Institutionen för systemteknik
Department of Electrical Engineering

Examensarbete

Performance evaluation of an easily retargeted C compiler using the LLVM framework

Examensarbete utfört i Datorteknik
vid Tekniska högskolan vid Linköpings universitet
av

Emil Nielsen

LiTH-ISY-EX--14/4781--SE

Linköping 2015

Linköpings universitet
TEKNISKA HÖGSKOLAN

Department of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings tekniska högskola
Linköpings universitet
581 83 Linköping

Performance evaluation of an easily retargeted C compiler using the LLVM framework

Examensarbete utfört i Datorteknik
vid Tekniska högskolan i Linköping
av

Emil Nielsen

LiTH-ISY-EX--14/4781--SE

Handledare: **Magnus Pettersson**
ISY, Linköpings universitet

Examinator: **Andreas Ehliar**
ISY, Linköpings universitet

Linköping, 12 June, 2015

Avdelning, Institution
Division, Department

Datorteknik
Department of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping, Sweden

Datum
Date

2015-06-12

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English
☐ _____

Rapporttyp
Report category

☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ _____

ISBN
—

ISRN
LiTH-ISY-EX--14/4781--SE

Serietitel och serienummer **ISSN**
Title of series, numbering —

URL för elektronisk version
http://www.da.isy.liu.se
http://www.ep.liu.se

Titel
Title

Prestandautvärdering av modifierbar C-kompilator i LLVM

Performance evaluation of an easily retargeted C compiler using the LLVM framework

Författare
Author

Emil Nielsen

Sammanfattning
Abstract

When considering processor architectures (either existing ones or when developing new ones), native code for functional testing and performance evaluation will generally be required. In theory, the work load involved in developing such code can be alleviated by compiling existing test cases written in a higher level language.

This thesis focuses on evaluating the feasibility of this approach by developing a basic C compiler using the LLVM framework and porting it to a number of architectures, finishing by comparing the performance of the compiled code with existing results obtained using the CoreMark benchmark. The resulting comparison can serve as a guideline when deciding which approach to choose when taking on a new architecture. The developed compiler and its back end ports can also serve as reference implementations.

While not conclusive, the final results indicate that the approach is highly feasible for certain applications on certain architectures.

Nyckelord
Keywords

compiler, llvm, back end

Abstract

When considering processor architectures (either existing ones or when developing new ones), native code for functional testing and performance evaluation will generally be required. In theory, the work load involved in developing such code can be alleviated by compiling existing test cases written in a higher level language.

This thesis focuses on evaluating the feasibility of this approach by developing a basic C compiler using the LLVM framework and porting it to a number of architectures, finishing by comparing the performance of the compiled code with existing results obtained using the CoreMark benchmark. The resulting comparison can serve as a guideline when deciding which approach to choose when taking on a new architecture. The developed compiler and its back end ports can also serve as reference implementations.

While not conclusive, the final results indicate that the approach is highly feasible for certain applications on certain architectures.

Acknowledgments

I'd like to thank the people who encouraged me to finish this after so long, in particular Andreas Ehliar who has gone beyond the call of duty to help me wrap this up.

Contents

Chapter 1

Introduction

This master's thesis covers the topic of writing a portable LLVM compiler, designed to be easily adapted to new architectures. While not optimized for any specific platforms, the generated code should be functional and contain some basic optimizations which will look roughly the same across all of the intended targets.

1.1 Motivation

The department of Electrical Engineering (ISY)[1] at Linköping University, in its research and to some extent undergraduate education, occasionally develops new processor architectures, as well as modifying existing ones. This thesis primarily covers the use case of *ASIPs*[1], which often do most of their actual work on highly specialized calculation CPU cores.

Tang et al. [2] show that a hand-coded assembler implementation of a benchmark program running on a Texas Instruments Digital Signal Processor is 8 times faster than a compiled one, despite using TI's own compiler. Since a basic, easily ported compiler is almost guaranteed to be even slower when compared to hand-coded assembler, the performance-critical code running on the calculation cores will need to be hand-coded. However, much of the code around it can be written in a high-level language, since the optimization and speed of this code is much less important. Writing code in the processor's assembler is time-consuming and more error-prone compared to writing or reusing existing code in a higher-level language such as C, so ideally, we will want to do the latter when possible.

This, however, requires a compiler capable of producing functioning machine code for the new target. Since it would be desirable to test the implementation as soon as possible after developing a prototype, the compiler should be easy and quick to port as soon as the specifications of the instruction set and other relevant features of the architecture are known [3].

Research in compiler technology for embedded processors is still highly relevant [4]. This thesis focuses on the second of the two major ambitions of current

[1] *Application-specific instruction-set processor*

compiler research[2], namely improving the flexibility of compilers to enable easier retargeting. This is to be accomplished by defining a generic compiler structure, porting it by specifying architecture-specific parts and finally evaluating the result using benchmarks, comparing it with results on other platforms where applicable.

1.2 Methods

With the main goal of the project being the production and evaluation of an easily portable compiler, it was important to approach the concept of generalization from an early point. At the same time, we had to make sure that the basis of the compiler actually worked, and that the parts that were to be rewritten for a new target were clearly identified and explained.

As a basis, thus, we needed a working compiler for a specific architecture. As we had access to partially finished back ends for the Senior architecture, this seemed a logical place to start. This working compiler was then to be generalized and ported to other platforms.

A more detailed discussion on the choice of compiler framework, along with descriptions of each, can be found in section 3. Comments and conclusions about the chosen approach are to be found in section 6.

When working ports had been developed, the next step was to test them. The goals of testing are twofold, as follows:

- That the ports generate correct code for a number of different test cases, which are meant to cover most of the ”corner cases” of the compiler.
- Evaluating the performance of the generated code, comparing the ports with each other and with publicly available results.

The primary tool for evaluating these metrics is a known benchmarking suite called CoreMark, described, along with a motivation for the choice of benchmark, in section 3.4, has been used. For some platforms, simulators have been used rather than real hardware.

1.3 Expected Results

While the primary focus of the project was to evaluate the feasibility of the portable compiler approach, the work done during the project brought with it some other useful results. Section 4, *Implementation* compares various approaches for implementing different parts of the LLVM backend and can serve as a guideline for future attempts at doing so. Appendix A summarizes some things to consider when developing a new architecture, in order to make it a good compiler target. Finally, appendix B contains a readme with information useful to a developer attempting to port the generalized compiler to a new architecture.

[2]The first one being finding more architecture aware code optimizations in the compiler to improve performance, particularly on specialized architectures such as DSPs.

1.3.1 Performance data

The expectations on the performance of the compiler are fairly low, especially for architectures that have unorthodox features that make them hard (or at least time-consuming) to support in the compiler. The thesis aims at providing an example of just how much worse one can expect the performance to be, compared to a more specifically targeted compiler, or, indeed, hand-written assembler code. Getting an estimation of the performance of non-critical parts will provide a guideline for which types of applications might be suitable for porting using this approach. For example, an application spending 90% of its execution time in 10% of the code might be suitable, even if the performance drop in the other 90% of the code is significant.

The numbers and comparisons will be followed by a more discussion-oriented section, which aims at determining the real-world feasibility of the approach.

Chapter 2

Overview of the Compiler Structure

The following chapter will describe in rough detail how a compiler works, and where the subject of this thesis fits into the process of producing machine code from a high-level source language. It largely follows the well-established structure from Aho et al.'s *Compilers: Principles, Techniques and Tools* [5].

A typical compiler is divided into two parts, the *front end*, or *analysis*, and the *back end*, or *synthesis*. The front end is responsible for breaking up the source code into its constituent pieces, analyzing the parts in several steps and, in the end, producing intermediate code used by the back end. The back end uses the information gathered by the front end (often in the form of a so-called *symbol table*) along with the intermediate code to put together, or synthesize, machine code. A flowchart depicting a simplified version of a typical compiler can be found in figure 2.1.

2.1 Front End

The front end of a compiler can be thought of as the platform-independent part. Stage by stage, it analyzes the original source code and transforms it into a format that contains increasingly more information. This intermediate format can then be used to generate efficient machine code in the back end.

Lexing and parsing stages analyze the source code, checking for syntax errors and controlling the semantics of the code. If errors are detected, it is the responsibility of the front end to generate useful messages to help the programmer detect the fault in the code. The result of these stages is typically contained in a so-called *syntax tree*, a form of intermediate representation.

Normally, one or more stages of intermediate code generation will follow. Different intermediate representations may be suitable for different types of optimization, and front end optimizations can make a great deal of difference. An example of this can be found in section 5.

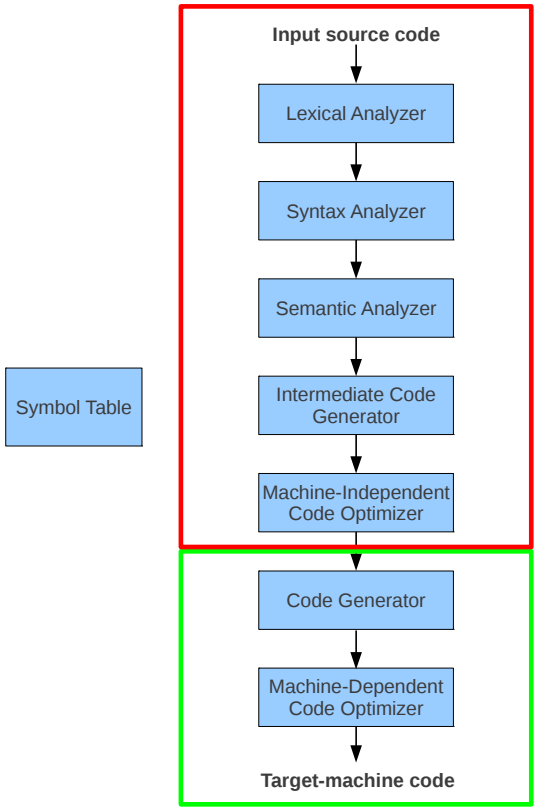

Figure 2.1. Simplified overview of a compiler. The red area denotes the front end, the green area denotes the back end.

Finally, the output of the front end is a form of intermediate representation suitable for conversion into machine code by the back end. A good example of such an intermediate representation is the *LLVM assembly language*, described in section 3.1.1.

Optimizations

In many of the stages of the front end, it is possible to perform a number of optimizations. These are typically intended to modify the intermediate representation output by the front end in a way that makes the final output of the back end faster or otherwise better. A short discussion on performance metrics can be found in section 4.2.2.

Most modern compiler front ends will provide the user with options to control which optimizations are made. Even though the optimizations are obviously intended to preserve the functionality of the program, it can be a good idea to

disable front end optimizations during the initial phases of back end porting, since the structure of the IR will be easier to follow and map to actual source code, which will alleviate debugging.

2.2 Back End

The second part of the compiler, and the focus of this thesis, is what is generally known as the *back end*. It is responsible for converting the IR generated by the earlier parts of the compiler into machine code that runs on the target architecture. This is not necessarily an easy task - the resulting code needs to be semantically correct and should preferably be at least somewhat optimized towards the target machine.

In general, there are three main general areas of interest when designing and implementing the back end. These are, in no particular order, instruction selection, register allocation and instruction ordering.

Instruction selection can be more or less simple depending on the design of the architecture. If the target machine has a consistent, mostly orthogonal[1] instruction set it may, in many cases, be as simple as mapping IR directives directly to machine instructions.

Register allocation concerns the problem of mapping symbolic registers in the IR to actual registers in the target machine. This can be made simpler if the IR is well designed and the register set of the target machine is reasonably complete and consistent. An example of a suitable IR for the purpose is the *LLVM assembly language*, which is SSA-based[2], making dependencies between registers easy to track. A good register set[3] will contain enough registers to enable the compiler to handle at least a reasonably sized subroutine without having to store temporary values in memory. Supporting most of the basic data types of the program is another convenient feature, although some type conversion with splitting of values into smaller registers or extending them into larger ones will very often be necessary.

Ordering the selected instructions can be a (relatively) simple matter if taking a naïve approach, but there is immense room for optimization. Instruction ordering is also crucial for semantic correctness of the program. Not only do the instructions have to be ordered to correspond with the intentions of the programmer - on many platforms, various hazards related to, among other things, memory access and delay slots have to be avoided. Some architectures (but far from all) will handle this in hardware, making the life of the assembler programmer or compiler developer a little easier.

[1] In an orthogonal instruction set, all instructions can utilize every register.

[2] Static Single Assignment [6] form, a property of IRs which implies that each variable is only assigned once. If the same variable is assigned several times, it is split into unique versions.

[3] An architecture's collection of registers

Chapter 3

Background

During the early phases of this project, two frameworks, LLVM [7][1] and GCC [8][2], were considered. Both are large, widely used frameworks, and we had access to the foundations of back end ports written in both. LLVM had a number of interesting properties which made it more suitable as a general compiler solution, most importantly the fact that its IR, the LLVM assembly language, is the one and only product of the front end and the only input needed to the back end [9]. In GCC, the relationship between front end and back end is somewhat more complex, although there have been attempts to separate the two in the past.

As mentioned in the introduction, the general approach in both cases was to finish existing, half-finished, back ends for the Senior architecture. The working back ends would then be generalized to include as few target-specific details as possible, giving us good platforms upon which the back ends for new architectures could be built with relative ease.

One thought originally was to compare the performance of the LLVM and the GCC back ends upon having two finished versions, deciding which one would be the better target for continued development. Due to time constraints, the scope was later limited to LLVM alone, but a GCC implementation with a comparison between the two would make an interesting follow-up project.

3.1 Overview of LLVM

First of the two considered frameworks was LLVM. LLVM is an SSA-based[3] compiler framework for arbitrary programming languages, and began as a research project at the University of Illinois [7] but has grown to include many more subprojects.

[1] Formerly an acronym of *Low-Level Virtual Machine*, but the project has grown to encompass so many things that the acronym is no longer valid, and LLVM is now officially considered not to be an acronym at all.

[2] The *GNU Compiler Collection, formerly known as the GNU C Compiler*

[3] Static Single Assignment

LLVM primarily provides the middle layer of a compiler, taking as input the intermediate form provided by a front end (such as Clang [10] or **llvm-gcc** [11]) and emitting an optimized intermediate form for the back end to use for target-specific code generation, as depicted in figure 3.1. The front end used in this project is Clang. LLVM also has its own infrastructure for assembly code generation, ie. the back end, which is the part we focus primarily on.

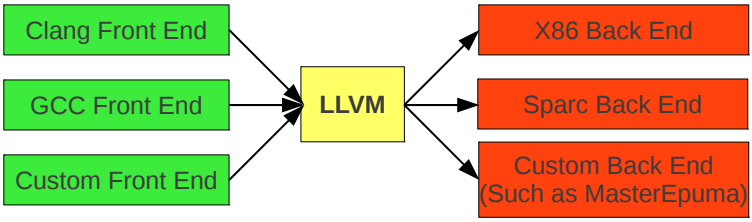

Figure 3.1. Simplified overview of the LLVM structure.

This "pure" three-phase structure is what primarily sets it apart from other compiler structures. In other frameworks, attempts have often been made to adopt a modular approach, but without achieving the full separation of front end and back end that LLVM accomplishes [9].

3.1.1 LLVM Assembly Language

The SSA-based LLVM assembly language is a form of IR, or intermediate representation, and is what is used by the back end to produce the machine code. It has three purposes [12], outlined below.

1. It serves as an in-memory baseline for the back end. This is our primary point of interest.

2. When saved to disk, it can be used by JIT compilation[4].

3. The structure of the IR can make it useful for debugging faulty programs, in some cases more than the source itself or the machine code.

A sample of the LLVM assembly language, together with a short explanation of the syntax, can be found in example 3.1 below. Understanding the details of the language is not needed in the scope of the report, but they are included anyway as an example.

[4] Just In Time compilation, compiling selected parts of a program during runtime for speed and efficiency

Example 3.1

A simple example of a C function that adds two numbers, followed by the main routine of the program which invokes the function with two parameters and returns the result.

```
1  int add(int a, int b)
2  {
3    return a+b;
4  }
5
6  int main ()
7  {
8    int i = 1;
9
10   i = add(i, i+1);
11
12   return i;
13 }
```

The following code is the result of compiling the C example above into the LLVM assembly language, using the Clang front-end.

```
1  ; ModuleID = 'example.c'
2  target datalayout = "E-p:32:32-i:32:32:32-f:32:32:32-n32"
3  target triple = "simplemips--"
4
5  define i32 @add(i32 %a, i32 %b) nounwind {
6  entry:
7    %a.addr = alloca i32, align 4
8    %b.addr = alloca i32, align 4
9    store i32 %a, i32* %a.addr, align 4
10   store i32 %b, i32* %b.addr, align 4
11   %tmp = load i32* %a.addr, align 4
12   %tmp1 = load i32* %b.addr, align 4
13   %add = add nsw i32 %tmp, %tmp1
14   ret i32 %add
15 }
16
17 define i32 @main() nounwind {
18 entry:
19   %retval = alloca i32, align 4
20   %i = alloca i32, align 4
21   store i32 0, i32* %retval
22   store i32 1, i32* %i, align 4
23   %tmp = load i32* %i, align 4
24   %tmp1 = load i32* %i, align 4
25   %add = add nsw i32 %tmp1, 1
26   %call = call i32 @add(i32 %tmp, i32 %add)
27   store i32 %call, i32* %i, align 4
28   %tmp2 = load i32* %i, align 4
29   ret i32 %tmp2
30 }
```

The file starts with a number of general target-specific definitions, such as data layout (line 2) and the type of target triples (line 3).

Lines 5-6 and 17-18 contain the declarations of the two function definitions made in lines 1 and 6 of the C code. `i32`denotes the type and size of the return value and the formal parameters of the function. The `nounwind` keyword indicates that the function doesn't end with an unwind or other "unusual" control flow modifier.

The `add` function definition starts with lines 7-10, where stack space is allocated for the two arguments and the values are stored into memory. In lines 11-14, the values are loaded into temporary variables, normally mapped to registers in the final code, before the addition is performed and the result is returned.

As for the main function, lines 19-22 declare and initialize the `retval` and `i` variables, and their values are stored in memory. The values are then loaded into intermediary registers (lines 23-24), a value of 1 is added to the second argument (line 26) and the call to the `add` function is finally made. At line 27, the return value of the function is stored in memory, a value which is immediately loaded again at line 28 and returned by the program at line 29.

3.2 LLVM Back End

The back end of LLVM is responsible for converting the intermediate form into machine code[5]. This section describes the various steps and passes involved in doing so.

3.2.1 TableGen

TableGen is a language used by the LLVM back end to simplify and organize the definitions of domain-specific information, replacing C++ with a simpler and more concise description of the target domain when applicable. It uses inheritance and what is known as *multiclasses* to group together common features of a specific property, such as, for example, variations of an `ADD` instruction which nonetheless expand to the same machine instruction. The .td source files are parsed and then processed by a TableGen back end, which in our case is the LLVM code generator.

Aside from multiclasses, TableGen also contains regular *classes* and *definitions*, which are both *records*. A record is basically a collection of values grouped together under a unique name. It also contains a list of superclasses. A class is an abstract record, and can be used to describe a kind of property, such as a group of machine instructions. For instance, ALU instructions in a specific processor might have common enough values and behavior to justify making a common class for them. Classes are denoted by the `class` keyword. A definition is more specific, and generally contains no unspecified values. A specific instruction is a good example of a definition. Definitions are marked by the `def` keyword.

[5]The back end can also be used to convert LLVM assembly code back into C, C++ or some other language.

TableGen is used for more than just instructions, however. Registers and register classes are commonly described using the language, as well as calling conventions and, to some extent, compiler subtargets.

Example 4.2 shows how TableGen can be used to define several instructions with shared attributes and structure.

3.2.2 Target Machine

The main idea behind the target-specific back end is to generate code that is fully optimized for the target architecture, while at the same time not being too bulky and complicated to design [9], especially not in our case, where we want the back end to be simple and portable. Some assumptions can generally be made regarding common features of the target architectures.

For instance, the register allocation phase may look very similar between different processors. The main differences will usually be how many registers are available, the width of these registers and which registers that go with which instructions and data types. With proper generalization of code, most of this can be described in a fairly concise manner, without having to re-write all the code to fit the specific target. In LLVM, the register set is described using the TableGen language, described in section 3.2.1. Also, see section 4.1.2 for more information on the register set implementation.

These two concepts, optimizing heavily for the target architecture and finding ways to describe common features in generic terms, will have to be balanced well to produce good machine code for new architectures.

LLVM tries to achieve this balance in several ways. The target description files, written in the TableGen language, provide a simple and powerful way of describing target-specific features. The control flow of the back end itself is divided into *passes*, each of which takes as input the IR produced by the previous pass, processes it in some way (such as applying an optimization algorithm) and then outputs the modified IR. A summary of the changes and additions that were made to each of the two target architectures can be found in section 4.

3.3 GCC

As mentioned in the introduction to this chapter, it was decided to focus on LLVM, but a good follow-up project would be to implement a similar generalized, easily-ported, back end in GCC. GCC is a compiler system developed by the GNU Project [8]. It provides a complete toolchain for converting source code into machine code, with mostly separate front end and back end, without quite attaining the full modularity of LLVM. The back end uses RTL [13] descriptions of the target machine to generate appropriate assembly instructions from the source.

Its wide-spread use, good documentation and continued support makes it a good alternative to LLVM, but LLVM's "pure" 3-phase structure and modular approach to compiler tools made it the primary alternative.

3.4 Benchmarking

We needed to somehow measure the performance and correctness of the code produced by our compiler. A mostly comprehensive benchmarking program is a good way of testing both, and, when the more basic test cases pass, gives a concrete goal to work towards while porting.

3.4.1 CoreMark

We chose CoreMark as our benchmarking solution for several reasons, outlined below:

1. It is designed to be easily retargeted, even for simple embedded platforms. Thus, it can very likely be used for future benchmarking of new platforms.
2. It is written in C, which is our source language of choice. Using another language would require us to use a different front-end and possibly to make modifications to our back end as well.
3. There is a large amount of benchmark scores available on the web to use as reference values in evaluation.

Correctness

The payload of CoreMark is divided into three major parts: List handling, a state machine processing string data, and a series of matrix operations, all on predetermined data. To verify correctness, a CRC[6] value, is calculated on the results of each individual part of the benchmark. These are then compared to known values.

Performance

CoreMark is divided into iterations, each of which perform the benchmarks outlined in section 3.4.1. The iterations are identical, and the performance of the processor is primarily measured in iterations per second. For the sake of measuring the performance of the compiled code, equation 3.1 was used to remove the dependency on CPU speed, leaving the number of cycles executed per iteration.

$$C = \frac{1}{N} \cdot S \tag{3.1}$$

```
C = Cycles per iteration
N = Measured iterations per second
S = CPU Speed in Hz
```

This metric is very easy to calculate provided the CoreMark score, ie. N, and the CPU speed, S, are known. Since this information is provided in CoreMark's own repository of benchmark scores [14] there was plenty of material to choose from during the comparison stage.

[6] Cyclic Redundancy Check

3.4.2 Optional Benchmarks

Our primary alternative to CoreMark was Dhrystone [15], also a widely-used, lightweight benchmarking solution for embedded systems and the like. There are a few differences between the two [16]. Firstly, CoreMark runs algorithms that can be found in many real-world applications, whereas the payload of Dhrystone is typically completely synthetic, making it harder to understand and less useful as a reference. Additionally, some of the main payload of the Dhrystone benchmark can more easily be optimized away by the compiler than that of CoreMark. This second point can actually make it a suitable benchmark for a compiler, but since the focus of this thesis was not on heavy optimization of the target code but rather on making the benchmark run and comparing it with what a typical compiler can achieve, it was decided that CoreMark was the better alternative.

Chapter 4

Implementation

The following chapter describes the process of implementing an LLVM back end and the subsequent target-specific issues encountered when porting said back end to the two architectures covered in this thesis. The descriptions are meant to serve as examples, and will be unlikely to cover all the issues one can expect when targeting a new architecture. Merely observing the differences between the Senior port, described in section 4.3 and the MIPS one, described in section 4.4 gives the reader a good example of the aforementioned differences.

In section 4.2 we describe the steps taken towards a more general compiler back end, more suitable for retargeting than the finished implementations.

4.1 Implementing an LLVM Back End

The process of implementing a new back end for LLVM can be divided into a number of parts, as described in `llvm.org`'s own guide to writing an LLVM back end [17]. Here, they will be looked at from the perspective of the general compiler and with comments related to the platforms approached in this thesis project.

The files containing target descriptions are implemented in TableGen (files ending in `.td`, see section 3.2.1) where applicable, although some parts still need to be implemented directly in C++.

Note that the version of LLVM used in this project is 2.8. Newer versions (the current one at the time of writing this thesis being 3.0) were not compatible with the base compiler for Senior that was used as a starting point for the project.

4.1.1 General Properties

General properties of the target machine are described in the `TargetMachine` and `SubTarget` classes. A subtarget can be used to describe minor variations, eg. introduced by small differences in the instruction set, of a given platform without having to make a completely separate target. In our compiler, subtarget support has been added (with a default subtarget), not because it has strictly been needed

for our current ports, but because it makes the compiler more easily extensible with more subtargets, should the need arise.

4.1.2 Register Set

The register set of the target is described in the `RegisterInfo` and, to some extent, `ISelLowering` files. The description covers target-specific properties of the register set such as the number of registers, register types, names and sizes. Some C++ descriptions are required to model behavior of register handling operations such as moves and stack interaction. Example 4.1 shows a typical register definition.

These descriptions are then used by the LLVM framework to perform tasks such as register allocation (which can be a complex, but largely target-independent, task).

Example 4.1

```
1  class SimpleMipsReg<string n> : Register<n> {
2    let Namespace = "SM";
3    field bits<6> Num;
4  }
5
6  class Ri<bits<6> num, string n> : SimpleMipsReg<n> {
7    let Num = num;
8  }
9
10 def V0 : Ri< 2, "$v0">, DwarfRegNum<[2]>;
11 def V1 : Ri< 3, "$v1">, DwarfRegNum<[3]>;
```

Here, we define a general register class `SimpleMipsReg` which acts as a superclass for all register types of the SimpleMips implementation. The `Ri` class denotes the integer registers of the machine, and inherits from the `SimpleMipsReg` general superclass. `V0` and `V1` are examples of specific integer registers, ie. instantiations of the register class[1].

The `CallingConv.td` file is used to model calling conventions, which details which registers can be used for passing arguments, which registers need saving on function return and so forth.

4.1.3 Instruction Set

The instruction set is mostly described via TableGen in the `InstrInfo.td` file. Here, we get to utilize some of the true strengths of the TableGen language, as described in section 3.2.1. Instructions are described using definitions, and groups of instructions that share the same properties can be described using multiclasses. A typical instruction definition includes opcode, a list of operands, an assembler representation and the DAG[2] pattern to match (for more information about the

[1] The V-registers are used for function return values and expression evaluations.
[2] Directed Acyclic Graph

matching, see section 4.1.4). An example of an instruction definition can be found in example 4.2.

Example 4.2

```
let isBranch = 1, isTerminator = 1 in {
  def BCCr : F1<(outs), (ins brtarget:$dst, cc:$cc, GenRegs:$src1,
                                                    GenRegs:$src2),
                    "b$cc␣$src1,␣$src2,␣$dst",
                    [(SMbrcc bb:$dst, imm:$cc,
                      GenRegs:$src1, GenRegs:$src2)]>;

  def BCCi : F1<(outs), (ins brtarget:$dst, cc:$cc,
                         GenRegs:$src1, i32imm:$src2),
                    "b$cc␣$src1,␣$src2,␣$dst",
                    [(SMbrcc bb:$dst, imm:$cc,
                      GenRegs:$src1, imm32:$src2)]>;
}
```

The code snippet above is the TableGen description of conditional branches in our MIPS implementation. The `let` statement surrounding the definition itself includes special properties of these types of instructions. The first part of the definition itself contains the name of the definition, what class it instantiates, and the operands of the instruction. The quoted part is the assembler representation of the instruction, with variables represented by $ signs. Finally, the square brackets enclose a DAG pattern to match for the instruction to be selected.

For operations that are not supported natively by the architecture, or that are too complex to model in the TableGen description, some custom handling will be needed. This is mostly done in the `ISelLowering.cpp` file.

4.1.4 Instruction Selection

In one of the earlier passes of the back end, the LLVM IR is converted into a DAG, with graph nodes containing operation information for the specific node. The instruction descriptions detailed in the previous section are used along with selection of addressing modes in the `ISelDAGToDAG.cpp` file to match subsets of this graph. Matched parts of the graph are either mapped directly into machine instructions or handled manually in the `ISelLowering.cpp` file. For instance, in the case of function calls, a number of things, such as stack allocated arguments, calculation of the function's total stack space, cannot (at least not yet in LLVM) be handled automatically or using TableGen attributes.

Type Handling

In `ISelLowering.cpp`, we specify to the compiler which data types that are supported natively by the target architecture. Operations handling other types than the ones supported will have to be changed somehow to conform to the requirements of the architecture. This can be done in a number of ways, the first one

being to break them down into smaller bits which are natively supported, using the `expand` keyword. They can also be extended into operations handling larger types with the `promote` keyword. If we need to manually define a specific behavior for an instruction, the `custom` keyword can be used, and a C++ routine to invoke will be specified. To explicitly tell LLVM that an operation is supported, we can use the `legal` keyword.

4.1.5 Code Generation

In the final phase of the LLVM back end, the optimized IR generated by the previous passes is converted into the target assembler, using the `AsmPrinter` module. It is possible to make last-minute patches to the target code in this section, but the ideal is to avoid it, as this pass should solely be about converting the IR into its corresponding target machine assembler representation.

4.2 Generalizing and Porting the Compiler

When a (nearly) working implementation on the Senior architecture had been finished, the next step was to generalize this implementation, giving us a good platform upon which to write a compiler for a new platform, with the general strategy of reusing as much as possible of the current implementation and not worry too much about performance hits suffered from not fully utilizing the architecture.

In practice, this meant creating a new target based on the existing Senior back end, stripping it of its target-specific features and providing some guidance as to how to implement them anew, as described in the following sections.

Namespaces

Target-specific names for functions, condition codes, calling conventions and the like needed to be stripped away, but we still wanted to make them easy to replace, so a “DU/Dummy” namespace was added, which would only need replaced by whatever the new target was to be named and abbreviated.

Instruction Set

The instruction set, while obviously having a highly target-specific assembler syntax, usually has a number of common features and instructions between architectures. Template TableGen definitions for many common instructions have been added. Also, the structure of these should be clear enough to allow adding new instructions easily.

Treatment of special instructions that cannot be handled by TableGen alone is trickier, since it will vary more between architectures, but some common cases are described along with proposed solutions.

Register Set

Much like the instruction set, the register set will vary between platforms, but the basic structure will likely remain the same. Template register definitions and comments where things have been omitted have been added to the generalized implementation. Providing guidance for defining special purpose registers like the stack pointer has also been prioritized.

Supported data types

An architecture will very seldom contain registers of every possible size and type that can be used in the source program. The appropriate way to handle conversion into supported data types varies between platforms, and it depends greatly on the instruction. Some suggestions for typical conversions are provided in the generalized compiler, but care will have to be taken to get things right in the specific case.

Function calls

Function calls bring with them a fair bit of complexity to the compiler. The solutions to some of the issues can often be reused, where some others are unique for certain architectures. Many processors will have dedicated registers for passing function arguments. These registers need to be specified in the target-specific part.

An example of a special target-specific consideration is described in section 4.4.1, where MIPS needs special handling of return addresses for nested function calls.

4.2.1 Readme

To make it easier knowing where to start porting, a readme was written specifying the main areas needing modification. A summary of this readme can be found in appendix B. Additionally, stubs and descriptive comments were added in areas of interest, in order to make it easier to know where to start porting and to make sure that nothing was missed.

4.2.2 Running CoreMark

To run the CoreMark benchmark (and get a meaningful result), a few additional requirements on the target architecture (or, more accurately, the testbench) have to be made. The two primary ones, a way to measure execution time and printing of results, will, in the general case, be presented in this section, while the specifics about the target-specific implementations for Senior and MIPS are presented in 4.3.2 and 4.4.4 respectively.

Timing

As CoreMark is meant to measure the performance of the architecture (and, in our case, the compiler) certain metrics have to be provided in one way or another.

The one which concerns us the most is the execution time, or rather, since we'll be running our tests almost exclusively in simulators, the cycle count. If relevant, we could calculate a theoretical execution time by using predicted values of the CPU clock, but for measuring compiler performance, this is not strictly necessary. Thus, some way of accessing a CPU clock or measuring the cycle count between CoreMark's calls to `start_time()` and `stop_time()` are required.

In the general case, it would obviously be preferrable if this could all be handled from the program itself, as this would enable one or several subsequent runs without interruption. With proper implementation of printing (a separate problem, addressed in the following section) this would also enable the program to print the cycle count, perform basic calculations on it and present it in some useful way to the user.

Printing

Merely measuring the time CoreMark takes to execute will not suffice to provide us with what can be considered a reliable reading. The benchmark contains a number of sub-tests, the results of which need to be verified with a CRC. If these checks fail, we'll need some way of conveying this to the person running the tests, meaning, if we want things to be reasonably user friendly, printing the results. Of course, we'll also want to let the user know if all tests were successful. This is normally accomplished either by some sort of standard output interface that conveys the specified information during runtime, or by file I/O.

4.2.3 Other considerations

Cycle count alone might not always suffice - other metrics may be necessary or at least relevant for some tests. For example, if the processor has a cache memory we don't want to restrict ourselves to cycle count as our only measure of the execution time, but would prefer a real time measurement (depending on how memory fetches are implemented with stalling and so forth).

Dynamic memory usage, stack depth for different algorithms and register utilization are other examples of metrics that could be useful for certain applications.

4.3 Senior

Since there was already an existing (albeit unfinished) implementation of an LLVM backend for the Senior architecture, the job here mainly consisted of completing the implementation and adapting it to the CoreMark benchmark, to make sure a reliable reading could be taken.

A number of target-specific issues arose when porting, the more interesting of which are outlined below.

4.3.1 Limitations of the Register Set

Senior only has 16-bit data registers and no other documented hardware support for most 8-bit or 32-bit operations. This meant that 8-bit values had to be extended into the 16-bit registers. For sign extending operations, the safest, simplest way of doing this was to shift the value 8 bits left and right with arithmetic shifts. 32-bit values were handled using two 16-bit registers and keeping track of carry bits when performing operations on the 32-bit value.

Furthermore, the Senior architecture has some severe limitations when it comes to the relationship between address registers and general-purpose data registers. Some addressing modes, most notably addressing with an offset, requires the usage of address registers, whereas regular arithmetic operations, such as additions, can only be performed on data registers.

Senior also lacks regularity in the sense that the stack pointer, `SP` (which is essentially an address register) does not support all the operations that a regular address register does. For instance, offset addressing with an offset of 0 can for any `AR` be written as either `(ARx,0)` or `(ARx)` whereas the shorter version is not supported for the SP, which further complicates trying to write a simple compiler.

4.3.2 CoreMark on Senior

Getting CoreMark to run was the primary objective of the port. Aside from requiring correctness of the compiler, ways of printing and measuring execution time are needed, as further discussed in section 4.2.2.

String Handling and printf in Senior

As the Senior assembler had no standardized way of handling string input and output directly to the simulator, and otherwise no standard implementation of printf (or similar), this had to be implemented manually, in a form that fit the Senior assembler.

Seeing how there was a python script in place for converting the default GAS code into the Senior assembler format, we opted to extend this script to convert the default representation of strings (normally using the `.ascii` or `.asciz` directives [18]) into a simplified array of words, each containing one character, represented by its respective ASCII code.

This made it easy to implement library functions for printing, building on CoreMark's default implementation of printf for architectures lacking inherent inherent library support, such as Senior. A few modifications had to be made to accomodate the fact that `vararg` was not implemented and to combat the over-optimization problem described in section 4.3.3, as well as to implement the most important part, the actual printing.

In Senior, through the simulator, no way of sending a string or a char directly to the screen was available. Instead, we had access to the `OUT $0111,rx` command, which sends the content of register `rx` (16 bits) to a file named `IOS0011` in hexadecimal format, appending it row by row at the end of the file.

This file was then converted into a readable format by another script, also written in python, producing a file containing the same text that would normally be presented in stdout when calling printf on a "standard" architecture.

Timing in Senior

Senior has an implementation of hardware timers, but these are too small (16 bits with a maximum prescaler of 1024) to be of much use when measuring things like cycle count, since the value will very quickly vastly exceed the maximum capacity of the timer.

A suitable workaround for this, which ended up being the one that was used, was to use the simulator's measured cycle time which is displayed when the program's execution is halted. This meant having to halt the execution when the timer was due to start, and again when it was due to stop, and manually take readings of the cycle count from the simulator. These readings could then be used to measure the performance of the compiler.

4.3.3 Over-optimization problem

One of the remaining bugs in the compiler causes it to optimize overly eagerly in some situations. In example 4.3, the LLVM optimizer incorrectly removes instructions that are necessary for correct program execution.

Example 4.3

Consider the following code:

C code

```
1 unsigned int y = 0x8000;
2 unsigned int x = 0x6666;
3 y >>= 8;
4 x = testfun(1,2,3,4,5,6,7,8);
```

LLVM IR

```
1   %shr = lshr i16 %tmp, 8
2   store i16 %shr, i16* %y, align 2
3   %call = call i16 @testfun(i16 1)
```

Generated assembler

```
1   nop
2   st1 (sp,-8), r0
3   set r0, 1
4   nop
5   call testfun
```

Notice the lack of a `lsr` instruction before the function call in the generated assembler code, despite the IR containing a `lshr` node (which corresponds to a logic right shift). The issue seemed to occur when a shift instruction was followed by a function call in the original C code.

The cause is still unknown, but a temporary work-around, which involved volatile variable declarations, was found. This work-around was employed temporarily in the CoreMark source code to get the benchmark running. The modified source code, and the result after compiling, is presented below:

C code

```
volatile unsigned int y = 0x8000;
unsigned int x = 0x6666;
y >>= 8;
x = testfun(1,2,3,4,5,6,7,8);
```

LLVM IR

```
%shr = lshr i16 %tmp, 8
volatile store i16 %shr, i16* %y, align 2
%call = call i16 @testfun(i16 1)
```

Generated assembler

```
nop
lsr r0, r0, 8
st1 (sp,-8), r0
set r0, 1
nop
call testfun
```

The problem seems to reside in the back end, as the LLVM IR code evidently still contains the appropriate nodes. Due to a shortage of time, it has not been fixed and will have to be added to the list of bugs and weaknesses with the current implementation.

4.4 MIPS

MIPS is a popular benchmark and research architecture. The assembler provides abstraction for things like string handling, making it significantly easier to get the benchmark running. The combination of popularity and simplicity made it a good target for our second port, which was meant to be relatively simple and quick while still being comparable to other similar implementations. Our implementation is (temporarily) called *SimpleMips*.

4.4.1 Function Calls

Function calls in MIPS are mostly straightforward. There are a number of registers dedicated to passing arguments, and stack offsets can be calculated in a fairly straightforward manner. The main issue that had not been considered at all previously in the compiler (simply because it worked differently on the Senior platform) was the fact that the return address was automatically saved in a register upon calling the `JAL`, or jump to subroutine, instruction. Since the same register was always used, nested subroutine calls would overwrite the previous return address. This was solved by saving the value of the return address register to the stack when needed, ie. when we knew that there would be nested calls.

4.4.2 Data types and variable sizes

MIPS mostly holds 32 bit values, but supports 8 bit loads and stores. As CoreMark handles 1, 8, 16 and 32 bit integer values, some care had to be taken to use the right size operator at the right time, in particular concerning 16 bit values, which had to be either handled with sign extensions and 32 bit operators or by splitting values and use 8 bit operators.

The framework already has built-in target-independent algorithms in place for taking care of type conversion. A certain amount of care is still required to make sure that no information is lost.

Offsets in structs and the stack are another thing that needs to be considered carefully when converting between types. In MIPS, offsets are measured in bytes, and they need to be aligned correctly, so a 32-bit word can only be stored and loaded with an offset divisible by 4 (unless the offset is zero), as that's the minimum correctly aligned offset. A byte, however, can be loaded or stored with any integer offset, but care needs to be taken to make sure that the correct part of the word is loaded.

4.4.3 Phi Nodes

In an SSA-based language like the LLVM assembly language, variables can only be assigned once (resulting in every assignment producing a uniquely named variable). An assignment where the value depends on the result of a previous conditional statement requires a special construct, as it is impossible to know until runtime which variable to use.

To handle these cases, a special function, denoted ϕ (or simply `phi` in program code), is used. Consider a situation where an `if` statement decided whether the variable `y` should have the value 1 or 0. After this is decided, we execute the `x = y` statement. Using SSA, there will be two versions of `y` depending on which branch of the `if` statement that was executed, for instance $\texttt{y}_1$ and $\texttt{y}_2$. As we cannot know at compile time which one of these will hold the value we want, we instead assign whatever is returned by $\phi(y_1, y_2)$ to `x`.

In the back end, an easy way to convert a `phi` node into machine code is to make use of conditional branches. This is how it was approached in our LLVM back end, implementing custom handling of the `SelectCC` DAG operation by a C++

function in the `SimpleMipsISelLowering` file. There are still some unresolved bugs, however, causing failures in compound conditional expressions, typically those grouped together with a logical connective such as `&&` or || [3].

Compiling CoreMark at low front end optimization levels (typically O0), we can mostly avoid having to work with complex `phi` nodes, enabling us to circumvent these bugs for verification runs.

4.4.4 CoreMark on MIPS

Running CoreMark on MIPS obviously bore with it the same requirements as in the general case - we needed to ensure correctness (which includes printing) and we needed a way to measure the cycle count of a run.

A few pieces of code had to be rewritten in the CoreMark source to achieve a correct CRC reading, mostly due to the over-optimization mentioned in section 4.3.3 and the `phi` node issues described in section 4.4.3.

This meant two major types of changes to the code: Firstly, that some variables needed to be declared as volatile, since some operations, particularly bitwise operations such as `shift`, `or`, and `and`, would otherwise be removed completely by the optimizer for certain operand values (see section 4.3.3 for a more detailed description of the problem, which occurs in both ports). Secondly, complex conditions in loop statements such as `while` and `for` needed to be broken up into `if` statements with corresponding `break` statements to avoid the currently broken behavior of `phi` nodes.

[3] Due to the nature of these connectives, they can be a bit more complex to implement than other operators. A correct implementation will involve adding conditional jumps between the evaluation of the two clauses, since, in the case of &&, if the first clause evaluates to false, the whole expression returns false, regardless of the value of the second clause. Similarly, in the case of ||, if the first clause evaluates to true, the whole expression returns true.

Chapter 5

Performance and Results

The following chapter presents the results of running the CoreMark benchmark on the two target architectures, compiled with our LLVM back end. The two main purposes, as described in section 3.4, of running the benchmark are to verify correctness and to provide relevant performance data. Section 5.1 outlines the scores chosen in the comparison with motivations. Figure 5.1 depicts the various scores obtained.

5.1 Choice of Compared Scores

Five different platforms were initially considered in the final comparison, of which four ended up being actually useful, as follows:

Senior

The original target of our compiler. At the time of this thesis, a number of errors were discovered in the original implementation, and the numbers cannot really be trusted in absolute terms. For this reason, the results of the Senior runs were not included in the final report[1].

SPIM, SimpleMips

Compiled with our compiler and run on the MIPS R2000 architecture emulated by SPIM [19]. As mentioned in section 4.4.4, these results are CRC verified but cannot be reported as official scores, since the source has been slightly modified to circumvent a number of shortcomings in the compiler, described in more detail in the 4.4.4. Coremark compiled with an optimization level above O0 did not execute correctly in SPIM, so no results could be obtained for these levels.

[1] The test results, however incorrect, still show a consistent increase in performance at higher optimization levels

SPIM, GCC 4.8.2

Another run was performed using SPIM, this time compiled using GCC 4.8.2 and converted to a format SPIM could execute. Interestingly, Coremark compiled using optimization levels above O0 could not be executed on this platform either, which might suggest an issue with the simulator. This is discussed more in depth in section 6.

WRT54GL Router

To put the SimpleMips numbers in context, a test run was done on real MIPS hardware, namely a WRT54GL router running a 200 MHz Broadcom BCM3302 CPU. The benchmark was compiled with GCC 4.8.2, and numbers using when using log levels O0, O1 and O2 are provided. The results are approximate, since the measured value of iterations per second was reported at a fairly low resolution.

Intel Core 2 Duo

Compiled from the CoreMark source using GCC 4.4.5. This implementation was useful as a reference when porting the compiler, since debug information could easily be extracted using printf statements and the like on a platform known to produce correct results. Performance data was gathered using varying levels of optimization in the front-end, and will serve as an example of a typical X86 implementation run without parallelism.

Xilinx MicroBlaze

Finally, a run from the list of officially submitted scores on the CoreMark website [14] will be included in the comparison. We chose an embedded system running on an FPGA without parallelism, to provide another useful comparison and to show what kind of performance one can expect in an embedded implementation. This run was compiled using GCC 4.1.2.

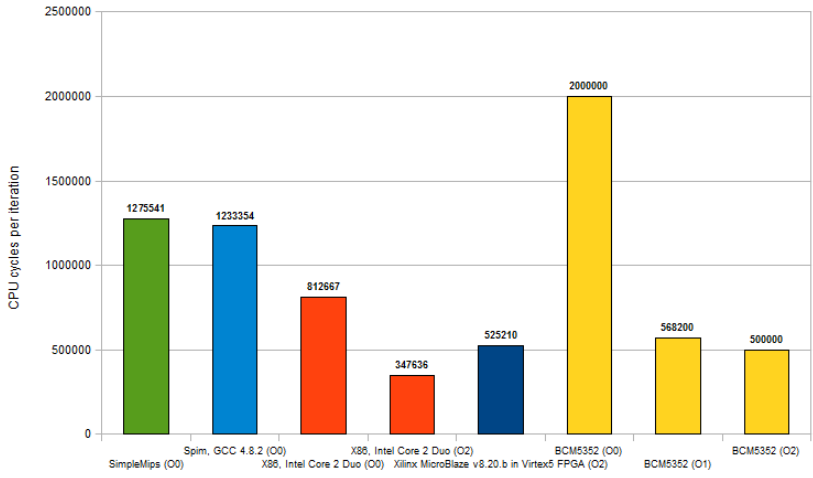

Figure 5.1. Performance comparison between a number of architectures and optimization levels, color coded by platform. Note that some results are approximations.

Chapter 6

Conclusions

Over the course of the thesis, the base compiler was retargeted to two different architectures, Senior and MIPS. The Senior implementation was trickier to achieve than expected, and in the end it didn't quite manage to provide a reliable benchmark reading. However, the generalization and subsequent MIPS implementation was successful.

As described in section 5, compiled with our compiler, CoreMark executes about 1.5 the amount of CPU cycles per iteration (simplifying things by assuming one instruction/cycle without parallelism) compared to x86 machine code produced by GCC 4.4.5. This is a substantial performance loss, but it may still be good enough for many purposes, considering the fact that performance critical parts will still likely be implemented in assembler and that the majority of the "structural" parts of the program will be considerably easier to follow and debug.

The difference between the SimpleMips and GCC 4.8.2 O0 runs made using the SPIM simulator are much smaller, however. This suggests that the difference between a simulator and real hardware is very significant, in absolute terms. For a truly useful comparison, the various compiled benchmarks have to be run on the same hardware or simulator platform.

Finally, the Mips hardware results, using a WRT54GL router, are somewhat anomalous for the O0 optimization level, as the results here are much worse than the readings seen in the SPIM runs. There are a number of different possible causes for this, and without more investigation, which is out of the scope of this project, it is hard to draw any in-depth conclusions. One possible cause is that the minor back-end optimizations done in the SimpleMips compiler are actually enough to make this much of a difference. Another possibility is that the results of the SPIM runs are skewed to the better due to inaccuracies in the implementation of the simulator.

In hindsight, the decision to use SPIM for the Mips processor emulation as opposed to another simulator, or actual hardware, may have been a mistake. The errors seen when running Coremark for higher optimization levels, both compiled with our custom SimpleMips compiler and with GCC 4.8.2, indicate that SPIM is

simply not a capable enough platform to perform this type of evaluation[1]. A better choice would probably have been QEMU[20] or compiling to actual hardware, e.g. the WRT54GL router. However, due to time constraints, this was not possible to do within the scope of this thesis.

On balance, even when accounting for minor errors in the benchmark setup, the Mips simulator issues, and the lack of data, this initial analysis indicates that the approach is feasible, especially for a certain set of use cases. For instance, a typical application to run on an embedded processor might be an image, video or audio decoder. These programs can be rather large, tens of thousands of lines of code or more, and the majority of the cpu time is spent in a comparatively tiny part of the code. For such a program, the initial cost of porting the compiler will be outweighed by the substantial gain of not having to re-write the majority of the code in native assembler. Performance-critical parts could still be re-written, making the loss in performance very small.

As far as the approach of using a pre-existing compiler implementation goes, it saved a great deal of development time, but may have caused the result - especially for the Senior port - to be worse than it might otherwise have been, simply because the base of the back end wasn't designed with portability in mind. It was also a work in progress in itself, with parts that hadn't yet been properly documented or tested. The base was also implemented using an older version of LLVM, which may have affected the quality of the end result negatively (though the exact effects of this have not been determined).

6.1 Future Work

While successful in certain areas, the compiler is in need of further improvement to make it practically usable. The proposed improvements and additions to the project are divided into short-term- and long-term goals. The short term goals primarily consist of bug fixes and relatively minor implementations in the back end. The long term goals are more focused on major additions and changes to the framework and project as a whole.

Short-Term Goals

- Implement proper support for ϕ nodes in the DAG node lowering part of the LLVM back end. This would enable complex conditionals to work properly, as well as likely allow higher optimization levels in the front end, as these tend to rely heavily on ϕ nodes.

- As discussed in section 4.3.3, certain operations are optimized away if the keyword `volatile` is not used. The cause of this has been hard to pinpoint, but it could potentially be a bug in the target-independent part of LLVM, which is slightly outdated (see section 4.1 for more discussion on this).

[1] There's still a possibility that the process of translating the generated assembly code into a format executable by SPIM was what caused these errors for both compilers, but without trying another simulator, it's hard to say for sure.

- Perform the evaluation using a simulator other than SPIM. This would either eliminate the problems seen with higher optimization levels or show that the error is not in the simulator.

Long-Term Goals

- Testing the compiler more or less extensively on real hardware. This would provide more accuracy than the results obtained in simulation, as we make certain simplifying assumptions in order to obtain meaningful measurements that are comparable to those of other platforms.
- Porting the compiler to more platforms, ideally to eventually get the chance of doing so with a relatively newly developed architecture to evaluate the core idea.

Bibliography

[1] Department of Electrical Engineering (ISY), Linkoping University, 2012. URL http://www.isy.liu.se/.

[2] Yiyan Tang, Yuke Wang, Jin-Gyun Chung, S. Song, and M. Lim. High-speed assembly fft implementation with memory reference reduction on dsp processors. In *Electronics, Circuits and Systems, 2004. ICECS 2004. Proceedings of the 2004 11th IEEE International Conference on*, pages 547–550, 2004. doi: 10.1109/ICECS.2004.1399739.

[3] Sejong Oh and Yunheung Paek. A quantitative comparison of two retargetable compilation approaches. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pages 29–36, 2003. doi: 10.1109/ICPP.2003.1240563.

[4] Rainer Leupers. System level mpsoc design: a bright future for compiler technology? In *Proceedings of the 13th International Workshop on Software & Compilers for Embedded Systems*, SCOPES '10, pages 9:1–9:1, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0084-1. doi: http://doi.acm.org/10.1145/1811212.1811225. URL http://doi.acm.org/10.1145/1811212.1811225.

[5] Aho, Lam, Sethi, and Ullman. *Compilers: Principles, Techniques and Tools*. Pearson Education, 2 edition, 2007.

[6] Jeremy Singer. SSA bibliography. URL http://www.dcs.gla.ac.uk/ jsinger/ssa.html.

[7] The LLVM Compiler Infrastructure Project, february 2012. URL http://llvm.org.

[8] GCC, the GNU Compiler Collection. URL http://gcc.gnu.org.

[9] Amy Brown and Greg Wilson. The Architecture of Open Source Applications, Chapter 11, LLVM.

[10] "clang" C Language Family Frontend for LLVM, february 2012. URL http://clang.llvm.org.

[11] Building the LLVM GCC Front-End, march 2012. URL http://llvm.org/docs/GCCFEBuildInstrs.html.

[12] Chris Lattner. LLVM Assembly Language Reference Manual, february 2012. URL `http://llvm.org/docs/LangRef.html`.

[13] Hans-Peter Nilsson. Porting GCC for Dunces, May 2000. URL `ftp://ftp.axis.se/pub/users/hp/pgccfd/`.

[14] CoreMark Scores, march 2012. URL `http://www.coremark.org/benchmark/index.php?pg=benchmark`.

[15] Dhrystone, march 2012. URL `http://www.sccs.swarthmore.edu/users/08/ajb/tmve/wiki100k/docs/Dhrystone.html`.

[16] CoreMark FAQs, march 2012. URL `http://www.coremark.org/faq/index.php`.

[17] Mason Woo and Misha Brukman. Writing an LLVM Compiler Backend, November 2010. URL `http://llvm.org/docs/WritingAnLLVMBackend.html`.

[18] Zeljko Juric, Sebastian Reichelt, and Kevin Kofler. The GNU Assembler, march 2012. URL `http://tigcc.ticalc.org/doc/gnuasm.html`.

[19] SPIM MIPS Simulator, february 2012. URL `http://spimsimulator.sourceforge.net/`.

[20] QEMU, open source processor emulator, june 2015. URL `http://wiki.qemu.org/Main_Page`.

[21] S. Daud, R.B. Ahmad, and N.S. Murhty. The effects of compiler optimizations on embedded system power consumption. In *Electronic Design, 2008. ICED 2008. International Conference on*, pages 1–6, 2008. doi: 10.1109/ICED.2008.4786702.

[22] Paul J. Dorweiler et al. Design Methodologies and Circuit Design Trade-Offs for the HP PA 8000 Processor. *Hewlett-Packard Journal*, august 1997.

[23] *Senior Instruction Set Manual*, September 2004.

Appendix A

Considerations for Future Architectures

When developing a new architecture (typically an ASIP[1]) there are several metrics to take into consideration. The traditional ones, such as size, complexity, performance, power consumption[2] and design time will usually be the most important [22], and balancing them alone can be very difficult.

One that is not always given due consideration is ease of development; that is, how easy it is to write useful code for a given platform. As discussed in the introduction in section 1, it would be very convenient if working, widely-used test cases could be recompiled for the new architecture as it was due for testing. Writing new test cases and even parts of the "live" code in C or another reasonably high-level language could also sometimes be preferable to writing them directly in assembler.

The ease of porting a compiler, as well as the speed and performance of the compiled code, is highly dependent on the hardware and instruction set of the new architecture. Sometimes, a design decision that will have seemingly little impact on the other metrics will make a monumental difference in how easy it is to retarget the compiler. This chapter, thus, describes a few basic guidelines to keep in mind when considering the retargetability of a new architecture.

A.1 Instruction Set

The instruction set of the processor is often tailor-made to suit a specific need, with added hardware to perform more complex tasks than a single instruction can normally handle, such as multiply-accumulate, certain matrix operations or even

[1] Application-Specific Instruction-set Processor

[2] Power consumption is a topic in itself in compiler research, and it has been shown that targeted compiler optimizations can have a large impact on power consumption [21].

convolution. Utilizing such instructions can be difficult at an early stage[3] in the porting of the compiler, but they can be very important for performance. Bear in mind, however, that it depends on the purpose of the compiler. The one designed and discussed in this thesis is mostly meant to be used in non-performance critical parts of the system, with the main workload routines, or large parts of them, still written directly in the native assembler.

It can sometimes be tempting to design your processor to perform certain operations in an unconventional way to save space and/or time, but it often creates problems for the compiler. Consider example A.1 from the Senior architecture:

Example A.1

The Senior architecture has two separate data memories, `dm0` and `dm1`[23]. Loading from and storing to data memory *x* is done using separate commands, `ld`*x* and `st`*x* respectively.

Allocating space for data in these memories is done using dot directives in the Senior assembler. `.ram0` and `.ram1` together with the `.skip` directive allows the programmer to reserve space for data in the RAM part of these memories. It being RAM, it will not contain any actual information before the program has started running.

To allocate constants, such as string data (which was used to some extent in the benchmark) ROM had to be used. Now, only `dm0` contains a ROM part. Data is allocated on it using the `.rom0` directive together with, for instance, `.dw` to allocate a 16-bit integer or `.df` for a floating-point number. Data is then accessed using `ld0`.

The problem in the case of the compiler is that there is no way in the back end to tell whether a load is performed on an address pointing to a pre-allocated constant or an address pointing to temporarily allocated RAM data, meaning there was no way to differentiate between the two memories. This would not have been a problem if it wasn't for the fact that `dm0` wasn't addressable using stack pointer related addressing modes, so we had to use `dm1` for stack allocated data.

We eventually solved it by appending assembly code to every program that reserves space for all the constant data using the `.skip` directive and then moves all the constant data into the RAM part of `dm1` at program start-up.

For the sake of the compiler, it can sometimes be a good idea to enable certain addressing modes and operations on registers that were originally not meant to have them. For instance, some (most) addressing modes enabled for the stack pointer, `SP`, that we needed `dm1`, with its corresponding load and store instructions, to use (see example A.1) were actually enabled for other address registers, of which `SP` is a subset. Enabling the same addressing modes for `dm0` would have reduced

[3] Though perfectly possible in theory, by recognizing patterns of instructions or modifying the front-end somewhat. See section 4.1.4 for more information about the instruction selection process.

the complexity of the problem significantly (and is very likely easy to change by tweaking the assembler program).

A.2 Hardware

The hardware of ASIPs, while not quite as customizable as ASICs[4], can still be modified to a large degree. It can be a good idea to take the issue of compiler retargetability into consideration when comparing hardware configurations.

A good starting point in hardware is to have a consistent, and sufficiently large, register set. It should be consistent in the sense that most registers should preferably be general-purpose, and work the same way. Even if the register types are different, using a common interface to read from and write to them will help in the design of a clean, efficient compiler for the architecture. Having many small subsets of highly specialized registers can be very efficient and make for good (albeit hard to write and understand) assembler programs, but will greatly increase the difficulty of writing a compiler that utilizes this functionality.

A.3 Things to change in Senior

This section contains a summary of what things could be changed in the Senior architecture to make it easier or better suitable for the compiler approach discussed in this thesis. Most of the points here have already been mentioned in the thesis, and will, if applicable, be referred to.

- **Addressing modes**. Enabling things like arithmetic calculations for address registers (very useful for stack/call frame handling) and offset addressing in general-purpose registers would simplify the porting a great deal. As matters currently stand, a fair amount of custom instructions have to be added when encountering situations where an addressing mode is not supported by a specific register class. For an example of a specific situation in which this occurs, see section 4.3.1.
- **More versatile instructions**. Making individual instructions more versatile would make the compiler backend leaner and simpler to write, and make it easier to create relatively optimized machine code from more varied source files. This includes making certain instructions support more addressing modes, or adding instructions to better support structures like function entry and exit.
- **Better programmer interfaces**. This point is mostly a matter of convenience - as can be seen in section 4.2.2, interfaces for printing values, measuring cycle count (or real time), but also more advanced debugging capabilities, would be a welcome addition to the platform.

[4] Application-specific integrated circuit, a piece of hardware fully customized for a specific purpose.

Appendix B

Readme for Porting the Compiler

The following appendix contains the readme that was written to help explain the parts of the porting process that are not entirely self-explanatory. It is divided into descriptions of the constituent files of the LLVM back end. Note that the original readme contains a table of contents, which has been omitted here.

B.1 Contents of the Readme File

B.1.1 <target>InstrInfo.cpp

A few machine-independent (given compliance with the requirements) optimizations that shouldn't need much modification. Begins with some stack- and register-related operations that should be scrutinized a bit closer.

B.1.2 <target>InstrInfo.td

This file is one of the main ones with regards to target-specific data. It contains definitions of formats for calls, conditional jumps, definitions of which types can be included in different instructions, addressing modes and the instructions themselves, or rather information about conversion between DAG nodes and assembler instructions is done.

Aside from making sure that the proper namespace for the target is used, make sure that the formats of the nodes for calls and conditional jumps corresponds to something that can be used by the new target.

As for instructions, start by implementing basic arithmetics and logics, reg/memory operations to get things started for simpler examples, and move on to other instructions as needed, as per specifications in the target instruction set.

B.1.3 <target>ISelDAGToDAG.cpp

This file describes a pass for converting certain instructions between DAGs. This file contains definitions for how to select nodes for specific addressing modes. Much of this will have to be closely examined when porting, to make sure the basic target-specific addressing modes are handled correctly.

B.1.4 <target>ISelLowering.cpp

Probably the file in which the most time porting will be spent. Contains definitions on how to replace and remove data types and operations that aren't natively supported by the target. This includes lowering (converting) certain special types of operations, such as function returns, function calls and their arguments, condition code handling and custom functions for certain operations that aren't natively implemented.

If a certain operation isn't supported for a certain type, this is where we specify how to attain the same functionality without it, such as expanding the operation into several operations for smaller types or promoting the type into a larger, supported, one by sign extending.

B.1.5 <target>MCAsmInfo.cpp

Some of the properties defined here may vary depending on target. The default values might work, but make sure to know roughly what the different values mean.

B.1.6 <target>RegisterInfo.td

Here, we'll specify the register set of the target. The example code shows how this is implemented for the Senior target, with 32 general registers, all of the same register class. Some special registers, such as the stack pointer, are specified, but for simplicity's sake they've all been added to the same register class. Limitations or special functions of these registers have been handled elsewhere.

B.1.7 <target>RegisterInfo.cpp

In this file, we define which registers should be saved by the callee during a function call and which registers to reserve (ie. won't be used by the register allocator at all).

It also has a number of functions related to stack frame allocation as well as entry and exiting of functions. Most of these will need some manual modification in order to work for the new target.

B.1.8 <target>SubTarget.cpp

If we want different subtargets with different features, we define most of them in this file. The primary concern with only one subtarget is the data layout definition, which specifies register and pointer size, alignments and the like.

B.1.9 <target>TargetMachine.cpp

A general description of the target machine. The most important thing to get right here is to use the correct subtarget and to make sure that its definition of the data layout is used.

B.1.10 AsmPrinter

The AsmPrinter folder primarily contains the <target>AsmPrinter.cpp class which determines how to write things like memory operands, condition codes and inline assembler operands. May be considered for tampering in case of last-minute hacks. The folder also contains a Makefile that may need to be modified.

B.1.11 TargetInfo

Contains a target info file, which contains info needed for LLVM's Target Registry, which is used by LLVM to look up info about targets at runtime.

Like other subfolders, it also contains its own Makefile which needs modified with the name of the new target.

B.1.12 Makefile

Needs modified to reflect the new name of the target, and what to call the fields generated from the .td files.

B.1.13 Additional Configuration

In order to be able to use the new target as a back end, some changes in the configuration will have to be made. In the "configure" file, add <targetname> to the `TARGETS_TO_BUILD` string.

Add the target name to the following files:

- `include/llvm/ADT/Triple.h`
- `lib/Support/Triple.cpp`
- `tools/clang/Basic/Targets.cpp`