Extending Simplic with an LLVM backend

Project in computer science, EDAN70
January 16, 2017

Kasper Bratz
Lunds Tekniska Högskola
dat12kbr@student.lu.se

Erik Hedblom
Lunds Tekniska Högskola
tpi11ehe@student.lu.se

Abstract

This paper explores the concept of extending an already existing compiler with a back-end for generating LLVM-intermediate code in place of x86-assembler code. A LLVM back-end will be written for an existing compiler for the the language SimpliC, a bare bones implementation of C, to examine what benefits can be made using LLVM. The SimpliC language will also be extended with some new functionality to compare complexity of implementation for x86-assembly and LLVM.

1. Introduction

When implementing a compiler there are a several important decisions to make and one of those is how to implement code generation. The compiler could use an intermediate representation (IR) to facilitate modularization, where the different modules use the IR as the common format. Another choice is whether to to target assembly directly or another lower level language, such as C.

LLVM is a compiler infrastructure with the goal of providing a strategy for optimizing and generating code for arbitrary programming languages [5] described further in section 3. The main focus of this article will be to explore the differences of using LLVM intermediate representation instead of X86-assembly as the target for code generation in a compiler.

To explore the potential benefits of using LLVM we decided to implement a LLVM back-end for an existing compiler for a language called SimpliC, a bare bones implementation of the C language further described in section 2. The existing compiler for SimpliC is implemented to generate unoptimized x86-assembly code, making it an ideal starting point to investigate the gains from optimizing code with LLVM. This paper will also briefly investigate the complexity of implementing a compiler back end in x86-assembly and LLVM-IR. To do this we will extend SimpliC with several features while taking note of what problems arise when implementing.

In addition to trying to asses how easy or difficult it is to implement a back end using LLVM-IR or x86-assembly as the target, we will also evaluate performance of both back ends. Performance will be measured in run time of the compiled programs, compilations time and the size of the binary output files. The result of our evaluation can be found in section 6 should convince you of the superiority of using LLVM.

2. The SimpliC compiler

The SimpliC compiler is a lab compiler developed in and used in the compiler course at Lunds Tekniska Högskola. It implements a subsets of the C language and only supports one data type, integers. Furthermore it supports variables, functions, if and while statements, Boolean and arithmetic operations. It is implemented in Java.

2.1 Scanner

The scanner is implemented using JFlex, which is a scanner generator implemented in Java[3]. This scanner is used for our lexical analyze.

2.2 Parser

A grammar on Extended Backus-Naur form, EBNF, is used to specify the SimpliC language and from the EBNF grammar the parser is generated. The parser generation is facilitated by Beaver, a Java LALR parser generator.

2.3 Code generation

The SimpliC compiler performs no optimizations and outputs x86-assembly. The x86 assembly code is generated from the abstract syntax tree using JastAdd which is a meta compilation system which supports reference attribute grammars[2]. Before generating code the SimpliC compiler also does some basic static analysis like checking number of arguments in function calls and name analysis.

3. LLVM

LLVM is a compiler infrastructure with the aim to supply a multi-platform strategy for compilation and was originally created to be used as a replacement to the existing code generator in the GCC stack. Today LLVM works as a large framework for compiling and optimizing software, providing a source and platform independent optimization strategy.

The LLVM infrastructure was designed to solve some problems of traditional compiler strategies, where code was represented as either high level abstract syntax trees or very low level assembly, not allowing for aggressive multi stage optimizations. [5].

LLVMs approach to these problems were the LLVM intermediate representation, a code representation similar in abstraction level to the assembler language while still including high level constructs for data flow, control flow, and type information. [5]

The LLVM-IR language is constructed with optimization and flexibility in mind and is designed to facilitate LLVMs multi-stage optimization system [5]. LLVM currently supports compile-time, link-time, run-time and idle-time optimization, providing high performance optimization. LLVM-IR is a cross-platform representation, abstracting away most machine specific instructions and can today be assembled to most common hardware architectures.

4. Implementing the LLVM back-end

There are a couple of key differences making LLVM-IR code generation both easier and more complex as opposed to x86 assembly generation. First of all LLVM-IR requires all register variables to be on single static assignment (SSA) form[4]. SSA-form require each variable to be defined exactly once and each use of a variable must be dominated by it's definition[1], meaning a variable must be defined before it can be used. We have circumvented part of this requirement by explicitly allocating all named variables in the stack frame, using the alloca instruction. By doing it this way we do not have to translate the variables to SSA-form. Only temporary variables introduced by the compiler have to be on SSA-form and since those are only used within a single basic block we can also avoid using Phi-functions, a way to determine what value a variable in SSA-form should assume after e.g a conditional expression.

An example of how variables are stored in LLVM could look like

```
//Assignment of 1 to the variable a.
%a = alloca i32
%tmp1 = add i32 0, 1
store i32 %tmp1 , i32* %a
```

Secondly LLVM has a higher abstraction level than x86-assembly, providing some higher lever constructs such as functions, if and while statements. These constructs correspond very well to the C equivalents, making them easy to implement. Furthermore LLVM-IR overloads functions[4] and hides register types. This did not have much impact on the original SimpliC LLVM back end since it only handled integers, but when we moved forward with implementing booleans and floating point values it eased implementation. This because we didn't have to worry about what registers to put what type of values in, like we did when working with x86. For instance integers and floating point values sometimes had to be stored in the same registers but had to have different mathematical operations applied to them. Instead, when working with LLVM, we could use typed variables to handle this problem.

We implemented SimpliC's predefined print functions in C, making standard library and which all programs are linked against during compilation. For consistency we also removed the x86-assembly implementations of the predefined functions and linked against the same library.

5. Extending SimpliC

With two working back ends we set out to extend the simpliC compiler with new functionality, including floats, booleans and structs.

5.1 Booleans

With the inclusion of Booleans, variables that can only evaluate to either true or false, the simpliC language now supports expressions like

```
    boolean a = true;
    boolean b = false;
    while(a){ */Statements/*}
```

When implementing booleans for the LLVM back end we utilized the built in typ i1, a 1 byte type used for boolean values. In the x86 back end we represent booleans as integers with either 0 or 1 values.

5.2 Floating point values

The initial version of the simpliC only supported integer values. With the inclusion of floats we can now write statements like

```
float a = 1.7;
float b = 1.2;
a = a+b;
```

LLVM supports a full instruction set for floating point values letting us implement floats in a similar way as booleans, using the float keyword instead.

In order to include floating point values in the x86 back end all floating point constants had to be pre-defined as quad-words like

```
.FLOAT_X:
        .long   1717986918
        .long   1074816614
```

where the floating point value is represented as two 32 bit integers, with the first value representing the 32 most significant bits, and the second the least significant bits. These values when concatenated and interpreted as a double precision float represent our value. These values can then be handled within the xmm registers of x86.

In both the x86 and LLVM back end we were also forced to use different instruction sets when performing arithmetic operations on floating point values.

5.3 Structs

With the inclusion of structs the SimpliC language now supports C-like variable containers like

```
struct foo = {
    int a;
    bool b;
    float c;
    struct bar = {
        int a;
        int x;
    };
};
```

Accessing these variables is done by navigating through the struct using dot notation like

```
foo.a = 1;
foo.c = 1.1;
foo.bar.a = 1;
printInt(foo.bar.a);
```

The most time consuming part of implementing structs for SimpliC was actually extending the front end of SimpliC. Making sure the scanner and parser correctly handled structs and that the abstract syntax tree had the required helper functions. What we did when implementing the x86 back end was to save field in the structs as regular variables, but made sure struct field access after this point pointed to the right memory address. In the LLVM back end fields in structs were instead saved with the dot notation they would later be accessed with.

```
struct a = {
    int x;
}

//Saved as
%a.x = alloca i32
```

The last problem we had to deal with in both back ends was how to pass structs as arguments. What we ended up doing was to simply "flatten" the struct to its individual fields, passing them one by one as arguments. Thanks to some previous work with the AST the arguments maintained their addresses/full names.

6. Evaluation

Evaluating the new LLVM back end in comparison to our existing x86 back end was performed in two steps. First we evaluated the performance of our compiler and generated code. Secondly we evaluated the complexity of the back end implementations. This was done similarly to other evaluations of LLVM back ends.[6]

The complexity of the back ends will primarily be interesting for developers wanting to implement their own back end, while code performance will be interesting for users of the compiler.

6.1 Performance evaluation

When evaluating performance of our generated code we considered run time, but also other metrics such as compile time and size of the compiled code. To test these metrics we introduce two test suites, $T^{(1)}$ and $T^{(2)}$.

- $T^{(1)}$
  Test suite 1 consists of compiling and running a source code file with a very naive implementation of the Fibonacci sequence.

```
int main(){
    int n = 40;
    int index = 0;
    while(index<n){
        printInt(fibonacci(index));
        index = index +1;
    }
    return 0;
}
int fibonacci(int i){
    if(i == 0){
        return 0;
    }
    if(i == 1){
        return 1;
    }
    return fibonacci(i-1) + fibonacci(i-2);
}
```

  The Fibonacci sequence is implemented with a running time of $\mathcal{O}(2^n)$. $T^{(1)}$ was run for $n = [35, 40, 41, 42, 45]$.

- $T^{(2)}$
  Test suite 2 consists of compiling a very large source code file generated to be filled with identical methods, each method is called once and the result is printed.

```
int f0() {
        int   i = 0;
        bool  b = true;
        float f = 0.0;
        while(i < 1000) {
                if (b) {
                        i = i + 1;
                        b = false;
                } else {
                        b = true;
                        f = 1.2 * f;
                }
        }
    return i;
}

int main() {
    printInt(f0());
}
```

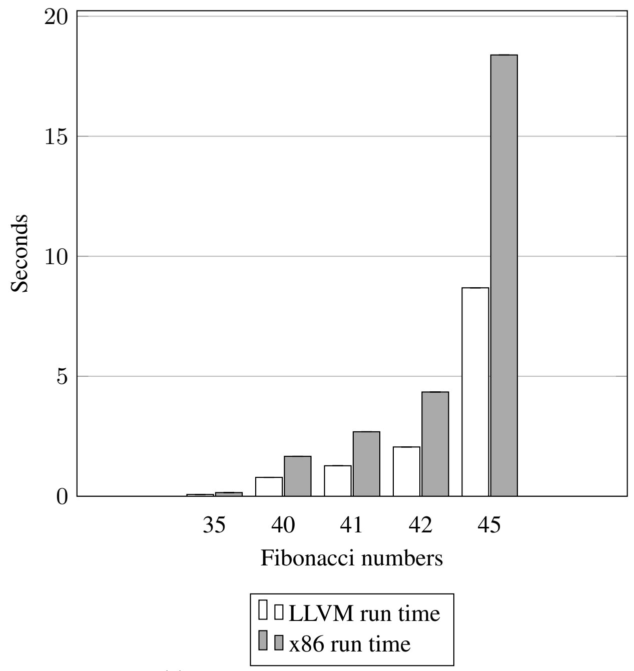
  $T^{(2)}$ was run for $[5000, 10.000, 20.000]$ functions.

$T^{(1)}$ is focused on measuring run time differences of the generated code from the x86 and LLVM back ends while $T^{(2)}$ is used to measure the speed of compilation and optimization.

6.1.1 Results

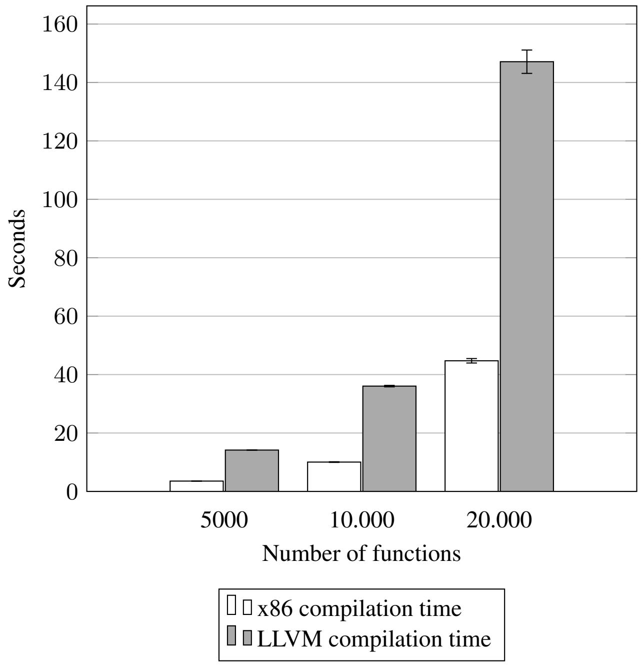
To ensure our results were consistent we ran each test suite 20 times and constructed a 95% confidence interval.

Result of $T^{(1)}$



The results of $T^{(1)}$ clearly show LLVM outperforming the unoptimized x86 code by $\approx 50\%$. Both LLVM and x86 does however still display the same exponential time behaviour introduced by the test suite. This indicates that LLVM is able to optimize away about half the execution time for each recursive call in the Fibonacci sequence, but not the naive implementation of the algorithm itself.

Result of $T^{(2)}$:



The results shows a compile time increase of $\approx$ 60% when compiling with our LLVM back end. When compiling with the LLVM back end the compiler will perform the same actions as the x86 compiler, but after generating LLVM IR code also run optimizations and code generation for a target hardware structure. With this in mind an increased compile time is to be expected.

Size of compiled code: The size of the binary file can be of importance when working in a resource constrained environment. For our test cases LLVM is able reduce the size of the binary file by a significant amount, up to 82%.

Code Size(bytes)	LLVM	x86
$T^{(1)}$	8 793	11 178
$T^{(2)}$ 5.000 functions	558 816	3 190 592
$T^{(2)}$ 10.000 functions	1 106 128	6 404 152
$T^{(2)}$ 20.000functions	2 218 944	12 853 784

6.2 Usability evaluation

When evaluating the usability of the LLVM infrastructure we are going to look at a few indicators that might give some insight in the difference in complexity when implementing a LLVM back end compared to an x86 back end.

One indicator can be how many source lines of code (SLOC) are required to implement a working back end for a language. This will give a general idea of the complexity behind an implementation.

Back-end	SLOC for simpliC back end
x86	316
LLVM	263

These numbers are quite modest, since the x86 back end more heavily relies on utilities introduced when building our abstract syntax tree such as node indexing.

As for the experience we had with implementation during this project , we found that outputting LLVM IR was significantly easier. This mainly due to the fact that LLVM IR is a higher level language, and such is more readable and intuitive to understand. There is also no need to worry about the stack frame, memory addresses or what registers is most suitable for each data type.

A good example is how a function definition could look. In x86 assembly you would have to worry about the stack and base pointers, pushing and pulling the correct arguments to appropriate registers and so on. In LLVM a function definition could instead look something like

```
define i32 @add(i32 %x, i32 %y) {
entry:
  %tmp = add i32 %x, %y
  ret i32 %tmp
}
```

It is clearly readable what variables are sent as arguments, what return type the function must have and no mentions of the stack or base pointers have to be made.

When extending SimpliC with support for functionality like floats and booleans we also found that the support for data types, floating point instructions and structs LLVM offers made the implementation several times easier.

For instance when adding two floating point values stored as variables we could write something like

```
%c = fadd float %a, %b
```

When implementing this in x86 we had to first predefine the floating point values on the stack and then save them in a specific memory adddress. When using the add operation the values would have to be moved to the xmm registers to be able to do floating point addition like

```
addsd %xmm1, %xmm0
movq %xmm0, %rax
```

With all things considered we fell that generating LLVM-IR is easier than outputting x86 code from a usability point of view. Implementing the LLVM back end was mostly a straight forward process, while the x86 back end involved allot of problems involving registers, data representation and fiddling with the stack frame.

7. Related work

Today LLVM is used to develop front and back ends for several languages. Some notable projects are

- Clang [1] - a compiler using a LLVM back end for C based languages like C and C++. Clang has the ambition to be able to be a full replacement for the GCC stack. Clang is, like our project, based on a LLVM IR back end.
- The Glasgow Haskell Compiler offers to use a LLVM back end when compiling [2], and has reported about 30% reduction in program run time when applied[3]. In our implementation we saw a $\approx$ 50% decrease in run time, but this was from completely unoptimised code, while GHC started with already optimised code.
- The OpenJDK zero assembler project offers a LLVM based JIT-compiler called Shark [4]. Shark is stable on platforms with a stable LLVM JIT.

High level virtual machines such as the JVM also offer a virtual instruction set that can be targeted by several programming languages, which can be considered an IR. Targeting high level virtual

[1] https://clang.llvm.org/

[2] https://downloads.haskell.org/~ghc/7.6.3/docs/html/users_guide/code-generators.html

[3] http://blog.llvm.org/2010/05/glasgow-haskell-compiler-and-llvm.html

[4] http://openjdk.java.net/projects/zero/

machines come with the the benefit that you can include libraries built for the VM, but has to follow a certain object model and often offer limited run time optimisations. Languages like Clojur [5] target the JVM with great success. Projects performed in earlier years of the course EDAN70 have implemented back ends generating java byte code, the IR for the JVM, instead of LLVM IR finding similar improvements in run time.

8. Discussion

After implementing a working back end generating LLVM IR instead of x86 assembly code and extending the compiler with new functionality we were able to come to some conclusions on weather a LLVM back end is a suitable replacement for the previous x86 back end. Based on the performance evaluation it appears to be a trade off between compile time and run time, with the LLVM back end outperforming the x86 back end by $\approx 50\%$ in run time with an increase of about $\approx 60\%$ in compile time due to code optimization. For most systems however, run time will be prioritised over compile time, since compile time is often trivial in comparison. Compiling with the LLVM back end will also in most cases reduce the size of the generated binary file, being an important aspect if the file is to be run in a resource constrained environment.

From a usability point of view we found that implementing in LLVM IR was more intuitive and produced more readable code. This mainly due to the higher abstraction level of LLVM IR and built in support for typing. This should become even more obvious when building compilers for more complex languages, with LLVM IR offering an extensive, well developed instruction set.

The fact that LLVM is both platform and source code independent is also a big advantage over x86.

The one downside that we found made LLVM more complex to work with is the requirement of variables to be in SSA form. We were able to circumvent this requirement due to the simplicity of the SimpliC language, but for a more complex language you would probably need to fully implement SSA form and Phi functions to support them.

As for the question if our LLVM back end is a suitable substitute for our x86 back end we find that the LLVM back end comes out ahead in almost every aspect and is a good choice.

9. Future work

To further improve the SimpliC compiler while still exploring the benefits of using LLVM, we feel it would be interesting to rebuild our compiler using the LLVM API [6]. This would allow us to further customise the optimisation passes of the LLVM optimiser and also allow us to extend the compiler with JIT support. This would be the best way to fully explore what gains can be made by using LLVM and what optimisations would be optimal for the SimpliC language.

It would also be interesting to write a third back end generating some other intermediate representation, such as java byte code. This would allow us to not only compare LLVM IR to unoptimised x86 code, but to other intermediate representations offering optimisations.

10. Acknowledgements

Finally we would like to extend our thanks to Alfred Åkesson of LTH for mentoring us through this project.

[5] https://clojure.org/

[6] http://llvm.org/doxygen/index.html

A. Appendix

References

[1] 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 Trans. Program. Lang. Syst.*, 13(4):451–490, 1991. doi: 10.1145/115372.115320. URL `http://doi.acm.org/10.1145/115372.115320`.

[2] G. Hedin and E. Magnusson. Jastadd–an aspect-oriented compiler construction system. *Sci. Comput. Program.*, 47(1): 37–58, 2003. doi: 10.1016/S0167-6423(02)00109-0. URL `http://dx.doi.org/10.1016/S0167-6423(02)00109-0`.

[3] G. Klein. Jflex user's manual. *Available on-line at www. jflex. de. Accessed August*, 2010.

[4] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88, 2004. doi: 10.1109/CGO.2004.1281665. URL `http://dx.doi.org/10.1109/CGO.2004.1281665`.

[5] C. A. Lattner. *LLVM: An infrastructure for multi-stage optimization*. PhD thesis, University of Illinois at Urbana-Champaign, 2002.

[6] D. A. Terei and M. M. T. Chakravarty. An llvm backend for GHC. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell, Haskell 2010, Baltimore, MD, USA, 30 September 2010*, pages 109–120, 2010. doi: 10.1145/1863523.1863538. URL `http://doi.acm.org/10.1145/1863523.1863538`.