Exploring SSA 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

2 Intermediate representation used

The intermediate representation (IR) used here is an SSA implentation, partially based on the LLVM IR as described by LLVM [1]. 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.

Concretely, 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.

3 Phiability 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 ϕ (phi) nodes.

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.

A system that would allow turning this `alloca` system into a more SSA-appropriate system would get rid of all these cases where alloca would form an exception, by morphing an advanced `load`/`store` mechanism into a mechanism using mostly temporaries for storage.

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[1] default to using the first system all of the time, relying on the middle-end to optimize it into a ϕ node system. If an `alloca` variable can convert its `load`/`store` system it shall be considered *phiable*.

3.2 Implementation[2]

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 phiable? It's been established that an `alloca` system is not phiable if the memory is actually required to exist. This means (naively) that a system is not phiable 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

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

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

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[3]

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

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

```
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);
```

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, as described by Wikipedia [2]. 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.

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[4]

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:     ...
```

Becomes (assuming L1 has no other predecessors):

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

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

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

[2] *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