcode, MCTOLL can use existing optimization passes executed by the LLVM optimizer. This is demonstrated in Figure 2.10.

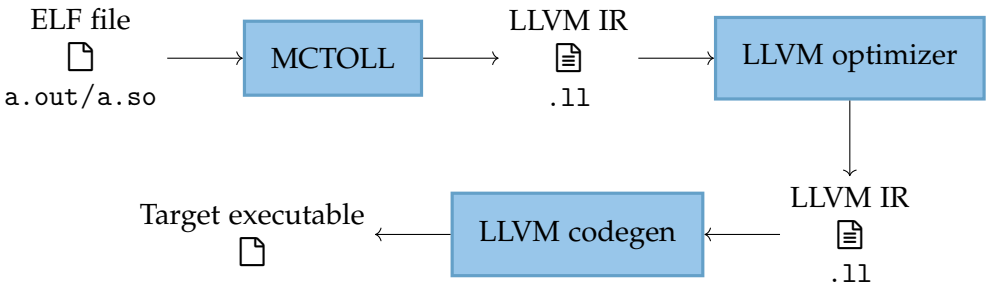


Figure 2.10: MCTOLL workflow

MCTOLL processes binaries on a function level. While this requires reconstructing the complete CFG, which is more effort than just raising on a block- or instruction level, the produced bitcode is much better suited for optimizations by LLVM.opt.

MCTOLL leverages data structures used in LLVM.codegen, gradually processing them and raising their level of abstraction. First, the source binary is disassembled to an array of `MCInsts`. The control flow graph is only constructed in the second step, after raising `MCInsts` to `MachineInstrs`. LLVM bitcode is then generated in four CFG walks:

1. discover function prototype (cf. Section 2.5.1),
2. discover jump tables,
3. raise non-terminator instructions,
4. raise terminator instructions.

The emitted bitcode can be compiled by clang to the target architecture.

In the following sections, we will dive deeper into some aspects of MCTOLL's raising process.

2.5.1 Discovering Function Prototypes

MCTOLL needs to discover function prototypes before raising instructions, as this pass provides information about which argument registers hold argument values. Additionally, subsequent passes need to know function prototypes when raising function calls to construct the LLVM `call` instruction.

The algorithm described in Section 2.5.1 always looks at super-registers. A super-register is the full-sized register, which the subregister is a part of. In Figure 2.11 all subregisters of `RAX` are visualized.

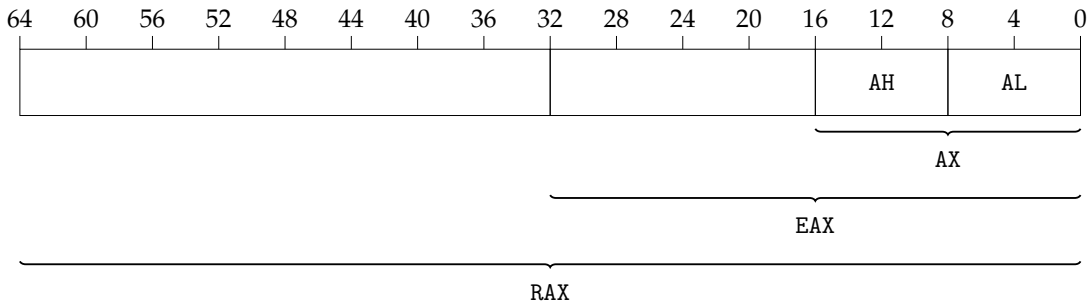


Figure 2.11: `RAX` with its subregisters

Terminology In the following sections we will use the following terminology:

Register definition A write to a register, for example: `mov eax, 0` defines `EAX`

Register usage An explicit or implicit read from a register, for example: `mov eax, edx` uses `EDX`

Instructions may define or use one or more registers implicitly. `idiv edi` has an explicit register usage of `EDI` and an implicit usage of `EAX`, and is an implicit register definition of `EDX` and `EAX`.

Parameter Discovery

To detect the parameters of a function, MCTOLL traverses the function's basic blocks via a depth-first search. Each register usage is then checked. If the register is one of the argument registers defined in Table 2.1 and has not been defined in the current or in all predecessor blocks, it is considered an argument. Since the CFG may contain loops, MCTOLL uses LLVM's `LoopTraversal` class, which traverses blocks that are part of CFG loops twice, to make sure each block's predecessor has been visited at least once.

For each basic block, MCTOLL creates a union of all registers defined in its predecessor blocks. If two blocks define the same super- but a different subregister, MCTOLL retains the greater one to ensure all incoming values fit the resulting type.

MCTOLL then begins checking the current basic block, iterating over all instructions, and checking each operand.

- If the operand is a register use, MCTOLL checks if a predecessor or a previous instruction in the current block defined the super-register. If that is not the case, the register should be considered as a potential argument register.
- If the operand is a register definition, MCTOLL saves it as defined in the current block. In this case, successive instructions and blocks will not consider the register as a potential argument register.

Once MCTOLL processed all basic blocks, it checks all potential argument registers. If the register is a valid argument register, MCTOLL looks up the number and type of the argument and inserts it into the function signature. If argument n is missing, but argument $n+1$ is defined, it is assumed that argument n was unused and optimized away. The type of the argument is considered to be `i64`.

Type Discovery

Values passed in general purpose registers are of integer type, with the width of the register as the width of the raised type (`EDI` $\rightarrow$ `i32`). This also means that MCTOLL raises parameters that were initially pointers to `i64`[5]. See Figure 2.12.

```
void test(int, char, char *) {
  ...
}
```

(a) Original code

```
define dso_local void @test(
  i32 %1, i8 %2, i64 %3) {
...
}
```

(b) Raised code

Figure 2.12: Discovered arguments passed in general purpose registers

Return Type Discovery

To discover the return type of a function, MCTOLL looks at the return blocks and checks if one of the two supported return registers `RAX` or `XMM0` has a reaching definition in all of them. A block is considered a return block if it ends with a return instruction, ignoring padding instructions. The algorithm to check if all return blocks have a reaching definition of a return register is as follows: A working list of basic blocks is initialized with the return blocks. For each block in the working list, its instructions are iterated in reverse order and checked.

[5] llvm-mctoll only supports raising 64 bit binaries.

- If the instruction defines a return register, the return register is also defined for the end of the block. Stop iterating over further instructions.
- If a call instruction is encountered, stop, as return registers are temporary and not preserved across function calls.
- If all instructions have been processed but no return register is defined, add the predecessors of the current block to the working list.

The return type of the function is set to that corresponding to the discovered return register. For functions where no return register has been found, the type is set to `void`. Should two return blocks return different-sized integers, the larger one is retained. This behaviour can be seen in Figure 2.13. Since the only return block `.exit` does not define the return register, we have to look at its predecessors `.bb.1` and `.bb.2`. They define different subregisters, `EAX` and `RAX` respectively. The return type is set to the larger `i64` instead of `i32`.

```
test:
  cmp edi, 0
  je .bb.2:
.bb.1:
  mov eax, 0
  jmp .exit
.bb.2:
  mov rax, 1
.exit:
  ret
```

(a) Original code

```
define dso_local i64 @test(i32 %1) {
  ; ...
}
```

(b) Raised code with detected return type

Figure 2.13: Return type detection example

2.5.2 Discovering Non-Terminator Instructions

During the second CFG walk, MCTOLL raises all non-terminator instructions. To keep track of values stored in registers, a register to SSA map is constructed for each function. This map is updated while raising basic blocks and keeps track of which LLVM value is stored in which register. Figure 2.14 shows an example of how this map might look while raising a program.

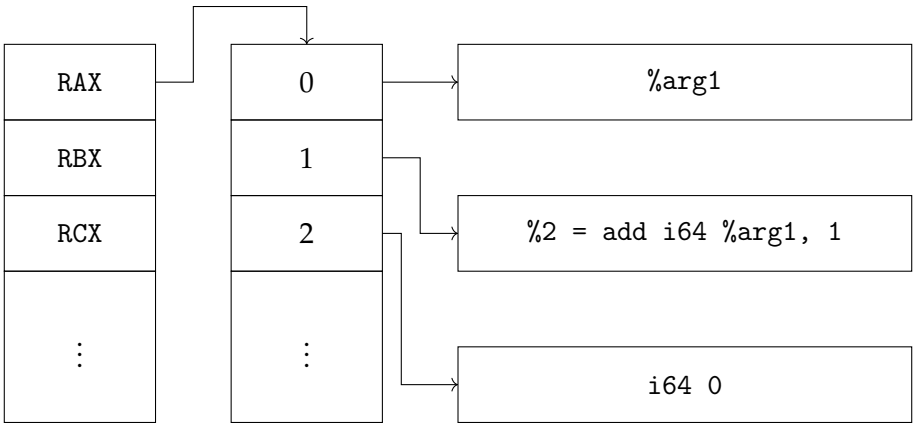


Figure 2.14: Register-SSA map keeping track of register values in different blocks

Each `MachineInstr` can be translated to zero, one, or more LLVM instructions. Register moves are not translated to an LLVM instruction, only the register-SSA map is updated. Instructions such as add operations, where a simple LLVM counterpart exists, translate directly to a single LLVM instruction. In addition, instructions that implicitly set processor status flags will result in more than one instruction in the raised bitcode. The value of the processor flags is also stored in the register-SSA map, where successive instructions will be able to access them. This can be seen in Figure 2.15, with the initial and updated register-SSA map shown in Figure 2.16. If successive instructions do not access generated instructions, LLVM will remove them in its optimization phase when generating code.

```
add edi, 1
```

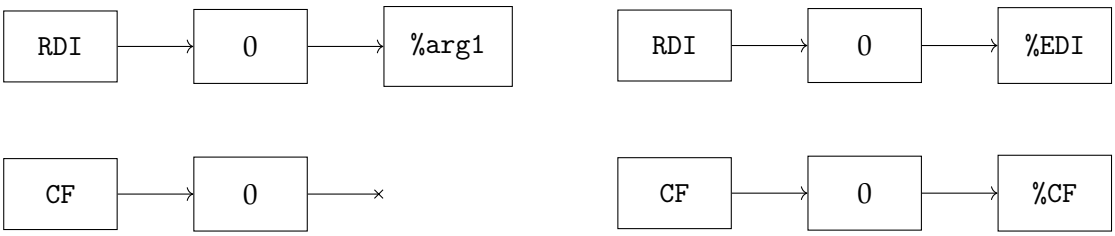
(a) Original code

```
%EDI = add i32 %arg1, 1
%0 = call { i32, i1 } @llvm.uadd.with.overflow.i32(i32 %arg1, i32 1)
%CF = extractvalue { i32, i1 } %0, 1
...
```

(b) Raised code

Figure 2.15: Raised add operation

Terminating instructions such as return or jump instructions are not yet raised in this pass, MCTOLL is gathering information about them to raise them in a subsequent pass.



(a) SSA map before raising `add` instruction (b) SSA map with new value for `RDI`

Figure 2.16: Register-SSA map before and after raising the `add` operation in Figure 2.15

Raising Function Calls

If an encountered instruction is a function call, MCTOLL first needs to look up the function prototype. Function prototypes for external functions need to be passed in the form of header files. For every argument, MCTOLL looks up the reaching value for the appropriate argument register. If there is no reaching value, it assumes the argument has been optimized and passes a constant 64 bit integer of zero.

MCTOLL can then construct the LLVM call instruction with the discovered arguments. If the function is not a `void` function, MCTOLL updates the register-SSA map for the return register and sets its value to that of the function call.

Vararg Calls If the called function is a vararg function, MCTOLL needs to check for additional arguments after the normal arguments. It does that by iterating over the remaining argument registers and checking for a reaching value for that register. If a reaching value exists, that value is added to the list of arguments. Otherwise, the code does not look for further arguments and constructs the function call.

2.5.3 Promoting Registers to Stack Slots

In Figure 2.17 we see an example where multiple values reach a block. In LLVM, there exists a `phi` instruction, that selects a value depending on which predecessor block was executed before entering the current block. MCTOLL cannot use this instruction, as not all predecessor are necessarily raised when processing the current block. As a solution, MCTOLL allocates a stack slot, where the value is stored in all predecessor blocks. This slot is read in the block where the value is needed. The code in Figure 2.17 is raised to the bitcode shown in Figure 2.18. If a predecessor block is not raised at the time when the stack slot for a register is created, it is marked as incomplete and processed after all blocks have been translated.

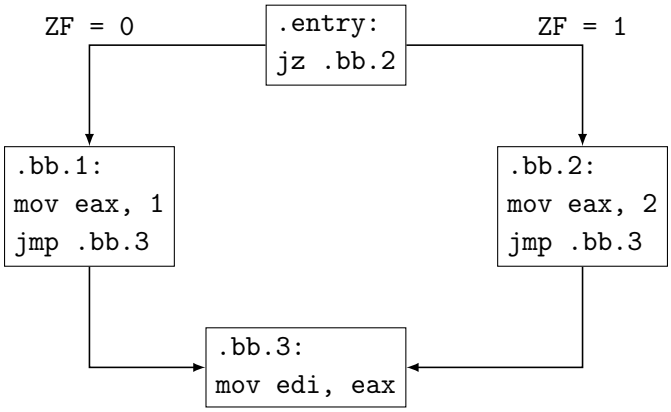


Figure 2.17: Control flow graph with a register being defined in multiple predecessors.

```
.entry:
    EAX-STK-LOC = alloca i32, align 4
    ; ...
    br %CF, .bb.1, .bb.2
.bb.1:
    store i32 1, i32* %EAX-STK-LOC, align 4
    br .bb.3
.bb.2:
    store i32 2, i32* %EAX-STK-LOC, align 4
    br .bb.3
.bb.3:
    %EDI = load i32, i32* %EAX-STK-LOC, align 4
```

Figure 2.18: Stack promotion of `EAX`

While code with these stack slots is potentially more inefficient, as it will access memory, while `phi` instructions may be compiled to use registers, LLVM.opt is able to convert these stack accesses to `phi` nodes, resolving this issue.

2.5.4 Peephole Optimizations

Peephole optimizations are compiler optimizations that operate on a small set of instructions, analyzing them, and potentially replacing them with a new set of instructions offering better performance [McK65]. In MCTOLL, peephole optimizations are used to replace certain instruction patterns to make sure the LLVM optimizer is able to optimize them. One example are memory accesses with an offset as shown in Figure 2.19.

```
%1 = ptrtoint i8* %0 to i64
%2 = add i64 %0, 16
%3 = inttoptr i64 %0 to i32*
```

(a) Original code

```
%1 = getelementptr i8, i8* %0, i64 16
%2 = bitcast i8* %0 to i32*
```

(b) Optimized code

Figure 2.19: Example for code optimized by the MCTOLL peephole pass

2.5.5 Limitations

While MCTOLL allows raising a large set of programs, there are some limitations on what it can do. Indirect jumps are not supported, as jump targets cannot be detected ahead of time.

Before our contributions to the project, MCTOLL was not able to raise programs from benchmarks such as phoenix-2.0 [Ran+07] and others, as many integer instructions were not supported. Additionally, almost no SSE instructions were implemented, meaning programs using floating-point arithmetic could not be raised. In Chapter 4 we will discuss our contributions to MCTOLL, the instructions that were implemented and bugs that we fixed.

3 Related Work

3.1 Direct Translation

With direct translation, a one-to-one mapping from the source to the target instructions is created. This is the approach used by projects such as Aries [ZT00] (HP-PA to IA64), Rosetta 1 (PowerPC to x86/IA32) and Rosetta 2 (x86_64 to ARM), IA-32EL [Bar+03] (x86/IA32 to IA64), and others [Che+08]. These projects were developed to aid the adoption of new CPU architectures by allowing legacy binaries to run on new machines without barriers or having to wait for developers to release an updated version of their programs. Rosetta 2 also implements parts of the translator in hardware, significantly speeding up the translated binary.

3.2 IR-Based Translation

A more flexible approach is achieved by translating the source into an intermediate language before it is compiled to the target architecture. Projects such as MCTOLL [YS19] and LLBT [She+12] use LLVM IR as their intermediate language, while others like the UBQT framework [Cif+02] develop their own intermediate representation.

The advantage of IR-based translation is that adding support for an additional source or target architecture takes less effort than for the direct approach, as the front- and backend are decoupled. Widely used IR's such as LLVM IR already come with a wide range of supported target architectures as well as an optimizer that is able to significantly improve the runtime performance of the generated code.

3.3 Peephole-Based Translation

Projects such as the tool developed by Bansal and Aiken implement a static binary translation approach without having to manually implement the mapping for every instruction. It uses a peephole superoptimizer to find mappings from the source to the target architecture. This superoptimizer approach works by extracting a list of instruction sequences from a training set, creating a possible list of replacements for them, and checking which of these replacements is equivalent to the source sequence. If

the replacement sequence is equivalent to the source sequence, it is saved as a mapping. From this, a lookup table with replacement sequences from the source to the target architecture is created [BA08].

3.4 Translation of Floating-Point Instructions

Floating point instructions pose a challenge not only in hardware but also for binary translators, as different architectures may implement different behaviour regarding rounding, exceptions, and more. To keep performance overhead low, binary translators should aim at keeping software emulation of instructions down to a minimum.

Support for floating-point instructions in LLBT [She+12] has been implemented by You, Lin, and Yang. ARM supports different rounding modes for floating-point instructions, for both intermediate and final results. As LLVM does not allow setting rounding modes for either of those, this behaviour needs to be implemented with software emulation. In order to replicate the original behavior in the raised code, intermediate results may need to be calculated for some instructions to get to the correct result. To check for hardware exceptions, the operands and the result of instructions need to be checked by the generated LLVM code [YLY19].

4 Contributions

Within the scope of this project, we have implemented previously unsupported instructions and fixed bugs that existed when raising programs, particularly ones with higher optimization levels (`-O2`, `-O3`).

4.1 Support for Floating-Point Arguments and Return Values

One of the more extensive modifications to MCTOLL we have implemented is support for floating-point arguments and return values. This required some modifications to both function prototype discovery and argument discovery while raising a call function as described in Sections 2.5.1 to 2.5.2. The algorithm described in Section 2.5.1 does not apply to SSE registers, as they do not expose subregisters, and different data types may be stored in the same register. Additionally, both floating-point values and vector values may be stored in SSE registers.

4.1.1 Function Prototype Discovery

To determine the type stored in a register, we need to look at the instruction using the register. To achieve this, we differentiate between two types of SSE instructions:

Packed instructions operate on the full vector register. They operate on vectors of integers or floating-point values.
The discovered type is a 128 bit wide vector, the type and count of elements depends on the instruction.
Example: `ADDPD` (Add packed double, works on a vector of two `double` values) → `<2 x double>`

Scalar instructions operate on the lower 32 or 64 bits of the register. They operate on single floating-point values, the discovered type is either a `float` or `double`.
Example: `ADDSD` (Add scalar double, works on a single `double` value stored in the lower 64 bits of the SSE registers) → `double`

The same type discovery is done for both the arguments and return type. Figure 4.1 shows how two functions that use the same argument and return discovery are raised to different function prototypes, depending on which instruction uses the register.

```
add_double:
  addsd xmm0, xmm0
  ret
add_float:
  addss xmm0, xmm0
  ret
```

(a) Original code

```
define dso_local double @add_double(double %1) {
  %2 = add double %1, %1
  return %2
}
define dso_local double @add_float(float %1) {
  %2 = add float %1, %1
  return %2
}
```

(b) Raised code

Figure 4.1: Discovered arguments passed in SSE registers

4.1.2 Call-Argument Discovery

The existing code had to be extended to look up SSE registers' values to support floating-point arguments. For vararg function calls, additional changes had to be implemented.

System-V requires to pass the number of SSE registers used in the `AL` register when calling a varargs function. Compilers usually generate an instruction that sets `AL` to a constant, e.g. `MOV AL, 1`. We leverage this behaviour and search for a reaching value for the `AL` register. If this value is a constant, we look up the value and search for that amount of SSE registers. If we do not find a constant stored in `AL`, we fall back to the approach used for general purpose registers.

Parameter Ordering For vararg function calls, we run into the problem of parameter order: since System-V does not preserve the order of arguments passed in general purpose and SSE registers, we cannot precisely reconstruct the function call with all parameters in their original position. We assume that arguments passed in general purpose registers come before those passed in SSE registers. In Figure 4.2 we show how a raised `printf` call will have a different parameter ordering than the original one. While this is not an issue when re-compiling to ARM64 or another architecture that separates argument registers for integral and floating-point types, we cannot re-compile for architectures that pass integral and floating-point values in the same registers.

This issue only arises for external vararg functions, not those raised by MCTOLL, since the order of arguments for those is set by MCTOLL.

```
printf("%f␣*␣%d␣=␣%f", 1.5, 2, 3.0);
```

(a) Call to `printf` with mixed `int` and `double` arguments

```
%1 = call i32 (i8*, ...) @printf(i8* %0, i32 2, double 1.5, double 3.0)
```

(b) Raised code, with all `int` arguments preceding the `double` arguments

Figure 4.2: Raised function call with mixed general purpose and SSE arguments

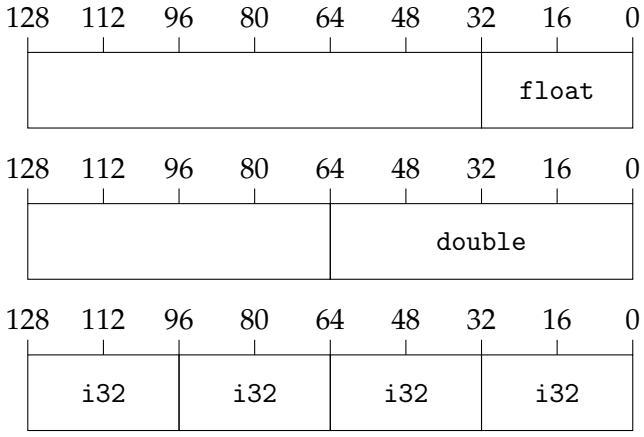


Figure 4.3: Different values stored in an `XMM` register

4.1.3 Handling of SSE Register Values

SSE registers are 128 bit wide and may hold different scalar or packed values, as can be seen in Figure 4.3. These values can be 16, 32, 64, or 128 bit wide. While scalar types only occupy the lower n bits of the register, packed types use the entire register.

We can deduce the LLVM type from the register used for integer types by looking at the register size. This is not possible for SSE registers, as they do not expose subregisters for each type. Instead, we use the instruction to check which type they operate on.

`addsd` This instruction operates on a scalar `double` value stored in the lower 64 bits of the register. The produced result is stored in the lower 64 bits again, while the upper 64 bits are set to zero.

`paddw` This instruction operates on four packed 32 bit integer values.

Since the programmer may mix SSE instructions operating on different types, we cast the value to the appropriate type for the instruction being raised on-demand. If an instruction only operates on the lower n bits of a register and sets the remaining bits to zero (as most scalar floating-point instructions do), we do not save those upper bits that are zero. This has the advantage that consecutive SSE instructions that operate on the same type do not have to cast anything and can be translated into LLVM with less overhead. If an instruction operates on higher bits that are missing in the value stored in the register-SSA map for the current register, we assume that those bits were set to zero by a previous instruction.

Casting Values in LLVM Since a cast is implicit in x86 assembly and should not modify any bits, we use bitcasts and LLVM vector instructions to get to the required type. In the following sections, *src* is the source type, *dst* is the destination type, and $|x|$ is the type's bit width. Three cases need to be handled.

1. $|src| = |dst|$: In this case, a simple `bitcast` instruction is used to cast the value to the desired type.
2. $|src| < |dst|$: Since we assume missing bits to be zero, we first create a 128 bit wide vector of the type <n `x` *src*>, where $n = \frac{|dst|}{|src|}$. Then, we insert the source value into position 0 of the created vector and bit-cast the value to the destination one.
3. $|src| > |dst|$: Here, we first bitcast the source value to a bit vector of type <n `x` *dst*>, where $n = \frac{|src|}{|dst|}$ and return the element at position 0.

In Figure 4.4 we show generated LLVM bitcode for cases 2 and 3.

```
%0 = insertelement <2 x double> zeroinitializer, double %arg1, i64 0
%1 = bitcast <2 x double> %0 to <4 x i32>
```

(a) `double` → `<4 x i32>`

```
%0 = bitcast <4 x i32> %arg1 to <4 x float>
%1 = extractelement <4 x float> %0, i64 0
```

(b) `<4 x i32>` → `float`

Figure 4.4: SSE register type casting

4.1.4 Stack Promotion of SSE Registers

In order to support programs using SSE registers, the stack promotion algorithm described in Section 2.5.3 needed to be updated to support all types of SSE values. If we encounter an SSE value that needs to be promoted, we allocate a 128 bit `<4 x i32>` stack slot. Should a value be less than 128 bits wide, we pad the remaining bits with zeros. When the value is reread, we re-interpret the value as described in Section 4.1.3.

4.2 List of other Contributions

We have added support for instructions found in the phoenix-2.0 benchmark, the execution of which we will evaluate in Chapter 5. To find instructions not yet supported, we first tried raising benchmarks that were compiled without any optimizations. Once we could raise non-optimized binaries, we continued raising the same benchmark compiled with higher optimization levels, each time raising and looking for new instructions, implementing those, and continuing. With optimization levels `-O2` and `-O3` in particular, subtle bugs in the code became visible. Compilers optimize very aggressively, and the produced code is not straightforward.

At the time of writing, we have submitted 45 pull requests to the MCTOLL repository[1], 43 of which are merged while 2 are still under review.

4.2.1 SSE Floating-Point Arithmetic Instructions

The following instructions are the basic SSE floating-point instructions.

- `addsd`, `addss`
- `subss`, `subsd`

[1] `https://github.com/microsoft/llvm-mctoll/pulls`

- `mulsd`, `mulss`
- `divsd`, `divss`
- `sqrtsd`, `sqrtss`

With the exception of the square root instructions, these operations translate directly to LLVMs `fadd`, `fsub`, `fmul`, and `fdiv` instructions. LLVM does not provide an instruction to calculate a floating-point square root, but it provides an intrinsic function we can call. If one operand is a memory operand, the instructions to load the value from memory are inserted before the arithmetic instruction.

```
addsd xmm0, xmm1
subsd xmm0, xmm2
mulsd xmm0, xmm1
divsd xmm0, xmm2
sqrtsd xmm0
```

(a) Arithmetic instructions

```
%3 = fadd double %0, %1
%4 = fsub double %3, %2
%5 = fmul double %4, %1
%6 = fdiv double %5, %2
%7 = call double @llvm.sqrt.f64(double %6)
```

(b) Raised code

Figure 4.5: Raised SSE floating-point arithmetic

4.2.2 SSE min/max Instructions

There are two instructions to select the minimum/maximum value of two registers:

- `maxsd`, `maxss`
- `minsd`, `minss`

The first argument is compared to the second one according to the following rules:

$$dst = \begin{cases} arg_2 & \Leftrightarrow arg_1 = arg_2 \vee arg_1 = \text{NaN} \vee arg_2 = \text{NaN} \\ arg_1 & \Leftrightarrow arg_1 < arg_2 \ (> \text{ for } \texttt{max}) \end{cases}$$

This comparison can be mirrored in LLVM with an `fcmp` instruction with `ogt` (ordered greater than) and `olt` (ordered less than) and a `select` instruction. Figure 4.6 shows how a `minsd` instruction will be raised.

```
minsd xmm0, xmm1
```

(a) Arithmetic instruction

```
%cmp = fcmp olt double %0, %1
%min = select i1 %cmp, double %0, double %1
```

(b) Raised code

Figure 4.6: Raised SSE min/max instructions

4.2.3 SSE Floating-Point Bitwise Instructions

The following bitwise instructions operate on packed floating-point values stored in the `XMM` registers. The semantics for this instruction are the same as those for packed integers (see Section 4.2.7). However, the processor may have different data buses and execution units for integer and floating-point types. Mixing types may cause a delay of a few clock cycles when switching execution units [Fog21, p. 119].

- `andpd`, `andps`
- `orpd`, `orps`
- `xorpd`, `xorps`

These instructions can not be directly translated to LLVM, as LLVM does not support bitwise operations on floating-point values. We can, however, bitcast the values to integer values, perform the bitwise operation, and bitcast the value back to its original type. In Figure 4.7 we can see an example where the two input arguments are assumed to be `doubles`. We first have to cast them to the expected input type (`<2 x double>`) by zero-extending the current value. Once this is done, we bitcast the values to `i128`, where we then perform the `and` operation. Afterward, the value is cast to `<2 x double>` again, as this is the input and output type for this particular instruction. There is space for optimization left here, as the generated instructions are somewhat redundant in some cases.

Setting a Register to 0 with xor Often, instructions like `xorps xmm0, xmm0` are generated to zero-out a register. We recognize patterns like these and update the register-SSA map to set the register value to zero.

4.2.4 SSE FP Comparison Operations

There are two kinds of SSE floating-point operations we support:

- `ucomisd`, `ucomiss` (Unordered Compare Scalar Set EFLAGS)
- `cmpsd`, `cmpss` (Compare Scalar)

```
andpd xmm0, xmm1
```

(a) Packed bitwise instruction

```
%0 = bitcast double %arg1 to i64
%1 = zext i64 %0 to i128
%2 = bitcast 128 %1 to <2 x double>
%3 = bitcast double %arg2 to i64
%4 = zext i64 %3 to i128
%5 = bitcast 128 %4 to <2 x double>
%6 = bitcast <2 x double> %4 to i128
%7 = bitcast <2 x double> %5 to i128
%8 = and i128 %6, %7
%result = bitcast i128 %8 to <2 x double>
```

(b) Raised code

Figure 4.7: Raised SSE bitwise instruction

Result	ZF	PF	CF
Unordered	1	1	1
Greater than	0	0	0
Less than	0	0	1
Equal	1	0	0

Table 4.1: Status flag values for `ucomisd/ucomiss`

Unordered Compare Scalar Set EFLAGS

The `ucomis` instructions compare their arguments and set the processor's status flags according to the result. In Table 4.1 the value of the different flags for all possible results is shown.

In LLVM, we can use the `fcmp` instruction with the following comparison condition to calculate the flag's value:

- `ZF`: `ueq` (unordered or equal)
- `PF`: `uno` (unordered)
- `CF`: `ult` (unordered or less than)

In Figure 4.8 we show how a `ucomisd` instruction is raised.

```
ucomisd xmm0, xmm1
```

(a) Compare instruction

```
%CF = fcmp ult double %0, %1
%ZF = fcmp ueq double %0, %1
%PF = fcmp uno double %0, %1
```

(b) Raised code

Figure 4.8: Raised SSE Unordered Compare Scalar Set EFLAGS instruction

Predicate immediate	Pseudo-Op	Description	LLVM `fcmp` predicate
0	`cmpeq`	ordered and equal	`fcmp oeq`
1	`cmplt`	ordered and less than	`fcmp olt`
2	`cmple`	ordered and less or equal	`fcmp ole`
3	`cmpunord`	unordered	`fcmp uno`
4	`cmpneq`	ordered and not equal	`fcmp one`
5	`cmpnlt`	not less than	`fcmp olt` and `not`
6	`cmpnle`	not less and equal	`fcmp ole` and `not`
7	`cmpord`	ordered	`fcmp ord`

Table 4.2: Compare Scalar comparison predicates

Compare Scalar

This instruction supports eight comparison predicates that dictate the comparison semantics. The destination register is set to a quad- or doubleword mask of all ones if the comparison is true or a mask of all zeros if the comparison is false.

We then use a `select` instruction to get the correct bitmask, as shown in Figure 4.9.

```
cmpeqsd xmm0, xmm1
```

(a) Compare instruction

```
%0 = fcmp oeq double %arg1, %arg2
%1 = bitcast i64 -1 to double
%2 = bitcast i64 0 to double
%result = select i1 %0, double %1, double %2
```

(b) Raised code

Figure 4.9: Raised SSE Compare Scalar instruction

4.2.5 SSE Move Packed FP Instructions

The following SSE instructions are used to move a vector of packed floating-point values from and to SSE registers.

- `movapd`, `movaps` (Move Aligned Packed Double/Single-Precision Floating-Point Values)
- `movupd`, `movups` (Move Unaligned Packed Double/Single-Precision Floating-Point Values)

The aligned version of these instruction is used when the memory is known to be aligned on a 16 byte boundary, which offers a performance benefit. The unaligned instructions do not require the memory to be aligned, but are slower than the aligned ones. Support for non-packed versions (`movsd`, `movss`) of these instructions was already implemented in MCTOLL.

If the move is a register-register move, we update the register-SSA map, analogous to non-SSE register-register moves. If one of the operands is a memory location, however, we need to create a `load` or `store` instruction. We can reuse the existing code here, as the semantics are the same as for non-SSE moves. Figure 4.10 shows how a move from a read-only region to the stack will get translated. Note that `%rodata` is a global variable containing all read-only data raised from the source binary.

```
movupd xmm0, [.L.val]
movupd [rsp], xmm0
```

(a) Assembly instructions

```
%0 = bitcast i8* %rodata to <2 x double>*
%1 = load <2 x double>, <2 x double>* %0, align 1
%2 = bitcast i64* %stack to <2 x double>*
store <2 x double> %1, <2 x double>* %2, align 1
```

(b) Raised code

Figure 4.10: Raised SSE Move Packed FP

4.2.6 SSE Conversion Instructions

We support the following conversions between `double` ↔ `float` and `i64/i32` ↔ `double/float`. We do this by utilizing the following LLVM instructions:

- `double` ↔ `float`: `fpext/ftrunc`
- `i64/i32` ↔ `double/float`: `fptosi/sitofp`

Figure 4.11 shows an example of three raised instructions.

4.2.7 SSE Integer Bitwise Operations

The `pand`, `por`, and `pxor` instructions can be mapped to a single LLVM instruction, as LLVMs `and`, `or`, and `xor` instructions work on integer and integer vectors [LLV].

```
cvtsi2ss xmm0, rsi
cvtss2si esi, xmm0
cvtss2sd xmm0, xmm0
```

(a) Assembly instructions

```
%0 = sitofp i64 %arg1 to float
%1 = fptosi float %0 to i32
%3 = fpext float %0 to double
```

(b) Raised code

Figure 4.11: Raised SSE convert instructions

4.2.8 SSE movq/movd Instructions

The `movq/movd` instructions copy a quad- (64 bits) or double word (32 bits) from the source to the destination operand without changing any of the bits. These instructions allow moving data from general purpose to SSE registers. Unlike the conversion instructions described in Section 4.2.6, we do not change any of the bits in the input values. We insert a bitcast to cast the value to the appropriate type for the destination register. This is shown in Figure 4.12.

```
movq rax, xmm0
movd xmm0, edi
```

(a) Assembly instructions

```
%2 = bitcast double %0 to i64
%3 = bitcast i32 %1 to float
```

(b) Raised code

Figure 4.12: Raised SSE `movq/movd` instructions

4.2.9 Bit Test Instructions

We have added support for `BT` (Bit Test), `BTS` (Bit Test and Set), `BTR` (Bit Test and Reset), and `BTS` (Bit Test and Complemented). These instructions take one register or memory operand to check against and a second register or immediate operand specifying the index of the bit to check. If the first operand is a register, the index of the bit to check should be calculated by taking the modulus of 16, 32, or 64 (depending on the register size). The instruction sets the carry flag to the value of the specified bit. Additionally, the different variants change the bit in the source operand:

- `BTS`: Set the bit to 1
- `BTR`: Set the bit to 0
- `BTC`: Flip the bit value

We use a series of shift and bitwise instructions to emulate the behavior described above. Figure 4.13 shows how the `BT` and `BTS` instructions are raised. `BTR` and `BTC` are

calculated similarly to BTS, except that they use a logical and and a not to unset the bit, and a xor to flip the bit.

```
bt rax, 4
```

(a) Assembly instructions

```
%0 = urem i32 4, 32
%1 = shl i32 1, %0
%2 = and i32 %arg1, %1
%CF = icmp ne i32 %2, 0
```

(b) Raised code

```
bts rax, 4
```

(c) Assembly instructions

```
%0 = urem i32 4, 32
%1 = shl i32 1, %0
%2 = and i32 %arg1, %1
%CF = icmp ne i32 %2, 0
%result = or i32 %arg1, %1
```

(d) Raised code

Figure 4.13: Raised Bit Test instructions

4.2.10 Multiplication Instructions

While integer multiplication instructions were already available in MCTOLL, the raising of single-operand instructions was broken. After our patch, the following mul/imul instructions are supported:

- IMUL16r, IMUL32r, IMUL64r (signed multiplication)
- MUL16r, MUL32r, MUL64r (unsigned multiplication)

The instructions take one operand and multiply it with either AX, EAX, or RAX, depending on the size of the operand. Since the result may overflow, it is stored in the registers DX:AX, EDX:EAX, or RDX:RAX, again depending on the size of the operand. To signal an overflow, OF and CF are set if the result does not fit into the register used to store the lower half of the result and cleared otherwise.
For the unsigned variant, this is done by checking if the upper half is zero. For the signed instruction, we sign-extend the lower half and check if the sign-extended value is the exact result of the multiplication. This is shown in Figure 4.14.

4.2.11 Vararg Argument Discovery

When a vararg function call is discovered, we need to check how many arguments are passed to that function. MCTOLL does this by iterating over all argument registers and

```
imul rdi
```

(a) Assembly instructions

```
%0 = sext i64 %RAX to i128
%1 = sext i64 %RDX to i128
%2 = mul nsw i128 %0, %1
%3 = lshr i128 %2, 64
%RDX1 = trunc i128 %3 to i64
%RAX1 = trunc i128 %2 to i64
%4 = sext i64 %RAX1 to i128
%CF = icmp ne i128 %4, %2
```

(b) Raised code

Figure 4.14: Raised multiplication instruction

checking if they have a reaching value.

This is done by walking back the reconstructed CFG and checking each instruction for a register definition. If a call instruction is encountered, MCTOLL does not look further ahead, as function calls do not preserve argument registers.

The algorithm to determine if there was a reaching value for an argument register was implemented as follows:

1. Check if the current basic block contains a register definition for the register.
2. Otherwise, recursively check all predecessor blocks if there are as many reaching register definitions as the current basic block has predecessors.

This algorithm has a flaw, as it does not consider that predecessor blocks may not define a register, but one of their predecessors does.

An example for an incorrectly identified parameter register is shown in Figure 4.15. We are checking for vararg registers in `.bb.2`. Since `.bb.2` does not define `rdx`, and `.bb.2` has one predecessor, we check if there is exactly one reaching definition. This is the case, since `.bb.3` defines `RDX`, although it is not an argument register. In order for `RDX` to be a valid argument register, either `.entry` and `.bb.2` *or* `.bb.1` would need to define `RDX`.

The new algorithm works recursively:

1. If the current block defines the register, return true.
2. Otherwise, call the function recursively for each predecessor block and check if all of them return true.

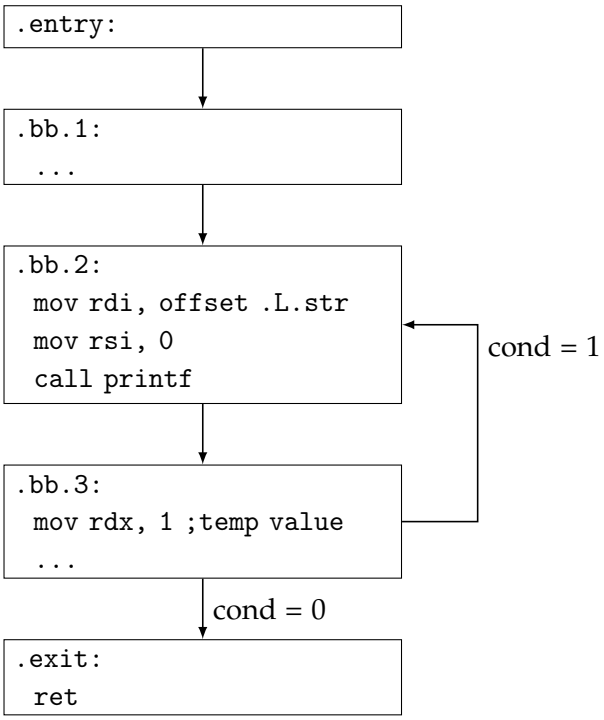


Figure 4.15: Control flow graph with a vararg function call

4.2.12 Various Bug Fixes

This section will describe a number of the bug fixes we submitted[2] to MCTOLL. Most of these bugs occurred when trying to raise optimized programs and benchmarks.

Identifying Implicitly-Set Registers for Vararg Calls

Registers that were set implicitly by instructions such as `idiv` or `div` were not considered as arguments for vararg functions previously. We fixed this by checking all definitions of an instruction instead of only explicit ones for register definitions when searching for potential argument registers.

Detecting Return Types for Functions Tail-Calling other Functions

An optimization often encountered in functions which return the result of another function call are tail calls. A tail call is the last instruction of a function that calls another function or itself. Instead of owning a stack frame, calling the next function, and then returning, no stack frame is set up; a jump instruction is used to jump to the next function. In Figure 4.16 the function `my_strlen` calls `strlen` with a modified parameter. The result is immediately returned. We have patched MCTOLL to allow detecting these cases and setting the function's return type to the one of the called function.

```
int my_strlen(char *str) {
    return strlen(str + 1);
}
```

(a) Original code

```
my_strlen:
    add rdi, 1
    jmp strlen
```

(b) Compiled code using a tail-call

Figure 4.16: Tail-call optimization

Support for External Variables

Variables which are not declared in the program itself but in included system header files should be declared as external in the raised bitcode.

Variables such as `stdout`, `stdin`, or others are declared as `extern` variables in the C header sources. In the compiled program, they are declared in the `.bss` section of the program, which means the operating system will initialize them with zero once it loads

[2] All submitted fixes can be viewed at `https://github.com/microsoft/llvm-mctoll/pulls`.

the program into memory. The C runtime is responsible for initializing these variables. While raising a program using these variables, MCTOLL initializes the variables with zero, as they are declared in the `.bss` section. This produces the following bitcode:

```
@stdout = common dso_local global i64 0, align 8
```

The correct way would be to declare the variable as `external` and let the linker figure out where and how the variable is initialized. Since the user of MCTOLL already has to pass a list of system headers to the tool in order for it to know about external function declarations, we patched MCTOLL to scan for variable declarations in these header files and mark global variables declared there as external instead of initializing them. This results in the following variable declaration:

```
@stdout = external dso_local global i64, align 8
```

Stack Promotion of Moved Registers

Since register-register moves are a no-op when raising code with MCTOLL, the stack promotion algorithm encountered a problem when promoting a register that contained a value moved from another register. After a register-register move, the entries in the register-SSA map for both registers would point to the same LLVM value.

Promoting a register value would generate a store instruction for each incoming reaching value and a load instruction for each usage of the reaching value. Since we updated the register-SSA map with the register-register move, the entry for both registers pointed to the same LLVM value, usages of the first register were updated. This led to read operations from the stack slot before the stack slot was written to, which is incorrect. We fixed this bug by only replacing usages of the value if the store instruction dominates (see Section 2.1.1) the instruction that will be replaced.

Support for assert

Asserts are implemented by checking for the passed condition, and if the condition is evaluated to false, call `__assert_fail`, which terminates the program. Every basic block in LLVM needs to be terminated by a terminator instruction. The bitcode generated by MCTOLL was incorrect when raising a program containing an assert, as the call to `__assert_fail` was the last instruction for the basic block containing it. We fixed this by inserting an `unreachable` instruction after every call to `__assert_fail`, as `__assert_fail` should never return.

5 Evaluation

In this section, we evaluate the runtime overhead of raised programs and how re-optimizations of the raised bitcode impact the overall execution time of the raised programs.

5.1 Setup

To run the benchmarks for ARM and x86_64, we set up a test machine running on both platforms to run the raised benchmarks. The benchmarks were compiled using clang 14.0.0 (commit hash ef976337f581[1]), which was set up to be able to cross-compile for ARM and x86_64.

$$S \xrightarrow{\text{clang}} \mathbb{P}_{\text{target}} \tag{5.1}$$

$$S \xrightarrow{\text{clang}} \mathbb{P}_{\text{X86}} \xrightarrow{\text{mctoll}} \mathbb{P}_{\text{IR}} \xrightarrow{\text{LLVM.codegen}} \mathbb{P}'_{\text{target}} \tag{5.2}$$

$$S \xrightarrow{\text{clang}} \mathbb{P}_{\text{X86}} \xrightarrow{\text{mctoll}} \mathbb{P}_{\text{IR}} \xrightarrow{\text{LLVM.opt}} \mathbb{P}'_{\text{IR}} \xrightarrow{\text{LLVM.codegen}} \mathbb{P}'_{\text{target}} \tag{5.3}$$

$$S \xrightarrow{\text{clang}} \mathbb{P}_{\text{X86}} \xrightarrow{\text{mctoll}} \mathbb{P}_{\text{IR}} \xrightarrow{\text{peephole+LLVM.opt}} \mathbb{P}'_{\text{IR}} \xrightarrow{\text{LLVM.codegen}} \mathbb{P}'_{\text{target}} \tag{5.4}$$

We build four binaries for each test, which we will evaluate in Section 5.2.

1. A native binary directly compiled to the target architecture from the source code
2. A raised binary without optimizations after raising
3. A raised binary where LLVM.opt optimizes the generated bitcode
4. A raised binary where the peephole pass runs and then LLVM.opt optimizes the generated bitcode

[1] `https://github.com/llvm/llvm-project/tree/ef976337f581`

ARM For ARM, we used a Raspberry Pi 4 Model B with 4 GB of RAM running 64 bit Debian 11.1. This model features an ARM Cortex-A72 CPU with four cores that implements the ARMv8 64 bit instruction set. To minimize thermal throttling, which might skew our results, we installed an active cooling system consisting of an aluminum heatsink and a small fan. This kept the temperature of the Pi below 35°C for single-threaded benchmarks and below 45°C for multi-threaded benchmarks, which used all CPU cores.

x86_64 To run the x86_64 benchmarks, we used a Linux machine with an AMD Ryzen 7 3700X CPU with 8 physical cores and 16 threads and 64 GB of DDR4 RAM running Debian 11.1.

5.1.1 Benchmarks

To evaluate our program, we used the sequential and pthread versions of the phoenix-2.0 benchmark[2]. To get a baseline, we cross-compiled each program to aarch64 and x86_64 using clang with optimizaiton level `-O3`. The x86_64 program was then raised to LLVM bitcode and re-compiled for both target architectures. To check how much overhead was introduced by MCTOLL, we re-compiled the raised bitcode with optimizations turned off (`-O0`), with optimizations turned on (`-O3`) and with optimizations turned on as well as the peephole compiler pass.

5.2 Results

In Figures 5.1 to 5.2 we compare the runtime performance of the phoenix-2.0 benchmark on ARM, in Figures 5.3 to 5.4 phoenix-2.0 on x86.

5.2.1 Native Binary

We ran the native binary to get a baseline for the execution time, which will be used to normalize runtimes (the native runtime will be 1).

5.2.2 Raw Overhead Introduced by Translation

To measure the impact of the raw overhead introduced by translating every instruction, we re-compile the raised bitcode with optimizations turned off (`-O0`). This does not remove unused instructions generated to calculate implicitly-set processor flags that

[2] `https://github.com/kozyraki/phoenix`

will not be used in the code. As discussed in Section 2.5.3, reaching register values may be stored in stack slots. Accessing these produces a significant overhead that LLVM.opt can optimize.

5.2.3 Optimized Binary

The most important optimizations LLVM.opt performs on the generated bitcode are

- removing unnecessary instructions (e.g., calculating processor flags),
- promoting values from the stack to registers with the help of `phi` nodes, and
- further optimizing generated code.

These optimizations manage to almost entirely wipe out the introduced overhead in all tested benchmarks with the sequential histogram test being the exception. For some benchmarks, the optimized binaries manage to outperform their native counterpart consistently.

5.2.4 Optimized Binary with Peephole Pass

The peephole pass does not introduce runtime performance in most test cases, and the execution time stays the same as in the run without the peephole pass. However, the peephole pass manages to significantly speed up the runtime of the sequential kmeans benchmark to the point where it consistently outperforms the native version and runs in about 70% of the native version's time.

5.2.5 Cross-Architecture Translation

In this subsection, we compare the benchmarks compiled natively for ARM against the version compiled for x86 and translated to ARM. Figures 5.1 to 5.2 show the runtime overhead for the sequential and pthread versions of the phoenix-2.0 benchmark.

We can see that the unoptimized translated version has an overhead that varies between 11% (pca-seq) and 380% (kmeans-seq). Benchmarks where dead code and many stack-demotions of registers are generated by MCTOLL show a high overhead here. By running LLVM.opt with its highest optimization level gets rid of almost all of this overhead introduced by MCTOLL and we are able to achieve a runtime very close to the native version. Some benchmarks run even faster than the native version. If we run MCTOLLs peephole pass and LLVM.opt, we are able to improve the performance of kmeans-seq and string_match-pthread to be faster than their native counterpart.

We believe we can achieve these performance gains over the native binaries as we are effectively running LLVM.opt twice over the generated binary.

By default, LLVM.opt runs a set of predefined optimization passes. By running these passes twice, LLVM may be able to optimize a binary further.

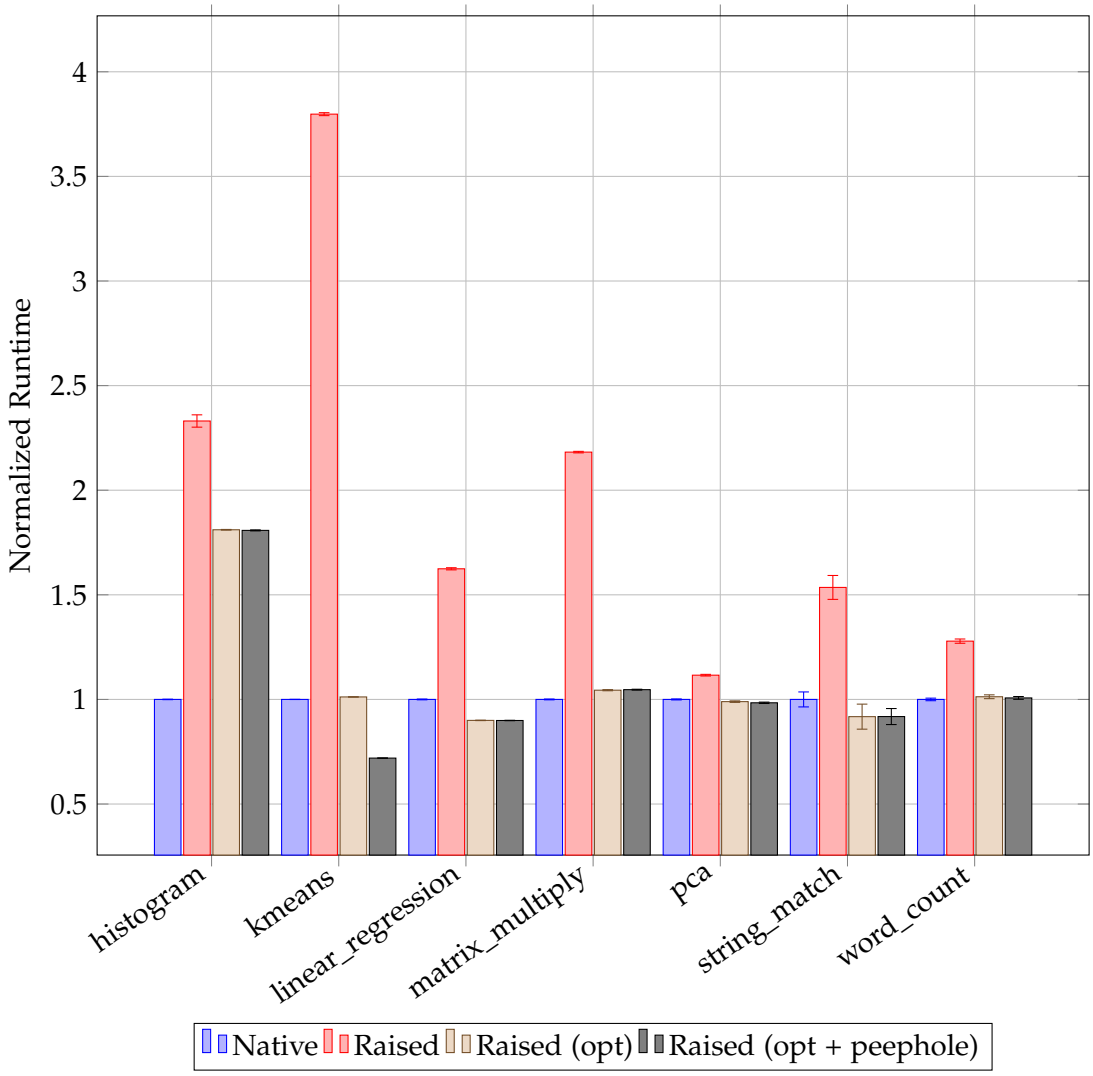


Figure 5.1: Runtime overhead of sequential phoenix-2.0 benchmarks on ARM

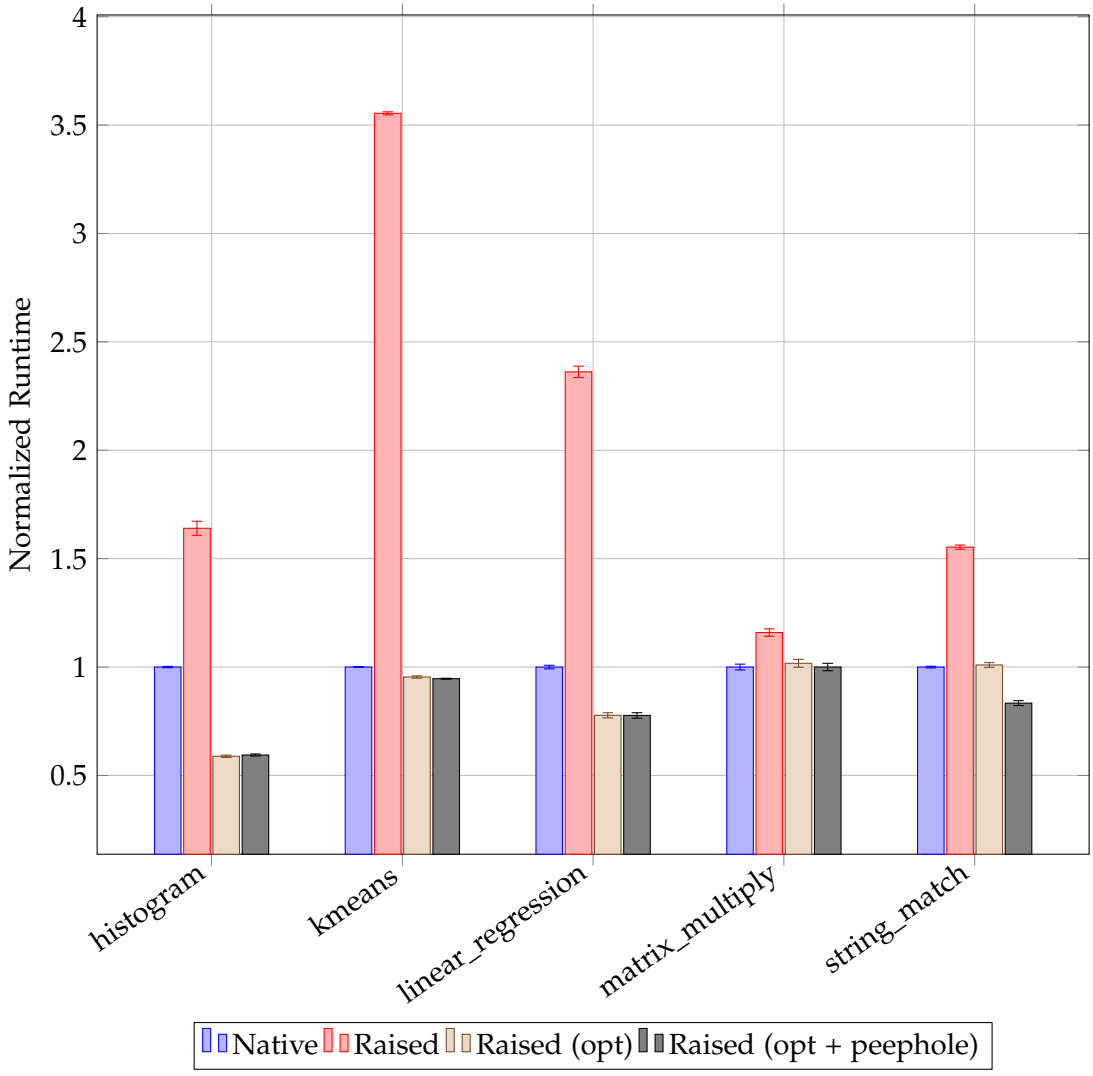


Figure 5.2: Runtime overhead of pthread phoenix-2.0 benchmarks on ARM

5.2.6 Same-Architecture Translation

In this subsection we compare a native x86 binary against the same binary raised and re-compiled for x86 again. Figures 5.3 to 5.4 show the overhead of raised binaries comapred to the native counterpart. For most benchmarks, we see the same result as we saw for the ARM versions in Section 5.2.5, but with a greater confidence interval for the pthread programs. This is probably caused by the background load being higher on the x86 machine than it was on the ARM machine.

We can conclude that recompiling to gain code size is not worth beneficial for the user.

5.2.7 Binary Size

In Figure 5.5, we compare binary sizes of the native ARM binaries against the translated binaries optimized for performance (`-O3`) and for code size (`-Oz`). In the plot we only display the sequential version of the benchmarks, as the results for the pthread ones are similar. We can see that for some benchmarks there is a marginal improvement in code size, for others such as histogram, linear_regression, and matrix_multiply there is no improvement.

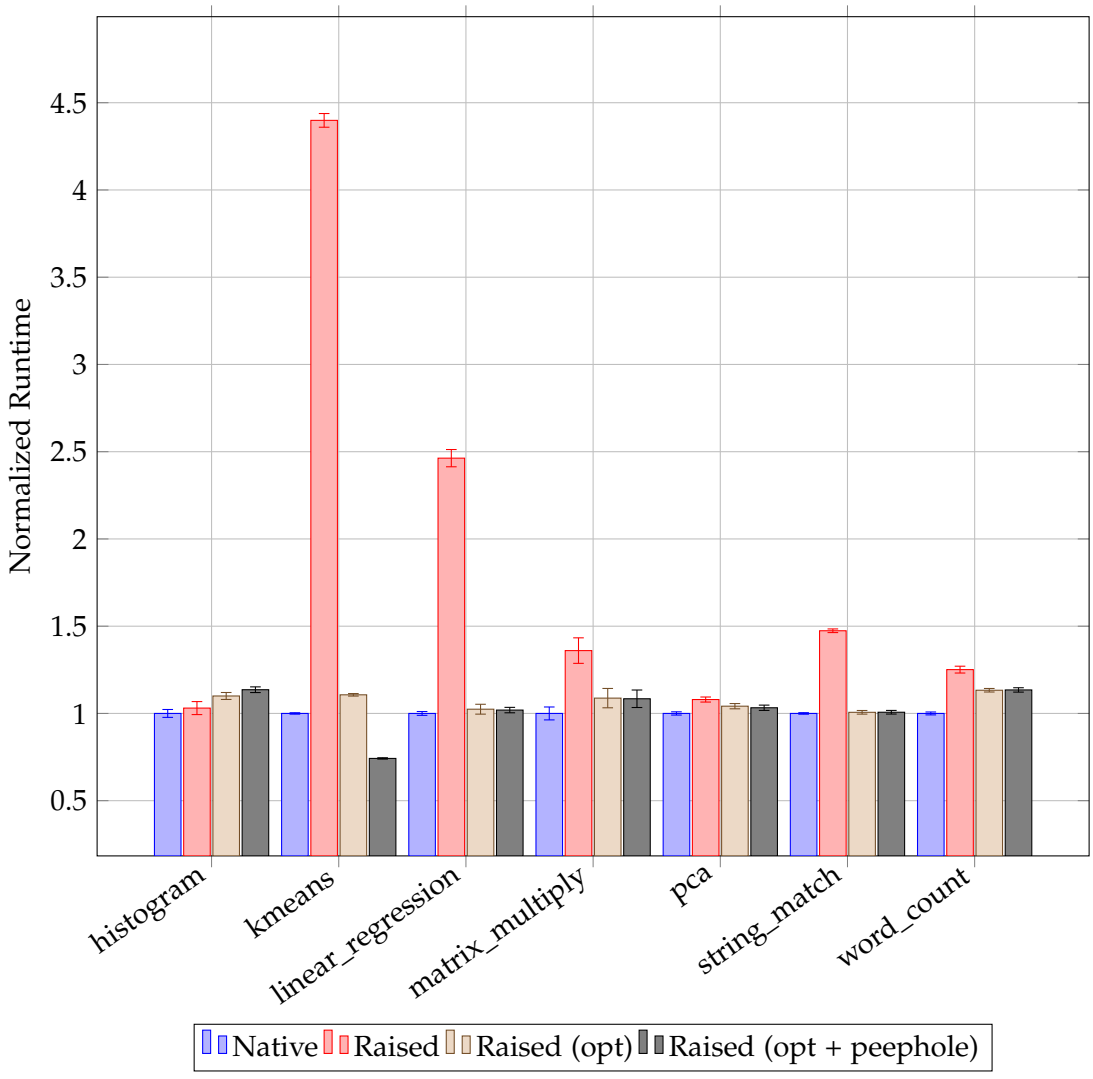


Figure 5.3: Runtime overhead of sequential phoenix-2.0 benchmarks on x86

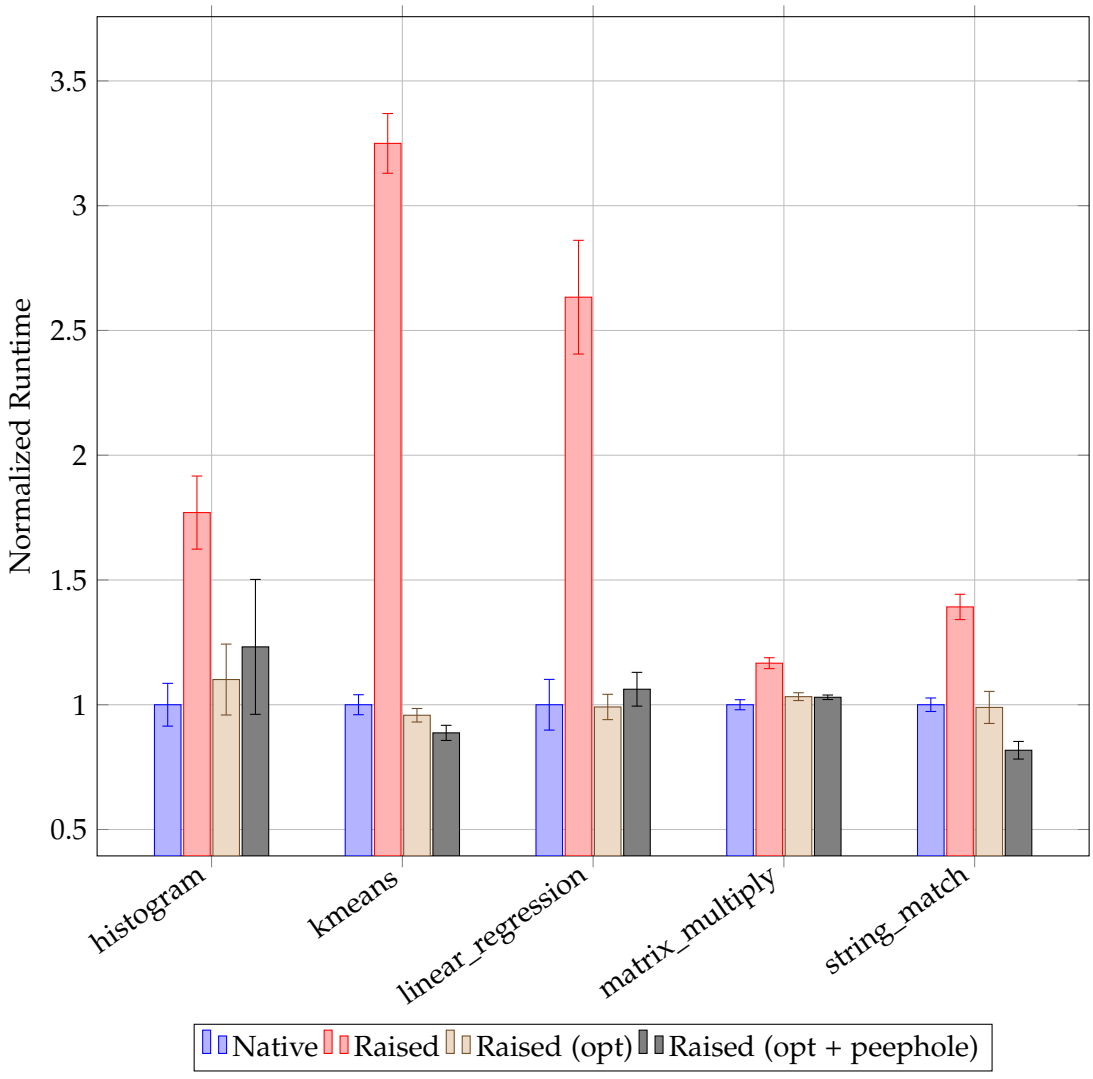


Figure 5.4: Runtime overhead of pthread phoenix-2.0 benchmarks on x86

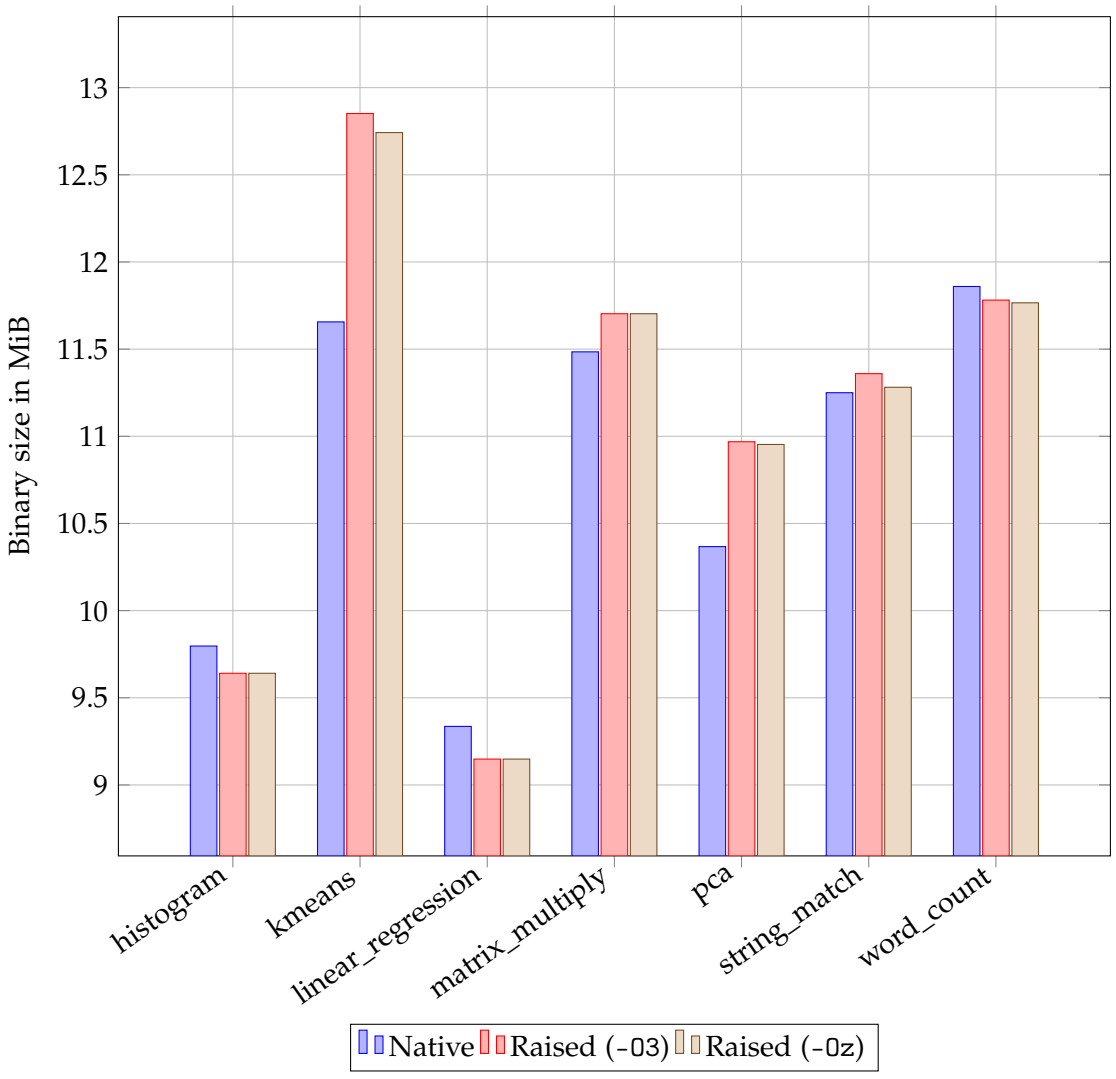


Figure 5.5: Binary sizes of native and raised benchmarks on x86

6 Conclusion

With our work on MCTOLL, we managed to improve its capabilities allowing users to raise more real-world programs than before. The support for floating-point arguments, return types, and instructions allow a wide range of programs to be raised, which was impossible before.

Additionally, we managed to fix issues that prevented programs which used externally defined variables (e.g., `stdout`) to be raised.

6.1 Future Work

At the time of writing, the most common instructions are supported, but there are still thousands of less frequently used instructions that are not implemented in the raiser, both from the x86 set and from later extensions such as AVX or AVX2. Implementing those would be working towards raising real-world programs and running them efficiently on different architectures.

As raising programs with indirect jumps is not possible with this static approach, a hybrid approach where static translation does most of the heavy lifting and dynamic binary translation is used for the part where static translation cannot translate the binary. This work could allow running programs on other platforms without significant overhead.

To correctly raise multi-threaded programs from ISAs using a strong memory model to a weak one (e.g., x86 to ARM), the raiser would need to insert memory fences, which is not done at the moment. This leads to potentially incorrect program execution on the weak memory model architecture. In this thesis, we do not evaluate programs which access shared memory in a way that violates these assumptions, but this is not true for all multi-threaded programs.

List of Figures

List of Tables

Bibliography

[All70] F. E. Allen. "Control Flow Analysis." In: *Proceedings of a Symposium on Compiler Optimization*. Urbana-Champaign, Illinois: Association for Computing Machinery, 1970, 1–19. ISBN: 9781450373869. DOI: 10.1145/800028.808479. URL: https://doi.org/10.1145/800028.808479.

[App] Apple. *x86-64 Code Model*. URL: https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/MachOTopics/1-Articles/x86_64_code.html (visited on 09/09/2021).

[BA08] S. Bansal and A. Aiken. "Binary Translation Using Peephole Superoptimizers." In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, 177–192.

[Bar+03] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach. "IA-32 execution layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium/spl reg/-based systems." In: *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* 2003, pp. 191–201. DOI: 10.1109/MICRO.2003.1253195.

[Bel05] F. Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX annual technical conference, FREENIX Track*. Vol. 41. Califor-nia, USA. 2005, p. 46.

[Che+08] J.-Y. Chen, W. Yang, J. Hung, C. Su, and W. C. Hsu. "A Static Binary Translator for Efficient Migration of ARM based Applications." In: (Jan. 2008).

[Cif+02] C. Cifuentes, M. Van Emmerik, N. Ramsey, and B. Lewis. *Experience in the Design, Implementation and Use of a Retargetable Static Binary Translation Framework*. Tech. rep. USA, 2002.

[Fog21] A. Fog. *Optimizing subroutines in assembly language*. 2021. URL: https://www.agner.org/optimize/optimizing_assembly.pdf (visited on 11/05/2021).

[LA04] C. Lattner and V. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California, Mar. 2004.

[LLV] LLVM. *LLVM Project*. URL: https://llvm.org (visited on 09/09/2021).

[LM69] E. S. Lowry and C. W. Medlock. "Object Code Optimization." In: *Commun. ACM* 12.1 (Jan. 1969), 13–22. ISSN: 0001-0782. DOI: 10.1145/362835.362838. URL: https://doi.org/10.1145/362835.362838.

[Lu+] H. Lu, M. Matz, M. Girkar, J. Hubička, A. Jaeger, and M. Mitchell. *System V Application Binary Interface*. URL: https://gitlab.com/x86-psABIs/x86-64-ABI/-/jobs/1438137053/artifacts/file/x86-64-ABI/abi.pdf (visited on 09/09/2021).

[McK65] W. M. McKeeman. "Peephole Optimization." In: *Commun. ACM* 8.7 (July 1965), 443–444. ISSN: 0001-0782. DOI: 10.1145/364995.365000. URL: https://doi.org/10.1145/364995.365000.

[Pro59] R. T. Prosser. "Applications of Boolean Matrices to the Analysis of Flow Diagrams." In: *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*. IRE-AIEE-ACM '59 (Eastern). Boston, Massachusetts: Association for Computing Machinery, 1959, 133–138. ISBN: 9781450378680. DOI: 10.1145/1460299.1460314. URL: https://doi.org/10.1145/1460299.1460314.

[Ran+07] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. "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.

[RWZ88] B. Rosen, M. Wegman, and K. Zadeck. "Global value numbers and redundant computations." In: *15th Annual ACM Symposium on Principles of Programming Languages* (Jan. 1988), pp. 12–27. DOI: 10.1145/73560.73562.

[She+12] B.-Y. Shen, J.-Y. Chen, W.-C. Hsu, and W. Yang. "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, 51–60. ISBN: 9781450314244. DOI: 10.1145/2380403.2380419. URL: https://doi.org/10.1145/2380403.2380419.

[YLY19] Y.-P. You, T.-C. Lin, and W. Yang. "Translating AArch64 Floating-Point Instruction Set to the X86-64 Platform." In: *Proceedings of the 48th International Conference on Parallel Processing: Workshops*. ICPP 2019. Kyoto, Japan: Association for Computing Machinery, 2019. ISBN: 9781450371964. DOI: 10.1145/3339186.3339192. URL: https://doi.org/10.1145/3339186.3339192.

[YS19] S. B. Yadavalli and A. Smith. "Raising binaries to LLVM IR with MCTOLL (WIP paper)." en. In: *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems - LCTES 2019*. Phoenix, AZ, USA: ACM Press, 2019, pp. 213–218. ISBN: 978-1-4503-6724-0. DOI: 10.1145/3316482.3326354. URL: http://dl.acm.org/citation.cfm?doid=3316482.3326354 (visited on 09/09/2021).

[ZT00] C. Zheng and C. Thompson. "PA-RISC to IA-64: transparent execution, no recompilation." In: *Computer* 33.3 (2000), pp. 47–52. DOI: 10.1109/2.825695.