Advanced techniques

$Revision: 1.2 $
$Date: 1999/02/07 05:06:14 $

This chapter describes a grab-bag of miscellaneous linker techniques that don't fit very well anywhere else.

Techniques for C++

C++ presents three significant challenges to the linker. One is its complicated naming rules, in which multiple functions can have the same name if they have different argument types. Name mangling addresses this well enough that all linkers use it in some form or another.

The second is global initializers and destructors, routines that need to be run before the main routine starts and after the main routine exits. This requires that the linker collect the pieces of initializer and destructor code, or at least pointers to them, into one place so that startup and exit code can run it all.

The third, and by far the most complex issue involves templates and "extern inline" procedures. A C++ template defines an infinite family of procedures, with each family member being the template specialized by a type. For example, a template might define a generic hash table, with family members being a hash table of integers, of floating point numbers, of character strings, and of pointers to various sorts of structures. Since computer memories are finite, the compiled program needs to contain all of the members of the family that are actually used in the program, but shouldn't contain any others. If the C++ compiler takes the traditional approach of treating each source file separately, it can't tell when it compiles a file that uses templates whether some of the template family members are used in other source files. If the compiler takes a conservative approach and generates code for each family member used in each file, it will usually end up with multiple copies of each family member, wasting space. If it doesn't generate that code, it risks having no copy at all of a required family member.

Inline functions present a similar problem. Normally, inline functions are expanded like macros, but in some cases the compiler generates a conventional out-of-line version of the function. If several different files use a single header file that contains an inline function and some of them require an out-of-line version, the same problem of code duplication arises.

Some compilers have used approaches that change the source language to help produce object code that can be linked by ‘‘dumb’’ linkers. Many recent C++ systems have addressed the problem head-on, either by making the linker smarter, or by integrating the linker with other parts of the program development system. We look briefly at these latter approaches.

Trial linking

In systems stuck with simple-minded linkers, C++ systems have used a variety of tricks to get C++ programs linked. An approach pioneered by the original cfront implementation is to do a trial link which will generally fail, then have the compiler driver (the program that runs the various pieces of the compiler, assembler, and linker) extract information from the result of that link to finish the compiling and relink, Figure 1.

Figure 1: Trial linking

input files pass through linker to trial output plus errors, then inputs plus info from errors plus maybe more generated objects pass through linker to final object

On Unix systems, if the linker can’t resolve all of the undefined references in a link job, it can still optionally can also produce an output file which can be used as the input to a subsequent link job. The linker uses its usual library search rules during the link, so the output file contains needed library routines as well as information from the input file. Trial linking solves all of the C++ problems above in a slow but effective way.

For global initializers and destructors, the C++ compiler creates in each input file routines that do the initialization and destruction. The routines are logically anonymous, but the compiler gives them distinctive names. For example, the GNU C++ compiler creates routines named `_GLOBAL_.I.__4junk` and `_GLOBAL_.D.__4junk` to do initialization and destruction of variables in a class called `junk`. After the trial link, the linker driver examines the symbol table of the output file and makes lists of the global initializer and destructor routines, writes a small source file with those lists in arrays (in either C or assembler). Then in the relink the C++ startup and exit code uses the contents of the arrays to call all of the appropriate routines. This is essentially the same thing that C++-aware linkers do, just implemented outside the linker.

For templates and extern inlines, the compiler initially doesn't generate any code for them at all. The trial link has undefined symbols for all of the templates and extern inlines actually used in the program, which the compiler driver can use to re-run the compiler and generate code for them, then re-link.

One minor issue is to find the source code for the missing templates, since it can be lurking in any of a potentially very large number of source files. Cfront used a simple ad-hoc technique, scanning the header files, and guessing that a template declared in `foo.h` is defined in `foo.cc`. Recent versions of GCC use a "repository" that notes the locations of template definitions in small files created during the compilation process. After the trial link, the compiler driver needs only scan those small files to find the source to the templates.

Duplicate code elimination

The trial linking approach generates as little code as possible, then goes back after the trial link to generate any required code that was left out the first time. The converse approach is to generate all possible code, then have the linker throw away the duplicates, Figure 2. The compiler generates all of the expanded templates and all of the extern inlines in each file that uses them. Each possibly redundant chunk of code is put in its own segment with a name that uniquely identifies what it is. For example, GCC puts each chunk in an ELF or COFF section called

`.gnu.linkonce.t.mangledname` where mangled name is the "mangled" version of the function name with the type information added. Some formats identify possibly redundant sections solely by name, while others such as Microsoft's ECOFF use type flags. If there are multiple copies of a section with the same name, the linker discards all but one of them at link time.

Figure 2: Duplicate elimination

Input files with redundant sections pass into the linker which collapses them into a single result (sub)section

This approach does a good job of producing executables with one copy of each routine, at the cost of very large object files with many copies of templates. It also offers at least the possibility of smaller final code than the other approaches. In many cases, code generated when a template is expanded for different types is identical. For example, a template that implemented a bounds-checked array of <TYPE> would generally expand to identical code for all pointer types, since in C++ pointers all have the same representation. A linker that's already deleting redundant sections could check for sections with identical contents and collapse multiple identical sections to one.

Database approaches

The GCC respository is a simple version of a database. In the longer run, tool vendors are moving toward database storage of source and object code, such as the Montana environment in IBM's Visual Age C++. The database tracks the location of each declaration and definition, which makes it possible after a source change to figure out what the individual routine dependencies are and recompile and relink just what has changed.

Incremental linking and relinking

For a long time, some linkers have permitted incremental linking and relinking. Unix linkers provide a `-r` flag that tells the linker to keep the symbol and relocation information in the output file, so the output can be used as the input to a subsequent link.

IBM mainframes have always had a ‘‘linkage editor,’’ rather than a linker. In the IBM object format, the segments in each input file (IBM calls the segments control sections or CSECTs) retain their individual identities in the output file. One can re-edit a linked program and replace or delete control sections. This feature was widely used in the 1960s and early 1970s when compiling and linking were slow enough that it was worth the manual effort needed to arrange to relink a program, replacing just the CSECTS that had been recompiled. The replacement CSECTs need not be the same size as the originals; the linker adjusts all of the relocation information in the output file as needed to account for the different locations of CSECTs than have moved.

In the mid to late 1980s, Quong and Linton at Stanford did experiments with incremental linking in a UNIX linker, to try and speed up the compile-link-debug cycle. The first time their linker runs, it links a conventional statically linked executable, then stays active in the background as a daemon with the program’s symbol table remaing in memory. On subseqent links, it only treats the input files that have changed, replacing their code in-place in the output file but leaving everything else alone other than fixing up references to symbols that have moved. Since segment sizes in the recompiled files usually don’t change very much from one link to the next, they build the initial version of the output file with a small amount of slop space between the input file segments, Figure 3. On each subsequent link, so long as the changed input files’ segments haven’t grown more than the slop amount, the changed files’ segments replace the previous versions in the output file. If they have grown past the end of the slop space, the linker moves the subsequent segments in the output file using their slop space. If more than a small number of segments need to be moved, the linker gives up and relinks from scratch.

Figure 3: Incremental linking

picture of inclink-ed object file with slop between segments, and new version's segments pointing to replace old ones

The authors did considerable instrumentation to collect data on the number of files compiled between linker runs in typical development activities and the change in segment sizes. They found that typically only one or two files change, and the segments grow only by a few bytes if at all. By putting 100 bytes of slop between segments, they avoided almost all relinking. They also found that creating the output file's symbol table, which is essential for debugging, was as much work as creating the segments, and used similar techniques to update the symbol table incrementally. Their performance results were quite dramatic, with links that took 20 or 30 seconds to do conventionally dropping to half a second for an incremental link. The primary drawback of their scheme was that the linker used about eight megabytes to keep all of the symbols and other information about the output file, which at the time was a lot of memory (workstations rarely had more than 16MB.)

Link time optimization

On most systems, the linker is the only program in the software building process that sees all of the pieces of a program that it is building at the same time. That means that it has opportunities to do global optimization that no other component can do. For example, in a language with class inheritance, calls to class methods generally use indirect calls since a method may be overridden in a subclass. But if there aren't any subclasses, or there are subclasses but none of them override a particular method, the calls can be direct. A linker could make special case optimizations like this to avoid some of the inefficiencies otherwise inherent in object oriented languages. Fernandez at Princeton wrote an optimizing linker for Modula-3 that was able to turn 79% of indirect method calls into direct calls as well as reducing instructions executed by over 10%.

A more aggressive approach is to perform standard global optimizations on an entire program at link time. Srivastava and Wall wrote an optimizing linker that decompiled RISC architecture object code into an intermediate form, applied high-level optimizations such as inlining and low-level optimizations such as substituting a faster but more limited load instruction for a slower and more general one, then regenerated the object code.

Other linkers have done other architecture-specific optimizations. The Multiflow VLIW machine had a very large number of registers, and register saves and restores could be a major bottleneck. An experimental tool used profile data to figure out what routines frequently called what other routines. It modified the registers used in the code to minimize the overlapping registers used by both a calling routine and its callee, thereby minimizing the number of saves and restores.

Link time code generation

Many linkers generate small amounts of the output object code, for example the jump entries in the PLT in Unix ELF files. But some experimental linkers do far more code generation than that.

XXXXXXXXXXXXXXXXXXXXXXXXXX The Srivastava and Wall optimizing linker starts by decompiling object files back into intermediate code. In most cases, if the linker wants intermediate code, it'd be just as easy for compilers to skip the code generation step, create object files of intermediate code, and let the linker do the code generation. That's actually what the Fernandez optimizer described above did. The linker can take all the intermediate code, do a big optimization pass over it, then generate the object code for the output file.

There's a couple of reasons that production linkers rarely do code generation from intermediate code. One is that intermediate languages tend to be related to the compiler's source language. While it's not too hard to devise an intermediate language that can handle several Fortran-like languages including C and C++, it's considerably harder to devise one that can handle those and also handle less similar languages such as Cobol and Lisp. Linkers are generally expected to link object code from any compiler or assembler, making language-specific intermediates problematical.

Link time assembler

An interesting compromise between linking traditional binary object code and linking intermediate languages is to use assembler source as the object language. The linker assembles the entire program at once to generate the output file. Minix, a small Unix-like system that was the inspiration for Linux did that.

Assembler is close enough to machine language that any compiler can generate it, while still being high enough level to permit useful optimizations including dead code elimination, code rearrangement, and some kinds of strength reduction, as well as standard assembler optimization such as choosing the smallest version of an instruction that has enough bits to handle a particular operand.

Such a system could be fast, since assembly can be very fast, particularly if the object language is really a tokenized assembler rather than full assembler source. (In assemblers, as in most othter compilers, the initial tokenizing is often the slowest part of the entire process.)

Load time code generation

Some systems defer code generation past link time to program load time. Franz and Kistler created "Slim Binaries", orignally as a response to Macintosh "fat binaries" that contain object code for both older 68000 Macs and newer Power PC Macs. A slim binary is actually a compactly encoded version of an abstract parse for a program module. The program loader reads and expands the slim binary and generates the object code for the module in memory, which is then executable. The inventors of slim binaries make the plausible claim that modern CPUs are so much faster than disks that program loading time is dominated by disk I/O, and even with the code generation step, slim binaries are about as fast to load because as standard binaries because their disk files are small.

Slim binaries were originally created to support Oberon, a strongly typed Pascal-like language, on the Macintosh and later Windows for the x86, and they apparently work quite well on those platforms. The authors also expect that slim binaries will work equally well with other source languages and other architectures. This is a much less credible claim; Oberon pro-

grams tend to be very portable due to the strong typing and the consistent runtime environment, and the three target machines are quite similar with identical data and pointer formats except for byte order on the x86. A long series of "universal intermediate language" projects dating back to the UNCOL project in the 1950s have failed after promising results with a small number of source and target languages, and there's no reason to think that slim binaries wouldn't meet the same result. But as a distribution format for a set of similar target environments, e.g. Macs with 68K or PPC, or Windows with x86, Alpha, or MIPS, it should work well.

The IBM System/38 and AS/400 have used a similar technique for many years to provide binary program compatibility among machines with different hardware architectures. The defined machine language for the S/38 and AS/400 is a virtual architecture with a very large single level address space, never actually implemented in hardware. When a S/38 or AS/400 binary program is loaded, the loader translates the virtual code into the actual machine code for whatever processor the machine on which it is running contains. The translated code is cached to speed loading on subsequent runs of the program. This has allowed IBM to evolve the S/38 and then AS/400 line from a midrange system with multi-board CPUs to a deskside system using a power PC CPU, maintaining binary compatibility throughout. The virtual architecture is very tightly specified and the translations very complete, so programers can debug their program at the virtual architecture level without reference to the physical CPU. This scheme probably wouldn't have worked without a single vendor's complete control over the virtual architecture and all of the models of the computers on which it runs, but it's a very effective way to get a lot of performance out of modestly priced hardware.

Exercises

How long does the linker you use take to link a fairly large program? Instrument your linker to see what it spends its time doing. (Even without linker source code you can probably do a system call trace which should give you a pretty good idea.)

Look at the generated code from a compiler for C++ or another object oriented language. How much better could a link time optimizer make it?

What info would the compiler have to put in the object module to make it possible for the linker to do interesting optimizations? How badly do shared libraries mess up this plan?

Sketch out a tokenized assembler language for your favorite CPU to use as an object language. What's a good way to handle symbols in the program?

The AS/400 uses binary translation to provide binary code compatibility among different machine models. Other architectures including the IBM 360/370/390, DEC VAX, and Intel x86 use microde to implement the same instruction set on different underlying hardware. What are the advantages of the AS/400 scheme? Of microcoding? If you were defining a computer architecture today, which would you use?