A New ELF Linker

Ian Lance Taylor
Google
iant@google.com

Abstract

`gold` is a new ELF linker recently added to the GNU binutils. I discuss why it made sense to write a new linker rather than extend the existing one. I describe the architecture of the linker, and new features. I present performance measurements. I discuss future plans for the linker. I discuss the use of C++ in writing system tools.

1 Motivation

For large programs the linker is a significant part of the compile/edit/debug cycle. When you change a single source file, other than a header file, the compiler need only be invoked once, and it need only look at that source file. The linker needs to look at every input file. When you change many files, or a single header file included by many source files, the compilation may be run in parallel on a cluster of machines via `distcc` or a similar program. The linker must be run serially.

In this paper I will consider a program P. P is based on a Google internal program; the source code is not available. It is written largely in C++. P has over 1300 input objects. When fully linked, P is over 700M. There are over 400,000 global symbols in the global symbol table built by the linker.

Building P from scratch using a compilation cluster using the GNU linker takes 11 minutes, 22 seconds. Of that, the link time is 2 minutes, 28 seconds, 21% of the total time. When a single source file, other than a header file, is modified, less memory is required by the build system, more of the input objects are available in the disk cache, and the link time is 1 minute, 11 seconds. However, since the largest single file in P takes just 25 seconds to recompile without optimization, the link time dominates the total time required to rebuild for testing.

Most programmers do not think about the linker and do not use any linker features. The linker is nothing more than a speed bump in the path of the compile/edit/debug cycle. Smoothing out the bump is a useful step in making programmer's lives easier.

Note: this paper assumes an understanding of basic linker concepts such as symbols, sections, and relocations. It also assumes a basic understanding of the ELF object file format.

2 Investigation

In May, 2006 I began planning how to speed up the linker. Since every modern free software operating system uses the ELF object file format, my focus was on improving linker performance for ELF. I was already familiar with the implementation of the GNU linker. I was the primary author of the core of the GNU linker, which I implemented in late 1993 and early 1994 based on a design by Steve Chamberlain. At the time the GNU linker did not fully support the ELF object file format. The design and implementation were optimized for the a.out and COFF formats. In mid-1994 Eric Youngdale and I added ELF support to the GNU linker, fitting it into the existing design.

At a very high level, the GNU linker follows these steps:

- Walk over all the input files, identifying the symbols and the input sections.
- Walk over the linker script, assigning each input section to an output section based on its name.
- Walk over the linker script again, assigning addresses to the output sections.
- Walk over the input files again, copying input sections to the output file and applying relocations.

This structure makes several aspects of ELF linking awkward.

ELF includes relocation types which do not simply modify section contents, but also create entries in special linker created tables such as the Global Offset Table (GOT) and the Procedure Linkage Table (PLT). The final definition of a symbol—whether it is defined locally or in a dynamic object, whether it is hidden—determines the exact treatment of these relocations. The GNU linker reads the ELF relocations during the first pass over the input files. At that time, of course, the final definition of a symbol is not yet known. The GNU linker uses some mildly complicated bookkeeping to track relocations applied to each symbol in order to figure out how to handle them.

This bookkeeping is merely awkward. What is more serious is that in order to implement it, the GNU linker must walk the entire symbol table. Recall that program P has over 400,000 symbols. Each walk over the symbol table is a time consuming process.

In fact, the GNU linker is profligate when it comes to walking the symbol table. A typical ELF link will walk the symbol table for each of these tasks:

- Assign versions to symbols.
- Set values of symbols defined in dynamic objects.
- Build the GOT and PLT tables.
- See whether any relocations apply to a read-only section.
- Find symbol version dependencies.
- Compute the hash code of dynamic symbols.
- Adjust the values of symbols in merged sections.
- Allocate common symbols.
- Number dynamic symbols which are forced local.
- Number the remaining dynamic symbols.
- Set the string offsets for dynamic symbols.
- Output symbols which were forced local.
- Output the remaining symbols.

While some of these walks could be eliminated fairly easily, their existence is a signal of the awkward fit of ELF in the GNU linker. The ELF support is implemented with a set of hooks, and the symbol table is the only common data structure available to every hook. (By comparison, `gold` walks the symbol table three times in a normal link.)

The GNU linker is implemented on top of the BFD library. This means that an entry in the ELF linker symbol table is implemented by layering on top of the standard BFD linker symbol table. Most of the ELF backends then layer additional information on top of the basic ELF information. The multiple layers duplicate information, and prevent optimizing for alignment. The result is that for a 32-bit i386 link, each entry in the symbol table is 80 bytes, and for a 64-bit x86_64 link, each entry is 156 bytes. (By comparison, in the `gold` symbol table, for a 32-bit link the entries are 48 bytes, and for a 64-bit link the entries are 68 bytes.) Another consequence of this layering is that the GNU linker must encode ELF symbol versions in the symbol name, which leads to some complexity.

As seen above, the linker script is at the heart of the GNU linker. The linker script is based on section names, whereas ELF makes most decisions based on section types and flags (in fact, the only decisions the ELF linker must make based on section names are GNU extensions added to the GNU linker without considering how to best implement them in ELF). The result is a rather complicated linker script, and, more importantly, a lot of section name pattern matching. (`gold` avoids this in the normal case by not using a default linker script.)

The linker script also means that the linker must operate in terms of sections. ELF has both segments and sections, and it is more natural for the linker to operate in terms of segments. The GNU linker must assign addresses to sections rather than segments, which creates a complicated and historically buggy piece of code which assigns the sections to segments.

A basic design decision of BFD, and therefore of the GNU linker, is to run the same code for both normal and cross linking. This means that all linking information read from an object file is read by calling through a function pointer which reads the data byte by byte and shifts it into place. It's not possible to reliably use byte-swapping macros like `htonl` because the alignment is not known. (`gold` uses C++ templates to avoid this byte swapping overhead, as described further below).

The ELF support in the GNU linker is split into four different parts which communicate via a loosely-defined set of function pointers.

- The target independent linker code, which handles the linker script and drives the linking process.
- The ELF linker emulation code.
- The ELF linker proper in BFD.
- The processor specific support being used for a particular link.

These interfaces do not necessarily make the linker slow, but they do make it difficult to implement substantial changes. The module separation is historical rather than logical.

BFD and the GNU linker are written in C. This makes it harder to change data structures, such as from a linked list to a hash table. It also makes it harder to implement an automatically resizing hash table. Thus the GNU linker can run into significant slowdowns on large programs, as the code uses data structures which turn out to be inadequate but are nontrivial to change. For example, BFD provides a nice automatically resizing hash table, but the key must be a string. Any code that wants to use a different key type is out of luck.

I've known about these difficulties with the GNU linker for some time. Many of them are the fault of the original implementor of the code—that is to say, they are largely my fault. While many people worked on the GNU linker, I did much of the original implementation and thus laid the groundwork for these issues. That said, I think I first proposed a rewrite to avoid using BFD in 1993. Only recently was it possible for me to devote the time required for the task.

It would be possible to make significant incremental improvements to the performance of the GNU linker for ELF. However, I have believed for some time that the best choice is to redesign the linker with ELF in mind. Removing the BFD library and taking linker scripts out of the heart of the linker are not incremental improvements; they are a complete rewrite. I decided that it made more sense to implement a new linker from scratch.

3 Architecture

As mentioned above, the main goal of `gold` is faster linking. Accordingly, it is ELF only. ELF is the only object file format which matters on free software systems. There is no way to support multiple object file formats without running at least slightly slower.

I also decided to write it in C++. While it is possible to argue performance of C vs. C++ on many levels, C++ has a feature which is not easily available in C: template specialization. I used this to avoid the slowdown due to byte swapping; I discuss this in more detail below. Using C++ also permitted easy use of different types of data structures; this could have been done in C as well, but is easier in C++.

At a high level, `gold` follows these steps:

- Walk over the input files, identifying the symbols and the input sections.
- Walk over the input files again, reading the relocations and building the PLT and GOT.
- Assign output sections to output segments, and assign addresses to the output segments.
- Walk over the input files again, copying input sections to the output file and applying relocations.

As can be seen, the main differences from the GNU linker are the second walk over the input file to read the relocations, and the omission of the linker script.

Using a second walk over the input files to read the relocations simplifies the symbol handling. It also avoids a walk over the symbol table. It is better to focus on the subset of symbols which need GOT or PLT entries.

Because there is no default linker script, input sections can be assigned to output sections as they are read. The output sections are then easily grouped into output segments based on their type and flags (read-only, writable, executable) rather than according to the addresses set in the linker script. (`gold` does also support linker scripts for compatibility purposes; linker scripts use a different code path for section and segment layout.)

3.1 Threads

gold is multi-threaded. The intent of the threading support is to permit the linker to overlap disk tasks (e.g., reading the input symbol table) and CPU tasks (e.g., processing the symbol table). The link process is split up into tasks. Tasks essentially form a directed graph, in that a given task may not be started until one or more other tasks have completed. There are very few fine-grained locks. A pool of worker threads picks up tasks as they are available. Unfortunately, using multiple threads has not proved to be a significant performance improvement in practice. It is a minor improvement in some cases, a minor slowdown in others. I am continuing to investigate whether this is due to bugs in the implementation or is somehow inherent in the linking process or the workqueue model.

3.2 C++

C++ template specialization permits the same code to be compiled in multiple different ways. The closest similar feature in C is preprocessor macros. However, template specialization is type safe, and has much better debugger support.

gold uses this to implement byte swapping. Any function which reads linking information from an input file—e.g., section headers, symbol table entries, relocation entries—is implemented as a template based on the class of the output ELF file (an integer, either 32 or 64) and on whether the output ELF file is big or little endian. The function then reads values using the template function Swap<size,big_endian>::readval. When the endianness of the host and the target are the same, this is specialized into a simple load from memory. When the endianness differs, this is a memory load followed by a byte swap. The linker ensures that the ELF data is aligned as needed. This is a time/space tradeoff, in that it makes gold itself larger, since multiple versions of various functions are compiled (although configure options may be used to only build specific versions).

It follows that, unlike the GNU linker, gold does not execute the same code on all systems. The code is customized based on the endianness. The __BYTE_ORDER macro from <endian.h> is used to select the appropriate version of the template function at compile time. This does raise the possibility of unexpected bugs in cross linkers. Fortunately the code in question is relatively small and easy to inspect.

This approach should be more clear if you look at the source code in the figures.

Not all of the code in gold is templatized. There are various places where non-templatized code must call templatized code. This generally takes the form of a switch statement on the target, returning an enumerated value, e.g., TARGET_32_LITTLE for a 32-bit little endian target. This switch is evaluated at runtime, and leads to a call to a function where the choices are evaluated at compile time.

As mentioned above, the other advantage of C++ is easy access to flexible data structures. For example, gold currently uses 22 different hash tables. They have several different key types; only three of them have char* keys (three have std::string keys). Adding a new hash table to gold is as easy as writing a typedef. Adding a new hash table to BFD generally requires writing four functions or macros. Similarly, gold uses many instances of std::vector for lists of items where the final number is unknown. The corresponding code in BFD requires explicit size checks and calls to realloc, the details of which are different for each instance.

Using C++ does require that gold be able to find the C++ library at runtime. This is not a problem normally, but is something to consider when building gold with an uninstalled compiler.

3.3 Merge sections

When linking code written in C or C++, one of the slower parts of the linker is the handling of merge sections. These are sections with the SHF_MERGE flag set, and are used to hold read-only constants, including string constants. When the linker finds the same constant in different input files, it merges them into a single instance of the constant in the output file. This reduces the size of the program and hopefully improves cache behaviour. A similar sort of merging is used for exception frame data found in .eh_frame sections.

This requires building a hash table to hold the constant values in order to find duplicates, which is easy enough. The harder part is that there are, naturally, relocations

```
struct Endian
{
 public:
  static const bool host_big_endian = __BYTE_ORDER == __BIG_ENDIAN;
};

// Valtype_base is a template based on size (8, 16, 32, 64) which
// defines the type Valtype as the unsigned integer, and
// Signed_valtype as the signed integer, of the specified size.

template<int size>
struct Valtype_base;

template<>
struct Valtype_base<32>
{
  typedef uint32_t Valtype;
  typedef int32_t Signed_valtype;
};

// Convert_endian is a template based on size and on whether the host
// and target have the same endianness.  It defines the type Valtype
// as Valtype_base does, and also defines a function convert_host
// which takes an argument of type Valtype and returns the same value,
// but swapped if the host and target have different endianness.

template<int size, bool same_endian>
struct Convert_endian;

template<int size>
struct Convert_endian<size, true>
{
  typedef typename Valtype_base<size>::Valtype Valtype;

  static inline Valtype
  convert_host(Valtype v)
  { return v; }
};

template<>
struct Convert_endian<32, false>
{
  typedef Valtype_base<32>::Valtype Valtype;

  static inline Valtype
  convert_host(Valtype v)
  { return bswap_32(v); }
};
```

Figure 1: Template specialization for efficient swapping 1

```
// Convert is a template based on size and on whether the target is
// big endian.  It defines Valtype and convert_host like
// Convert_endian.  That is, it is just like Convert_endian except in
// the meaning of the second template parameter.

template<int size, bool big_endian>
struct Convert
{
  typedef typename Valtype_base<size>::Valtype Valtype;

  static inline Valtype
  convert_host(Valtype v)
  {
    return Convert_endian<size, big_endian == Endian::host_big_endian>
      ::convert_host(v);
  }
};

// Swap is a template based on size and on whether the target is big
// endian.  It defines the type Valtype and the functions readval and
// writeval.  The functions read and write values of the appropriate
// size out of buffers, swapping them if necessary.  readval and
// writeval are overloaded to take pointers to the appropriate type or
// pointers to unsigned char.

template<int size, bool big_endian>
struct Swap
{
  typedef typename Valtype_base<size>::Valtype Valtype;

  static inline Valtype
  readval(const Valtype* wv)
  { return Convert<size, big_endian>::convert_host(*wv); }
};

// This is how the template specializations and function inlining
// works, assuming Endian::host_big_endian is false (i.e.,
// __BYTE_ORDER == __LITTLE_ENDIAN in <endian.h>.

//    Swap<32, false>::readval(wv)
// => Convert<32, false>::convert_host(*wv)
// => Convert_endian<size, true>::convert_host(*wv)
// => *wv

// In other words, the call to Swap::readval turns into a memory load
// at compile time.
```

Figure 2: Template specialization for efficient swapping 2

which refer to the constants. When processing those relocations, the output must refer to the location of the constants in the merged output section. This processing is much more complicated than ordinary relocation processing, which just needs the value of a symbol. In my initial implementation looking up the address of a merged constant took over 25% of the total CPU time when linking program P.

The GNU linker implements this lookup by looking for the constant in the input section and doing a hash table lookup in the constant table. This is not a bad approach, but it requires saving the input section contents in memory, and it require recomputing the hash code each time a relocation refers to the constant. I chose instead to build a mapping from an input section offset to an output section offset. Given a relocation, the input section offset can be determined from the symbol and the addend. The mapping can then be used to efficiently find the output section offset. As it turned out, to make this efficient in practice, I had to build a separate mapping for each input merge section, and I had to build a temporary cache while processing an input file. Getting the right data structures to make this efficient was one of the most difficult parts in `gold`.

4 Performance

Using `gold` when building program P from scratch takes 1 minute 15 seconds, which is 50% of the time taken by GNU ld. Using `gold` when one source file is changed takes 13 seconds, which is 18% of the time taken by GNU ld. The difference is primarily how much of the input is in the disk cache. These comparisons were done with a version of GNU ld between 2.17 and 2.18, which had itself been tuned for performance on large C++ programs by increasing some hash table sizes.

For a simple hello world program, there is less scope for improvement. When linking dynamically, the GNU linker takes 0.063 seconds and `gold` takes 0.040 seconds, 63% of the time. When linking statically, the GNU linker takes 0.178 seconds, and `gold` takes 0.083 seconds, 46% of the time.

`gold` itself a medium sized C++ program, some 50,000 lines of code. When linking `gold`, the GNU linker takes 1.137 seconds and `gold` takes 0.522 seconds, 45% of the time.

In general testing `gold` ranges from twice as fast to five times as fast, depending on the input. Generally the larger the program, the bigger the improvement.

5 Features

Although `gold` is focused on speed, it does have a couple of new features.

`gold` has a new `--detect-odr-violations` option, which looks for potential violations of the C++ ODR rule. The C++ ODR rule says that a given name may only refer to a single object. It is fairly easy to violate in a large C++ program by, say, defining two different template classes with the same name in different input files. This is forbidden, but it is difficult to detect in practice, and the C++ standard does not require the compiler to report it.

`gold` uses a heuristic to find potential ODR violations: if the same symbol is seen defined in two different input files, and the two symbols have different sizes, then `gold` looks at the debugging information in the input objects. If the debugging information suggests that the symbols were defined in different source files, `gold` reports a potential ODR violation. This approach has both false negatives and false positives. However, it is reasonably reliable at detecting problems when linking unoptimized code. It is much easier to find these problems at link time than to debug cases where the wrong symbol is used at runtime.

`gold` has a new `--compress-debug-sections` option, which compresses the debugging information using the `zlib` library. The debugging information is highly compressible, and this option typically reduces it to half the original size. Parsing the debug information to eliminate duplicates would also be effective, and should be implemented in the future; it will decrease the time required for the debugger to read the information, but likely at the cost of increasing link time.

6 Future plans

The future plans for `gold` are, naturally enough, to make it faster. Two main approaches are planned.

6.1 Concurrent Linking

When using a compilation cluster, the object files will be generated in parallel. It is possible to run the linker in parallel with the compilations. As each object is compiled, the linker can read the symbols and sections out of it and do an initial link. It can put the sections into place in the output file, and record relocations against symbols which have not yet been resolved. As more object files are seen, more of the relocations can be resolved.

The resulting output file will not be identical to the non-concurrent output, and will in fact be slightly less efficient—there will be wasted space between output sections. The expectation is that it will be OK to pay a small penalty in executable size and runtime in return for much faster link times. When using distributed compilation, the actual link time might be reducible to near-zero, as the linker will operate while the system would otherwise be waiting for compilations to complete.

6.2 Incremental Linking

Concurrent linking is optimized for the case where many objects are being rebuilt. When only a few objects are being rebuilt, incremental linking is a more useful approach. Incremental linking modifies an existing executable by inserting new object code. Where the new object code is the same size or smaller it can simply overwrite the existing object code. When it gets larger the incremental linker will have to find new space to store it.

Incremental linking requires storing relocation information in the executable so that all relocations can be updated as required. It also requires storing additional symbol information, so that the symbol table may be adjusted appropriately in case the set of symbols defined in the object changes.

As with concurrent linking, the resulting output file will be slightly less efficient. Again this seems a price worth paying to significantly decrease link times for large programs. Incremental linking is a particularly natural fit with incremental compilation approaches.

7 C++ System Tools

`gold` was among other things an experiment in writing a system tool in C++. I believe it was a successful one. `gold` does run significantly faster than GNU ld. The `gold` executable itself is nearly four times larger than GNU ld, but that is at least in part due to an explicit tradeoff: much of the code in `gold` is in fact compiled four times, for 32-bit vs. 64-bit and for big vs. little endian. (A version of `gold` configured to only build one version of that code was 2.5 times larger than GNU ld.)

I encountered some problems with different versions of `g++`. `g++ 3.2` is unable to compile `gold` at all, getting confused by template specializations with no parameters. `g++ 4.1` had trouble with uninitialized const iterators for at least one STL type. `g++ 4.3.0` did not recognize attributes on template function declarations.

When experimenting with different compilers, I occasionally ran into trouble with not being able to find a sufficiently new version of libstdc++. This was easily fixed by setting `LD_LIBRARY_PATH`. However, it might be desirable to have an option analogous to `-static-libgcc` which requests a static link of libstdc++.

`gcc` does not need to use the template specialization tricks that `gold` uses. However, `gcc` would benefit from easy access to flexible data structures—just take a look at `vec.h` to see what is being done in C. I believe that it would be appropriate to now start using C++ in `gcc`, and I would like to encourage everybody to consider that seriously.

8 Acknowledgements

I am the primary author of `gold`, but as with any program it could not have been done without many other people. Cary Coutant worked on several pieces, notably including support for generating shared libraries and for TLS. Craig Silverstein wrote the initial x86_64 support with Andrew Chatham, and also implemented the new options for ODR violations and debug info compression, as well as the general option handling, among other things. They and others helped with debugging problems, which were found by many hardy Googlers willing to test an experimental linker. David Miller contributed the SPARC port.