New SSA Backend for the Go Compiler

Keith Randall
2/10/2015

Introduction

So the compiler is soon to be translated into Go. … then what?

There will be a lot of simple cleanups that can be done once the compiler is in Go, and of course we'll do those. I suspect most of these cleanups will happen in the 1.5 time frame.

For the 1.6 release, I'd like to propose doing some more radical surgery on the compiler. I'd like to convert from the current syntax-tree-based IR to a more modern SSA-based IR. With an SSA IR we can implement a lot of optimizations that are difficult to do in the current compiler.

The plan is to convert the compiler in stages, back end first. By converting in stages we have smaller chu
nks to get working, test, and deploy.

Stage 1

Convert the compiler backend to SSA. We'll leave all the frontend and most of the Go-specific processing as is. We branch from the current IR after the "walk" phase, at which point the IR is fairly low-level yet machine-independent (mostly). At that point we convert to an SSA representation of the function. We then use the SSA form for lowering to machine-dependent operations, instruction scheduling, register allocation, and code generation.

I think this is the right place to split, at least for stage 1, because most of the easy improvements happen after this point, and most of the hard Go-specific stuff happens before this point.

Improvements possible during this stage:

- better common subexpression elimination
- better dead code elimination
- better register allocation
- better stack frame allocation

I believe that with all of these areas we can do substantially better than the compiler does today.

This stage must also contain reimplementations of:

- stack pointer map generation
- stack DWARF info generation

because information for these data structures is not complete until the stack frame layout is complete.

What's not part of this stage:

- Escape analysis
- Inlining
- Lowering Go syntax to runtime calls (chan ops, select, defer, …)

A common question that comes up is "why not just convert to the IR of {llvm,gcc,...} and go from there?" I think there are 3 major reasons:

1. Compiler speed. I intend our SSA IR, and the optimizer passes implemented on top of that IR, to be fast by design. We'll aim for linear-time algorithms wherever possible. The backends of other possible compilers weren't really designed with fast compile speed in mind.
2. Runtime information. The runtime needs accurate maps of stack frames for both GC and stack copying, something that is not available from standard backends today.
3. An additional dependence. Do we want a {llvm,gcc,...} backend as a prerequisite for building Go? (Note that this would only be an issue for those developing Go, not those writing Go programs.)

I'm happy to be convinced otherwise - maybe we should throw our effort into getting frame maps out of {llvm,gcc,...} and figuring out how to configure the backend (e.g. which optimizations to turn on) to make it compile quickly.

Stage 2

We'll move more of the middle part of the compiler, as needed. See the bullet points for "not part of stage 1" above.

Stage 3

Convert the frontend? Use go/types or go/ssa as the frontend and plug it into the new ssa backend.

I'm not entirely convinced that stages 2 & 3 are necessary, maybe we stop at stage 1. It all depends on what optimizations we'd like to do that can't be done because the IR is in the wrong form. Proceeding with stages 2 and 3 might gain some efficiency in the compiler itself because then we don't have to generate the old IR at all. I suspect that effect will be small, however.

Appendix 1: current prototype

I've hacked up a prototype that is a start down the direction of stage 1. First is a patch to the current compiler to dump a simple intermediate representation to a file just after the walk pass. It uses a simple S-expression language to serialize the program. (In the real thing we won't need this step - I use it only because the current compiler is in C and the prototype I'm writing is in Go. Once the compiler is in Go this will be unnecessary). The prototype reads this simple IR into memory, generates the SSA IR from it, does a couple of optimization passes, lowers to machine code, regallocs, and generates assembly output. It handles only a small subset of opcodes. It is complete enough to compile some toy programs up to the regalloc phase.

The IR is pretty simple. Here are the definitions of `Value`, `Block`, and `Function`.

```
// A Value represents a value in the SSA representation of the program.
// The id and typ fields must not be modified.  The remainder may be modified
// if they preserve the value of the Value (e.g. changing a (mul 2 x) to
// an (add x x)).
type Value struct {
	// A unique identifier for the value.  For performance we allocate these IDs
	// densely starting at 0.  There is no guarantee that there won't be occasional
	// holes, though.
	id int

	// The type of this value.  Normally this will be a Go type, but there
	// are a few other pseudo-types, see type.go.
	typ Type

	// The operation that computes this value.  See opcode.go.
	op Opcode

	// Auxiliary info for this value.  The type of this information depends
	// on the opcode (& type).
	aux interface{}
	// A value that determines how the block is exited.  Its value depends
	// the kind of the block.
	control *Value

	// The set of Values contained in this block.  The order in this list

	// Arguments of this value
	args []*Value

	// Containing basic block
	block *Block

	// Storage for the first two args
	argstorage [2]*Value
}
```

There are lots of places in the compiler where we would want a map[Value]X for some X. We use the ids to implement such a map efficiently by instead using []X indexed by the value id. This is in preference to the current compiler where we pile all possible ancillary information fields into a Node, only a few of which are used by any kind of Node and any phase of compilation.

```
// Block represents a basic block in the control flow graph of a function.
type Block struct {
	// A unique identifier for the block.  The system will attempt to allocate
	// these IDs densely, but no guarantees.
	id int

	// The kind of block this is.
	kind BlockKind

	// Subsequent blocks, if any.  The # and layout depends on the block kind.
	successors []*Block

	// Inverse of successors.
	// The order doesn't matter, BUT phi arguments correspond to this order.
	// So if you reorder the predecessor blocks you must also reorder
	// the arguments of all the phi values in this block.
	predecessors []*Block

	// the immediate dominator of this block.  For the entry block this
	// points to itself.
	idom *Block

	// is irrelevant.
	values []*Value

	// containing function
	function *Function
}
```

Blocks have ids just like values do (in a separate namespace).

```
type Function struct {
	name   string
	typ    Type     // type signature of the function
	blocks []*Block // all basic blocks
	entry  *Block   // the entry basic block
	bid    idAlloc  // block ID allocator
	vid    idAlloc  // value ID allocator
}
```

The id allocators attempt to allocate ids densely for a given function. Passes like the dead code eliminator will recycle ids back to these allocators for reuse.

I don't have any feel for the speed of compilation at the moment.

Appendix 2: sampling of currently suboptimal code

I just paged through the `go` binary and found a few instruction sequences where it looks easy to do better than the current compiler. With an SSA IR, most of these would be easily fixable.

```
lea    0x28(%rsp),%rbx
movq   $0x0,(%rbx)
movq   $0x0,0x8(%rbx)
… don't use %rbx again

mov    %rsi,(%rsp)
cmpq   $0x0,(%rsp)

lea    0x6e95ad(%rip),%rsi
mov    %rsi,0x8(%rsp)
addq   $0x88,0x8(%rsp)
… don't use %rsi again

mov    %rax,0x80(%rsp)
mov    %rax,0x10(%rsp)
callq  47e090
… never use 0x80(%rsp) again

lea    (%rbx,%rcx,8),%rbx
mov    (%rbx),%rax
… never use %rbx

imul   $0x10,%rsi,%rsi

lea    0x88(%rsp),%rbx
movq   $0x0,(%rbx)
movq   $0x0,0x8(%rbx)
lea    0x88(%rsp),%rbx
cmp    $0x0,%rbx
je     42d6e2
… use %rbx

mov    $0x1,%rdx
mov    $0x1,%rcx
mov    %rdx,0x280(%rsp)
```

```
        mov    %rcx,0x288(%rsp)
        … don’t use %rdx,%rcx again

        movzbl 0x0(%rbp),%ebx
        mov    %rbx,%rsi
        …
        movsbq %sil,%rbp
        cmp    $0x43,%rbp

        and    $0x80,%rbx
        cmp    $0x0,%bl
        je     47bd24
```

Total guesstimate, I think it would be fairly easy to make the generated programs 20% smaller and 10% faster. Just as an experiment I cleaned up the runtime function mapaccess1_fast64 by hand in assembly and made it 29% smaller and with a speed improvement too small to measure.