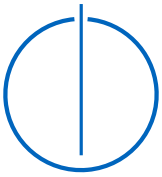
School of Computation, Information and Technology – Informatics
Technical University of Munich

Bachelor's Thesis in Informatics

Efficient Compilation of an Extensible Intermediate Representation

Jim M. R. Teichgräber



School of Computation, Information and Technology – Informatics
Technical University of Munich

Bachelor's Thesis in Informatics

Effiziente Kompilierung einer erweiterbaren Zwischensprache

Efficient Compilation of an Extensible Intermediate Representation

Author:	Jim M. R. Teichgräber
Supervisor:	Prof. Dr. Thomas Neumann
Advisor:	Dr. Alexis Engelke
Submission Date:	15.09.2023

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

Ich versichere, dass ich diese Bachelor's Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Ort, Datum

Jim M. R. Teichgräber

Acknowledgments

Most of all, I would like to thank my advisor Dr. Alexis Engelke for his continued support, advice, and confidence throughout this thesis, especially at times when I was lacking the latter. Thank you to Mona Wich and Tom von Allwörden for interesting (and occasionally hilarious) discussions, and now and then enduring a bit of rubber-ducking[a]. I would also like to thank everyone else with whom I got to share my time at TUM, especially Peter Shan, Daniel Haselbeck, Tobias Kamm, Johannes Stöhr, Vincent Limbach, Jana Huhne, Robin Marchart, and Florian Kraupa.

[a] Special thanks to Tom for reminding me that plastering `printf` statements everywhere can solve almost any problem.

Abstract

Compiler frameworks are used across applications and domains to speed up and simplify the development of compilers. LLVM has long been the framework of choice for domains that aim to accelerate their workloads through the superior run-time performance of native machine code. The extensible MLIR framework has stepped in to provide more flexibility in modeling domain-specific semantics right in the IR, and allows users to leverage the potential of domain-specific transformations on a common platform. However, MLIR currently lacks the capability to directly lower operations to machine code, which makes it hard to map domain-specific semantics straight to machine code for improved compile-time performance.

We present MoNaCo, a fast native code generator that directly converts MLIR to machine code, capable of letting users specify exact lowerings for performance critical operations. At the same time, MoNaCo explores the viability of MLIR for JIT compilation, where compile-times are critical.

MoNaCo outmatches MLIR's existing code generation path in terms of extensibility and outperforms it in terms of compile-time: across the MCF benchmark from SPEC CPU 2017 and the Dhrystone benchmark, MoNaCo's compilation times are about 48% lower on average, while the resulting code is about 4 times slower on average. Our approach shows that MLIR is a viable option for efficient compilation, but several issues in the design and implementation of MLIR hold back its potential in JIT compilation.

Contents

1 Introduction

Motivation

Today, *just-in-time* (JIT) compilation is a pivotal technique to speed up the execution of programs with dynamic source, or source code that remains unknown until execution. This includes applications in database query execution [1], machine learning [2], and programming language implementation [3, 4, 5]. In JIT compilation, low compile-times are essential, as they contribute to the overall execution latency.

Compiler frameworks can reduce engineering effort in (JIT) compiler development substantially by providing data structures, a common platform for transformations and analyses, and machine code generation. But these attempts at generalizing and abstracting the compilation process can lead to compromises in representing the semantics of the program. Translating to a common ecosystem defined purely by the compiler framework means losing domain-specific knowledge that is crucial to transforming programs in order to optimize their performance characteristics [6].

This has led users of compiler frameworks to develop components dedicated to domain-specific information. These not only require extra effort to build, but each component still only represents the program on a single level of abstraction. The novel MLIR compiler framework [7] provides its users with the ability to combine representations of multiple levels of abstraction in the same code. It allows them to define custom operations and types which can be incorporated into the program representation. MLIR also includes a state-of-the-art framework for applying generic program transformations through user-defined patterns. In combination, this allows domain-specific transformations to be defined on domain-specific information, alongside the rest of the program.

However, MLIR's ability to express domain-specific information does not extend to machine code generation: the current code generation path requires translation of all program information to the LLVM [8] instruction set, which operates at a single, fixed level of abstraction. This makes it hard to influence the final machine code that is generated to execute the program; thus the vital advantage of JIT compilation, native machine code execution, cannot utilize the full potential of domain-specific information, as it, once again, gets lost in translation.

Approach

To this end, we present the MoNaCo code generator, which employs a novel approach to generating machine code directly from MLIR. MoNaCo defines an MLIR dialect representing x86-64 instructions, allowing them to be used like any other MLIR operations, without the constraints of fixed register assignment. We then implement a code generation

schema for MoNaCo that traverses the x86-64 dialect program a single time. Within this single traversal it assigns registers to values, converts MLIR constructs such as block arguments to machine code, and encodes the finalized instructions. This empowers the user to define exactly how their MLIR operations should be lowered to machine code and therefore allows taking advantage of domain-specific information for machine code generation. As MoNaCo solely represents the program as MLIR before emitting machine code, it avoids multiple different code representations and thus allows for faster compilation times than the existing code generation path. Performance results show that MoNaCo can compile the Dhrystone benchmark [9], as well as the MCF benchmark from the SPEC CPU 2017 suite [10] between 1.34 times and 2.7 times faster than the established LLVM code generation path, while incurring up to a 4.6x run-time slowdown.

Contributions

The key contributions of this thesis include:

- An alternate machine code generation route for MLIR, which does not rely on the LLVM compiler infrastructure.
- JIT-focused x86-64 code generation for MLIR with fine-grained control over the generated machine code. This includes both predefined conversions from common operations to an MLIR dialect representing machine instructions, and the ability to define custom conversions for any operation.
- Compile-time improvements over MLIR's current code generation route. In our tests, MoNaCo required between 25% and 63% less compile-time than the baseline.

Outline

In Chapter 2, we will first describe the compilation process in the abstract, the place of compiler frameworks within it, and the emergence of MLIR. We will then detail how we built MoNaCo to address issues with the current code generation process for MLIR in Chapter 3. Thereafter, in Chapter 4, we test and identify performance characteristics of the MoNaCo code generator and compare it to the existing code generation path for MLIR. Then, we cover work related to MoNaCo and MLIR code generation in Chapter 5, and finish by summarizing our work and findings, and looking out to future work in Chapter 6.

Throughout the thesis, we will be referring to the source code of MoNaCo and the LLVM project; specifically, we will refer to the MoNaCo commit `8a3c8e01`[1] and the LLVM commit `a403d75be7`[2], except where otherwise noted.

[1] https://github.com/J-MR-T/MoNaCo/tree/8a3c8e01274c93304bf4f64902fdf3692c1dd397
[2] https://github.com/llvm/llvm-project/tree/a403d75be7add73f3e34032d73c81b8e1dcba3b9

2 Background

In this chapter, we will outline key steps and data structures in the compilation process, and then proceed to examine the need for compiler frameworks in general, as well as their extensibility. We then briefly describe the MLIR compiler framework.

2.1 Lowering, Compilation, and just-in-time

Lowering describes the process of reducing a program representation to another program representation, which is more aligned with the hardware that is meant to execute the program. The term lowering is used because the conversion usually takes place from a higher-level representation to a lower-level one, although it can be hard to quantify what makes a representation high- or low-level. This means *compilation* includes a sequence of lowering steps, with *transformations* in between. These transformations are often colloquially referred to as 'optimizations', but this term is misleading: first, the objective of the optimization (run-time, code size, compile-time) is unclear, second, not all transformations are guaranteed to actually improve the metric they are targeting. We thus prefer the term 'transformations', although we use some established terms deriving from 'optimizations'.

To enhance flexibility, compilers typically employ architectures that can be divided into front-end, optimizer, and back-end. Each of these components can be seen as taking a program representation as input, transforming it, and passing it to the next component [11]. As for the beginning and end of this pipeline, the front-end gets its input as a program string from the user, while the back-end typically emits machine code. This code is then either linked and executed immediately, or written to a file in object form, for linking at a later stage. Representations of the program in-between the input string and output code are called *intermediate representations* (IRs)[1]. The front-end is responsible for transforming the input language, which is often designed to be well understood by humans, into a language which is more suitable for the compiler. The optimizer performs transformations to try to improve various aspects of the IR. The back-end lowers the IR into actual machine code; this process is called *code generation*.

This architecture simplifies bridging the *semantic gap* between human understandable input language and machine understandable output language. Note that in many cases, each of these components conceptually divides into further parts that also transform an input representation into an output representation.

When the program source or semantics are unknown until just before the execution, traditional, so called *ahead-of-time* (AOT) compilation is not possible. These kinds of programs, like database queries and programs in highly dynamic languages, can either be

[1] Also *intermediate languages*, *intermediate codes*.

interpreted or JIT compiled. For JIT compilation, low compile-times are a priority, as JIT compile-time adds to the end-to-end delay between requesting execution and the result of execution. JIT compilation usually results in faster execution time than interpretation for hot code, although compilation is never worth the time on sections of code which are only ever executed once [12]. For these sections, interpretation should be preferred.

2.2 Static Single Assignment Form

The textual program representation supplied by the user is generally not feasible for use throughout the compilation process:

- the representation is highly language specific;
- there is no inherent notion of transformations;
- the representation does not expose information about context, or types;
- data flow, meaning information about which values are used where, is not explicit.

The last point is especially important, as data flow information is the basis for many transformations. This leads us to consider alternate, more suitable program representations.

2.2.1 Structure

Static single assignment form (SSA form, or simply SSA) [13, 14] is the intermediate representation used by most modern optimizing compilers and compiler frameworks, including GCC [15], LLVM [8], HotSpot JVM [4], and the V8 JavaScript engine [3]. Fundamentally, it consists of *values*, that have exactly one definition. There is no variable assignment to these values afterwards. Consequently, values form a chain starting at their definition, leading to their uses. This chain establishes a data flow DAG. Control flow is typically represented by dividing the program description into so-called *basic blocks*, these are the largest possible units of code that do not contain control flow themselves.[2] Each basic block is *terminated* by exactly one control flow instruction.

As there is always only one definition of a value, conditional logic requires special handling: a value cannot simply be assigned to multiple times, based on the condition. Take Listing 2.1 as an example: the control flow of the program determines which of the two definitions of x is used for the call to `print`.

In SSA form, these values can be represented as arguments to the basic blocks (*block arguments*)[3]. Listing 2.2 shows Listing 2.1 transformed to SSA; basic block d has an argument x_d, with which it can then call the function `print`. Blocks b and c terminate by unconditionally branching to block d, and passing their respective definition of x as an argument to the block.

This means that SSA basic blocks can also be interpreted as a set of functions, (conditionally) calling each other with particular arguments and operating on some common

[2] For this definition we consider function calls to eventually return to execution to the call site; they are not viewed as control flow.

[3] ϕ-nodes are another possible representation. As MLIR exclusively uses block arguments, we do not discuss ϕ-nodes further.

```
int x;    // end basic block a
if(condition())
  x = 1; // end basic block b
else
  x = 2; // end basic block c
print(x);// basic block d
```

Listing 2.1: *Conditional logic in C*

```
block a:
  v1 ← condition()
  cond_branch v1, block b, block c
block b:
  branch block d(1)
block c:
  branch block d(2)
block d(xd)
  print(xd)
```

Listing 2.2: *Listing 2.1 transformed to SSA*

state. Each SSA value is seen as occupying one of an infinite set of *virtual registers*, each of which stores exactly one SSA value.

SSA values can only be used after they are defined. If a value is defined in a basic block y, but used in a different block z, every possible path from the entry block x to block z must go through y. On a path from x to z that does not visit y, the use of the value in z would rely on an undefined value. This property is called *dominance*, here y dominates z. Compilers employing SSA thus need to perform dominance checks to confirm that the IR is well-defined.

The main advantage of SSA over other program representations, is that it makes data flow explicit. This allows transformations to easily eliminate dead code or redundant computations, as SSA provides direct access to all data dependencies of a value definition [13, 14].

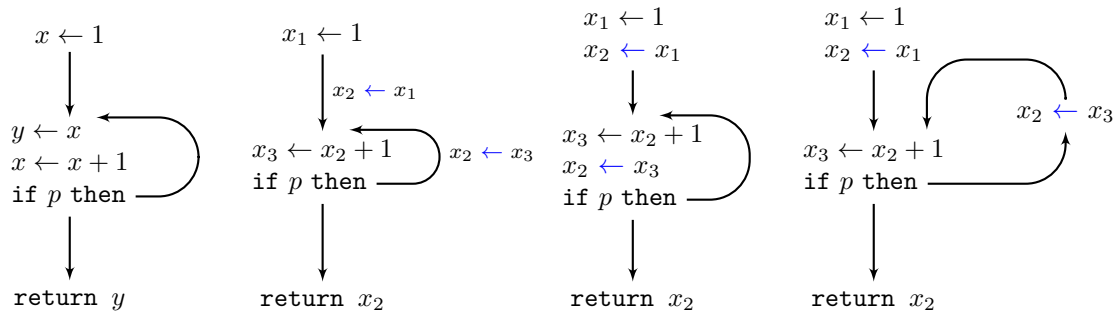
2.2.2 Challenges

Although SSA has many advantages for transformations and analyses, it leads to challenges in generating machine code. Because processors have a limited set of registers, it is necessary to decide which of the SSA values occupy which hardware register at what time. If, at one point, there are more values needed than there are machine registers, some values need to be *spilled* to stack frame slots to free up hardware registers.

In addition, as machine code has no notion of block arguments, these need to be resolved to hardware registers or stack slots. By writing the block argument's possible values to the same location, its values are merged into one. This is similar to the original representation as one variable in the source, like x in Listing 2.1.

However, simply inserting instructions to move values to the correct location is not sufficient for correctly lowering SSA to machine code. Consider the example in Figure 2.3: the original code returns the value of x in the second to last iteration of the loop. When transformed to SSA, x becomes a block argument to the main block of the loop (see Figure 2.3b). If we now naively insert move instructions to copy the value that is passed to the block argument to the block arguments location, we accidentally change the semantics of the code; now the value of x in the last iteration is returned (see Figure 2.3c).

This issue is known as the *lost-copy* problem [16]. It occurs only on edges of the *control flow graph* (CFG) that lead from a block with multiple successors, to a block with multiple predecessors, so-called *critical edges*. If the target block had only one predecessor, it would



(a) *Original code* **(b)** *Code in SSA form with block arguments* **(c)** *Copies inserted (incorrect)* **(d)** *Copies inserted, critical edges split (correct)*

Figure 2.3: *Lost-copy problem: naive copies lead to incorrect code (adapted from [16])*

not need block arguments to merge values from its predecessors. If, on the other hand, there were no other successors to the block, i.e. the block had no other incoming CFG edges, the unconditional moves performed to place the block argument's value could not overwrite a value set by another edge. One way to solve the lost-copy problem is to split the critical edge by inserting another block that is only executed when the edge is actually taken; this block then houses the unconditional moves for this edge, ensuring that they do not overwrite a value from another edge (see Figure 2.3d).

The other problem during SSA destruction is the *swap problem* [16]: consider that arbitrarily many block arguments can be passed concurrently to a block in the abstract model of SSA; but setting arbitrarily many values in machine code cannot be done concurrently on known architectures, and simply setting them sequentially can lead to cyclic dependencies. Take the example in Listing 2.4: if we naively insert copies into this code to move the block arguments, we end up with the incorrect Listing 2.5, where v_1 and v_2 now contain the same value after the first iteration.

```
block(v1, v2):
  print(v1)
  print(v2)
  branch block(v2, v1)
```

Listing 2.4: *SSA loop printing alternating values (without copies)*

```
block: // v1, v2 implicit
  print(v1)
  print(v2)
  v2 ← v1
  v1 ← v2
  branch block
```

Listing 2.5: *Listing 2.4 in SSA-destructed form (with naive copies, incorrect)*

In the original code, they simply alternate values. To address this, we need to temporarily save the original value of v_2 to assign to v_1 afterwards. Note that this is not a critical edge, as the block has only one successor: itself. Also note that this instance of the swap problem conveys the general idea, but does not cover all possibilities. More details on SSA can be found in [17].

2.3 Lowering an Intermediate Representation to Machine Code

Lowering from an IR to machine code is usually divided into three steps [18]: *instruction selection*, *instruction scheduling*, and *register allocation*.

In instruction selection, IR operations need to be mapped to actual machine instructions provided by the target ISA. This is largely done through pattern matching. In its simplest form, often called *macro expansion*, every IR operation is expanded into one or more ISA instructions, meaning each individual operation is simply matched by one pattern. More complex patterns, which operate on data flow trees, DAGs, or arbitrary graphs, attempt to combine more complex IR behaviors into more efficient instruction sequences.

Instruction scheduling describes the process of optimizing the ordering of instructions, to take advantage of instruction-level parallelism. The goal is to improve run-time performance of the code. Instruction scheduling is therefore an optional part of code generation and can be omitted where compile-time is critical, such as in JIT compilation.

During register allocation, the set of infinite virtual registers used by the IR needs to be mapped to ISA registers and stack memory. The target ISA and ABI typically impose additional constraints on register allocation:

- Some ISAs, such as x86-64, feature instructions that constrain the registers in which their operands must reside.
- ISAs restrict which instructions can have memory operands. This can necessitate insertion of additional instructions to spill-load or spill-store values, as well as temporary registers to hold values that should be moved between memory locations.
- Two-address instructions that destroy one of their operands (*two-operand instructions with destructive source*), such as `ADD rax, rcx` in x86-64, require the register allocator to allocate the same register to the destroyed operand and the destination.

SSA is destructed at the latest during register allocation, minding the problems described in the previous section.

Each of these processes can employ additional *peephole optimizations*. These analyse a small part of the code at a time (the so-called *window*), and try to combine or transform instructions. Aho et al. [19] provide a more detailed explanation.

2.4 Compiler Frameworks

To reduce engineering effort in compiler development, compiler frameworks can provide data structures for a complex IR, code generation facilities, and handling of peripheral tasks such as object file writing. The instruction set of a compiler framework can either be fixed, or extensible. An extensible instruction set allows users of the framework to add and use custom operations.

2.4.1 LLVM

LLVM [8], one of the most widely known compiler frameworks, chose to provide a prespecified instruction set for its SSA-based intermediate representation known as LLVM-IR. It captures ideas from high-level languages as well as the key operations of hardware ISAs:

- the LLVM-IR `switch` instruction models a C-like `switch` statement;
- there are arithmetic instructions such as `add`, `sub`, ...;
- specialized instructions, like `cmpxchg` to atomically modify memory, map to hardware equivalents.

LLVM thus models an abstract machine designed for ease of transformations and code generation.

LLVM has simplified compiler development substantially. Any operation that is needed in C, can be expected to have a fitting equivalent in LLVM-IR, as the Clang C compiler compiles to LLVM-IR. After translating their source language into LLVM-IR, the user mostly does not need to concern themselves with the details of code generation. In addition, they are not faced with the design challenge of defining their own custom IR instructions.

A fixed instruction set also simplifies LLVM's development, as instruction selection only has to be implemented once per IR instruction per target ISA. Afterwards, any LLVM-IR program can be instruction selected, and the rest of machine code generation can ensue.

Generally, LLVM has achieved a lot of success and has established itself as the backbone of numerous processes involving code generation, from databases employing code generation for queries [1] to programming language implementations for Rust, Swift, or Julia [7]. A downside of a fixed instruction set is that transformations can mostly only happen on one fixed level of abstraction, namely LLVM-IR.[4] To work around this, some LLVM users, such as LLPC [20], have inserted function calls to pseudo-functions representing their custom operations; but these are opaque to the optimizer, and cumbersome to do custom transformations on. The alternative is to represent the operation more verbosely in its individual parts, by utilizing the instructions provided by LLVM-IR; although this ultimately leads to more complex and less expressive code.

LLVM itself tries to address this issue using *intrinsic functions* (*intrinsics* for short), which use the same mechanism: defining pseudo-functions that represent particular semantics not found in the instruction set itself. Intrinsics range from high-level operations like `llvm.memcpy`, to some that model abstract functionality that is often represented in ISAs, like `llvm.prefetch`, and others that correspond to a specific ISA instruction, such as `llvm.x86_ssse3.pshuf_b` which maps to the x86-64 `PSHUFB` instruction. As intrinsic functions are defined by LLVM itself, they are better integrated into the compilation process, but they are ultimately still a workaround.

The need for domain-specific transformations has thus led many LLVM users, like Rust, Swift, and Julia, to opt to build their own, custom, higher-level intermediate representations. This allows them to perform transformations that would be difficult to do on LLVM-IR with its limited expressiveness hiding the original semantics of higher-level operations. This negates some of the advantage of using a compiler framework, as the user now does have to implement at least the IR data structures from scratch, as well as a basic infrastructure for performing transformations.

[4] Some transformations can be done in the back-end during code generation.

2.4.2 MLIR

These issues led to the development of MLIR [7], which provides an infrastructure for building an intermediate representation with user-registerable instructions, as well as some predefined instructions for common use cases, referred to as *in-tree*[5]. Fundamentally, MLIR is also SSA-based. MLIR code can arbitrarily combine user-defined and predefined instructions from different so-called *dialects*, which group instructions and data types.

This allows domain-specific knowledge to be encoded by representing it as native MLIR types and operations, these make transformations on multiple levels of abstraction easily possible with MLIR's built-in generic pattern rewriting infrastructure.

Extensibility also leads to higher complexity, as it requires an all-around modular design; the compiler developer needs to be able to store various kinds of information about an instruction. As a result, there are competing ways of accomplishing the same task. This includes two separate domain-specific languages to define transformation patterns, in addition to being able to define them in C++ code; two ways of defining compile-time-known information on an operation; and redundant ops between dialects, for example, the `constant` operation in the `arith` [21] and `llvm` [22] dialects, or the `funcop` operation in `func` [23] and `llvm`.

Internals

We will only provide a brief overview of MLIR concepts and terminology. For further details, refer to the documentation [24], including the language reference manual [25].

Generally, each instruction is represented by a class deriving from the MLIR `Op` class; instructions are thus referred to as 'ops' or 'operations', we use these terms interchangeably. An operation has zero or more SSA values as results, zero or more SSA-valued operands, as well as zero or more attribute operands, which represent operands that are set at compile-time for each instance of an operation. An integer comparison operation, for instance, could have

- one result, `true` or `false`;
- two SSA-valued operands, the two values to compare;
- one attribute operand — the comparison predicate — for instance 'less than'.

Attributes like this are independent of any SSA value, but tied to the instance of the operation; another instance of the compare op might use 'greater than' as its predicate.

Parts of an operation that are not specific to the particular instruction, like the sequence of results and SSA value operands are stored in the `Operation` class. An `Op`-derived class is implemented as a smart pointer wrapper around a pointer to `Operation` [24]. Note that therefore, no classes derive from `Operation`, but each op derives from `Op`. Attributes and types, on the other hand, are stored in the `MLIRContext` and only *referenced* in the `Operation` class. The storage of the `MLIRContext` deduplicates any values; as a result, all references to attributes that share the same value point to the exact same storage. For example, in one `MLIRContext`, there is only ever one point in storage that stores the bool attribute with the value `false`; every op that uses a bool attribute with the value `false` actually has a pointer to this original `false` attribute. This allows attributes to

[5] This refers to the directory tree of the MLIR source.

be compared for equality using a simple pointer comparison. The context also stores miscellaneous options, such as whether multithreading is enabled. We can now more clearly describe an MLIR dialect as a collection of definitions of operations, attributes, and types.

Note that around May 2023, MLIR gained the ability to use so-called properties instead of *inherent* attributes [26] — these are the only kind of attributes we present here. Like attributes, properties can store any compile-time-known data. However, properties have the advantage of being stored together with the rest of the op in the `Operation` class, unlike attributes. Accessing attributes thus requires additional indirection through the `MLIRContext`, and adding or removing them results in the recomputation of a hash map. The storage for properties, on the other hand, is known at the time the op is constructed, and there is no hashing involved when accessing or updating them. The main advantage of attributes over properties is that comparing their pointers for equality is often cheaper than comparing properties, which requires a bytewise comparison that is expensive for aggregate data such as strings.

MLIR's notion of compilation units, *modules*, are also represented as ops. We call a module consisting of arbitrarily many different dialect operations *heterogeneous*, and a module consisting of one or a select few dialects *homogeneous*.

To guarantee the availability of common information or functionality on an operation, i.e. to enable generic interaction with the extensible instruction set, MLIR provides *interfaces*. For example, the `mlir::BranchOpInterface` can be used on different types of branching operations, to gain access to the supplied block arguments for a successor block.

3 MLIR-based code generation

In this chapter, we describe how we leverage MLIR's potential to build a code generator for the x86-64 (AMD64) architecture.

3.1 Problem and Approach

The existing path to generate machine code from an MLIR module lowers the module to LLVM-IR and then reuses the LLVM code generator. This lowering from MLIR to LLVM-IR is divided into two steps (see the start of the left path in Figure 3.1): first, the heterogeneous MLIR module consisting of ops from any dialect is converted to an MLIR module only containing ops from the `llvm` dialect, which implements a subset of features of the LLVM instruction set. As this conversion depends on the semantics of the ops, conversions for ops of custom dialects need to be specified by the user. Then, an existing converter translates the `llvm` dialect MLIR module to an LLVM module. Differences between MLIR and LLVM regarding metadata, function signatures, global values, and more[1] are handled here, and individual op translations provided by the op definitions themselves are executed. Finally, the LLVM code generator [27] takes care of generating actual machine code. The entire process is configurable through the LLVM C++ API.

Using LLVM as the code generation toolchain for an MLIR module has several shortcomings:

1. Speed: this process involves at least 60 passes over the program representation, translation between MLIR and LLVM data structures, as well as conversion between several LLVM-internal code representations. All of this leads to subpar compile-time performance.
2. Loss of semantics: the precise modeling of op semantics via custom operations that MLIR enables, is lost in the translation to the fixed LLVM instruction set and only *then* to machine code.[2]
   In addition, certain transformations lose some of their effectiveness in the translation. For example, an MLIR module with eliminated common sub-expressions may produce an LLVM module that could benefit from running another common sub-expression elimination transformation. This results in unnecessary effort.
3. Loss of analysis information: both MLIR and LLVM track information gained through their analyses in order to minimize recomputation of the same information. However, this information is not compatible between MLIR and LLVM pass infrastructures, thus any transformations run on the LLVM-IR generated from the MLIR module have to do their own analyses from scratch.

[1] See MLIR's `lib/Target/LLVMIR/ModuleTranslation.cpp`, starting around line 1380.

[2] Besides the aforementioned domain-specific information, this also extends to certain low-level semantics, for instance information about arithmetic carries.

LLVM, however, has the following advantages for the user:

4. Comfort: the user typically does not need to concern themselves with the specifics of code generation after lowering to the `llvm` dialect.
5. Portability: lowerings to the `llvm` dialect suffice to be able to execute the code on any LLVM-supported platform.

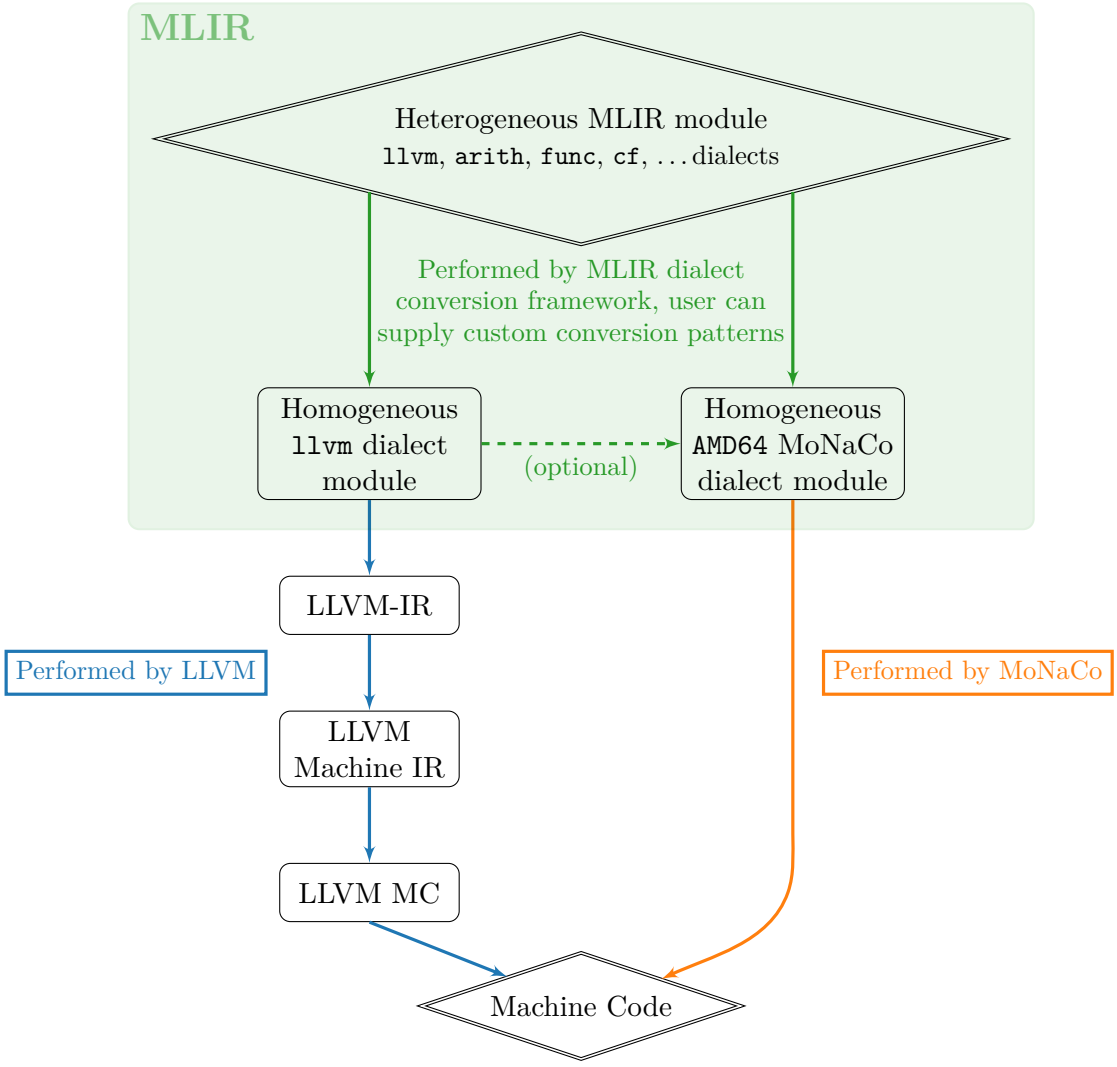


Figure 3.1: *LLVM-based MLIR code generation (left) vs MoNaCo (right)*

We propose the ***MLIR to Native Code*** *generator* (MoNaCo), which uses a different approach to address these shortcomings. MoNaCo leverages MLIR's existing dialect conversion infrastructure to first perform instruction selection, by converting the heterogeneous MLIR module to a homogeneous module. This module only consists of functions that are made up of MLIR ops from a target-specific dialect. Each op from this dialect corresponds to exactly one opcode in the target ISA encoding. We chose to implement the first MoNaCo back-end for the x86-64 (AMD64) architecture, so in the `AMD64` dialect,

each op representing an instruction corresponds to exactly one AMD64 opcode. In a second pass, MoNaCo destructs SSA form, allocates hardware registers to each value, emits a data section for global variables, and encodes the instructions (see the right path in Figure 3.1). It thus addresses the shortcomings of the LLVM-based code generation process as follows:

1. Speed: MoNaCo was designed with JIT compilation in mind and can emit JIT-execution-ready machine code in only two passes over the whole IR.
2. Semantics: a MoNaCo user can provide conversions of their custom operations to the `AMD64` dialect, similar to the conversions needed to lower custom operations to the `llvm` dialect. As this lowering directly targets the ISA, the user can decide precisely how to bridge the semantic gap between their operations, and the target, without being constrained by the LLVM instruction set in between.
3. Analyses: MLIR's pass infrastructure can be used to perform transformations such as instruction scheduling, or peephole optimizations on the instruction selected IR, although this is currently not implemented.
4. Comfort: a MoNaCo user also does not need to concern themselves with destructing SSA, or the later stages of code generation. SSA destruction, register allocation, and encoding are handled seamlessly by the MoNaCo back-end.
5. Portability: MoNaCo's portability, however, is limited to supported targets (currently only x86-64). MoNaCo provides partial support for lowering the `llvm` dialect, so it can be used as a drop-in replacement for the LLVM code generator. To leverage MoNaCo's full potential, the user can then decide to provide custom lowerings for performance critical operations. Due to their precision, these lowerings are platform-specific and need to be performed for every relevant target-specific dialect.

3.2 Target-specific dialect design

We will illustrate the design of a target-specific dialect for MoNaCo using the example of the `AMD64` dialect. As we want the dialect to represent an instruction-selected MLIR module, the dialect has to model machine code closely in order to be easily translatable, while still maintaining and profiting from SSA form.

The `AMD64` dialect is designed to represent all semantic information necessary to lower an op from this dialect to an instruction in machine code. The simplest solution to this problem is to map each MLIR op to exactly one opcode in the target encoding. This way, the encoding routine can decide which opcode to use based solely on the C++ type of the instruction it is currently encoding. As an x86-64 opcode also specifies an instruction's operand type, as well as its operand and result bit widths, the `AMD64` dialect consists of ops that are fully qualified in regard to these properties, for example:

- The `ADD8rr` op corresponds to an 8-bit addition of two registers.
- The `MOVSXr64m32` op corresponds to a sign-extending move of a 32-bit value in memory, to a 64-bit register
- The `SUB16ri` op corresponds to a subtraction of a 16-bit immediate from a 16-bit register.

This representation is close to other compiler back-ends like LLVM Machine IR (MIR) [27, 28] or the `gc` Go compiler's back-end [29].

The information about the opcode of an instruction is one example of information that needs to be represented in the ops of the `AMD64` dialect. Generally, all information that is needed can be split across two axes. First, it matters whether the information is static, i.e., independent of a specific op instance; the second axis is whether the information is attached to a specific result of an op, or the whole op. MLIR has first-class support for storing static, as well as non-static information per op, but not per op result. We will describe each of the four possible cases in turn, along with their implementation using MLIR facilities; the following table provides an overview of these coming sections:

	Static	**Per Instance**
Whole Op	Example: opcode Subsection 3.2.1	Example: immediate operand Subsection 3.2.2
Op Result	Example: result register constraint Subsection 3.2.3	Example: register allocated to result Subsection 3.2.4

Table 3.2: *Overview of necessary op information in `AMD64` across two axes*

3.2.1 Static information per op

Information that is static per op includes the aforementioned opcode of the instruction, by design of the dialect; register constraints that constrain the register an instruction operand needs to be in (*operand register constraints*); comparison predicates used by conditional instructions, such as `Jcc`, `CMOVcc`, and `SETcc`.

The only remaining pieces of static information per op are whether the instruction has an immediate operand, and whether it belongs to a group of generically modeled special cases. For example, all variants of the `DIV` instruction (`DIV8r`, `DIV8m`, ...) inform the encoder to zero-extend the first of their two x bit wide operands to $2x$ bits. This is necessary because the `DIV` instruction always performs $2x$ bit / x bit = x bit division. For instance, if we try to represent an unsigned 64-bit/64-bit = 64-bit division in the IR semantics with `DIV`, it assumes `rdx:rax` as the 128-bit dividend. So to ensure that only the 64 bits in `rax` are used, `rdx` needs to be zeroed. The same applies for `IDIV`, but here, sign-extension needs to be performed.

Implementation We implement static information per op using static methods on each of the ops, and an MLIR interface to specify which ops provide what information. For example, every op representing an instruction in the `AMD64` dialect implements the `getFeMnemonic` method, which returns a mnemonic that informs the encoder which opcode to choose for this instruction. The `InstructionOpInterface`, implemented by all ops representing an instruction (*instruction ops*), indicates that an opaque `Operation` implements the `getFeMnemonic` method. Likewise, the `InstructionOpInterface` also enables querying the operand register constraints of an instruction; comparison predicates are only retrievable on ops that implement the dedicated `PredicateOpInterface`.

Simple, boolean-like information, like whether the op has an immediate operand, or whether it belongs to a certain class of special cases, can also be attached to the ops using MLIR *traits*. In fact, MLIR internally uses traits to indicate that an op implements a specific `OpInterface`. Op interfaces can thus be viewed as traits that also guarantee that an op implements a set of static and non-static methods specified by the interface.

3.2.2 Information per op instance

The most common kind of information that needs to be saved per op instance, is information about instruction immediates. For example each instance of `ADD32ri` can have a different immediate to add to a 32-bit register.

Implementation As outlined in Section 2.4.2, information varying per op instance can be stored inline with the op using MLIR properties or attributes; attributes' main advantage is the ability to compare them cheaply. As we require little to no comparisons of attribute-like data, we almost exclusively opt for properties in the `AMD64` dialect. Symbol references are the only kind of information we store as attributes instead. Typically MLIR symbol references are stored in an `mlir::FlatSymbolRefAttr`, so it was more idiomatic for us to store them as attributes as well. Additionally, symbol references in the `AMD64` dialect never change after construction, so we also avoid the aforementioned recomputation of the attribute dictionary.

To indicate that an op has a certain kind of property, we also use MLIR interfaces. The `ADD64ri` op's immediate operand is stored in its `InstructionInfo` property; the `InstructionOpInterface` indicates that an op has this property. Note that as instruction ops implement the `InstructionOpInterface`, all of them allocate the space for an immediate in their `InstructionInfo`. Not every instruction needs an immediate, but this overapproximation simplifies the compilation process at the cost of a higher memory footprint, specifically 8 additional bytes per operation.

3.2.3 Static information per op result

Static information per op result includes the bit width of the result, and register constraints specifying the exact register an instruction result is provided in (*result register constraints*). For a 64-bit division, the `DIV64r` instruction returns the quotient in `rax`, and the modulus in `rdx`.

Implementation The only information that is directly accessible on an MLIR op result, is the result's type, which can also provide methods accessible through MLIR interfaces. We chose to model the bit width and register bank of a value as a type, with a hierarchy of MLIR type interfaces that allow access to this information about the type. The `GPRegisterTypeInterface` is implemented by the `gpr8/16/32/64Type` types representing general purpose registers; the `FPRegisterTypeInterface` is implemented by the `fpr32/64Type` types, which represent floating-point registers. Both of these interfaces inherit from `RegisterTypeInterface`, which supplies a method to access the bit width of the register.

We chose to model only this information as types, as these types are close to a one-to-one representation of types in the typical source dialects like `arith` or `llvm`. Modeling register constraints as types would be quite far from the typical use of types in MLIR, and would thus increase the complexity of the code dealing with these types.

In addition, the integer types directly model an exact class of registers, thus providing simplicity in translation. We decided to differentiate between 32-bit and 64-bit floating-point types in the dialect, even though they are both stored in the same `XMM` class registers, as this makes it easy to operate separately on single- and double-precision floats, as most instructions do.

Any per result information that is not represented using types, needs to be stored on the op itself instead. This is the case for the result register constraints; they are retrievable as a `std::pair`-like structure through a static method on the op, as there are never more than two result register constraints in the instructions that are currently in use.

3.2.4 Information per op result per op instance

The register allocated to a result is the primary kind of information saved per op result per op instance. It is important to note that this is non-static information that typically varies over the course of the compilation process. It always represents where this result is found *at each point of use*. Consider this simplified sequence of ops in the `AMD64` dialect:

```
v3 ← ADD64rr v1, v2
v4 ← CALL @print (v3)
...
RET v3
```

Due to ABI constraints, this example first requires the value v_3 in the `rdi` register, to pass as argument to the call, and later in the `rax` register, to use as the return value. So at the time of encoding the `CALL` (`RET`) instruction, the register information on the op result of the `ADD64rr` will be in the `rdi` (`rax`) register respectively.

Implementation Storing information per op result per op instance is not directly supported by MLIR. Here, we employ the same strategy as with result register constraints, and use a custom fixed-size collection to store allocated registers per result in the properties of the op.

3.2.5 Special cases

There are some exceptions and special cases in the dialect's design that need to be dealt with.

Register information on block arguments Unfortunately, register information cannot just be stored on ops. We also need to save register information on block arguments, which have no inherent storage attached, as MLIR basic blocks are not extensible. This is handled through the use of a separate hash dictionary mapping block arguments to their register in the register allocator. An alternative would be storing this information about block arguments on the function op that owns the block. This would introduce more

complexity by requiring a custom `AMD64` function op, a custom non-fixed size collection as a property, and added indirection when updating the information on the block arguments. Thus, we opted to first implement the simpler solution using the hash map.

Meta ops and memory In addition to instruction ops, the `AMD64` dialect defines a small number of non-instruction ops (*meta ops*) that represent other information about the resulting machine code, such as addresses of globals and functions, as well as memory operands or stack slots allocations. All of these ops require special casing in the register allocator or encoder, which is why they are employed sparsely, only when there is no simpler option. These ops do not implement the `InstructionOpInterface`, this way they are distinguished generically, in addition to the required special casing. So to confirm that an op is a normal instruction, it is not necessary to check all non-instruction special cases to see that none of them apply. Instead, presence of the `InstructionOpInterface` can be checked as the first condition.

Memory operands Memory operands are represented as meta ops returning the special `memLoc` type representing a memory location. This allows memory ops to be used as `memLoc`-type operands of instruction ops. Modeling memory operands as ops allows them to have their own properties and SSA-valued operands, which, in combination, allows memory ops to mimic the scale-index-base-displacement (SIBD) memory operands allowed by x86-64: the scale and displacement values are both fixed at compile-time and can therefore be represented as properties on the memory op. Both the index and the base are specified in a register in machine code, so they can be an integer SSA value in the SSA representation of an SIBD operand. Note that pointers to memory are not represented using the `memLoc` type, as they are fundamentally integers that can also be used to perform arithmetic. But when a pointer is used as a memory operand, it is converted to a `memLoc`, by using a memory op that consists only of the base part of an SIBD operand. Conversely, if the address of a `memLoc` type is needed, a `LEA64rm` instruction is used to retrieve it, as it only performs address computation with its memory operand.

Modeling memory operands as ops enables more than just SIBD addressing, other meta ops can take advantage of the flexibility of the `memLoc` type, for example the `AllocaOp`:

Allocas The `amd64::AllocaOp` can be used to allocate an n-byte wide stack slot in the entry block of a function. It returns a `memLoc`, so it can be used like any other memory operand, as can be seen in Listing 3.3. If the alloca replaces an equivalent alloca from a source dialect, like `llvm.alloca`, its original uses are replaced with a `LEA64rm` instruction with the alloca as its operand, as outlined above. As this `LEA` is not always necessary, we take special care to try to forgo it in favor of using the alloca directly, wherever possible.

Conditional branches Another special case pertains to conditional branching. To keep the fundamental SSA structure intact, conditional jumps in the `AMD64` dialect are not trivially equivalent to their machine code counterparts. As with all other instruction ops, there is exactly one conditional jump op per opcode, i.e. there are `amd64::JZ`, `amd64::JNZ`, etc. In our model, every conditional jump has two successors blocks, and still has all their

```
// allocate 8 bytes on the stack, returning a memory location
%0 = amd64.Alloca 8:i32 -> !amd64.memLoc
// create the 64-bit integer constants 42, 43
%1 = amd64.MOV64ri 42:i64 -> !amd64.gpr64
%2 = amd64.MOV64ri 43:i64 -> !amd64.gpr64
// save 42 to the stack slot
amd64.MOV64mr %0, %1 -> ()
// add 42 to 43, by loading 42 from the stack slot
%3 = amd64.ADD64rm %2, %0 -> !amd64.gpr64
```

Listing 3.3: *Simplified MLIR example of the `AllocaOp`*

block argument operands attached, like a typical MLIR branch op does. This has key advantages, over trying to represent lower-level conditional jumps in the dialect:

- The SSA structure is maintained. Consider the example in Listing 3.4, which shows an alternate modeling of conditional branches that uses two jumps at the end of a block, each with one destination block. If we chose this alternate modeling, there would be control flow in form of a `Jcc` *inside* the basic block. This would be difficult to represent in MLIR, and ultimately not in the spirit of modeling machine code as SSA. Instead, we maintain clear SSA structure as can be seen in Listing 3.5.

```
block a:
  v3 ← CMP64rr v1, v2
  JLE v3, block b
  JMP block c
```

Listing 3.4: *Alternate modeling of conditional branches using two ops*

```
block a:
  v3 ← CMP64rr v1, v2
  JLE v3, block b, block c
```

Listing 3.5: *Listing 3.4, like modeled in `AMD64`*

- During instruction selection, we do not need to worry about moving the destination operands to registers yet, this task can be left to the register allocator. This also means the user does not need to concern themselves with the exact lowering of conditional jumps.
- In the encoding phase, we can try to optimize the block ordering to encode fewer jumps by potentially inverting the predicate of a conditional jump and leveraging fall-through.

Information separate from MLIR The last special cases we want to describe again relate to information and how it is represented in the dialect. Specifically, we chose not to model some necessary information using the MLIR facilities.

First, use of the `RFLAGS` register is not tracked in the dialect, all flag-setting instructions do so implicitly, meaning that the IR is not aware of flags, except for some special cases that will be demonstrated in the following. In the provided conversion patterns, flags are only used to save the result of a `CMP` using a subsequent `SETcc`, or to perform a conditional jump or move, using `Jcc` or `CMOVcc` respectively. Not modeling flags in the IR made the dialect simpler, but it means that, when flags are used, the instruction order becomes relevant, and the pattern must pay attention not to overwrite the flags between their

computation and their use. For example, if the condition of a branch is produced by a `CMP`, the `CMP` will be duplicated right before the `Jcc` instruction; otherwise an `AND8ri` is emitted to logically AND the arbitrary `i1` condition with 1, before a `JNZ` is performed to mimic the behavior of conditional branches in the IR. Thus the use of flags must be fully specified by the conversion patterns that produce the flag-setting and flag-using instructions. Another drawback of not modeling flags explicitly arises in conjunction with dead code: if flags set by instructions other than `CMP` or `AND` are used, MoNaCo should be configured with `-fno-codegen-dce`, as it could omit these instructions, if they are trivially dead except for their impact on flags. For this reason, `-fno-codegen-dce` is the default. Fully modeling flags in the `AMD64` dialect is left as future work.

Global values are also not represented using MLIR. These are implemented through a custom data structure representing each global, containing its address, data, and alignment. All globals are inserted into an `llvm::StringMap`, indexed by their symbol name. This way, globals can simply be accessed from anywhere using their symbol name, without requiring the indirection through MLIR's symbol table functionality. As we emit the `.data` section before the `.text` section, all globals need to be handled in encoding, before any function is; if we represented globals as ops, we would have to either maintain a strict ordering to keep all globals before all functions in the `AMD64` dialect module, or iterate twice over the entire module, handling globals on the first iteration, and functions on the second. Both of these options would add unnecessary complexity to the encoding process. Instead, we can simply iterate over the aforementioned `llvm::StringMap`'s values to emit all globals for the `.data` section, and then iterate over the module to emit the functions to the `.text` section. Furthermore, globals in the `AMD64` dialect are only ever referenced using their address, which is retrieved using the special `amd64::AddrOfGlobalOp`. This allows us to keep the SSA structure intact at the point of accessing and using a global; it also better represents that globals are fundamentally just data in the final machine code, compared to viewing them as global ops, suggesting that they are instructions of some kind.

3.2.6 Dialect implementation in TableGen

The TableGen language is a metaprogramming language developed for use in the LLVM code generator [30]. MLIR employs TableGen to generate C++ source code which can define dialects, including operations and types. The goal is to make it easier to write the TableGen source, than it would be to write generated C++ code. For example, at one point in the development process of MoNaCo, the `AMD64` dialect specified in TableGen was about 266 times smaller in terms of ASCII characters than the combined C++ header and implementation files generated from it.

To demonstrate how to specify an instruction in TableGen, we will use the `ADD` instruction in all its forms as an example. We make heavy use of the TableGen `class` and `multiclass` facilities, which generically specify information about so-called *records*, in this case, these are the resulting C++ op classes. Note that it is technically not necessary to use (`multi-`)`class`es, they only serve to reduce code duplication and improve readability. A TableGen `class` collects common definitions through *fields* which are read by the MLIR TableGen back-end, on each of the records that use this `class`. The C++ code that is generated by

the back-end determines the exact meaning of each field. To make classes more generic, they can use template arguments, which can then be used in the definition of fields. MoNaCo defines the `Instruction` class which bundles most of the functionality for the `AMD64` dialect instructions, including `ADD`. `Instruction` derives from the MLIR-provided `Op class` and contains definitions of static methods used to represent static op information, like the opcode:

```
class Instruction<string mnemonic>: Op<mnemonic,[InstructionOpInterface]>{
  let extraClassDeclaration = "static constexpr FeMnem getFeMnemonic() {
      return FE_" # mnemonic # "; }";
  // ...
}
```

`let` overrides one of the fields of a `class` higher up in the hierarchy, in this case the `Op class`. `extraClassDeclaration` is a field on the `Op class` which can be used to inject arbitrary declarations into the generated C++ source for an op. This value is supplied as a string containing C++ source code, in this case it is constant for all parts except the exact name of the `FE_...` macro defining the opcode. The instruction that is currently being defined passes this last part to the `Instruction` class via its `mnemonic` template argument. The mnemonic's name is then concatenated to the existing string using the TableGen concatenation operator `#`. This way, each op in the dialect can pass its mnemonic to the `Instruction class`, which then declares an appropriate static method to retrieve the opcode from the Fadec encoder library. Note that the real `Instruction` class declares more methods through the `extraClassDeclaration` field than shown here, and derives from a subclass of `Op`, that bundles more functionality.

Defining all forms of the `ADD` instruction really means defining multiple ops. This is because `ADD8rr` has a different mnemonic from `ADD8ri` or `ADD16rr`, and each op corresponds to exactly one opcode, as explained in Section 3.2. We define multiple ops at once using TableGen `multiclass`es. The `ADD` instruction is defined through a more complex version of the following `multiclass`:

```
multiclass Instr_rmi<string mnemonic, Type t> {
  let operands = (ins t:$src1, t:$src2) in
  def "rr" : Instruction<mnemonic # "rr">;

  let operands = (ins t:$src1, memory:$src2) in
  def "rm" : Instruction<mnemonic # "rm">;
  // ... for "mr", "ri", "mi"
}
```

This way, we can define variants of the `ADD` instruction with different operands, using the `let ... in` statment to specify the types and names of the operands, and the `def` statement to define records for ops with the *suffix* `rr`/`rm`/etc.

To use this multiclass to define all variants of `ADD` for all four bit widths, we use the `foreach` statement. The following shows a simplified version:

```
foreach bitwidth = ["8", "16", "32", "64"] in
  defm ADD # bitwidth : Instr_rmi<ADD # bitwidth, "gpr"#bitwidth#"type">;
```

The `defm` statement here defines an op using a multiclass. We have now defined `ADD8rr`, `ADD8rm`, `ADD8mr`, `ADD8ri`, `ADD8mi`, `ADD16rr`, ... ops. As most x86-64 arithmetic instructions are similar, we can reuse the `class`es outlined here for instructions such as `SUB`, `AND`, `OR`, `XOR`, and more, to define these instructions with just one additional line of TableGen source each.

To specify more complex behavior and special cases, such as the aforementioned register constraints and information about comparison predicates, we again use the `extraClassDeclaration` field.

3.3 Instruction Selection

Instruction selection in the context of MoNaCo and the `AMD64` dialect, means translating the ops of a heterogeneous module, including any dialects, to a homogeneous module in the `AMD64` dialect. More specifically, ops that structure the code, such as module and function ops, can remain the same or be translated one-to-one to an equivalent in the `AMD64` dialect. On the other hand, ops that define the semantics of the program need to be lowered to actual instructions of the `AMD64` dialect. As dialect conversion is a typical use case in MLIR, and is also necessary in the LLVM-based code generation process, MLIR provides a dialect conversion framework.

3.3.1 Use of the MLIR Conversion Framework

We will now detail how the MoNaCo instruction selector uses the MLIR conversion framework. An `mlir::ConversionTarget` indicates to the conversion framework which operations the converted module is allowed to use. Operations that are allowed in the final module are referred to as *legal* operations, the others as *illegal*. Legality can also be defined on types, as well as entire dialects, to declare all their ops and types legal or illegal at once.

To perform pattern matching on the IR, the user of the conversion framework provides patterns defined as C++ classes, which inherit from one of the subclasses of `mlir::ConversionPattern`. These implement either a `matchAndRewrite` or separate `match` and `rewrite` methods, which are used by an `mlir::ConversionPatternRewriter` to perform pattern matching on the IR. Patterns can have *benefits*, which indicate how beneficial it is to match a certain pattern. Prior to the matching itself, the pattern rewriter sorts patterns in descending order of benefit. Please see Listing 3.6 for an example of such a pattern.

An `mlir::TypeConverter` can be used to register and apply type conversions. In translating to the `AMD64` dialect, general purpose register types can be expressed using the `amd64::gpr8/16/32/64Type` types, while the `amd64::fpr32/64Type` types can be used for floating-point register types. We chose to translate the MLIR `i1` one-bit-wide integer type representing boolean values to the `amd64::gpr8Type` for consistency with other operations, e.g., the `SETcc` instructions, which conceptually produce a boolean value and also operate on the 8-bit wide variants of the general purpose registers.

To allow custom behavior, MoNaCo's instruction selector takes a pattern set and a type converter as arguments, which the user can populate with patterns and type

```
using namespace mlir;
                          // only try to match arith::AddIOp
struct Add64Pattern: public OpConversionPattern<arith::AddIOp> {
  // Constructor to specify lowest possible benefit of 0
  Add64Pattern(TypeConverter& tc, MLIRContext* ctx)
    : OpConversionPattern<arith::AddIOp>(tc, ctx, 0){}

  LogicalResult matchAndRewrite(arith::AddIOp addOp, OpAdaptor addAdaptor,
      ConversionPatternRewriter& rewriter) const override {
    // This pattern only matches 64-bit additions
    if(addOp.getType().getIntOrFloatBitwidth() != 64)
      return rewriter.notifyMatchFailure(addOp, "bitwidth");

    rewriter.replaceOpWithNewOp<amd64::ADD64rr>(
      addOp, // replace the original add with an ADD64rr
      addAdaptor.getLhs(), // use converted left-hand side
      addAdaptor.getRhs());// and right-hand side

    return success();
  }
};
```

Listing 3.6: *An MLIR conversion pattern which matches a 64-bit addition*

conversions for the dialects that they wish to convert to `AMD64`. Users can employ predefined conversions of in-tree dialects and types to `AMD64` via `populate...` methods such as `populateArithToAMD64ConversionPatterns`. This way the user can decide which dialects and types should be included in the conversion; to simplify prototyping we also provide a method which adds all predefined conversions to a pattern set and type converter. The instruction selector finally invokes the conversion framework to perform the actual pattern matching.

Although the MLIR conversion framework supports generic DAG-to-DAG rewriting, we decided to constrain almost all the patterns we implemented for simplicity: each pattern tries to match a single type of operation for a single bit width, regardless of its arguments, and expands it to one or more instructions, similar to macro expansion. MoNaCo first supported the essential parts of the `func`, `arith` and `cf` dialects; now it includes support for lowering parts of the `llvm` dialect as well. Due to their simplicity, these patterns do not employ some more efficient instruction encodings: in most cases, immediate operands are materialized using a `MOV__ri` instruction, instead of being encoded into the instruction itself; the same applies for operands operating on memory and `MOV__rm` or `MOV__mr` instructions. Another consequence of the simple patterns is that for 8- and 16-bit operations, the code generated by MoNaCo includes partial register dependencies which can stall the pipeline [31].

MoNaCo's aforementioned support for instruction-selecting a subset of the `llvm` dialect allows users to use their existing lowerings to `llvm`, and then make use of MLIR's transitive lowering, to seamlessly lower to `AMD64` afterwards. This enables MoNaCo to be integrated easily into existing codebases. The user can incrementally add support for more detailed, direct lowerings of their custom operations to `AMD64` where it makes sense to take advantage

of an immediate end-to-end translation of user-defined operations to machine code. As we defined the patterns to lower parts of `llvm` after we already added support for other dialects, we could take advantage of MoNaCo's ability to perform instruction selection independent of the later stages of code generation: except for ops that required new features, like globals, lowerings for `llvm` were written without any changes to the register allocator.

3.3.2 Notable Special Cases

We would like to note a few special cases that we encountered during instruction selection, and how we handled them.

Globals As explained in Section 3.2, MoNaCo manages globals in a custom data structure. A reference to this data structure thus needs to be passed to all patterns that need read- or write-access to globals.

Additionally, external global symbols need to be resolved to their address. As MoNaCo is focused on JIT code generation, we first implemented this for the JIT mode; in AOT compilation, references to unknown symbols require ELF relocations, which can be resolved by the linker. Emitting these is not currently implemented.

In the `llvm` dialect, globals can have initializer regions, which can house ordinary `llvm` dialect ops to initialize complex global values. In typical use cases, these often employ operations that are not found in the rest of the module. Additionally, the initializer regions need to be executed before the register allocation and encoding stage, to determine their fixed initial value and write it to the `.data` section. As a result, we decided to implement the translation of complex initializer regions using LLVM, by first converting the global to LLVM-IR, and then extracting the necessary information from it. This could be improved by executing the MLIR ops in the initializer region with an interpreter at compile-time, and setting the initial value of the global to the interpretation's result; the development of such an interpreter is left as future work.

Switch statements The MLIR `cf` and `llvm` dialects provide a C-style switch statement as an IR operation. These switch statements operate only on integer values, which simplifies the lowering process, as only integers have to be compared. In addition, the IR switches always have a default case.

Generally, there are three main ways to lower a switch statement to machine code; we will only outline these, a more detailed explanation can be found in [32, 33].

1. Use a linear sequence of `if`-style comparisons of the base value to each case. This is referred to as a *linear search*.
2. Employ *binary search* to reduce the asymptotic number of comparisons to $\log(n)$, where n is the number of cases. This is also referred to as using a *comparison tree*.
3. Generate a *jump table* where the i-th entry contains the address of the target block for `case i`, or the default block if `case i` is not specified; then use the base value to index into the table, to find the target block.[3]

[3] To reduce the size of the jump table, one can subtract the smallest case value from the base value.

In their most common form, jump tables require max (`case value`) − min (`case value`) entries, and can thus not be used on highly sparse switch-case statements. Furthermore, implementing jump tables in MoNaCo would also require a new n-way branch instruction to emulate the indirect branch in SSA, which would then require additional special casing during encoding, to actually generate the table.

We thus opted to use the binary search technique over jump tables, as it can handle all switches regardless of their case layout, and it could be implemented without changes to the rest of the compilation process. In comparison to the linear `if`-sequence, binary search has better asymptotic complexity, thus improving worst-case performance.

Note though, that both the jump table and the linear search might yield better performance in some cases. Especially for highly predictable switches that almost always branch to the same one or two blocks, a linear search ordered by estimated frequency would excel in performance.

Address computation We will demonstrate how to perform address computation to index into aggregate data structures in the `AMD64` dialect using the example of lowering the `llvm` dialect's `getelementptr` (GEP) op. This op models the LLVM-IR instruction with the same name. Broadly, `getelementptr` uses successive indices, to compute the exact address of a nested sub-element, in some cases with arbitrary depth. For a detailed explanation, please refer to the LLVM language reference manual [34] as well as the page dedicated to explaining the `getelementptr` instruction on the LLVM website [35].

In general, we iterate through the indices and emit instructions to add the computed offset for the current index to the base address. The final base address obtained in this way is then returned as the result of the lowered `getelementptr` op. Each GEP op can have a mixture of fixed, attribute-like indices, and dynamic, SSA-valued indices. We will first cover the fixed case, and then extend the solution to the dynamic case.

For fixed indices, MoNaCo currently handles `structs`, arrays, pointers, and integers as the type to index into (referred to as the *base type*). When we encounter an array or pointer base type, we multiply the supplied index by the size of the element type, and use this as the immediate to be added to the base address. For a struct base type, we use the same method as `LLVMStructType::getTypeSizeInBits`[4] to calculate the size and offset of individual members. The only difference between the two approaches is that `getTypeSizeInBits` traverses the entire `struct`, whereas we stop at the element with the supplied index. We use the size of the `struct` up to this point as the offset to add to the base address. In the case of an integer base type, we generate an instruction to add the supplied index, multiplied by the size of the integer type, to the base address.

When handling dynamic indices, there are only two differences to the above:

1. We generate an `IMUL64rri` instruction to perform any necessary multiplications, instead of performing them at compile-time.
2. Struct base types are not handled, as the `llvm` dialect forbids dynamic indices into structs.

Note that we skip generating the add instruction, if the computed offset is zero. This avoids unnecessary instructions. We would also like to point out, that we exclusively

[4] See MLIR's `lib/Dialect/LLVMIR/IR/LLVMTypes.cpp` lines 560 to 580

utilize the `ADD64rr`, `ADD64ri` and `IMUL64rri` instructions to do address computation. This was done to simplify the lowering process at the possible cost of run-time speed of the generated code; in particular, we do not employ the scaled-index addressing modes offered by the x86-64 architecture [36], as using them would not only increase the complexity of the lowering, but they also only cover a subset of all possible cases.

3.3.3 Custom lowerings to the target-specific dialect

This section details the process of providing custom lowerings from user-defined operations to a target-specific dialect, again using `AMD64` as an example.

1. To add user-defined conversions to the MoNaCo instruction selection process, first create an `mlir::RewritePatternSet`, and an `mlir::TypeConverter`.
2. Design and specify type conversions to lower types of custom dialects to register types. For an example of this, see MoNaCo's various `populate...TypeConversions` methods.
3. Create patterns lowering the custom dialect operations to `AMD64`; see the MLIR documentation [24] for details. The available patterns[5] can be used as a starting point.
4. Then call the appropriate `populate` functions provided by MoNaCo, to add relevant predefined type conversions and conversion patterns to the `TypeConverter` and `RewritePatternSet`. Also add the custom type conversions and patterns defined earlier.[6]

Note that this simple version of the process can only employ predefined `AMD64` instructions. Should the user wish to utilize more instructions, they can exploit the extensibility of certain MLIR dialects: the `AMD64` dialect is an `mlir::ExtensibleDialect`, a feature originally designed to provide users with the ability to modify a dialect at run-time. We have chosen to make the dialect extensible in order to provide full flexibility to a MoNaCo user, allowing them to define new instructions at run-time. For details, please refer to the MLIR documentation [24], as well as the C++ header and implementation files generated from MoNaCo's own instruction definitions.

3.4 Register Allocation, SSA Destruction, and Encoding

Register allocation and encoding are performed in a single pass over the instruction-selected IR; SSA is destructed on the fly.

This pass traverses the IR in reverse post-order (topological order) of the CFG. This is to ensure that the definitions of values are visited before their use, and that as many predecessors of a basic block as possible are visited before the block itself [17]. The latter property is desirable, as it provides us with the most information possible, allowing for better register allocation and spill decisions, although the current register allocator does not use this information.

[5] See the first section of MoNaCo's `src/isel.cpp`.
[6] See MoNaCo's `src/isel.cpp` around lines 1750 to 1800.

Note that in the following, we will not distinguish register constraints specified by the System V ABI [37] and register constraints imposed by the hardware, as they mostly require the same handling in practical terms.

Many x86-64 instructions constrain their operands or results to specific registers, or require that their result is in the same register as one of their operands. To simplify the register allocation process under these constraints, we devised a partition of the available registers. The three general purpose register families `AX`, `DX`, and `CX`, with 64-bit registers `rax`, `rdx`, and `rcx` are designated as *operand registers*, all other registers are *storage registers*.

As a heuristic, operand registers are never used to permanently store values; they are treated as entirely volatile, and in most cases it is assumed that their contents could change arbitrarily from one instruction to the next. These specific registers were chosen because almost all register constraints use these registers. For instance, `DIV` and `MUL` are constrained to parts of `rdx` and `rax`, and the variable shift instructions can only use the `cl` register as their second register operand.

This heuristic also allows operand registers to be used whenever a temporary register is needed, without needing to spill any value; as there are three of them, one is free in almost any situation in which the need for a temporary register could arise. Temporary registers are needed throughout the register allocation process, such as to solve the swap problem in SSA destruction, or to move a value from an arbitrary memory location to another, as can also happen in SSA destruction. Another example includes the calling of external functions in JIT compilation, here MoNaCo emits the following instruction sequence[7]:

```
MOVABS rax, // address of function to call
CALL rax
```

Due to the volatility of `rax` in our model, this can be done safely whenever required, and does not require spilling any values.

We treat storage registers and stack slots almost interchangeably: unlike operand registers, storage registers are never used in a way that could overwrite the value they hold, if that value is still used after the instruction; this further simplifies the allocation process. Both storage registers and stack slots combined are hereafter referred to as *value slots* or simply *slots*.

As for SSE registers, `XMM0`, `XMM1`, and `XMM2` are designated operand registers for the same reasons as the general purpose registers. The remaining registers are again storage registers. We still employ operand registers for SSE registers, to unify the approach between both register banks, even though most SSE instructions do not have operand register constraints, with the notable exception of `BLENDVPS`.

3.4.1 Register Allocation Strategy

To simplify register allocation, we decided to first implement an entirely stack-based allocator. This allocator effectively does not use any storage registers; it assigns a unique stack slot to every value. Note that this is a significant overapproximation of necessary

[7] Note that in the case of variable argument functions, the ABI [37] requires `al` to contain an upper bound on the number of SSE registers that contain arguments. This is not currently supported.

stack space, as the allocator does not compute at what point in the program values are not needed (*live*) anymore.

Whenever a value is used, it is loaded from the stack into a fixed operand register, regardless of the current contents of that operand register. This also means it is possible for the same value to be loaded into the same operand register, even though the register has not been cleared; in trivial cases, MoNaCo can be compelled to omit such unnecessary spill loads (`-fomit-one-use-value-spills`). The fixed operand registers are chosen in a way that respects operand register constraints of all instructions currently in use by the `AMD64` dialect.

As the current strategy does not use storage registers, we can use `rbp` to store the stack frame base pointer without truly sacrificing a register. This simplifies stack accesses, especially in the case of calls to functions that require stack arguments, as they result in a variably-sized stack frame for the caller.

It is clear, that this strategy results in code which inadequately uses both the available registers and the available stack-space. But this kind of stack-based allocator is simple to implement, and can be used to test the correctness of the rest of the compilation process and of a more sophisticated allocator that could be implemented in the future.

To allow a future register allocator to be implemented without unnecessary code duplication, we have divided the functionality of the register allocator. An abstract base class tries to make as few decisions regarding register allocation strategy as possible while bundling as much functionality as necessary. A concrete implementation class then actually makes allocation decisions, utilizing the generic functionality provided by the base class.

The abstract base class can thus take care of SSA destruction and the insertion of code to move values between slots, as well as to and from operand registers. It also determines the iteration order and calls the functionality of the concrete implementation class in the right order, accounting for necessary special casing.

The concrete implementation class provides methods to load an SSA operand of an instruction from its slot into an operand register, as well as to allocate and encode an instruction, once all its operands are loaded. To perform encoding, it calls into a further `Encoder` class, which again encapsulates functionality.

3.4.2 SSA Destruction

Whenever the register allocator encounters a CFG edge, it inserts instructions to move the values for the block arguments of the target block to the appropriate value slots; it considers both critical edges (lost-copy problem), and data dependencies (swap problem), as outlined in Chapter 2. The swap problem is handled equivalently to Briggs et al. [16].

Lost-copy problem As previously mentioned in Chapter 2, a critical edge flows from a block a with multiple successors, to a block c with multiple predecessors. Note that the `AMD64` dialect currently does not use indirect branches for internal control flow, so a control flow edge can either be caused by a conditional branch, or an unconditional branch. As a has multiple successors, an unconditional branch is impossible for the (a, c)

edge. Therefore, as the (a, c) edge is guaranteed to be produced by a conditional branch, we know the `RFLAGS` register contains a specific value if the branch is to be taken.

The simplest way to solve the lost-copy problem is often cited as *splitting* the critical edge into a separate block. Splitting the edge, and inserting the move instruction in the new block b, instead of at the end of a, ensures that the move is only executed when this edge is actually taken. In particular, this means that the move cannot overwrite a value from any of the other edges targeting c, which also insert move instructions.

Instead of splitting the critical edge, we propose to execute the moves on a critical edge using the `CMOVcc` instruction, and place them at the end of block a as normal. This ensures they are only executed when the edge is taken; there are no additional comparisons needed, as the `CMOVcc` instructions can simply use the same predicate and flags as the conditional jump itself. We can therefore avoid dealing with the management overhead of inserting a new split-block when some blocks have already been encoded and a fixed block iteration order has been computed beforehand. Another advantage of this approach is that, if the block argument resides in a register, the direct `CMOVcc` to a register does not require an additional temporary register. Although when moving from either a memory location or a register *to* a memory location, a temporary register `rT` is necessary: first, the *destination's current value* is unconditionally moved to `rT`. Then, the source value is conditionally moved to `rT`, potentially overwriting it. Finally, we unconditionally move the value from `rT` to the destination.

Another approach to solving the lost-copy problem requires a distinct temporary register *per CFG edge*, that stays live until block c [38]; our temporary register on the other hand can be reused between all CFG edges and only has fixed liveness of a few instructions. This same alternative approach would also require changing the register of the block arguments' uses in the following block, which would again add complexity.

Note that in the case of a memory-to-memory move, any traditional `MOV`-based approach would also require a temporary register. However, there are also some disadvantages to the `CMOVcc`-based approach: first, the allocator that is currently implemented never allocates registers to block arguments, for simplicity; so at the moment, only the approach involving the temporary register can be used. Second, the `CMOVcc` instructions perform worse than simple `MOV` instructions in a number of ways:

1. Due to their conditional nature, move elimination [31] is not possible for `CMOVcc` instructions, thus `CMOVcc`s cannot have zero latency.
2. On modern Intel architectures, `CMOVcc` instructions require the complex decoder (see [39] for detail), meaning that in most cases four times as many `MOV`s could have been decoded in the same time [40, 41].

All in all, the `CMOVcc` approach thus results in worse run-time performance on the critical path, but it requires only one temporary register and no additional blocks, simplifying the compilation process.

3.4.3 Encoding

Globals and `.data` As explained in Section 3.2, the first step in encoding is to emit the `.data` section, by iterating over all globals, and writing their data to the section, while leaving space to respect their alignment requirements. During this phase, the addresses of globals are also saved. Externally defined global symbols are ignored here, as their address is already known at this point.

Function prologue and epilogue Afterwards, the registers for each function are allocated, and the function is encoded using the Fadec encoder library. However, before this can happen, the function *prologue* needs to be emitted. The prologue includes the allocation of the function's stack frame, preservation of callee-saved registers, and saving the function's register and stack arguments. The aforementioned stack frame base pointer in `rbp` also needs to be set up here. The function *epilogue* emitted before each `RET` instruction needs to clean up the stack frame and restore callee-saved registers. Most of this is straightforward, but allocating the stack frame presents some challenges: at this point in the compilation process, we have not yet register-allocated the function, so we do not know the size of the stack frame in advance. We thus emit a phony stack frame allocation instruction, and save its location, to be corrected when we know the size of the stack frame at the end of the function; the same is done for all stack frame deallocations in all epilogues. But x86-64's variable-length encoding leads to another problem: encoding `SUB rsp, 0` as the phony allocation only occupies four bytes in the efficient encoding, because the smallest immediate encoding possible is used, which allows for an 8-bit immediate. If we were to rewrite this with a stack frame allocation larger than 255 bytes, a larger encoding would be used and would overwrite part of the code that follows. Instead, we use the seven-byte wide `SUB rsp, 0x01000000` as the phony allocation, to ensure the encoder chooses the biggest possible immediate size of four bytes. We can then manually overwrite the four byte immediate with the actual value of the stack frame allocation; we know this immediate to start exactly three bytes after the start of the instruction itself. During this allocation, we also take care of aligning the stack pointer to a 16-byte boundary like the System V ABI [37] requires.

Note that it would also be possible to fully re-encode this seven byte instruction with the new immediate, resulting in possibly less than seven bytes to be written, and proceed to fill the bytes that were not used with one-byte wide `NOP` instructions (`0x90`). However, it is both easier to just overwrite the immediate and better in terms of run-time, as `NOP`s still require execution time [42].

Function body To encode the body of the function, the aforementioned allocate and encode method in the register allocator calls the `Encoder` class to encode each instruction. The allocated register is first written to the value's register information as explained in Section 3.2, this is then used by the `Encoder` class to pass on to the Fadec encoding library. Note that MoNaCo can be configured to omit trivially dead instructions (`-fcodegen-dce`); these can either occur in the source module, or be generated by the instruction selection in some cases.

Passing registers to Fadec Not all registers used by an instruction need to be passed to Fadec though. First, there are the two-operand instructions with destructive source, i.e., that return their result in the register of the first operand; for example `ADD rax, 42`. Fadec only needs to be passed `rax` once to encode this instruction, even though there are actually two SSA values involved, as can be seen in the equivalent MLIR code: `%1 = amd64.ADD64ri <{immediate = 42}> %0`. This case is recognized using the `amd64::Operand0IsDest0` MLIR trait, which can be attached to an op in its TableGen definition; it compels the encoder to omit passing the register of the first operand. In addition, fixed *result* register constraints do not need to be passed to Fadec at all. The `MUL` instruction combines both of these properties: in the IR it can be used to multiply any two values and handle their result just as any other SSA value: `%2 = amd64.MUL64r %0, %1`. But the actual instruction can only be passed a single register, which is multiplied with `rax`; the result resides in `rax` as well. So the encoder first omits the `rax` destination register, because it is implicit, and then recognizes that `MUL` is conceptually a two-operand instruction with destructive source (although the destructive source is implicit), and thus omits the register of the first operand. As a result, the encoder ignores the registers of the `%2` and `%0` values from the example, to correctly call Fadec only with the appropriate `MUL` mnemonic, and the register of `%1`.

Note though, that some *operand* register constraints do need to be passed to Fadec, for example the `cl` constraint of the second operand of the variable shift instructions.

Encoding memory operands Memory operands are represented with the `memLoc` type in the `AMD64` dialect. Any op returning a `memLoc` implements an `encode` method, defined along with the rest of the op in TableGen through inline C++. An `AllocaOp`, for instance, uses Fadec's memory operand facilities to return an offset into the stack frame from its `encode` method. The `amd64::EncodeOpInterface` lets the `Encoder` know, that an operand supplies its own encoding, which the `Encoder` can then query to encode the memory operand.

Unlike the passing of registers to Fadec and the encoding of memory operands, some other special cases cannot be generalized and need to be handled on a per-instruction basis. As explained in Section 3.2, these are also indicated to the encoder via MLIR traits; the prominent example being the variants of the `DIV` and `IDIV` instructions.

Function calls Function calls need various special handling. As MoNaCo prioritizes JIT code generation, we chose to first support the handling of external functions for the JIT mode. External functions include those from the `libc` and other libraries. Like with externally defined global symbols, emitting relocations for non-JIT external calls is not implemented yet. During JIT compilation, `dlsym()` is used to retrieve the address of an externally defined function. If the symbol is not defined, compilation stops and `dlerror()` is used to detail the failure. `dlsym()` and `dlerror()` are also used for retrieving the address of an external function through the `amd64::AddrOfFunction` meta op.

When calling external functions, we decided against using a procedure linkage table (PLT) in favor of simply moving the address of the external function into a register (`rax`), and performing an indirect `CALL`. This is also the default strategy employed by widely

used JIT compilation frameworks like LLVM. Additionally, emitting and tracking a PLT would have increased the complexity of the encoder further.

Before a call is performed, function arguments need to be moved to the correct registers and stack slots. We currently support an arbitrary amount of integer arguments including those passed on the stack, and up to eight floating-point arguments (passed in `XMM0` to `XMM7`). Care needs to be taken, as stack arguments:

1. are rounded up to 8 bytes [37], we handle this by always performing a `PUSH` with 8-byte operand size;
2. can break the 16-byte alignment of the stack pointer [37] with their 8-byte size (we conditionally push 8 phony bytes to the stack to prevent this);
3. are passed in reverse order [37], so we collect the integer stack arguments and iterate over them in reverse order as soon as all are known.

Calls and branches to parts of the code which are not yet generated need to be noted, and their offsets need to be fixed up at the end of the encoding process. Fadec supports this with the `FE_JMPL` flag which compels Fadec to use the longest possible jump encoding. If this is supplied both when a phony branch to an unknown target is inserted, and when resolving this branch later on, the instruction sizes are guaranteed to be the same.

3.5 Limitations

Although MoNaCo does have support for lowering many operations, it is not a fully mature code generator yet. We would like to note a few of the features still missing due to the constrained scope of this thesis.

First, support for complex types like structs is limited. Struct types can be indexed into using the GEP instruction, but passing them as block arguments, function arguments, or returning them is not supported yet. Float-type block arguments are not supported either, and support for the lowering of floating-point operations is sparse. Vector types utilizing modern CPUs' SIMD capabilities are not supported at all.

Stack unwinding and thus C++ exceptions are unsupported. Debug information and the writing of ELF relocatable object files are not supported. Globals that get initialized using an MLIR initializer region can only be initialized to zero. We do not support functions with more than one return value.

There is only partial suport for the System V ABI [37] and its complex argument-passing and value-returning rules. In particular, defining functions with variable arguments is entirely unsupported, calling functions with variable arguments only works as far as it would be equivalent to calling a normal function.[8] Passing more than eight float arguments, meaning some would be passed on the stack, is unsupported, as is returning and passing of struct types as mentioned before.

All the limitations above are only due to the required implementation effort, and not because of any structural limitations in MoNaCo. However, MoNaCo currently only supports x86-64; support for other target architectures would require significant structural changes to support a new target-specific dialect and extend the register allocation plus encoding pass to support the new target.

[8] This turns out to be enough to reliably call `printf`.

4 Evaluation

To benchmark the performance of MoNaCo, and compare it to that of the LLVM-based code generation process for MLIR, we chose the `505.mcf_r` SPECrate benchmark ('MCF') from SPEC CPU 2017 [10], a suite designed to resemble real world applications; the rest of the benchmarks from SPEC CPU 2017 were omitted due to the limited scope of this thesis. MCF itself uses a network simplex algorithm to schedule public transport vehicles. We ran the benchmark in single-threaded mode and used the reference input workload size. We also used the Dhrystone benchmark [9], specifically version 2.1, modified to be strictly ANSI C compliant [43], compiled with `-DTIME` and `-DDHRY_HZ=100`. We set Dhrystone to 1,000,000,000 iterations for run-time tests.

We will first compare MoNaCo's compile-time performance with LLVM's, then compare the run-time performance of the generated code.

4.1 Setup

Each of the two C benchmarks was compiled with `clang -O0` and `clang -O2` to generate four sets of LLVM modules in total. Each set of modules was then linked into a single LLVM module with `llvm-link`. Next, every module was converted to MLIR using `mlir-translate -import-llvm`, resulting in a total of four MLIR modules in the `llvm` dialect. For the `-O2` compilations, we additionally added `-fno-vectorize` and `-fno-slp-vectorize`, to prevent Clang's autovectorization engine from emitting vectorized IR, which MoNaCo cannot handle. `mlir-translate -import-llvm` inserted dead code pertaining to globals into the MLIR module, which we eliminated manually, because it contained operations that MoNaCo does not support yet. Finally, the MLIR module was JIT-compiled using MoNaCo, once with its native code generation process, once in its fallback configuration in which it calls LLVM's MCJIT through an `llvm::EngineBuilder`. The engine was instructed to use LLVM's FastISel instruction selector which is the fastest one currently available, and to emit position independent code with a small code model[1]; LLVM was also configured to use the lowest possible code generation optimization level of `None`, to make LLVM as compile-time-performant as possible. MoNaCo was configured to omit trivial cases of unnecessary stack spill loads and dead code (`-fomit-one-use-value-spills`, `-fcodegen-dce`); this did not impact compile-time performance beyond the margin of error.

All tools and code of the LLVM project were built with `-fno-omit-frame-pointer` and `-fno-omit-leaf-frame-pointer` from commit `a403d75be7`. MoNaCo was built from commit `80c014e`. LLVM, MLIR, and MoNaCo itself were compiled with `gcc` version 13.1.1

[1] See the System V ABI [37] for a more detailed explanation

on `-O3`. MoNaCo was linked statically against MLIR and LLVM, but linked dynamically against glibc 2.37 and libstdc++ 6.0.31.

The benchmarks were run on an AMD Ryzen 7 5700U system clocked at 1.8 GHz, with CPU frequency scaling disabled; there were 16 GiB of main memory available, running at 3200 MT/s; the system ran Arch Linux kernel version 6.3.6.arch1-1 in 64-bit mode.

To ensure that measurements were reproducible and taken at least one second apart, compile-times were measured after 1000 repetitions, and run-times after 10 repetitions. The times for an individual run were obtained using the arithmetic mean. Note that we tried to only measure parts of the compilation process that occurred both in MoNaCo and in LLVM JIT compilation. However, due to the nature of MoNaCo's memory management, we were not able to include a call to `mmap` to allocate the memory in which the code is stored, and a call to `mprotect` to set the execute bit for pages containing code.

To further analyse individual components of the two compilation processes, we used the *perf* profiler [44]. Note that as a sampling profiler, perf does not provide exact measurements. Wherever possible, we thus validated these measurements through repetition and by using other tools, such as LLVM's `TimeTraceProfiler`.

4.2 Compile-time Performance

When measuring the compile-time performance of MoNaCo, we took combined measurements of the entire compilation process, as well as isolated measurements of the two phases of MoNaCo JIT compilation: instruction selection, and register allocation plus encoding. In all cases, the sum of the two isolated measurements deviated from the combined measurements by about 0.1% to 1%. As the individual measurements provide additional insight, we will only report these and their sums.

We will use the number of lines of MLIR code in the input module as a very rough estimate of code size. The major caveats of this estimate are:

- Not every line of IR code represents the same amount of information; some lines contain a constant op, while others contain a complex op like GEP.
- Between optimized and unoptimized versions of the module, some lines cannot get reduced, such as metadata, global declarations, and function declarations.

4.2.1 Results

Benchmark	LLVM (ms)	MoNaCo (ms)	$\frac{\text{MoNaCo}}{\text{LLVM}}$	LLVM FastISel failures	MLIR module lines
Dhrystone `-O0`	10.35	4.73	0.457	6	1077
Dhrystone `-O2`	9.81	3.61	0.368	2	773
MCF `-O0`	49.67	37.18	0.749	11	8419
MCF `-O2`	50.11	25.04	0.500	33	4990

Table 4.1: *Compile-time performance of MoNaCo and LLVM JIT compilation.*

MoNaCo vs. LLVM MoNaCo generally outperforms LLVM in JIT compilation, as can be seen in Table 4.1. MoNaCo takes approximately 37% to 75% of the time that LLVM takes to compile the benchmarks. Note that due to the nature of the benchmarking process, the input module to both MoNaCo and LLVM is already a homogeneous module in the `llvm` dialect. For LLVM, this saves the time of converting a possibly heterogeneous module to an `llvm` dialect module first, a step which MoNaCo does not require. This means the results are skewed in LLVM's favor.

Effect of input optimization We can see that MoNaCo benefits more from smaller input code, as LLVM takes almost the same time on the unoptimized and optimized modules in both cases. Notably, in the case of MCF, LLVM's FastISel instruction selector failed and fell back to the slower *SelectionDAG* selector three times more often in the `-O2` module; 18 of the 22 additional fallbacks were triggered by calls to intrinsics, the other 4 by uses of the LLVM-IR `select` statement.

Between the unoptimized and optimized modules, MoNaCo's performance improves nearly linearly with the reduction in code size. When compiling the `-O2` Dhrystone module, MoNaCo's compile-time is reduced by about 23.7% of the `-O0` time, while the code size is reduced by about 28.2% of the `-O0` module. In the case of the `-O2` MCF module, MoNaCo's compile-time is reduced by about 32.7% of the `-O0` time, with code size reduced by about 40.7% of the `-O0` module.

These discrepancies between the decreases in MoNaCo's compile-time and lines of code are unsurprising, as the number of lines is an imperfect measurement for code size, and the `-O2` input is of higher complexity, as mentioned before.

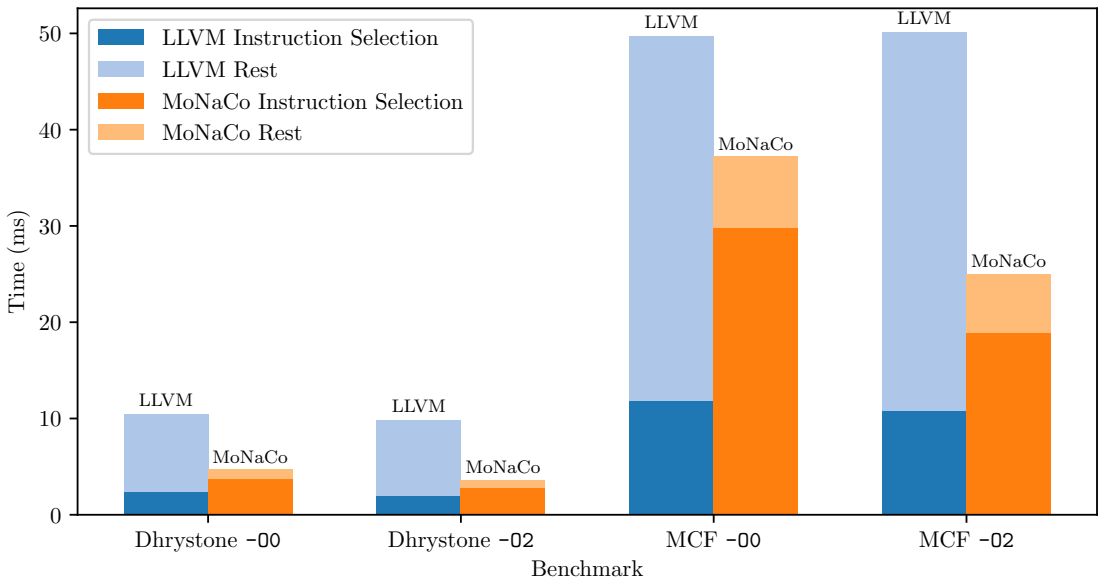


Figure 4.2: *Compile-time performance breakdown*

MoNaCo performance breakdown MoNaCo consistently spends 75% to 80% of its compile-time in instruction selection, while LLVM only takes around 20.5% to 24% for this (see Figure 4.2). LLVM thus outperforms MoNaCo in instruction selection both in relative and absolute terms without exception, especially on larger files; MoNaCo's overall performance lead stems solely from gains over LLVM in the rest of code generation. This poor instruction selection performance by MoNaCo is surprising, given that nearly all of its instruction selection patterns are trivially simple, and the other pass over the IR handles register allocation, SSA destruction, and encoding in one step.

For all code generation stages after instruction selection, MoNaCo only takes between 10.3% and 19.5% of the compile-time that LLVM needs for these stages.

4.2.2 Analysis and Discussion

In the following, any performance numbers refer to MoNaCo's compile-time performance on the MCF `-O2` module as it is the most complex module and covers the most cases. The performance measurements reported here were obtained using perf.

4.2.2.1 Optimization level

As seen in Subsection 4.2.1, MoNaCo's compile-times improve nearly linearly with the code size reduction in an optimized input module, compared to an unoptimized one. LLVM's compile-time performance remains almost unaffected by the optimization level of the input module. This is explained by a number of factors: first, LLVM's code generator is more complex, so for these semantically similar `-O0` and `-O2` modules, the kinds of ops that are employed affect LLVM's compile-time performance more than MoNaCo's, which scales more directly with the raw number of ops. Second, more complex IR structures present in the `-O2` modules also require more complex handling in the code generator: allocas for example are almost entirely eliminated by optimization of the input module, MCF contains 331 more alloca ops in the `-O0` module. This results in a more complex register allocation for the `-O2` module, as allocas do not already set the stack slots for variables in the IR anymore and this task now falls to the register allocator. This higher input complexity is less of an issue for MoNaCo, as it favors simplicity throughout the entire compilation process, although this comes at the expense of code quality. Third, the increased number of fallbacks to SelectionDAG negated some of the performance benefit of a smaller input: the FastISel selector took 2.5 ms less on `-O2`, but the 22 additional fallbacks resulted in SelectionDAG taking 2 ms more on the `-O2` module compared to `-O0`.

Note that it is unsurprising that `-O2` can produce a less FastISel-friendly module, as the combination of optimized IR compilation, but unoptimized machine code generation is uncommon.

4.2.2.2 MLIR design drawbacks

MLIR does not allow storing direct references to a function within another operation, instead it encourages referring to the function using its symbol name, causing some unnecessary duplicate symbol look-ups in MoNaCo. This is done to simplify multithreading [23],

although we want to note that multithreaded dialect conversion is unsupported at the time of writing.

Although many parts of the MLIR ecosystem are extensible, it is neither possible to store custom information on values, nor on blocks. Extensibility instead centers around ops, but this is not fine-grained enough for MoNaCo's usage of MLIR. Information that could be stored on values, individual op results and block arguments in particular, has to be stored in hash maps outside the framework instead. For example, MoNaCo's register allocation stores information about which stack slot or storage register a value resides in, in an instance of `mlir::DenseMap`. 38% of the total time spent in register allocation, SSA destruction, and encoding is spent solely on accessing this map, which would be avoidable if the information could be saved directly on values. Another 6% of total register allocation time is spent on looking up registers of block arguments in a separate `mlir::DenseMap`.

4.2.2.3 MLIR implementation issues

MLIR's interfaces enable generic access to many parts of the IR, but due to their implementation, accessing an ops representation as an `amd64::InstructionOpInterface` costs another 16% of the total register allocation time. The rest of register allocation time is spent making spill decisions, destructing SSA, and encoding instructions, and makes up 40% of the total register allocation time.

MLIR's dialect conversion infrastructure tries to fold operations into one another with the fold mechanism, which cannot be turned off. This feature can be helpful, but MoNaCo does not define any operation folding rules yet. 12.7% of the time spent in instruction selection, is spent solely on trying to fold unfoldable operations. Adding a `bool`-type parameter to the MLIR functions initiating dialect conversion could remedy this issue.

MLIR also takes care to always be able to revert any changes made during dialect conversion. This is a vital feature, but a dialect conversion driver which does not take these precautions could improve performance in the case of MoNaCo, which does not need error handling that recovers the original module. This driver could prevent additional passes over the rewritten operations, and save time creating, inserting, and later replacing `mlir::UnrealizedConversionCastOp`s, which could instead be realized immediately. In addition, 3% of instruction selection time is spent just requerying values that already had their rewrite determined, but not applied yet.

Overall, only about 68% of instruction selection time is actually spent matching patterns and determining their rewrites.

MLIR realizes all its patterns through virtual functions enabling dynamic polymorphism. This is the most flexible approach, but prevents inlining of pattern matching functions, meaning that every pattern that fails to match, still requires at least one virtual function call. MoNaCo employs one pattern per possible bit width of an operation, as this is the most idiomatic way to use the conversion framework. The four possible bit widths for a typical integer operation thus require up to four virtual function calls by the conversion framework to match a pattern. Through experiments with the order in which patterns are matched, we can quantify part of this overhead. For MCF `-O2`, we achieved a reduction of 9.1% in instruction selection time when matching in descending order of bit width, compared to the default ascending order. Note that this improvement still includes the

virtual function call overhead from the matching pattern. If these calls were inlinable, it would additionally be possible to eliminate the common sub-expression that computes the bit width for each result. MLIR could employ the C++ *curiously recurring template pattern* (CRTP)[2] to replace dynamic polymorphism with static polymorphism, allowing inlining, and eliminating virtual function call overhead.

4.2.2.4 Discussion

Overall, MoNaCo consistently outperforms LLVM in JIT compile-time, but is being held back by several issues pertaining to MLIR, most notably the performance of the dialect conversion framework, which MoNaCo's instruction selection is based on.

After instruction selection, MoNaCo excels at compile-time efficient code generation and beats LLVM by a large margin in all benchmarks. This is expected, as LLVM performs more steps like liveness analysis to improve code quality.

4.3 Run-time Performance

4.3.1 Results

	LLVM (s)	**MoNaCo (s)**	$\frac{\textbf{MoNaCo}}{\textbf{LLVM}}$
Dhrystone `-O0`	105.6	445	4.21
Dhrystone `-O2`	62.8	239.6	3.82
MCF `-O0`	23.76	108.34	4.57
MCF `-O2`	24.44	79.76	3.26

Table 4.3: *Run-time performance of MoNaCo and LLVM JIT compilation.*

MoNaCo consistently takes about 3.2 to 4.6 times as long as LLVM in the run-time benchmarks (see Table 4.3), and MoNaCo generally tends to profit more from an optimized input module. Both of these facts are to be expected: first, MoNaCo is not optimized towards run-time performance yet. Second, LLVM's FastISel still performs some code quality transformations that are beneficial to compile-time, like constant folding, while MoNaCo performs almost none. FastISel has less opportunities to try these on a more optimized input module, while MoNaCo directly profits from this, especially as less input ops also mean fewer stack spills have to be performed.

LLVM unexpectedly performs worse on the `-O2` MCF module, than on the `-O0` version. As mentioned in Section 4.2, almost all allocas were eliminated by the optimizer in MCF `-O2`, making register allocation more difficult and resulting in less efficient stack usage and ultimately worse performance. We experimented with a higher code generation optimization level equivalent to clang `-O2`, which resulted in a run-time of 15s on the MCF `-O2` module. This outcome confirms that the optimized input module alone was not well suited to LLVM's unoptimized code generation path.

[2] See `https://en.cppreference.com/w/cpp/language/crtp` for a detailed explanation.

4.3.2 Discussion

The factors that make MoNaCo perform worse in terms of run-time are its fully stack-based register allocation that does not utilize all available registers, as well as the unrefined instruction selection patterns. Despite that, MoNaCo's compile-time focused code generation still attains run-time performance within an order of magnitude of LLVM's most compile-time-oriented code generation configuration, which employs more sophisticated instruction selection, and register allocation that uses liveness information.

MoNaCo is thus a good fit for JIT compilation applications in which compile-time plays a larger part in end-to-end execution lag than run-time.

As explained in Chapter 3, both MoNaCo's instruction selection patterns, and its register allocation are designed to be extensible, to be able to provide code quality improvements and thus better run-time performance in the future.

5 Related Work

Machine code instructions as MLIR operations The `arm_sve` [45], `x86vector` [46], `arm_sme` [47], and `amx` [48] in-tree dialects are similar to MoNaCo in modeling machine code as MLIR dialects. They model instructions from ISA extensions for vector or matrix operations, as MLIR ops. In the existing code generation path, the operations of these dialects are translated to target-specific LLVM intrinsics. Like MoNaCo, they are also designed to be lowered to from higher-level dialects, such as the more generic MLIR `vector` dialect [49].

There are also in-tree dialects targeting code generation for GPUs at various levels of abstraction [50, 51, 52, 53, 54]; the NVVM dialect [53] and ROCm device library dialect [54] in particular model target-specific GPU operations; these get translated to LLVM intrinsics in the existing code generation path [22]. However, as MoNaCo is entirely focused on traditional CPU code generation, we will not study these dialects further.

LLVM intrinsics themselves also partly represent ISA instructions. However, the subset of intrinsics defined in the `llvm` dialect does not include these target-specific intrinsics; the task of representing these falls to other dialects such as those mentioned above. Thus, the `llvm` dialect is of a higher-level than MoNaCo's target-specific dialects.

To the best of our knowledge, there is only one other MLIR dialect that represents ISA instructions like MoNaCo — fully qualified with regard to their opcode. This approach was developed out-of-tree by Madhusudan et al. [55], but few details about it are publicly available. The approach focuses on porting LLVM's machine specific MIR representation to MLIR by introducing both a generic MIR dialect as a wrapper around LLVM's MC layer [27], as well as MIR-based target-specific dialects, such as an x86-64 dialect. Thus the lowering is divided into more steps compared to MoNaCo: first, the input is lowered to the `llvm` dialect, then to the generic MIR dialect, followed by a final MLIR dialect lowering to a target-specific dialect; the module can then be printed as assembly in another pass, to be assembled and linked by external tools. They plan to run various code generation passes that are also found in LLVM's back-end as further intermediate steps, to perform, e.g., register allocation; it is not clear whether a working register allocator is implemented yet. This architecture makes it structurally less efficient than MoNaCo's two-pass approach. Moreover, the focus of the project seems to be on porting the LLVM code generator to MLIR, rather than enabling custom user-defined lowerings from MLIR to machine code; there is not enough information available to assess whether this is supported by their approach. The project seems to have stalled since its presentation in 2021.

xDSL xDSL [56] is a Python-based sister framework to MLIR that also allows users to define custom IRs, and implements a dialect representing the instructions of the RISC-V ISA. Just like MoNaCo, register allocation is performed on this dialect, but in its own pass. Instead of directly emitting machine code, the xDSL RISC-V dialect emits textual

RISC-V assembly in another pass. This not only requires a subsequent pass over the code by an assembler, but the assembler needs to perform more work than just translating text to machine code one-to-one, as it needs to resolve jumps, process directives and resolve instruction aliases employed by the xDSL RISC-V dialect. The register allocator that is currently implemented for xDSL's RISC-V dialect is more sophisticated than the one implemented for MoNaCo, as it utilizes the available registers. xDSL dialects are specified in IRDL [57], an IR definition language similar to TableGen, but designed specifically for defining intermediate representations for SSA-based compilers.

Thus the biggest differences between xDSL's RISC-V dialect and MoNaCo are: xDSL is implemented in Python, and it emits textual assembly code instead of machine code, this means fully compiling xDSL RISC-V code after it has been emitted takes at least three passes: register allocation, assembly writing, and assembling. If JIT-compiling, another pass is typically required to link the assembly into an executable form. MoNaCo is implemented in C++, and emits machine code directly after instruction selection, in a single second pass over the IR. Due to its architecture, xDSL can thus not achieve the same level of performance. But as it is Python-based, it provides an easier entry to compilation for extensible intermediate representations than MoNaCo does.

LLVM MIR Finally, LLVM's own MIR format is also an IR designed for code generation and initially instruction-selected in SSA form. Fundamentally, MIR tries to be as machine agnostic as possible, which makes sense given LLVM's vast array of supported target architectures. MoNaCo's only target-specific dialect and register allocator on the other hand are specialized to operate on the x86-64 architecture only. This difference can be seen in the representation of register constraints and instruction special cases in MIR compared to MoNaCo. MoNaCo tries to minimize hassle for the user and its register allocation can handle register constraints specified on the instructions themselves; notices of special cases can be attached to a MoNaCo instruction op via MLIR traits. The combination of these two properties results in MoNaCo's transparent handling and smooth representation of a highly constrained instruction such as x86-64 `IDIV`:

```
// example constants for X, Y
%0 = amd64.MOV32ri <{immediate = 1337}> : !amd64.gpr32
%1 = amd64.MOV32ri <{immediate = 42}> : !amd64.gpr32
%quotient, %remainder = amd64.IDIV32r(%0, %1) : !amd64.gpr32, !amd64.gpr32
```

In comparison, MIR might instruction-select an `IDIV` instruction like this (adapted from [27], implicit register operands of `CDQ`, `IDIV32r` omitted for brevity):

```
;; example constants for X, Y
%reg0 = MOV32ri 1337
%reg1 = MOV32ri 42
;; start of 'idiv X by Y'
$eax = COPY %reg0            ;; Copy X (in reg0) into eax
CDQ                          ;; Sign extend X into edx
IDIV32r %reg1                ;; Divide by Y (in reg1)
%reg2 = COPY $eax            ;; Read the quotient out of eax
%reg3 = COPY $edx            ;; Read the remainder out of edx
```

This disparity in ease of use is a natural result of the design goals of LLVM and MoNaCo: with a fixed instruction set, any user who does not want to implement an LLVM back-end typically never needs to interact with MIR directly, so it makes sense to make usability sacrifices in order to make the functionality as generic as possible. A MoNaCo user however is expected to exploit the potential provided by code generation that is aware of high-level semantics, and thus needs to be able to lower their code without worrying about register constraints or the special behavior of the `IDIV` instruction explained in Section 3.2. This has the obvious downside of requiring more work per target-specific MoNaCo dialect.

Another difference to MoNaCo is that MIR's SSA is destructed just before the register allocation stage, in its own pass. As discussed in Chapter 4, LLVM's code generator is also more advanced than MoNaCo's in terms of code quality, as it implements more complex instruction selection patterns, instruction selectors, and register allocators, an instruction scheduling phase, and various peephole optimizations.

However, like MoNaCo, LLVM's back-end infrastructure also allows translation to machine code without the use of textual assembly.

6 Conclusion and Future Work

JIT compilation is crucial to improving the performance of programs whose exact semantics are not known until execution time. To simplify the development of JIT compilers, especially in domain-specific applications, compiler frameworks are often used, and can benefit from being able to precisely model the semantics of the target domain with custom operations and types.

In this thesis, we described problems with the current LLVM-based code generation path for the extensible compiler framework MLIR. In particular, we detailled how the limited expressiveness of the LLVM instruction set constrains the potential of modeling domain-specific information in an IR. To remedy these issues, we proposed MoNaCo, a JIT-focused native code generator for MLIR, that allows its users to directly bridge the semantic gap between their custom MLIR operations, and machine code. It does this while maintaining the convenience in automatic handling of register allocation, SSA destruction, and encoding, that the existing code generation path provides. We evaluated MoNaCo's compile-time and run-time performance in comparison to the LLVM code generation for MLIR, and showed that it is possible to efficiently (JIT) compile an extensible IR to machine code. MoNaCo consistently outperformed LLVM's code generation process for MLIR, reducing compile-time by between 25% and 63%. LLVM's more complex code generator outperformed MoNaCo in terms of run-time of the generated code, by a factor between 3.2 and 4.6.

We thus conclude that MoNaCo has the potential to be a viable alternative to LLVM's code generation process for MLIR, especially in those JIT applications that prioritize low compile-times over run-time performance. But our performance analysis also suggests, that despite its advantages, relying on MLIR and its dialect conversion infrastructure for instruction selection is not in the interest of compile-time performance. Implementing MoNaCo from the ground up with a custom extensible IR could have been a better choice in this regard.

6.1 Future Work

MoNaCo is now a working prototype, but there are still many ways in which it could be improved. We will discuss some of these in the following.

6.1.1 Features

To be able to translate more code, support for more features is desirable. We described some missing features in Section 3.5, and will now detail the most pressing ones.

Support for more platforms First of all, MoNaCo currently only supports the x86-64 platform. To be able to translate code for other platforms, more target-specific dialects would have to be defined, new encoding special cases handled, and the register allocator would require generalization to arbitrary general purpose and floating-point register banks.

Support for more types Struct-type arguments and returns, as well as vector types and instructions are not supported yet. Because of the complex handling of these types in the ABI, as well as their diverse usage patterns in the IR, supporting these is not trivial.

Translation of additional in-tree dialects Currently, only `arith`, `cf`, `func`, and `llvm` are supported, and only to a limited extent. Support for the `memref` [58] dialect would be beneficial, to avoid conversion of `memref` to `llvm` first. Support for the `vector` dialect, as well as the various platform-specific vector and matrix dialects mentioned in Chapter 5 would go hand in hand with the support for vector types and more platforms.

Debug information and ELF file handling Finally, more peripheral features like debug information, and ELF file handling could be the subject of future work. Writing ELF relocatable object files would not require much work, as all the necessary information is already present. However emitting debug information is a complex task that is highly platform-specific, and would require support from the MLIR source dialects, which is currently only implemented in the `llvm` dialect. The implementation could at least already make use of the source location information attached to each MLIR op.

6.1.2 Compile-time Performance

To improve compile-time performance, especially in instruction selection, which is the most time-consuming part of MoNaCo's compilation process, we aim to explore parallelizing the instruction selection process in the future. MLIR is designed with multithreaded compilation in mind, but its dialect conversion infrastructure does not support parallel conversion directly. In addition, MoNaCo's custom data structure used to store globals would need synchronization. Solving these challenges would allow compiling different functions in parallel and could improve compile-time performance on large modules, although the overhead of the additional synchronization necessary on the globals and the IR module itself might outweigh the gains.

6.1.3 Run-time Performance

A more sophisticated register allocator would achieve the highest improvement in code quality. Through combining MLIR's built-in support for liveness analysis and MoNaCo's extensible architecture for register allocation, this could be fairly straightforward to implement. However, in preliminary testing, the built-in liveness analysis has proven adverse to the goal of good compile-times, as it took more time than the entire second phase of code generation currently does.

In addition, more sophisticated instruction selection patterns would need to be implemented to improve run-time performance. This can be done entirely within the existing framework, and does not require additional design work.

Bibliography

[1] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, June 2011. DOI: 10.14778/2002938.2002940.

[2] XLA: optimizing compiler for machine learning. URL: https://www.tensorflow.org/xla (visited on Aug. 21, 2023).

[3] Google. V8 JavaScript engine. URL: https://v8.dev/ (visited on Aug. 27, 2023).

[4] M. Paleczny, C. Vick, and C. Click. The Java HotSpot server compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM '01, page 1. USENIX Association, 2001.

[5] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. Julia: a fast dynamic language for technical computing. *ArXiv*, abs/1209.5145, 2012. URL: https://api.semanticscholar.org/CorpusID:7044532.

[6] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. *SIGPLAN Notices*, 46(8):35–46, Feb. 2011. DOI: 10.1145/2038037.1941561.

[7] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. MLIR: scaling compiler infrastructure for domain specific computation. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '21, pages 2–14. IEEE Press, 2021. DOI: 10.1109/CGO51591.2021.9370308.

[8] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004.* CGO '04, pages 75–86, 2004. DOI: 10.1109/CGO.2004.1281665.

[9] R. P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Commun. ACM*, 27(10):1013–1030, Oct. 1984. DOI: 10.1145/358274.358283.

[10] Standard Performance Evaluation Corporation. SPEC CPU 2017. URL: https://www.spec.org/cpu2017/ (visited on Aug. 20, 2023).

[11] A. S. Tanenbaum, H. van Staveren, E. G. Keizer, and J. W. Stevenson. A practical tool kit for making portable compilers. *Communications of the ACM*, 26(9):654–660, Sept. 1983. DOI: 10.1145/358172.358182.

[12] J. Aycock. A Brief History of Just-in-Time. *ACM Computing Surveys*, 35(2):97–113, June 2003. DOI: 10.1145/857076.857077.

[13] 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, pages 12–27. Association for Computing Machinery, 1988. DOI: 10.1145/73560.73562.

[14] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991. DOI: 10.1145/115372.115320.

[15] GCC, the GNU compiler collection. URL: https://gcc.gnu.org/ (visited on Aug. 29, 2023).

[16] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software: Practice and Experience*, 28(8):859–881, July 1998.

[17] F. Rastello. *SSA-Based Compiler Design*. Springer Publishing Company, Incorporated, 1st edition, 2016.

[18] G. H. Blindell. *Instruction Selection: Principles, Methods, and Applications*. Springer Publishing Company, Incorporated, 1st edition, 2016.

[19] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2007.

[20] AMD. LLVM-based pipeline compiler (LLPC). URL: https://github.com/GPUOpen-Drivers/llpc (visited on Aug. 29, 2023).

[21] MLIR arith in-tree dialect. URL: https://mlir.llvm.org/docs/Dialects/ArithOps/ (visited on Aug. 21, 2023).

[22] MLIR llvm in-tree dialect. URL: https://mlir.llvm.org/docs/Dialects/LLVM/ (visited on Aug. 21, 2023).

[23] MLIR func in-tree dialect. URL: https://mlir.llvm.org/docs/Dialects/Func/ (visited on Aug. 21, 2023).

[24] MLIR code documentation. URL: https://mlir.llvm.org/docs/ (visited on Aug. 13, 2023).

[25] MLIR language reference. URL: https://mlir.llvm.org/docs/LangRef/ (visited on Aug. 13, 2023).

[26] Deep dive on MLIR internals, operation&attribute, towards properties. Feb. 9, 2023. URL: https://mlir.llvm.org/OpenMeetings/2023-02-09-Properties.pdf (visited on Aug. 13, 2023).

[27] The LLVM target-independent code generator. URL: https://www.llvm.org/docs/CodeGenerator.html (visited on Aug. 13, 2023).

[28] Machine IR (MIR) format reference manual. URL: https://www.llvm.org/docs/MIRLangRef.html (visited on Aug. 31, 2023).

[29] Gc: the Go programming language's default compiler. URL: https://github.com/golang/go (visited on Aug. 31, 2023).

[30] M. Brukman. LLVM commit `a7ec8e1a3e` introducing the first TableGen backend in the code generator. May 24, 2003. URL: https://github.com/llvm/llvm-project/commit/a7ec8e1a3e994a8136ed0f8500b578eec3098e3a (visited on Sept. 12, 2023).

[31] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. English. Intel Corporation. Aug. 2023. URL: http://www.intel.com/sdm (visited on Aug. 31, 2023).

[32] A. Sale. The implementation of case statements in pascal. *Software: Practice and Experience*, 11(9):929–942, 1981. DOI: https://doi.org/10.1002/spe.4380110905.

[33] R. L. Bernstein. Producing good code for the case statement. *Software: Practice and Experience*, 15(10):1021–1024, 1985. DOI: https://doi.org/10.1002/spe.4380151009.

[34] LLVM language reference manual. URL: https://llvm.org/docs/LangRef.html (visited on Aug. 27, 2023).

[35] The often misunderstood gep instruction. URL: https://www.llvm.org/docs/GetElementPtr.html (visited on Sept. 1, 2023).

[36] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. English. Intel Corporation. June 2023. URL: http://www.intel.com/sdm (visited on Aug. 12, 2023).

[37] System V application binary interface. AMD64 architecture processor supplement. URL: https://gitlab.com/x86-psABIs/x86-64-ABI/-/jobs/artifacts/357de358ba68eb779822dfcbb45f7ee2d9d09193/raw/x86-64-ABI/abi.pdf?job=build (visited on Aug. 15, 2023).

[38] K. Cooper and L. Torczon. *Engineering a Compiler: International Student Edition*. Morgan Kaufmann Publishers Inc., 2003. DOI: 10.5555/2737838.

[39] A. Fog. The microarchitecture of intel and amd cpus: an optimization guide for assembly programmers and compiler makers. May 26, 2023. URL: https://www.agner.org/optimize/microarchitecture.pdf (visited on Sept. 9, 2023).

[40] A. Abel and J. Reineke. Uops.info: characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *ASPLOS*, ASPLOS '19, pages 673–686. ACM, 2019. DOI: 10.1145/3297858.3304062.

[41] A. Abel and J. Reineke. uiCA: accurate throughput prediction of basic blocks on recent Intel microarchitectures. In L. Rauchwerger, K. Cameron, D. S. Nikolopoulos, and D. Pnevmatikatos, editors, *ICS '22: 2022 International Conference on Supercomputing, Virtual Event, USA, June 27-30, 2022*, ICS '22, pages 1–12. ACM, June 2022. URL: https://dl.acm.org/doi/pdf/10.1145/3524059.3532396.

[42] A. Fog. Optimizing subroutines in assembly language: an optimization guide for x86 platforms. June 22, 2023. URL: https://www.agner.org/optimize/optimizing_assembly.pdf (visited on Sept. 2, 2023).

[43] Dhrystone benchmark version 2.1, modified for strict ANSI C compilance. URL: https://github.com/nfinit/ansibench/tree/master/dhrystone (visited on Aug. 10, 2023).

[44] `perf`: linux profiling with performance counters. URL: https://perf.wiki.kernel.org/index.php/Main_Page (visited on Sept. 2, 2023).

[45] MLIR Arm scalable vector extension (SVE) in-tree dialect. URL: https://mlir.llvm.org/docs/Dialects/ArmSVE/ (visited on Aug. 21, 2023).

[46] MLIR x86Vector in-tree dialect. URL: https://mlir.llvm.org/docs/Dialects/X86Vector/ (visited on Aug. 21, 2023).

[47] MLIR Arm scalable matrix extension (SME) in-tree dialect. URL: https://mlir.llvm.org/docs/Dialects/ArmSME/ (visited on Aug. 21, 2023).

[48] MLIR Intel advanced matrix extensions (AMX) in-tree dialect. URL: https://mlir.llvm.org/docs/Dialects/AMX/ (visited on Aug. 21, 2023).

[49] MLIR vector in-tree dialect. URL: https://mlir.llvm.org/docs/Dialects/Vector/ (visited on Aug. 21, 2023).

[50] MLIR GPU in-tree dialect. URL: https://mlir.llvm.org/docs/Dialects/GPU/ (visited on Aug. 21, 2023).

[51] MLIR NVGPU in-tree dialect. URL: https://mlir.llvm.org/docs/Dialects/NVGPU/ (visited on Aug. 21, 2023).

[52] MLIR AMDGPU in-tree dialect. URL: https://mlir.llvm.org/docs/Dialects/AMDGPU/ (visited on Aug. 21, 2023).

[53] MLIR NVVM in-tree dialect for NVPTX. URL: https://mlir.llvm.org/docs/Dialects/NVVMDialect/ (visited on Aug. 21, 2023).

[54] MLIR ROCm device library in-tree dialect. URL: https://mlir.llvm.org/docs/Dialects/ROCDLDialect/ (visited on Aug. 21, 2023).

[55] V. Madhusudan, R. Kumar, and P. Nr. Moving LLVM's code generator to MLIR framework. Presentation at Fifth LLVM Performance Workshop at CGO. URL: https://llvm.org/devmtg/2021-02-28/slides/Vinay-MLIR-codegen.pdf (visited on Sept. 3, 2023).

[56] N. Brown, T. Grosser, M. Fehr, M. Steuwer, and P. Kelly. xDSL: a common compiler ecosystem for domain specific languages. Research Poster presented at SC22.

[57] M. Fehr, J. Niu, R. Riddle, M. Amini, Z. Su, and T. Grosser. IRDL: an IR definition language for SSA compilers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI '22, pages 199–212. Association for Computing Machinery, 2022. DOI: 10.1145/3519939.3523700.

[58] MLIR memref in-tree dialect. URL: https://mlir.llvm.org/docs/Dialects/MemRef/ (visited on Aug. 21, 2023).