Gold

Ian Lance Taylor
Google

June 17, 2008
What is gold?

- gold is a new linker.
- gold is now part of the GNU binutils (if you configure with `--enable-gold`, gold is built instead of GNU ld).
- gold only supports ELF, which is used by all modern operating systems other than Mac OS and Windows.
- gold is written in C++.
- gold currently supports x86, x86_64, and SPARC.
Why write a new linker?

- Almost all programmers use no linker features.
  - Exception: linker scripts on embedded systems
  - Exception: version scripts for libraries
- The linker is a speedbump in the development cycle.
- Compilation can be easily distributed; linking can not.
- The GNU linker is slow.
Why is the GNU linker slow?

- It was designed for the a.out and COFF object file formats. ELF support was added later.
- ELF includes relocations which build new data; this had to be shoehorned into the GNU linker.
- The GNU linker traverses the symbol table thirteen times in a typical link.
  - gold traverses the symbol table three times.
- The GNU linker is built on top of BFD, increasing the size of basic data structures like symbol table entries.
  - For x86_64, GNU linker symbol table entry is 156 bytes.
  - gold is 68 bytes.
- The GNU linker always loads values using byte loads and shifts.
Why not fix the GNU linker?

- The GNU linker source code is split in several parts which communicate by various hooks.
  - The linker proper (`src/ld`).
  - The ELF emulation layer (`src/ld/emultempl/elf32.em`).
  - The generic BFD library (`src/bfd`).
  - The ELF support in the BFD library (`src/elf.c`, `src/elflink.c`).
  - The processor specific ELF backend (e.g., `src/elf64-x86-64.c`).
- The GNU linker is designed around a linker script. All actions are driven by entries in the linker script.
Why not fix the GNU linker?

- The GNU linker source code is split in several parts which communicate by various hooks.
  - The linker proper (`src/ld`).
  - The ELF emulation layer (`src/ld/emultempl/elf32.em`).
  - The generic BFD library (`src/bfd`).
  - The ELF support in the BFD library (`src/elf.c`, `src/elflink.c`).
  - The processor specific ELF backend (e.g., `src/elf64-x86-64.c`).

- The GNU linker is designed around a linker script. All actions are driven by entries in the linker script.

Changing this design is not a fix; it is a rewrite.
Some notes on the gold implementation. For more information, see the paper. For details, see the source code.

- Over 50,000 lines of commented C++ code.
- Uses templates to avoid byte swapping for a native link.
- Multi-threaded.
- Not driven by a linker script.
  - Linker scripts are supported, though.
  - Linker script support is over 10% of the source code.
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);
    }
};

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);
    }
};
// Convert_endian<64, true>::convert_host(*wv)

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

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

// *wv
Performance

How long it takes gold to link compared to the GNU linker.

- Hello, world
  - Dynamic link: 37% faster
  - Static link: 54% faster
- Large program (700M, 1300 objects, 400,000 symbols)
  - Complete build from scratch: 50% faster
  - Change one input object: 82% faster
  - Difference is disk cache effects.
gold has some features which are not in the GNU linker.

- **C++ ODR detection.**
  - Uses debug info to look for two symbols with the same name defined at different source lines.

- **Debug info compression.**

- **Discard debug info other than source line information**
  - Backtraces work.
  - Local variables are not available.
Concurrent Linking

Problem: compilation can be easily distributed; linking cannot.

- Solution: concurrent linking.
- Start the link before starting the compilations.
- As each compilation completes, pass the object file to the linker.
- The linker lays each object down as it receives it.
- The linker stores relocations as it goes along.
- As the first objects are seen, the symbols are determined, and relocations can be applied.
- This is not implemented.
Incremental Linking

Problem: changing one object file only changes a small part of an executable. Recreating the entire executable is wasteful.

- Solution: incremental linking.
- The linker records symbol and relocation information in the executable.
- The linker checks which objects are newer than the executable.
- Only those objects are updated.
- If only object changes, there is significantly less relocation processing and significantly less I/O.
- This is not implemented.
Who

- Ian Lance Taylor
  - Design, bulk of implementation.
- Cary Coutant
  - Shared library generation, TLS.
- Craig Silverstein
  - x86_64 port, ODR detection, debug info compression.
- Andrew Chatham
  - x86_64 port.
- David Miller
  - SPARC port.