Exploring Compiler Optimization Techniques

A. D. Blom

Abstract

In this era of global digitalization, the need for performant software is larger than ever. For the production of fast software not only fast hardware and solidly written code are needed, but also a well-optimizing compiler. This paper explores some common optimization techniques based on an SSA intermediate language.

1 Introduction

When a compiler translates code from a programming language to assembly languages, it may apply a number of transformations to the source code, most often to make programs faster (but also to enhance debugging capabilities for example). Doing so is called code optimization.

Code optimization is most often done by first translating the source code (as read by a *lexer* and interpreted by a *parser*) into an *intermediate form*, which is subsequently converted into assembly.[1] Such an intermediate form must be designed in such a way that it can be analyzed and optimized easily. Such a intermediate form commonly consists of a list of instructions (using so-called *three-way address code*) grouped into *basic blocks*. Each basic block has one starting point and one end-point. The endpoint is always a *terminal instruction* which causes control flow to leave the block (`jmp`, `split`, `ret` and `leave` in the intermediate form used here).

The following C source code:

```
int sub(int a, int b)
{
        return a - b;
}
```

May be converted (by the *front-end*) into the following intermediate form (this intermediate form is described in section 2):

```
global int (int, int)* @sub(int p0, int p1) {
L0:     int* t1 = alloca(int);
        int* t2 = alloca(int);
        store(t1, p0);
        store(t2, p1);
        int t3 = load(t1);
        int t4 = load(t2);
        int t5 = sub(t3, t4);
        ret(t5);
L1:     leave();
}
```

Which is subsequently optimized (by the *middle-end*):

```
global int (int, int)* @sub(int p0, int p1) {
L0:     int t1 = sub(p0, p1);
        ret(t1);
}
```

And converted into assembly (by the *back-end*):

```
        .globl  sub
sub:
        movl    %edi, %eax
        subl    %esi, %eax
        ret
```

It is then that the job of the compiler is done, and the assembler takes over, and converts the assembly into numerical codes (*"ones and zeroes"*):

```
89 66 66 f8 f0 29 00 c3
```

One may ask oneself how the intermediate code is optimized from the first into the significantly faster second format, and how this is implemented concretelely in software. The *acc* project was written to find out, and this article describes some of the optimization techniques used by acc (and a few more).

2 Intermediate representation used

The intermediate representation (IR) used here is based on the *static single assignment* (SSA) principle: every temporary variable may be assigned to only once. It is partially based on the LLVM IR as described by LLVM [2]. There are a few differences: the IR used here uses a typesystem analogous to C's, and uses a C-like syntax for textual representation as well.

The custom IR used also lacks implicit blocks, so blocks are declared explicitly using labels, and follow temporary value numbering.

Concretlely, the following LLVM snippet:

```
%1 = add i32 1, i32 2
%2 = icmp eq i32 %1, i32 3
br i1 %2, label %3, label %0
ret i32 0
```

Is equivalent to the following custom-IR:

```
L0:     int t1 = add((int)1, (int)2);
        _Bool t2 = cmp eq(t1, (int)3);
        split(t2, L3, L0);
L3:     ret((int)0);
```

Both languages feature an `undef` constant, with the ability to take on any value. The custom IR lacks a `null` constant, it uses `0` instead.

A working C front-end is expected to be present, and snippets here will be either in custom IR or C.

2.1 Common instructions

A lot of the instructions are taken from LLVM. The non self-explanatory ones behave as described below.

2.1.1 alloca

Allocates memory of the specified type, and returns a pointer to that memory. The memory is deallocated when the function returns.

2.1.2 split

A conditional jump, `split(a, b, c);` jumps to `b` if `a`, and `c` if not.

2.1.3 leave

In a function returning `T`, `leave();` is equivalent to `ret((T)undef);`. It returns from a function but returns no useful value.

3 SSA conversion optimization

3.1 Problem

Imperative languages usually allow variables and memory to be rewritten. SSA inherently, does not allow temporary variables to change their value over time. SSA supports several ways to support this model. One is using the `alloca` instruction to allocate memory and use the `load`/`store` system, another, is cleverly using SSA and its ϕ nodes. The first translation uses rewritable memory and is therefore not a strict SSA representation of control flow.[1]

Consider the following imperative code:

```
int i;
if (condition)
        i = 0;
else
        i = 1;
return i;
```

A naive translation would be:

```
L0:     int* t1 = alloca(int);
        split(cond, L2, L3);
L2:     store(t1, (int)0);
        jmp(L4);
L3:     store(t1, (int)1);
        jmp(L4);
L4:     int t5 = load(t1);
        ret(t5);
```

It involves two memory accesses and a memory allocation. Pointers are involved so it's hard to optimize any further.

It also complicates register allocation a great deal. The allocator now not only needs to keep track of where its temporaries are, but also the registers used by the `alloca` instructions. Furthermore it's complicated by requiring a new lifetime analysis method, instead of the one already provided for temporaries, since most `alloca` memory needn't be alive for the entirety of the surrounding function.

[1]The intermediate form used is still SSA compliant, however, even though it allows rewriting of *memory*. The temporary values (t1 etc.) are still single-assignment. Allowing rewritable memory is a necessary evil for imperative languages.

For further optimization it's far more convenient to turn such a complex system with `alloca`, `load` and `store` into an pure system where each variable is truly written to once.

SSA features a mechanism that allows selecting a value based on the previously run block. This system is a ϕ node system, where a ϕ node is an instruction taking a map of blocks and expressions, selecting the appropriate expression based on the predecessing block.

```
L0:     split(cond, L1, L2);
L1:     jmp(L3);
L2:     jmp(L3);
L3:     int t4 = phi(   L1, (int)0,
                        L2, (int)1      );
        ret(t4);
```

Sometimes for imperative languages it is impossible to use the second system, for example in the case where actual memory is required:

```
        int i = 0;
        foo(&i);
        return i;
```

Can't use a ϕ node system, since it needs `i` to actually exist in memory. It is therefore hard for a front-end to decide which system to use, and many[2] default to using the first system all of the time, relying on the middle-end to optimize it into an SSA system. If an `alloca` variable can convert its `load`/`store` system it shall be considered *SSA-capable*.

3.2 Implementation[3]

Given a simple, one-block SSA graph:

```
L0:     int* t1 = alloca(int);
        store(t1, (int)0);
        int t2 = load(t1);
        int t3 = add(t1, (int)10);
        store(t1, t3);
        int t4 = load(t1);
        ret(t4);
```

Can `t1` be considered SSA-capable? It's been established that an `alloca` system is not SSA-capable if the memory is actually required to exist. This

[2]At least clang does so: *echo "void foo() { int a; }" | clang -x c - -S -emit-llvm -o /dev/stdout*

[3]This is the o_phiable() optimization pass in opt.c in acc

means (naively) that a system is not SSA-capable if the `alloca` instruction is used outside its own `load`/`store` instructions: that is if it is ever used in an instruction, except as the first operand of a `store` or `load`.

`t1` meets the phiability requirements. `load` instructions need to be replaced by its last `store`. This means that the `load` in line 3 (`t2`) needs to be replaced by its last stored value (line 2). `t4` similarly needs to be replaced by `t3`:

```
L0:     int t1 = add((int)0, (int)10);
        ret(t1);
```

This constitutes an enormous code shrinkage, and will speed up the code immensely.

Finding the last store is trivial for these one-block examples, it is more involved when considering a piece of code where the last store is in one of a `load`'s block predecessors. Consider this:

```
L0:     int* t1 = alloca(int);
        split(cond, L2, L3);
L2:     store(t1, (int)0);
        jmp(L4);
L3:     store(t1, (int)1);
        jmp(L4);
L4:     int t5 = load(t1);
        ret(t5);
```

For the load in line 7 for example, finding the last store is non-trivial, it has in fact got multiple last `store` instructions, one in `L2` and one in `L3`. It is now actually required to implement a ϕ node. It selects the value from `L2` if that was its predecessor, and the store from `L3` if that was its predecessor using a ϕ node:

```
L0:     split(cond, L1, L2);
L1:     jmp(L3);
L2:     jmp(L3);
L3:     int t4 = phi(L1, (int)0, L2, (int)1);
        ret(t4);
```

It is also possible for an `alloca` to be loaded without any previous `store`. In that case, the value of the `load` is undefined, and it is tempting to use the `undef` constant. It is important, however, that the result of the load is guaranteed to remain constant. That isn't the case if all instances are replaced by individual `undef` constants. Consider, for instance, the following example:

```
L0:     int* t1 = alloca(int);
        int t2 = load(t1);
        int t3 = load(t1);
        _Bool t4 = cmp eq(t2, t3);
        ...
```

The value of `t4` is well-defined, because the value of `t1` is guaranteed not to alter spontaneously. If the following translation would be used:

```
L0:     _Bool t1 = cmp eq((int)undef, (int)undef);
        ...
```

The result of the comparison is undefined as well.

It is therefore required to introduce a `undef` instruction. The code would therefore be optimized into:

```
L0:     int t1 = undef(int);
        _Bool t4 = cmp eq(t1, t1);
        ...
```

4 Constant folding

4.1 Problem

When a programmer writes something along these lines:

```
int i = 10 - 3 * 2;
```

The compiler can be expected to see that i should be initialised to four, rather than having it emit instructions for each mathematical operation. Moreover, if a programmer types:

```
int a = 10;
int b = a * 2;
```

The compiler can also be expected to simplify the initialisation of b into an initialisation to twenty. Although perhaps trivially optimised manually, these types of trivial constant expressions occur not so much in manually written code, but quite often in macro expansions.

Therefore the compiler may not expect all constants to be simplified as much as possible. Instead, the compiler evaluates these constants in a process known as constant folding, and subsequently propegates these constants further, filling them in for SSA variables along the way in a process known as constant propagation.

4.2 Implementation[4]

In order to perform any useful consant folding, the compiler needs to fill in constants for variables where possible, so code of the form:

```
int a = 10;
int b = a * a;
return b - a;
```

Becomes:

```
int b = 10 * 10;
return b - 10;
```

Once the value of `b` is determined, it should then also be filled in, to fold further. Since the value of `b` is 100, it can be used to fill in the return expression:

```
return 100 - 10;
```

This value can then be folded once more to yield the value 90:

```
return 90;
```

This algorithm might look quite involved, but its simplicity is actually staggering. It simply depends on phiability optimisation. Phiability optimisation fills in constants for variables automatically. Consider the first fragment's IR before phiability optimisation:

```
L0:     int* t1 = alloca(int);
        int* t2 = alloca(int);
        store((int)10, t1);
        int t3 = load(t1);
        int t4 = load(t1);
        int t5 = mul(t3, t4);
        store(t5, t2);
        int t6 = load(t2);
        int t7 = load(t1);
        int t8 = sub(t6, t7);
        ret(t8);
```

The variables still exist in their crude memory form. However, their values are propagated automatically once phiability optimisation occurs:

```
L0:     int t1 = mul((int)10, (int)10);
        int t2 = sub(t1, (int)10);
        ret(t2);
```

[4]This is the o_cfld() optimization pass in opt.c in acc

The constants can now be propagated with a pass that scans for computable instructions (arithmetic instructions of which both operands are constants) and computes their values, filling them in for all future occurrences:

```
L0:     ret((int)90);
```

4.3 Considerations

4.3.1 Platform incompatibilities

There is a way compiler-based constant folding might stand in the way of the programmer. Mostly the compiler can do this when folding away instructions operating on floating point operands, because different targets may compute floating point operations differently. Therefore cross-compilation becomes an issue; if a floating point instruction for target Y normally yielding V_y, it yields V_x when folded away by target X, causing different semantics before and after optimisation.[3]

A solution to this problem is to implement a floating point virtual machine for several targets, that use non IEEE floating point. Targets using IEEE floating point can use C99's internal way of computing IEEE floating point operations. Since implementing such a system is non-trivial, code duplication needs to be avoided. If any other optimisation would need to be able to calculate an operation on two constants, it should run the same code. Therefore, the actual folding computations are performed outside of the optimiser, by a separate folding system.

5 Constant split removal

5.1 Problem

After constant folding, some `split` instructions may branch on a constant condition:

```
La:     ...
        split((_Bool)1, Lb, Lc);
Lb:     ret((int)0);
Lc:     ret((int)1);
```

Could be converted easily into:

```
La:     ...
        jmp(Lb);
Lb:     ret((int)0);
```

```
Lc:        ret((int)1);
```

This has only minor implications for further flow, except that it removes a predecessor from block `Lc`. The only way that that affects SSA validity is that a block-expression pair may need to be removed from ϕ nodes in `Lc`.

Removing this predecessor may also have implications for further block inlining; if a block has only one predecessor and the predecessor has only one successor, the block could be merged with its predecessor.

5.2 Implementation[5]

The implementation of this optimization simply needs to check whether the first parameter of a `split` instruction is constant, and convert it into a `jmp` accordingly. It also needs to check for the presence of ϕ nodes in the block not covered by the `jmp` instruction, and remove them accordingly:

```
La:        ...
           split((_Bool)1, Lb, Lc);
Lb:        ...
Lc:        int tA = phi(La, (int)10, ...);
           ...
```

Needs to get rid of the `La` items from the `tA` ϕ node too:

```
La:        ...
           jmp(Lb);
Lb:        ...
Lc:        int tA = phi(...);
           ...
```

6 Block inlining

6.1 Problem and implementation

When a block has only one predecessor and its single predecessor also has one successor, its instructions can be inlined into the block it succeeds:

```
L0:        ...
           jmp(L1);
L1:        int t2 = add((int)0, (int)1);
           jmp(L3);
L3:        ...
```

[5]This is the o_uncsplit() pass in opt.c acc

Becomes (assuming L1 has no other predecessors):

```
L0:        ...
           int t1 = add((int)0, (int)1);
           jmp(L2);
L2:        ...
```

That way the amount of jumps and blocks is reduced without duplicating instructions.

It's a very trivial optimization but occurs quite a lot, especially considering the front-end may generate redundant blocks all the time. Consider an infinite `for` loop:

```
           for (; cond;)
                     ;
```

The front-end puts the initialization clause in the block it's currently writing to, but generates a new block for the condition, then generates (without knowledge of the loop body) a block for the final loop clause (this block is needed to jump to when compiling a `continue` statement). It inserts this block after it has generated the body block.

This would therefore be a possible translation:

```
           /* enter the loop */
L0:        jmp(L1);
           /* continue or break from the loop */
L1:        split(cond, L2, L4);
           /* go to the final clause */
L2:        jmp(L3);
           /* no final clause, jump to loop start */
L3:        jmp(L1);
L4:        ...
```

It can be noticed quite easily that `L3` only has one predecessor (`L2`), and its predecessor only one successor. It can therefore be merged with `L2`:

```
           /* enter the loop */
L0:        jmp(L1);
           /* continue or break from the loop */
L1:        split(cond, L2, L3);
           /* go to the final clause */
           /* no final clause, jump to loop start */
L2:        jmp(L1);
L3:        ...
```

This turns out to be quite an interesting case, however; it can also be noticed that this example might be optimized further, so the `split` in `L1` jumps to itself immediately. This is because `L2` is empty besides the `jmp` instruction: the condition for empty loops to be inlined is therefore more relaxed.

In fact, all empty blocks (except `L0` and empty blocks that are their own predecessor) can be inlined:

```
L0:     jmp(L1);
L1:     split(cond, L1, L2);
L2:     ...
```

References

[1] Aho et al. *Compilers: Principles, Techniques And Tools.* (1988)

[2] *LLVM Language Reference Manual.* (2014) Consulted on 2014/11/11, http://llvm.org/docs/LangRef.html

[3] *Constant folding and cross compilation.* (s.d.) Consulted on 2014/11/11, http://en.wikipedia.org/wiki/Constant_folding#Constant_folding_and_cross_compilation

A Implementation Details for acc

A.1 Introduction

acc (the antonijn/Antonie C Compiler) is a software project with the intent of one-day being self-hosting (able to compile itself). The only external library it depends on is the C99 standard library, and it's written in portable standard C99.

acc implements many of the optimizations mentioned in the main paper, and could serve as a reference implementation for them. It is however, much more than that, of course, since it has to provide not only an optimizer, but back-ends and a C front-end as well. Only the subsystems relevant to compiler optimization are described here in detail. The most relevant subsystem is the so-called *intermediate* subsystem (shortened to `itm` in code), implementing functions and data structures for defining and manipulating an intermediate form SSA tree. It also implements functions for writing such a tree to a file in text form.

A.2 Object oriented programming

Although the C language doesn't natively feature object oriented syntax, it doesn't exclude the possibility of writing clean object oriented code. For instance, the following hierarchy:

INSERT DIAGRAM

Could well be implemented in C as follows:

```
struct A {
        /* pointer to a B or C object */
        void *extended;

        void (*free)(struct A *self);
};

struct B {
        struct A base;

        int field;
};

struct C {
        struct B base;
```

```
        float field;
};
```

This style will be found a lot in the acc source code, although sometimes missing the `extended` field in a base class (in which case the addresses of both types are presumed compatible).

A.3 The AST and its elements

The `itm` abstract syntax tree (AST) is not very complex. It mostly uses linked lists (the `struct list *`, for instance, is a linked list containing only `void *` instances) for chaining instructions and blocks together.

The following intermediate code is internally represented through a few different data structures:

```
global int (int, int)* @gcd(int p0, int p1) {
L0:     jmp(L1);
L1:     int t2 = phi(L0, p0, L5, t7);
        int t3 = phi(L0, p1, L8, t9);
        _Bool t4 = cmp neq(t2, t3);
        split(t4, L6, L10);
L5:     _Bool t6 = cmp gt(t2, t3);
        int t7 = sub(t2, t3);
        split(t6, L1, L8);
L8:     int t9 = sub(t3, t2);
        jmp(L1);
L10:    ret(t2);
}
```

Global variables (as of yet unimplemented) and functions are represented through *container* structures (`struct itm_container *`). They contain an entry block, and are expressions themselves (as required to be called):

```
struct itm_container {
        /* expression base */
        struct itm_expr base;

        /* container identifier */
        char *id;
        /* entry block */
        struct itm_block *block;
};
```

Blocks are represented through `struct itm_block *` structures. They are expressions (as required to be a parameter to `jmp()` or `split()`), contain a pointer to the first instruction, the last instruction (terminal instruction), a pointer to the block that's lexically next (`L1` for `L0` in the first example), a pointer to the block that's lexically previous (`L0` for `L1` in the first example, `NULL` for `L0`), and two lists of blocks that are sementically next and previous (`L1` is `L10`'s semantic predecessor, for example).

```
struct itm_block {
        /* expression base */
        struct itm_expr base;

        /* the container the block's contained by */
        struct itm_container *container;
        /* first and last instructions */
        struct itm_instr *first, *last;
        /* blocks lexically next and previous */
        struct itm_block *lexnext, *lexprev;
        /* predecessor and successor lists */
        struct list *next, *prev;
};
```

Instructions are represented as `struct itm_instr` pointers. They are themselves a linked list: they contains pointers to the previous and next instructions. They also link to their parent block, and have a list of their operands. They contains a field of a strange type (`itm_instr_id_t`) which is an instruction identifier constant for each instruction type.

For instance, an instruction of the type `add` has a constructor called `itm_add()`. Its identifier can be obtained passing that function to the `ITM_ID()` macro: `ITM_ID(itm_add)`.

The structure looks roughly like this:

```
struct itm_instr {
        struct itm_expr base;

        itm_instr_id_t id;
        struct itm_block *block;
        struct list *operands;
        struct itm_instr *prev, *next;
};
```

Then there has to be a way to store literals (both floating point and integral) and `undef` constants. These structures are trivial:

```
struct itm_literal {
        /* expression base */
        struct itm_expr base;

        /* value, sharing memory */
        union {
                long long i;
                double f;
        } value;
};

struct itm_undef {
        /* expression base */
        struct itm_expr base;
};
```