MCLinker - the final toolchain frontier

Jörg Sonnenberger

March 16, 2013

Abstract

LLVM and Clang provide the top half of an integrated BSD licensed cross-platform toolchain. The MCLinker project implements the remaining parts. It is compact, modular and efficient. Currently supported platforms are ELF systems running X86 (32bit and 64bit), ARM and MIPS. MIPS64 and Hexagon are work-in-progress.

1 Introduction

Following the GPL3 announcement of the Free Software Foundation, the LLVM project gained a lot of traction. The Clang front-end allows replacing GCC for C and C++ code on many important platforms. The Machine Code layer provides an integrated assembler, replacing the dependency on external assemblers for most situations. The combination provides a powerful toolchain with full support for cross-compilation. For a long time, the only major component missing to replace GNU binutils has been a linker. This gap is now filled by the Machine Code Linker (MCLinker) project.

This paper first introduces the architecture of MCLinker and compares it to GNU ld. The performance for linking large programs is evaluated and compared with GNU ld and Google gold. A detailed status report of the implementation is presented as well as list of future projects.

2 Architecture

MCLinker is a modern, modular linker. It is designed to process input in three separate phases:

- Building the input tree based on command line and implicit default values.
- Resolve symbols.
- Determine output positions, apply relocations and write output files.

GNU ld doesn't have a clear phase separation; individual steps are re-applied iteratively until all input files are processed and all resolvable symbols are resolved. The result is complicated and cache-unfriendly code.

Google gold on the other hand mixes the first two phases. Symbols that can't be resolved in the first round are retried later. This choice improves the maintainability of performance of the linker code, but is still more complicated than the design of MCLinker. It is useful to note that gold supports multi-threading during the relocation processing to speed up linking large input sets.

2.1 Input tree

The input tree is the first intermediate representation (IR) used by MCLinker. It contains vertices for every input file and all positional options. Positional options are options like *–start-group* or *–as-needed* that change the handling of a subset of the input files. Input files can form subtrees, e.g. linker archives form a subtree with a child node for every archive member. Edges in the input graph are either positional relationships for options or inclusion links for archives.

2.2 Symbol resolution

Symbol resolution is implemented using tree iteration. As result of symbol resolution, the low-level IR is created: the fragment reference graph. This graph represents the individual sections found during input process and the symbolic dependencies between them.

The input tree is traversed and for every input file it is checked whether it must be included in the output. An input file is included if it was explicitly mentioned on the command line or if it provides a definition for a currently undefined symbol. If the input is to be included, its sections and symbol table are processed. When the tree iteration hits the start of a linker group, the current position is pushed onto a stack. Iteration continues until the corresponding end is seen. The linker now checks if any new undefined reference was found while processing this group. If so, the process starts again from memorized position.

The data structures used for symbol resolution are highly optimized to maximize cache locality. The symbol attributes and the initial part of the name share a cache line on modern CPUs. This helps the CPU when linking large programs with many smaller functions; C++ applications are well-known for such characteristics.

2.3 Layout

The layout step is the first part of the last linking phase. It is responsible for deciding what sections should be written in which order and to what position. It merges sections as necessary or drops them, if they are redundant (e.g. multiple inline definitions for C++ methods). At the end of this phase, the symbol values are finalized.

Defering the section merging until such a late step in the linking process has the advantage of avoiding many redundant computations. The list of all input sections is known at this point and after a single ordering pass, addresses can be assigned directly.

2.4 Relocation

The relocation step follows the layout step and is responsible for determining the symbol values in the various places that reference the symbols. It may prepare to modify the instructions to replace more costly code with simpler sequences. This helps platforms with limited immediate value ranges like ARM and MIPS to load short (relative) address directly instead of using constant tables. It can also be used to use more efficient access methods for Thread Local Storage (TLS).

2.5 Writing

Once the relocation step is done, all that is left is applying the relocations to the input sections and writing the result to the output file with any meta-data necessary for the file format. MCLinker aggressively uses memory mapped files when possible. This leverages the page lookup table (TLB) cache and improves page locality. It helps improves the use of the OS file system cache.

3 Performance

To measure the performance of MCLinker, the llvm-tblgen and clang binaries were built using the normal options in NetBSD (-O2, no -g). The final linker invocation was repeated using MCLinker, GNU ld and Google gold. For MCLinker the current development version of March 11 was used. GNU ld and gold are both from GNU binutils 2.22.90.

	MCLinker	GNU ld	gold
llvm-tblgen	0.05s	0.10s	0.04s
clang	0.69s	1.41s	0.44s

The resulting binaries have the following size:

	MCLinker	GNU ld	gold
llvm-tblgen			
.text	2,123,527	1,827,773	1,786,483
.data	2,408	2,664	2,520
.bss	5,360	5,912	2,520
clang			
.text	34,299,746	26,917,022	26,698,448
.data	21,984	22,112	22,112
.bss	47,624	47,736	47,704

Both MCLinker and gold are significantly faster than GNU ld. MCLinker is clearly pulling in too many objects, which is responsible for the much larger binaries. This is likely also reponsible for a good part of the performance difference to gold. Evidently, further analysis is needed.

Of special interest for embedded use is the memory footprint of the different linkers:

	MCLinker	GNU ld	gold
llvm-tblgen	17,508KB	17,700KB	17,528KB
clang	176MB	150MB	182MB

The peak Resident Memory Size shows clearly that both MCLinker and gold are optimised for in-memory processing. Since they are reading the full symbol tables of all input files first, they require more memory.

4 Implementation status

Linker functionality falls into two categories. The first is generic features like symbol versioning which are handled in the platform independent core. The second is features specific to a target platform like the handling of the different relocation types.

4.1 Platform independent

Most features necessary for ELF systems are implemented. This includes static and shared linkage as well partially linking objects to form a new relocatable object. Weak symbols are overwritten by strong definitions. The different visibility types are handled. Missing for shared libraries is the processing of *DT_NEEDED* entries and loading the symbols of the referenced objects. This means that currently only symbols in libraries explicitly requested on the command line are found, when standard ELF behavior is to recursively scan libraries and their dependencies. This omission is not absolved by recent versions of GNU ld switching to the same broken semantic.

C++ requires constructor and destructor support. The compiler uses *.ctors* or *.init_array* sections (or *.dtor* and *.fini_array* for destructors) to implement this. MCLinker has full support for either form. It doesn't try to convert one mechanism into the other due to the slightly different ordering rules. Support for COMDAT-sections is present. This allows defining the implemention of a function in more than one object file and letting the linker pick one. It is used for implicit template specialisations. The *.eh_frame* sections necessary for exception handling are correctly processed and the associated *.eh_frame_hdr* section is created. The latter provides a fast binary search table to reduce runtime overhead. One issue in this area is the missing support for relocations into the *.eh_frame* section. This is needed for support the older *__register_frame_info* style shared library interface.

The platform-independent part of TLS support is present. This concerns the proper merging of the *.tdata* and *.tbss* sections and the corresponding segments.

Support for symbol versioning is missing. For NetBSD, it is only used by *libgcc* at the moment, so it is primarily an issue for missing MCLinker builds with non-MCLinker ones. It is far more important for FreeBSD though.

Linker scripts are required in certain situations that need more granular control over the image layout. For NetBSD, this concerns RUMP and the kernel. It is planned to add this feature during the second quarter of 2013.

Support for other binary formats like Mach-O is planned, but not implemented.

4.2 NetBSD/i386

NetBSD/i386 needs a few workarounds for the lack of linker scripts. With a fallback to using GNU ld, a NetBSD release build can be finished. The result works well with few regressions seen in the ATF testsuite results. The only remaining platform specific issues is the incomplete support for TLS relaxations, i.e. converting dynamic TLS relocations into faster relocations for the main binary.

4.3 NetBSD/amd64

AMD64 is the youngest platform supported by MCLinker; the initial changes were added by H.J. Lu in early February. It is still maturing. NetBSD/amd64 shows the same issues as NetBSD/i386 and has some additional problems with certain relocation types. At the time of writing, this primarily affects *libc* and *libstdc++*.

4.4 ARM and MIPS

ARM and MIPS are the original platforms for most of the MCLinker work. Since last year, it is possible to build a full Android system on those platforms. NetBSD has not been tested yet though. Support is currently limited to the 32bit variants of the ARM and MIPS ABIs.

5 Future work

The biggest priority for the MCLinker project is to finish the missing platform independent features. This means adding symbol versioning and linker script support. Another important item is support for Link Time Optimisation (LTO). Open research questions include the impact of fine-grained layout decisions by creating individual sections for every function in the compiler and letting the linker figure out the best placement.

Support for the alternative 32bit ABI on X86 (X32), MIPS64 and Hexagon is work-in-progres.

6 Summary

MCLinker provides a cross-compiling friendly linker for many platforms targetted by the BSDs. It is making huge progress towards fully replacing GNU ld on i386 and AMD64 for NetBSD, with a similar state easily reachable for ARM and MIPS. A GPL free toolchain is within reach now.