How Clang Compiles a Function

I've been planning on writing a post about how LLVM optimizes a function, but it seems necessary to first write about how Clang translates C or C++ into LLVM. This is going to be fairly high-level:

- rather than looking at Clang's internals, I'm going to focus on how Clang's output relates to its input
- we're not going to look at any non-trivial C++ features

We'll use this small function that I borrowed from these excellent lecture notes on loop optimizations:

```
bool is_sorted(int *a, int n) {
  for (int i = 0; i < n - 1; i++)
    if (a[i] > a[i + 1])
      return false;
  return true;
}
```

Since Clang doesn't do any optimization, and since LLVM IR was initially designed as a target for C and C++, the translation is going to be relatively easy. I'll use Clang 6.0.1 (or something near it, since it hasn't quite been released yet) on x86-64.

The command line is:

```
clang++ is_sorted.cpp -O0 -S -emit-llvm
```

In other words: compile the file is_sorted.cpp as C++ and then tell the rest of the LLVM toolchain: don't optimize, emit assembly, but no actually emit textual LLVM IR instead. Textual IR is bulky and not particularly fast to print or parse; the binary "bitcode" format is always preferred when a human isn't in the loop. The full IR file is here, we'll be looking at it in parts.

Starting at the top of the file we have:

```
; ModuleID = 'is_sorted.cpp'
source_filename = "is_sorted.cpp"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"
```

Any text between a semicolon and the end of a line is a comment, so the first line does nothing, but if you care, an LLVM "module" is basically a compilation unit: a container for code and data. The second line doesn't concern us. The third line describes some choices and assumptions made by the compiler; they don't matter much for this post but you can read more here. The target triple is a gcc-ism that doesn't concern us here either.

An LLVM function optionally has attributes:

```
; Function Attrs: noinline nounwind optnone uwtable
```

Some of them (like these) are supplied by the front end, others get added later by optimization passes. This is just a comment, the actual attributes are specified towards the end of the file. These attributes don't have anything to do with the meaning of the code, so I'm not going to discuss them, but you can read read about them here if you're curious.

Ok, finally on to the function itself:

```
define zeroext i1 @_Z9is_sortedPii(i32* %a, i32 %n) #0 {
```

"zeroext" means that the return value of the function (i1, a one-bit integer) should be zero-extended, by the backend, to whatever width the ABI requires. Next comes the mangled name of the function, then its parameter list, which is basically the same as in the C++ code, except that the “int” type has been refined to 32 bits. The #0 connects this function to an attribute group near the bottom of the file.

Now on to the first basic block:

```
entry:
  %retval = alloca i1, align 1
  %a.addr = alloca i32*, align 8
  %n.addr = alloca i32, align 4
  %i = alloca i32, align 4
  store i32* %a, i32** %a.addr, align 8
  store i32 %n, i32* %n.addr, align 4
  store i32 0, i32* %i, align 4
  br label %for.cond
```

Every LLVM instruction has to live inside a basic block: a collection of instructions that is only entered at the top and only exited at the bottom. The last instruction of a basic block must be a terminator instruction: fallthrough is not allowed. Every function must have an entry block that has no predecessors (places to come from) other than the function entry itself. These properties and others are checked when an IR file is parsed, and can optionally be checked more times during compilation by the “module verifier.” The verifier is useful for debugging situations where a pass emits illegal IR.

The first four instructions in the entry basic block are “alloca”s: stack memory allocations. The first three are for implicit variables created during the compilation, the fourth is for the loop induction variable. Storage allocated like this can only be accessed using load and store instructions. The next three instructions initialize three of the stack slots: a.addr and n.addr are initialized using the values passed into the function as parameters and i is initialized to zero. The return value does not need to be initialized: this will be taken care of later by any code that wasn’t undefined at the C or C++ level. The last instruction is an unconditional branch to the subsequent basic block (we’re not going to worry about it here, but most of these unnecessary jumps can be elided by an LLVM backend).

You might ask: why does Clang allocate stack slots for a and n? Why not just use those values directly? In this function, since a and n are never modified, this strategy would work, but noticing this fact is considered an optimization and thus outside of the job Clang was designed to do. In the general case where a and n might be modified, they must live in memory as opposed to being SSA values, which are — by definition — given a value at only one point in a program. Memory cells live outside of the SSA world and can be modified freely. This may all seem confusing and arbitrary but these design decisions have been found to allow many parts of a compiler to be expressed in natural and efficient ways.

I think of Clang as emitting degenerate SSA code: it meets all the requirements for SSA, but only because basic blocks communicate through memory. Emitting non-degenerate SSA requires some care and some analysis and Clang's refusal to do these leads to a pleasing separation of concerns. I haven't seen the measurements but my understanding is that generating a lot of memory operations and then almost immediately optimizing many of them away isn't a major source of compile-time overhead.

Next let's look at how to translate a for loop. The general form is:

```
for (initializer; condition; modifier) {
  body
}
```

This is translated roughly as:

```
  initializer
  goto COND
COND:
  if (condition)
    goto BODY
  else
    goto EXIT
BODY:
  body
  modifier
  goto COND
EXIT:
```

Of course this translation isn't specific to Clang: any C or C++ compiler would do the same.

In our example, the loop initializer has been folded into the entry basic block. The next basic block is the loop condition test:

```
for.cond:                                         ; preds = %for.inc, %en
  %0 = load i32, i32* %i, align 4
  %1 = load i32, i32* %n.addr, align 4
  %sub = sub nsw i32 %1, 1
  %cmp = icmp slt i32 %0, %sub
  br i1 %cmp, label %for.body, label %for.end
```

As a helpful comment, Clang is telling us that this basic block can be reached either from the for.inc or the entry basic block. This block loads i and n from memory, decrements n (the "nsw" flag preserves the C++-level fact that signed overflow is undefined; without this flag an LLVM subtract would have two's complement semantics), compares the decremented value against i using "signed less than", and then finally branches either to the for.body basic block or the for.end basic block.

The body of the for loop can only be entered from the for.cond block:

```
for.body:
  %2 = load i32*, i32** %a.addr, align 8
  %3 = load i32, i32* %i, align 4
  %idxprom = sext i32 %3 to i64
  %arrayidx = getelementptr inbounds i32, i32* %2, i64 %idxprom
  %4 = load i32, i32* %arrayidx, align 4
  %5 = load i32*, i32** %a.addr, align 8
  %6 = load i32, i32* %i, align 4
  %add = add nsw i32 %6, 1
  %idxprom1 = sext i32 %add to i64
  %arrayidx2 = getelementptr inbounds i32, i32* %5, i64 %idxprom1
  %7 = load i32, i32* %arrayidx2, align 4
  %cmp3 = icmp sgt i32 %4, %7
  br i1 %cmp3, label %if.then, label %if.end
```

The first two lines load a and i into SSA registers; i is then widened to 64 bits so it can participate in an address computation. getelementptr (gep for short) is LLVM's famously baroque pointer computation instruction, it even has its own faq. Unlike a machine language, LLVM does not treat pointers and integers the same way. This facilitates alias analysis and other memory optimizations. The code then goes on to load a[i] and a[i + 1], compare them, and branch based on the result.

The if.then block stores 0 into the stack slot for the function return value and branches unconditionally to the function exit block:

```
if.then:
  store i1 false, i1* %retval, align 1
  br label %return
```

The else block is trivial:

```
if.end:
  br label %for.inc
```

And the block for adding one to the loop induction variable is easy as well:

```
for.inc:
  %8 = load i32, i32* %i, align 4
  %inc = add nsw i32 %8, 1
  store i32 %inc, i32* %i, align 4
  br label %for.cond
```

This code branches back up to the loop condition test.

If the loop terminates normally, we want to return true:

```
for.end:
  store i1 true, i1* %retval, align 1
  br label %return
```

And finally whatever got stored into the return value stack slot is loaded and returned:

```
return:
  %9 = load i1, i1* %retval, align 1
  ret i1 %9
```

There's a bit more at the bottom of the function but nothing consequential. Ok, this post became longer than I'd intended, the next one will look at what the IR-level optimizations do to this function.

(Thanks to Xi Wang and Alex Rosenberg for sending corrections.)

June 26, 2018 regehr Compilers, Computer Science, Education

7 responses to "How Clang Compiles a Function"

1. **Sebastian Redl** says:
   June 27, 2018 at 2:56 am

   Minor nit: printing the predecessors of a basic block is a feature of the IL printer, not Clang. It is just visualizing information that is implicit in the IR anyway.

   That is to say, Clang is not doing anything special to get this effect, and so other producers of IR do not need to worry about duplicating this effect.

2. **kernyan** says:
   June 27, 2018 at 6:17 am

Great detailed write-up, the line by line commentary of the IRs are really useful for someone who's not familiar with LLVM's keywords (e.g., the nsw flag). Would love to see your posts about clang going from source to IR, and from IR to assembly to complete the loop! Thanks!

3. **John Payson** says:
   July 6, 2018 at 11:32 am

   Although the C Standard says that arrayLvalue[x] is equivalent to *(arrayLValue+x), the two constructs sometimes generate different machine code. For example, given:

   ```
   union U { long l[10]; long long ll[10];} u;

   long long utest1(int i, int j)
   {
   if (u.l[i])
   u.ll[j] = 1;
   return u.l[i];
   }

   long long utest2(int i, int j)
   {
   if (*(u.l+i))
   *(u.ll+j) = 1;
   return *(u.l+i);
   }
   ```

   To be fair, the Standard regards the ability to use derived lvalues to access the objects from which they are derived as a Quality-of-Implementation issue rather than a conformance issue, and thus doesn't actually define the behavior of either function in *any* situation. Both functions access an object of type "union U" using lvalues of type "long" and "long long" in violation of 6.5p7, since nothing in the 6.5p7 would require an implementation to permit union object to be accessed using a non-character member type in *any* circumstances. On the other hand, I think they'd expect any quality implementation to be able to recognize an access to a visibly- and freshly-derived lvalue or pointer as being an access to the parent (which–if it was an aggregate–would in turn under 6.5p7 be allowed to access to all of the members thereof). Their failure to explicitly mandate that probably stems from a belief (stated in the rationale) that it isn't necessary to mandate that implementations be of sufficient quality as to actually be useful.

Most code that doesn't need to use unions in interesting ways won't be taking the addresses of union members, and a lot of code that does take the addresses of union members will be using them in "interesting" ways *in the context where their address is taken*, I would thus think a cheap way to support a quality dialect would be to have "getelementptr" on a union type mark the pointer produced thereby so that any cached values data associated with them will get flushed on the next "getelementptr" or other action involving the same union type within the same context, or when leaving the context.

I don't know how easy it would be to add a layer between the llvm generation and the optimizer that would allow llvm to process a quality dialect of C without disabling type-based optimizations completely, but that would seem like it might be an easier approach than trying to modify clang via other means. I find it sad that the authors of clang fail to recognize that the Standards' concessions for low-quality implementations that behave a certain way are not meant to invite other compilers to eagerly behave likewise, and would prefer that an llvm-patching program wouldn't be necessary. Still, it may allow for more efficient code generation than disabling optimizations entirely.

4. **regehr** says:
   July 9, 2018 at 12:46 pm

   Hi John, while I'm not sure about this specific issue, I agree that the standard is pretty frustrating in how to describes the semantics of situations like this.

   Regarding your last paragraph, adding an optimizable layer between the source language and LLVM is a useful and perhaps necessary step for high-level languages and both Swift and Rust do this.

5. **regehr** says:
   July 9, 2018 at 3:35 pm

John, you may be interested in this: https://bugs.llvm.org/show_bug.cgi?id=38108

6. **John Payson** says:
   July 9, 2018 at 3:51 pm

   What is “frustrating” about the Standard is the fact that the authors have deliberately avoided quality-of-implementation issues, but failed to adequately emphasize that the Standard makes no effort to fully everything that would be required to make an implementation be suitable for any particular purpose.

   If the Standard had said that a quality implementation suitable for a particular purpose should make a bona fide effort to behave as though all reads and writes of lvalues behave as reads and writes of the underlying storage in all cases that would be relevant to its intended purpose, and left things at that, that might have been seen as a bit condescending (the notion that quality compilers should behave that way ought to be self-evident) but would have avoided confusion.

   Going beyond that, the Standard could usefully have mandated support for certain patterns, but left it up to the implementations how such support should be realized. For example, it may be reasonable to require that implementations support the pattern:

   ```
   void copyBytes(char *dest, char *src, int n)
   { while (n–) *dest++ = *src++;
   … and then in some other function
   whateverType thing1,thing2;
   …
   copyBytes((char*)&thing1, (char*)&thing2, sizeof thing1);
   ```

   and disabling type-based aliasing analysis for character-type lvalues would be one way of achieving such support, but it’s hardly the only way. Disabling TBAA for all character-type lvalues shouldn’t be required if implementations support the construct in other ways, such as by flushing any register-cached object when the address-of operator is applied to it, flushing all register-cached objects at function-call boundaries, flushing all register-cached objects of type T when a T* is converted to any other type, etc. Some examples of patterns that implementations would be required to support, some that programmers would be strongly encouraged to avoid, and some which should likely be supported by some kinds of compilers but not others, would have been much more useful than a set of rules which–despite being incomplete–needlessly block what should be useful optimizations.

I don't know what gets done to a function's LLVM by different optimization stages, but achieving quality (and conforming, for that matter) semantics would require an ability to recognize that something like:

```
void convertLongToLongLong(void *p)
{ long long temp = *(long*)p; *(long long*)p = temp; }
```

will result in any subsequent read of *(long long*)p being a read of the last value stored by to *(long*)p. An optimizer could replace the read and write with a directive that would generate no more code than required to ensure the required behavior (which in most cases wouldn't require generating any instructions, nor computing the value of "p"), but it must leave sufficient information is needed to ensure the required semantics. Stripping out calls to the function without a trace would not be allowed in a conforming implementation, and would not be appropriate in a non-conforming one intended for any kind of low-level code.

7. **John Payson** says:
   July 9, 2018 at 4:39 pm

   The linked bug-report entry suggests that the authors of clang are more interested in what gcc does than in what the Standard requires or what could be done to minimize the need for -fno-strict-aliasing. I happen to agree with gcc's optimization of "array", and think the Standard should offer three examples of the function as cases where the optimization should be expected on most kinds of implementations, forbidden on all, or expected on some but not others. In brief:

   ```
   struct s {int x, arr[10];};
   ```

   ```
   int invite_optimization1(struct s *p, int index)
   { if (p->x) p->arr[index]=1; return p->x; }
   ```

```
int forbid_optimization2(struct s *p, int index)
{ if (p->x) ((int*)(p->arr))[index]=1; return p->x; }

int optional_optimization3(struct s *p, int index)
{ if (p->x) *(p->arr+index)=1; return p->x; }

int optional_optimization4(struct s *p, int index)
{ void *pp = p->arr; int *ppp = pp; if (p->x) *(ppp+index)=1; return p->x; }

int invite_optimization5(struct s *p, int index)
{ int (*pp)[] = &p->arr; if (p->x) (*pp)[index]=1; return p->x; }
```

Passing a pointer through a cast should generally put the compiler on notice that it may be used in "interesting" ways. Use of subscripting operators on lvalues of array type should be an indication that the resulting lvalue will be part of the actual array upon which the operator is used. Pointer indexing operations that lack either feature and conversions that go through "void*" without any casts should be handled based upon an factors like implementation's intended application field.