Dynamic Compilation using LLVM

Alexander Matz
alexander.matz@ziti.uni-heidelberg.de

***Abstract*—This paper presents and analyzes the compiler framework LLVM which aims to enable optimization throughout the whole lifecycle of a program without introducing excessive software overhead. Its key technique is the utilization of a platform independent low level presentation of the source code that is enriched with high level information to allow extensive optimization using as much high level information as possible. Despite this low level representation using the instruction set architecure of a virtual machine it is never supposed to be interpreted by an actual implementation of this virtual machine, but to be translated to native machine code at runtime with a Just-In-Time compiler. This allows aggressive optimizations to be performed during compile and link time and enables profile-driven optimizations at runtime with potentially no additional effort. This paper focuses on the implementation of the LLVM on x86 architectures running GNU/Linux as this is the most common system configuration for high performance computing where aggressive optimization is crucial to efficiently use the available computing power and reduce energy consumption.**

Keywords*—*LLVM, Dynamic Compilation, Optimization, Just-In-Time compiler, Profile-guided optimization, Low Level Intermediate Representation

I. Introduction

With the current trend of computer architectures becoming more and more sophisticated and computing platforms becoming heterogeneous systems containing various types of processors, platform dependent optimization by hand becomes less feasable. In the past applications automatically benefited from increasing clock frequencies of CPUs but this is not longer the case as the need for energy efficient systems and physical limiations lead to more cores per CPU instead of a faster single core CPU. Additionally, with power becoming more expensive, optimization is a critical aspect of every application as executing poorly optimized programs simply becomes too expensive. This leads to higher expectations on the compilers used to develop future application. The traits expected of an ideal compiler include, but are not limited to, the following:

- **Fast compilation** Computing power is a rare resource everywhere, including the systems used to develop applications. Fast compilation not only allows to use the available power more efficient but also speeds up the development of applications as results are produced faster and therefore can be tested and improved faster.
- **Platform** The trend towards more complex and heterogeneous computing platforms leads to the desire to not having to develop for one specific architecture. Besides the ability to use the same application on different platforms without additional development effort this also removes the need to optimize for a platform and helps to reduce the time to market of applications.
- **Low startup latency** The startup latency especially impacts programs that are to be executed often but do not run for a long time and also results in wasted computing power and energy.
- **Low runtime overhead** Even the most optimized application can be slowed down by a heavy runtime environment featuring a rich abstraction layer but competing for CPU cycles with the application itself. This negative impact of excessive abstraction becomes more visible at applications running for a long time and should be reduced to a minimum.
- **Aggressive optimization at compile time** As mentioned before optimization greatly helps to reduce overall energy consumption but is increasingly difficult to do by hand. This should therefore be one of the most important aspects of any compiler.
- **Profile-guided runtime optimization** Huge optimization potential lies in analyzing code paths executed often and spending additional effort on optimizing these paths as much as possible. With this optimization taking place at runtime applications can adapt to changes of patterns of use almost instantly.

The great majority of compilers fall into two categories which both do not have all these traits. Both categories are and they drawbacks are illustrated with the following two examples.

Statically compiled and linked languages like C and C++ produce platform dependent machine code without any high level information about the program left. These programs do not benefit of new CPU instructions without recompiling the application. With statically compiled languages the chances for optimizations are mainly limited to the compile time and identifying bottlenecks at runtime to focus optimization on these regions adds significant development effort. They follow a "compile-link-execute" build process, which is widely regarded as a good approach because it allows to rebuild parts of an application without the need to recompile source files that did not change.

More modern languages based on virtual machines (VM) produce so called "bytecode" which is a platform independent intermediate representation (IR) of the program with high level information about the source code. To execute these applications on a machine they are either interpreted by an implementation of their virtual machine or compiled to native code at runtime. This allows adaption to newer CPU architectures without the need to recompile the whole program from source and allows adaption to patterns of usage at runtime. They usually provide a runtime environment that puts an abstraction layer between the application and the CPU architecture and operating system. This results in the

applications being platform independent but comes at the cost of increased software overhead at runtime due to the big abstraction layer and complex high level representation of the code.

The LLVM aims to fill the gap between these two approaches by keeping a platform independent representation of the source code at all times and generating machine code only Just-In-Time but at the same time not introducing excessive software overhead through abstraction. This promises optimization over the whole lifecycle of an application [1].

This paper is structured as follows: Section II gives an introduction to the overall system archictecture of the LLVM infrastructure and the important design decisions. Section III briefly introduces the intermediate representation which is used as the basis for all optimizations. After the introduction to the internal code representation the Just-In-Time compiler in use and its features and limitations are explained in section IV. This is followed by an analysis of the profile guided optimization currently possible with LLVM in section V. The section VI presents some projects with high performance requirements which use the LLVM infrastructure. A final conclusion concerning the potential and current status of the LLVM project is drawn in section VII.

II. System architecture overview

The general design of the LLVM compiler framework is that of a modular library whose modules can be used independently of each other or in the form of a compiler toolchain similar to that of the GNU C compiler. The whole system revolves around the low level intermediate representation (LLVM IR) with the first step being the translation of arbitrary high level programming languages into the LLVM IR and the last step being the translation of the intermediate representation into native machine code at runtime [2]. This approach substantially differs from that of static compilers where the internal code representation is translated into native code after the compilation step but before linking the application. High level type information is lost at this point and makes the corresponding optimizations difficult but in return enables very low startup latencies of executables as all native code is already generated. It also differs from the approach of common virtual machine based compilation frameworks, as their intermediate representation is usally on a higher level than the LLVM. These VM based languages allow aggressive optimization throughout the whole lifecycle of a program at the cost of higher startup latencies. The higher startup latencies stem from the code generation being more complex from a high level intermediate representation than from a low level IR.

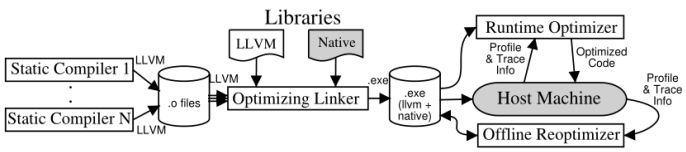

Figure 1. LLVM system architecture, taken from [3]

A common use case of the LLVM is its use as a compilation toolchain for languages like C, C++ and Objective-C. In this case the build process follows the well known compile-link-execute model which can be seen in figure 1:

- **Compile** A frontend like Clang translates all source files from their high level language to the LLVM IR and performs several optimizations. The optimization scope of this step is limited to one translation unit (source file). The results are one "object file" per input file containing a compressed form of the LLVM IR (LLVM bitcode).
- **Link** The resulting object files are now linked into a program still containing LLVM bitcode. Further optimizations are performed with the scope being the whole program this time, which significantly enhances the accuracy of some optimizations (for example dead-code elimination).
- **Execution** When the program starts, the LLVM runtime environment uses a Just-In-Time compiler (jit compiler) to generate native machine code for the target architecture. The native code generator applies platform specific optimizations not possible on the platform independent LLVM IR.
  With the (modifiable) LLVM IR still being present at this point, instrumented programs can collect information on the typical patterns of usage. This information can be used to adapt the program to its usage in the field during runtime and idle time.

Almost all optimizations are performed on this intermediate representation, which is therefore to be kept at all times to utilize its full potential. The low level nature of the IR and the fact that it is already very optimized allow cheap machine code generation without the need to reapply all standard optimizations at this point [3].

The differences to statically linked applications are the preservation of the intermediate representation throughout the whole lifecycle of the application and more extensive optimization crossing several translation units.

Although the differences to VM based languages are not directly visible in the system architecture diagram, they are not any less important. One difference is the intention to never interpret the intermediate representation on a virtual machine but to always generate native machine code before execution. This is easily possible because the intermediate representation used is deliberately kept on a low level which has the benefit of cheap code generation. The structure and reasoning of this code representation is introduced in the next section [3].

III. The LLVM intermediate representation

The design goal of the LLVM intermediate representation is to provide a representation low level enough to allow cheap machine code generation and at the same time providing high level information enough to allow aggressive optimization. It is a language in static single assignment form using a virtual instruction set architecture for a virtual machine with unlimited registers. It aims to support only the key operations of typical processors without introducing high level or machine specific features. This acts as a common ground and can implement arbitrary high level constructs without the need to have direct built-in support for them [3].

The static single assignment (SSA) form is a criteria for languages requiring that each variable is only allowed to be assigned once. This greatly simplifies data flow analysis as the determination of the origin of each value is trivial [4]. In the case of the LLVM IR variables are an unlimited number of registers which later have to be reduced to the physically available registers for machine code generation.

The distinctiveness of the LLVM IR is its strong type system. Every register and operation has a primitive type associated with it and operations are only allowed to use registers with the same type. These primitive types are integral, floating point and pointer values. All operations are polymorphic and need the type they operate on to be specified. Additionally the most basic high level constructs like arrays, structs and functions are also provided as well as functions to access their members. They do not, however, represent classes of object oriented programming languages. Classes and their properties have to be mapped to the available simple data types. This approach of enforcing the use of types ensures that all optimizations performed on the LLVM IR are completely aware of types in use and do not break the code although the intermediate representation is on a low level [1].

```
#include <stdio.h>

int main(int argc, char** argv)
{
  printf("Hello World!\n");
  return 0;
}
```

Figure 2. C source code of a hello world application

```
@.str = private unnamed_addr constant [14 x i8] c"Hello ↩
    World!\0A\00", align 1

define i32 @main(i32 %argc, i8** %argv) nounwind uwtable {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i8**, align 8
  store i32 0, i32* %1
  store i32 %argc, i32* %2, align 4
  store i8** %argv, i8*** %3, align 8
  %4 = call i32 (i8*, ...)* @printf(i8* getelementptr ↩
      inbounds ([14 x i8]* @.str, i32 0, i32 0))
  ret i32 0
}
```

Figure 3. Resulting program in LLVM IR (using LLVM 3.0)

As a byproduct of keeping the LLVM IR as long as possible and generating code being the last step or performed at runtime the resulting programs are, within limits, platform independent. To support a platform, only a code generator for that system and the libraries used in the programs have to be present on the target system. A hello world program shown in figures 2 and 3 is used to clarify this. The code in figure 2 shows a minimal c application that just executes a `printf` command. The resulting code in LLVM IR shown in figure 3 shows how the arguments to this call are prepared, the call performed and the exit code returned. No actually platform dependent command is present in the code fragment, the only prerequesit necessary to run this program on any platform without recompiling the bitcode is, that it is linked against a library containing a printf function with the same prototype. But this does not replace the abstraction layer providing access to operating system functions of programming languages designed to be platform independent. These features are deliberately not provided by the LLVM runtime to avoid overhead.

The instruction set architecture of the LLVM IR uses a memory layout which fits the overwhelming majority of CPU architectures. The memory is divided into a heap and a stack, with the stack being mostly used to keep track of the call stack and passing arguments to functions whereas the heap is used for all complex data structures and data not bound to the lifetime of a function. Memory traffic is exclusively possible through load and store operations. Memory on the heap has to be allocated with `malloc` and freed with `free` while memory on the stack is allocated with `alloca` and automatically freed on returning from the corresponding function.

Although the LLVM does not directly implement any specific form of exception handling, it does provide a mechanism to implement arbitrary exception handling models. This mechanism are two seperate operations to call functions. `call` calls a function without the possibility to throw exceptions while `invoke` takes two arguments, one being the function to call and one being a label to jump to in the case of an exception. One consequence of this approach is that there are no additional instructions executed in the case of no exception being thrown [1].

The final step from generating LLVM bitcode to executing a program is generating the native machine code and is explained in following chapter.

IV. LLVM Just-In-Time Compiler

A critical part of the LLVM system architecture is a Just-In-Time compiler that generates native machine code out of the LLVM IR because despite the acronym LLVM standing for Low Level Virtual Machine, the intermediate representation is acutally not supposed to be executed in a virtual machine. It differs from common VM based programming languages using interpreters where jit compilation is just used as a means of optimization. The virtual machines of these languages usually use a much higher level intermediate representation specifically tailored for the interpreter in use. As mentioned before, the intermediate runtime is only used to provide common ground for optimization algorithms which therefore not need to be customized to any source language or target instruction set [2].

As the LLVM jit compiler translates the LLVM IR into native machine code, it does not fit the common understanding of a compiler translating a high level programming language into an assembler language. It is therefore called a "code generator" in the context of LLVM. The process of generating native machine code at runtime is nonetheless still referred to as "jit compilation" in this paper.

Besides translating the intermediate representation into native machine code the code generator performs all optimizations that depend on the target architecture. An example for platform dependent optimizations on modern x86 architectures are vector operations like SSE and AVX which are supported through auto vectorization [5]. As long as the bit code uses vector operations, the most effective vector operations present on the target machine are used. The possibilities to generate machine code this optimized for the actual target machine is

new for languages that are usually statically compiled like C or C++.

The following analysis of the jit compilation with LLVM is focused on x86 using GNU/Linux, as this is currently the most common configuration used for scientific and high performance computing in general. There are two common approaches to Just-In-Time compilation: generating code the first time it is called, and generating the machine code for the whole program once at startup. LLVM currently uses the latter approach. Although it is generally propagated that both the LLVM IR and native code are present in an executable produced with the LLVM infrastructure, this is currently not the case and developers have the choice between the following two options:

- **Generating a native application** If the Clang frontend is used as a gcc drop-in replacement, the resulting executable is a completely native application without any LLVM IR bitcode present. The actual LLVM infrastructure is only used in the compilation step and produces regular object files. The object files are then linked with the default linker of the system. This completely eliminiates all Just-In-Time compilation, as machine code is already generated and no LLVM bitcode is available which could be used for this.
  Starting from the original idea of keeping both representations, this is a move in direction of statically compiled languages.
- **Generating LLVM bitcode** The alternative is using Clang exclusively as a frontend. In this case the LLVM bitcode is emitted and the corresponding machine code is generated once at startup of the executable. Startup latency is increased in this case due to the need of code generation but platform dependent optimizations are more accurate.
  As this also does not meet the original idea of the LLVM, it is a move towards VM based languages.

Due to platform dependent optimizations the generation of bitcode and using the jit compiler seems more beneficial. The increase in startup latency can usually be tolerated as code generation from the LLVM IR is fast due to its closeness to common CPU ISAs.

Compiler frameworks using high level intermediate representations can apply many low level optimizations (for example global value numbering) only at the time of jit compilation and have to reapply them every time they generate native code. This is not the case with LLVM as all optimizations possible on the LLVM IR have already been applied at this point so the only optimizations left are platform dependent ones. This further reduces the higher startup latency introduced with jit compilation and makes this model an even more viable option.

V. Profile-guided optimization

Profile-guided optimization is the sum of all optimizations that are impossible to perform without knowledge of the patterns of usage of the application in the field. This information is generally reduced to the so called "hot path" of a programm which is the portion of code the application spends the majority of its execution time in [6]. A typical example for such hot paths are inner loops in programs doing extensive calculations, such as simulations.

The goal of profile-guided optimization is to spend extra effort in optimizing an application for the case that the hotpath is traversed without "wasting" time on optimizing code regions that are barely used. On of these optimizations implemented in LLVM is basic block placements which moves code blocks so that caching effects are leveraged and jumps avoided. It is usally desirable to perform these optimizations at runtime to adapt the application as fast as possible to the patterns of usage of its user.

Statically compiled and VM based languages follow two different approaches to enable profile-guided optimization:

- **Statically compiled** They follow the classical "compile-link-execute" build model. To enable profile-guided optimization the application has to be instrumented to collect usage information during runtime. This information can only be incorporated into the program by recompiling and linking it again with the applied optimizations. This changes the build model to a "compile-link-profile-compile-link-execute" build model which introduces significant additional effort.
  As the goal of profile-guided optimization is to adapt applications to their users, they should be the ones profiling the application. This would be possible by either providing them the source code and build environment or collecting feedback from them, recompiling the application and redistributing it. Both of these approaches complicate the build process to a degree where it usually is no longer feasable.
  Additionally, using statically compiled and linked applications limits the possibilities of profile-guieded optimization to be performed during idle times between runs, as the application does not have a jit compiler to generate code at runtime.
  As a result profile-guided optimization is skipped in most cases.
- **VM based languages using jit compilation** In this case profile-guided optimization can be applied without additional development effort. The runtime environment can instrument and profile the application transparently and dynamically optimize and recompile the hot paths during runtime. A well known example for this is the Oracle Java runtime environment "Hot Spot" [7].
  One drawback is the large software overhead introduced with the Java runtime environment and the optimizer/jit compiler competing for CPU cycles with the actual application [3].

LLVM aims to fill the gap between these two approaches by keeping the "compile-link-execute" build model but allowing transparent profile-guided optimization in the field (on the target system) without additional development effort. This is to be achieved by the keeping the LLVM IR and native machine code at runtime together with mapping information about how the machine code maps to the intermediate represenation. The LLVM runtime environment can then instrument the application and gather information about the hot paths during execution. Once the hot paths are identified, cheap

optimizations are applied at runtime while more costly and potentially more effective optimizations are applied during idle time between runs of the program (see figure 1). As mentioned above, the cost of code generation is to be low due to the low level character of the LLVM IR.

The implementation on x86 GNU/Linux currently suffers from several limitations. The most important limitation is the lack of optimization during runtime. As all machine code is generated once at startup, it is not updated when the application is still running. This only leaves offline optimization between different runs of the program, although this is not critical for programs invoked. The other important limitation is that profile-guided optimization is not enabled automatically. It can easily be enabled and automated but it nonetheless is the not wished for zero-effort or transparent solution. The optimizations benefitting from profile information are currently limited to basic bock placement.

VI. Projects using LLVM

The approach of LLVM being designed as a library has lead to widespread usage in other projects. Especially the extensive optimizations available for the LLVM IR and the already implemented native code generators make it a welcomed basis for projects in need of a compiler. The following projects show two different use cases for the LLVM where the projects replace different components of the LLVM infrastructure.

- **Ocelot/PLANG** These two projects try to overcome the challenge of the increasingly complex systems with the upcoming of systems using General Purpose GPUs. They allow the execution of NVIDIA CUDA kernels on x86 architectures (Ocelot and PLANG) and different GPUs (Ocelot) [8], [9]. Only the execution on x86 architectures uses the LLVM infrastructure as CUDA kernels are already designed to be executed on GPUs. Their approach is to translate PTX programs, which is the intermediate representation of CUDA kernels, into the LLVM IR and modifying them to keep the behaviour of the GPU programming paradigm, bulk synchronous parallel programming. They then use the LLVM optimizers and code generators to generate x86 machine code which can then be executed on an arbitrary number of CPU cores [10].
  This project can be seen as an additional LLVM frontend as the focus of it is the translation from a source language (PTX) to the LLVM IR.
- **OpenCL to FPGA compiler** This project translates OpenCL kernels (similar to CUDA kernels) to a hardware description language which can then be used to program accelerators with excellent performance per watt. Due to the closeness of OpenCL to C, it uses the Clang frontend to compile OpenCL to the LLVM IR and then translates the LLVM IR to the hardware description language VHDL [11].
  As the translation into LLVM IR is already covered by clang and this project focuses on the translation from LLVM IR to VHDL, it can be seen as a LLVM backend.

Both these projects significantly benefit from the optimizations available for the LLVM and its front- and backends. Despite them not fitting into the build process of typical applications they still benefit from improvements on the LLVM infrastructure without additional effort.

VII. Conclusion

LLVM follows an interesting approach and tries to fill a gap between statically compiled and VM based languages that allows new optimizations especially for languages written in languages that are usually statically compiled (like C and C++). Clang and LLVM as a drop-in replacement for the gcc work well in most cases and generally result in smaller executables with comparable performance and greatly reduced compilation time. Projects making use of peculiarities of the gcc cause problems but this is acceptable as they are, per definition, not part of the official language standards. The modular design as a library allows great synergy effects between all projects using the LLVM and the LLVM project itself and may lead to rapid development.

It already is a mature compiler framework featuring extensive optimization, although not all promised features are implemented at this time. Especially the incomplete support of profile-guided and runtime optimization, which are the most promising advantages of using LLVM, do not give it the hoped-for headstart. Despite this, the fast progress of the project and the large user and developer base promise further progress and the implementation of the still missing features.

References

[1] Chris Lattner, Vikram Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," Mar 2004.

[2] Chris Lattner, Vikram Adve, "The LLVM Instruction Set and Compilation Strategy," Aug 2002.

[3] C. Lattner, "LLVM: An Infrastructure for Multi-Stage Optimization," Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, F. Kenneth Zadeck, "Efficiently computing static single assignment form and the control dependence graph," Oct 1991.

[5] Hal Finkel, "Auto vectorization with LLVM," Apr 2012.

[6] Andreas Neustifter, "Efficient Profiling in the LLVM Compiler Infrastructure," Apr 2012.

[7] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox, "Design of the java hotspot^{TM}client compiler for java 6," *ACM Trans. Archit. Code Optim.*, vol. 5, May 2008.

[8] Vinod Grover, Andrew Kerr, Sean Lee, "PLANG: PTX Frontend for LLVM," Oct 2009.

[9] Andrew Kerr, Gregory Diamos, S. Yalamanchili, "Dynamic Compilation of Data-Parallel Kernels for Vector Processors.," Apr 2012.

[10] Andrew Kerr, Gegory Diamos, Sudhakar Yalamanchili, "A Characterization and Analysis of PTX Kernels," Oct 2009.

[11] Markus Gipp, "Online- und Offline-Prozessierung von biologischen Zellbildern auf FPGAs und GPUs," May 2012.