Independent Study Report: Compiler Optimizations with LLVM

Brian Leibig

November 13, 2012

Introduction

The purpose of this independent study is to investigate compiler optimization techniques and experiment with implementing them with the Low Level Virtual Machine (LLVM)[1] compiler infrastructure. This study includes a report on the design and implementation of LLVM based on the paper that introduced it in 2002, and how it has evolved to what it is today. Many of the topics of study and projects are borrowed from a compiler optimization course from Carnegie Mellon University[2], as well as chapters 8 and 9 from the book *Compilers: Principles, Techniques, and Tools*[3], commonly known as the Dragon Book. The goal of these projects is apply the techniques I investigated and learned about to create optimization passes for LLVM capable of improving the performance of compiled programs. For this study, I used version 3.0 of LLVM.

LLVM Concepts

LLVM as it exists today is an umbrella project for many different compiler and compiler-related tools, but at the core is a language known as LLVM IR (Intermediate Representation). LLVM IR is a simple assembly-like language designed to be a target for compilers of languages that want high performance and portability. By having LLVM as a compilation target, the compiler can take advantage of its optimizations to improve runtime performance of the programs it compiles, and be portable amongst many different CPU architectures given LLVM's wide range of code generation targets.

LLVM IR has three representations: a textual assembly language, a compact binary compiled form (bitcode), and an in-memory form for analysis and transformation using its C++ API. These forms are isomorphic, meaning they can be converted amongst each other with no loss of information. For example, two of the tools the LLVM toolchain provides are *llvm-dis* and *llvm-as*, which disassemble bitcode to assembly and compile assembly to bitcode respectively. Also, the C++ library can read and write between persistent representations and the in-memory representation, which is a hierarchy of C++ objects.

```
define i32 @euclid(i32 %a, i32 %b) nounwind uwtable ssp {
bb:
  br label %bb1
bb1:                                              ; preds = %bb2, %bb
  %.01 = phi i32 [ %b, %bb ], [ %tmp3, %bb2 ]
  %.0 = phi i32 [ %a, %bb ], [ %.01, %bb2 ]
  %tmp = icmp ne i32 %.01, 0
  br i1 %tmp, label %bb2, label %bb4
bb2:                                              ; preds = %bb1
  %tmp3 = srem i32 %.0, %.01
  br label %bb1
bb4:                                              ; preds = %bb1
  ret i32 %.0
}
```

Figure 1: Example of the Euclidean algorithm in LLVM assembly. LLVM's type system has integers of fixed (but arbitrary) precision, so the standard int type is `i32` and booleans are of type `i1`.

The LLVM assembly language sits between C and real assembly languages with respect to how low-level it is. It is higher level than assembly in the sense that it is statically typed, supports abstractions for functions and function calls, and has unlimited registers, but is lower level than C in the sense that it lacks control-flow abstractions such as if and while, and all code must be written as simple assembly-like instructions. LLVM's type system supports nearly all of the types seen in C, including numerical primitives, pointers, arrays, and structs. Most LLVM instructions are polymorphic based on type, and thus take an extra parameter to specify the type of the result.

LLVM is written static single assignment (SSA) form, which means registers can be declared and assigned to only once, and the definition must dominate (come before) all of its uses. The reasoning behind SSA is that these constraints simplify many optimizations, especially with regards to data flow, as computing use-definition chains becomes trivial. Although registers are immutable, there is a way to bypass this restriction to handle mutable variables by allocating data for these variables on the stack with the registers holding pointers to this data. The value of these variables can then be read and written with load and store instructions using the register holding the pointer. Although this makes it easy to correctly handle mutable variables, it is inefficient, so LLVM provides an optimization pass called "mem2reg" that converts stack-allocated variables to registers to the best of its ability.

Code in LLVM is organized using *modules*, *functions*, and *basic blocks*. A module represents a translation unit of the higher level language being compiled, and contains declarations or definitions for types, static data, and functions. Functions typically represent the function of the higher level language it was compiled from, and contain the argument list and a collection of basic blocks that determine the control flow and code executed. A basic block starts with a label that identifies it, and contains a sequence of instructions, with the last one being a special terminator instruction that transfers control flow unconditionally or conditionally to one or more basic blocks (which may include itself), or returns from the current function. The basic blocks in a function form a directed graph called a *control flow graph*.

One of the most important types in LLVM defined in the C++ library is `Value`. `Value` represents a typed operand that can be used as an operand in an instruction. `Value`s also keep track of all of its uses by other instructions, for easy def-use analysis. The `User` class is a subclass of `Value` that represents nodes that refer to other values, and provides access to them through a list of operands. `Instruction` is a subclass of `User` that represents an instruction, holding an opcode for the type of instruction and a reference to its parent basic block. Because LLVM is in SSA form, a register is only defined by one instruction, so a user refers to a register assigned to by an instruction as the instruction itself, which is why `Instruction` is a descendent of `Value`. `Instruction` also has many subclasses to provide more specialized functionality to specific kinds of instructions, such as terminator instructions or binary operations.

Because LLVM is in SSA form, one issue that must be dealt with is the situation where a variable's value depends on the specific control flow path taken. For example, a loop's induction variable might have the value 0 if the loop is first being entered, or it will have an incremented value after an iteration of the loop. To deal with this, SSA (and LLVM) has a *phi instruction* that conceptually chooses the value for a variable amongst multiple alternatives depending on which predecessor block transferred to the current block. All phi instructions must come before all other instructions in a block. Since phi is not a real instruction supported by any ISA, this is translated to assembly by inserting a move instruction into each predecessor of a block with a phi instruction, assigning that block's value to the target of the phi instruction.

Analysis of Lattner's MS Thesis

Originally written between 2000 and 2002 as a MS thesis by Chris Lattner, the LLVM project was designed to address several shortcomings in the then-current state of compiler technology: insufficiently powerful link-time optimization due to lack of type information at link-time, lack of ahead-of-time compilation and optimization by virtual machines such as the JVM, and the cumbersome process of using profile-driven optimization to identify areas of programs that could benefit from optimization the most. The solution proposed was a *multi-stage* optimizer that does appropriate optimizations while code is being compiled, linked, run, and idle between runs, using a programming language- and architecture-independent IR format. At compile time, static optimizations that can be done within a translation unit are performed, such as strength reduction, constant propagation, and function inlining. At link time, LLVM employs inter-procedural optimizations now that it has all of the code from the whole program. At this point, the code is still in LLVM IR form, and thus still has high-level information such as types in it, so this allows for much more aggressive optimization than if it were in machine code. At the very end of the link stage, the optimized LLVM IR is translated into native machine code, and if post-link optimizations are enabled, the LLVM IR is included in the executable itself to allow for further profile-driven optimization.

LLVM code is analyzed and transformed using a simple modular tool called *opt* that reads an LLVM bitcode file, performs an analysis or transformation pass, and then outputs the bitcode file after processing. The passes can either be one of LLVM's built in passes or come from a dynamically loaded library given by the `-load` option. When writing a pass, it is declared to operate on a certain scope of the LLVM structure. For example, a `ModulePass` operates on the module as a whole, a `FunctionPass` operates on functions independently of each other, and a `BasicBlockPass` operates on the basic block level.

One interesting thing not mentioned in the thesis was that the LLVM project was not just a feat of compiler technology, but also a feat of good software engineering and architecture as well. One of the core design principles of LLVM and its related projects is the code should be designed in a modular fashion as a set of loosely-coupled libraries, rather than a monolithic tool. This not only allows the compiler to be well-engineered, but makes it easy to build a toolchain around the compiler as well. For example, the clang c compiler (built on top of LLVM) provides a "libclang" library that allows external tools to query for semantic information from C/C++/Objective-C programs to power editor and IDE features such as syntax highlighting, auto-completion and cross referencing. Also, it has made it easy for a new C debugger called LLDB to come into existence, which reuses much of clang and LLVM's technology. The key idea here is that compilers gather a lot of information about programs which are useful for tasks other than compilation itself, and this modular design allows this information to be easily used by these other programs to create powerful tools.

Local Optimizations

All compiler optimizations share the same goal of transforming code to be more efficient (or more generally, no less efficient) and still semantically identical, and this is usually done by reducing the amount of computational work done by a program. Some of the simplest optimizations include *local optimizations*, which work at the basic block scope that mend small inefficiencies, usually at the instruction level. Some of these include:

1. Algebraic identity optimization: This optimization simplifies or removes instructions if the instruction is a binary operation that performs an algebraic identity. For example, addition with 0, subtraction by 0, multiplication with 1, or division by 1 are all algebraic identities because the result

is the same as the other operand. This would simplify an instruction of the form `x = y + 0` to `x = y`.

2. Constant folding: This optimization replaces constant expressions with the value of the expression. This includes changing binary instructions whose operands are constants to a simple assignment. For example, this optimization would change `x = 2 + 3` to `x = 5`. Also, this replaces variables whose value is a compile-time constant with the constant itself, so to continue the previous example, all references to `x` from that point on would be changed to `5`.

3. Strength reduction: This optimization simplifies expensive operations that take multiple CPU clock cycles with an equivalent but less computationally-intensive operation. For example, multiplying or dividing a integer by a power of 2 is equivalent to a left or right bit shift, and the remainder operation by a power of 2 is equivalent to a bitwise-and and a right bit shift.

I implemented these optimizations as a basic block pass for LLVM. The pass simply iterates through each instruction in the basic block and checks the opcode to see if it binary operation that can be optimized. If so, the operands are then checked to see if they satisfy a certain criteria for the optimization (for example, constant folding would check to see if both operands were constants). If the criteria checks, the instruction is removed or replaced with a more efficient version.

Data-flow Analysis

More complex optimizations whose scope goes beyond a single basic block belongs in the family of *global optimizations*. At the heart of most global optimizations is a technique called *data-flow analysis*. Data-flow analysis is a family of algorithms that helps describe what the state of a program is like at a certain point with regards to the semantic significance of values at that point. Information from these analyses can enable optimizations such as constant folding, dead code elimination, efficient register allocation, and common subexpression elimination, as well as producing useful information for a debugger.

All data-flow algorithms follow a common pattern, known as the data-flow analysis schema. This is a general algorithm that takes a group of basic blocks (or a function), and determines which facts, known as *data-flow values*, hold at each point in the program. Data-flow values are typically a subset of all variables defined amongst all basic blocks, and the program points are the points before and after each basic block (or even each instruction for a fine-grain analysis), known as the IN and OUT points. All data-flow values including the kind previously described must be a *semilattice*, a special kind of set that has a meet operation $\wedge$ defined as a binary operation between two operations that is idempotent, commutative, and associative. Typically, meet is set union or

```
$\text{OUT}[\text{ENTRY}] = v_{entry}$;
for (each basic block $B$ other than ENTRY) $\text{OUT}[B] = \top$;
while (changes to any OUT occur)
    for (each basic block $B$ other than EXIT) {
        $\text{IN}[B] = \wedge_{P\ a\ predecessor\ of\ B}\ \text{OUT}[P]$;
        $\text{OUT}[\text{B}] = f_B(\text{IN}[B])$;
    }
```

Figure 2: Iterative algorithm for forwards data-flow, from the Dragon Book

intersection. Semilattices also have a *top* value $\top$ such that for all elements x in the semilattice, $\top \wedge x = x$, and bottom value $\bot$ such that for all elements x in the semilattice, $\bot \wedge x = \bot$. If $\wedge = \cup$, $\top$ is $\emptyset$ and $\bot$ is U (the set of all elements). If $\wedge = \cap$, $\top$ is U and $\bot$ is $\emptyset$. In the data-flow algorithm, the meet operation serves as a way to propagate data-flow values from a basic block's predecessors or successors, and the $\top$ and $\bot$ values initialize the IN and OUT values, depending on the specific application of the algorithm.

A data-flow analysis can be run using a generalized *iterative algorithm* which starts by initializing data-flow values for OUT[ENTRY] as either $\top$ or $\bot$ (known as the *boundary condition*), sets the OUT for all other blocks as $\top$, and then iterates over each basic block (in any order). It sets the IN value as the result of $\wedge$ over all of the OUT values of the basic block's predecessors, and then iterates through each instruction in the basic block, propagating the data-flow value and passing it through a *transfer function* f specific to the data-flow analysis being done, ending up with setting the OUT value for the basic block. After iterating over each basic block, the algorithm then repeats if any OUT value was changed.

The data-flow analysis can also be done in the backwards direction, which is the same as the previous algorithm but with IN and OUT reversed as with ENTRY and EXIT, and the meet operation is done over the successors of B instead of the predecessors. The backwards variant of the algorithm is useful for analyses that determine data-flow values based on facts about code that is run in the future.

I implemented this algorithm as a general data-flow analysis framework for LLVM. The data-flow values are represented as bit-vectors, with each variable being assigned to an index in the vector, and the value at that index (1 or 0) representing if that variable was present in the set. The framework as a whole is represented as an abstract base class. The inputs to the algorithm (transfer function, meet operation, top and bottom values, boundry condition, and direction) are kept as pure virtual methods to be implemented by subclasses that use the framework for a specific analysis.

Reaching Definitions and Liveness Analysis

One data-flow analysis built on the algorithm previously described is *reaching definitions.* This analysis computes all possible points a variable was last defined

```
class DFAF {
public:
  virtual DFV tf(BasicBlock *b, const DFV& x) = 0;
  virtual DFV tf(Instruction *i, const DFV& x) = 0;
  virtual DFV meet(const DFV& a, const DFV& b) = 0;
  virtual DFV top() = 0;
  virtual DFV bot() = 0;
  virtual DFV boundryCondition() = 0;
  virtual Direction direction() = 0;
  DFAFResult run(Function& f);
};
```

Figure 3: The interface of the `DFAF` class, the superclass of all data-flow analyses for LLVM. `DFV` is the data-flow value (implemented as `std::vector<bool>`), `DFAFResult` is a struct combining the IN and OUT values for each basic block and instruction, and `tf` is the transfer function. The `run` function implements the general algorithm described above with both forwards and backwards directions and works unmodified for all specific analyses.

before it is used. A variable is defined during any statement that assigns or could assign to that value. This is a forwards analysis with $\cup$ as the meet operation and $\emptyset$ as the boundary condition. The transfer function is $f(x) = gen_s \cup (x - kill_s)$, where x is the incoming data-flow value, gen_s is the set of definitions generated (usually a singleton set) by statement s, and $kill_s$ is the set of definitions that "kill" or assign to the left-hand side of s in the rest of the program. This transfer function (and all transfer functions that work at the instruction level) can be turned into a transfer function that works on a basic block simply by composing it over all instructions in the block, i.e. $f_B = f_{s_n} \circ f_{s_{n-1}} \circ \ldots \circ f_{s_2} \circ f_{s_1}$. This description of reaching definitions is for a non-SSA IR. In LLVM, reaching definitions is much simpler because a variable can only have one definition and cannot be "killed". Reaching definitions at a program point then becomes just a list of variables have been defined in all paths leading up to that point. Although reaching definitions is just an analysis pass, it is useful for the loop invariant code motion optimization pass, described later on.

Another data-flow analysis is *liveness analysis*, which seeks to discover the set of all "live" variables at each point in a program. A variable is *live* if it may be used by some other instruction in the future. Liveness is a backwards analysis because it relies on information about code yet to be run to determine if a variable is live, has $\cup$ as the meet operation and $\emptyset$ as the boundary condition. The transfer function for liveness is $f(x) = use_s \cup (x - def_s)$, where use_s is the set of variables used in instruction s, and def_s is the set of variables defined in instruction s (usually one). This transfer function works unmodified in my LLVM implementation, however there is a different caveat due to the unusual nature of the phi instruction. When computing the OUT value for a block that

```
define i32 @faint(i32 %x) nounwind {
bb:
  %tmp = add nsw i32 %x, 1
  %tmp1 = add nsw i32 %tmp, 2
  %tmp2 = sub nsw i32 %tmp1, 3
  %tmp3 = add nsw i32 %tmp1, %tmp2
  %tmp4 = mul nsw i32 %tmp3, %x
  ret i32 %x
}
```

Figure 4: LLVM code with multiple dead instructions. A naive dead code elimination pass will only remove `%tmp4`, when all instructions except `ret` can be removed.

transfers to another block with multiple predecessors (i.e. there are multiple paths to the first block's successor), not every one of the variables in the successor's IN would be considered live in the first block's OUT because some of them might be defined in a path that doesn't go through the first block. This is because the successor block has one or more phi instructions, and the operands of these instructions all count as uses. To mend this, the analysis iterated through each value in IN, and if it is used by a phi instruction and the phi instruction chooses that value when not transferring from the current block, then remove that value. Like reaching definitions, this analysis pass performs no optimizations, but provides useful information for dead code elimination, which would simply remove each instruction that is not live right after it is defined.

Faint Variable Analysis and Dead Code Elimination

Liveness analysis is a useful analysis for optimization as it enables *dead code elimination* in the scope of instructions, where instructions that assign to a dead value can be safely removed. However, dead code elimination through liveness checking is not the most effective, because removing a dead instruction also removes uses by other instructions, which can cascade into causing other instructions to be considered dead, requiring dead code elimination to be repeated to catch all dead instructions.

A solution to this problem is a new analysis called *faint variable analysis* that offers more aggressive detection of dead code. Faint variable analysis considers a variable *faint* if it is either dead or its only uses are in other instructions deemed faint. Like liveness analysis, it is a backwards data-flow problem, but unlike liveness analysis, the boundary condition is U and the meet operation is $\cap$ because a variable needs to be considered faint in all of a block's successors to be considered faint in that block. Also the data-flow value (as the set of faint variables) is not program-point specific; at the end of the analysis all

```
define i32 @licm(i32 %x) nounwind uwtable ssp {
bb:
  br label %bb1
bb1:                                                  ; preds = %bb7, %bb
  %a.0 = phi i32 [ 0, %bb ], [ %tmp6, %bb7 ]
  %i.0 = phi i32 [ 0, %bb ], [ %tmp8, %bb7 ]
  %tmp = icmp slt i32 %i.0, %x
  br i1 %tmp, label %bb2, label %bb9
bb2:                                                  ; preds = %bb1
  %tmp3 = mul nsw i32 %x, 3
  %tmp4 = sub nsw i32 %tmp3, 2
  %tmp5 = add nsw i32 %tmp4, %i.0
  %tmp6 = add nsw i32 %a.0, %tmp5
  br label %bb7
bb7:                                                  ; preds = %bb2
  %tmp8 = add nsw i32 %i.0, 1   br label %bb1
bb9:                                                  ; preds = %bb1
  ret i32 %a.0
}
```

Figure 5: Loop invariant code motion example. The `tmp3` and `tmp4` instructions are both loop invariant and can be moved out of `bb2` (the body of the loop) without changing semantics.

faint variables are in the IN value of the entry block by not filtering path-specific variables. By using faint variable analysis instead of liveness for dead code elimination, it will only take one pass instead of many to ensure all dead instructions are removed.

Loop Invariant Code Motion

The last optimization covered in this independent study was *loop invariant code motion*. This optimization finds instructions inside loops that compute the same thing with each iteration because they are not dependent on any loop-local variables, and moves them outside the loop to minimize redundant computation. Loops in a control flow graph are not simply cycles, they must have a single entry point (the *header* block) that dominates all other blocks in the loop, and one or more *back edges* that transfer to the header from a node dominated by the header. LLVM includes a `LoopInfo` library that aids with finding loops within a function.

The analysis searches for instructions that are *loop invariant*, which are instructions that only use values that are either defined outside the loop or are also loop invariant. Specifically, an instruction $a = b + c$ is loop invariant if the definitions of b and c are outside the loop or come from other loop invariant instructions (also, constant operands are always loop invariant). These instructions are then subject to *code motion*, which simply moves the instructions to a different basic block, namely a *preheader* block that is inserted before the loop's header block. The end result is that instructions that don't have to be inside a loop are moved outside the loop, reducing the amount of computational work that needs to be done with each iteration by avoiding redundant computations.

Conclusion

This study allowed me to start exploring the topic of compiler optimization using LLVM. Optimization is an increasingly important part of any compiler as it allows programs to achieve as much performance as it can get without any effort by the programmer. In many domains from systems software to computer graphics, it is important that programs run as efficiently as they can. Even on the large-growing class of battery-powered mobile computers and devices, performance is very important because CPU and memory usage has a direct correlation with battery life. LLVM serves as a useful tool in this field as its well-designed architecture is simple yet powerful enough to take advantage of many of the compiler optimization techniques pioneered in research. For this study, I expected to spend 4 to 8 hours per week, and ended up spending closer to 8 and going a little over for the more challenging project like creating the initial data-flow analysis framework. Though I have learned a lot in this quarter, I have only covered some of the basics in this area. Other important topics include pointer/alias analysis, register allocation, memory usage, and parallelization, and these can serve as the focus of future study. I also came across an existing data-flow analysis framework written in Haskell called Hoopl[4], which is also fair game to learn about and experiment with to see another way of doing data-flow analysis and optimizations.

References

[1] The LLVM Compiler Infrastructure Project. http://www.llvm.org/.

[2] CS745: Optimizing Compilers, Spring 2012. http://www.cs.cmu.edu/afs/cs/academic/class/15745-s12/www/.

[3] Aho, Alfred V., et al. Compilers: principles, techniques, and tools. Vol. 1009. Pearson/Addison Wesley, 2007.

[4] Ramsey, Norman, Joao Dias, and S. Peyton Jones. "Hoopl: Dataflow optimization made simple." ACM SIGPLAN Haskell Symposium. 2010.