GNU Rope – A Subroutine Position Optimizer

Nat Friedman nat@nat.org

October 1, 1998

1 Introduction

GNU Rope is a portable optimizer for GNU systems. It improves the running time and memory use of an executable by rearranging its subroutines so that related functions are placed on the same page. This paper gives some background information which is necessary to understand how the optimizer works, describes the motivation for and design of the optimizer, discusses in detail several algorithms which have been implemented to determine optimal function orderings, and outlines how you can go about using GNU Rope to optimizer your programs.

The most up-to-date version of this paper is available at:

http://grope.nat.org/

2 Background

2.1 Linking and Binary Formats

2.1.1 Overview

When a program is built, it passes through three stages: compilation, assembly, and linking. The link editor, or **linker** generally operates on multiple files of relocatable machine code, some of which are object files – the results of compilation and assembly – and some of which are libraries. A piece of relocatable code is code whose references to other pieces of code are not absolute addresses. For example, a subroutine `func1` which calls another subroutine, `func2`, but which references `func2` by name and not by its address in memory is a piece of relocatable code. `func1` may be in an object file, and `func2` in a library. During the final link, when an executable is created, it is the job of the linker to find the reference to `func2` in `func1` and "fix up" the reference; the symbolic reference to `func2` must be changed to the address at which `func2` will be loaded when the program is running. The linker performs these fixups for every reference to relocatable data or code that appears in the input files.

2.1.2 Symbols and Relocations

A file's symbolic references to code and data are stored in a two-part system in the linker's input files. Each relocatable object has an associated symbol which references it. The symbol provides a name for the relocatable object and a way to find it – an address or offset. Each reference to an object, such as a function call, references the object's symbol. In this manner, when an object moves, the only address that needs to be updated is the address stored in the object's symbol.

For each reference to a relocatable object, there is an associated piece of information called a **relocation**. A relocation contains three fields. First, it contains the location of the reference – be it a function call, a data access, or a jump. Second, the relocation contains a pointer to the symbol which describes the location of the referenced object. Finally, each relocation has a processor-specific type, sometimes referred to as a **howto**, which tells the linker how to go about resolving this reference. This is described in greater detail later.

2.1.3 Binary Formats

A binary file[1], be it an executable, a library, or an object file, is divided into **sections**. Generally, a file's code is all stored in one section, and its data in one or several other sections. If the file is relocatable, and can be processed by a linker, then the section containing code has a set of relocations associated with it. These relocations are grouped together and stored in their own section, called a **relocation section**. The relocation section is sometimes referred to as a **relocation table**. Similarly, all of the symbols in a file are stored in **symbol tables**, each of which has its own section. Each symbol can be uniquely identified by a handle to its symbol table and an index into that table, and this is how a relocation refers to a symbol.

2.1.4 Dynamic Linking

When a program is dynamically linked, not all of its relocations are resolved by the linker. Instead, references to objects in shared libraries are left for the dynamic linker to fixup at run time. This is very convenient as it allows shared libraries to be changed without requiring that the programs which use the libraries be relinked. Dynamic linking of functions is accomplished with a jump table

[1]The discussion of binary files in this paper is biased towards the Executable and Linkable Format (ELF), discussed later in this chapter. Many of these concepts apply equally well to other binary formats, but bear in mind this bias.

called a **procedure linkage table**, or PLT. The PLT effectively adds another level of indirection to references to dynamic objects.

When the linker builds an executable, all references to dynamic objects are made to point at a jump entry in the PLT[2]. Each dynamicaflly linked function has its own PLT entry. Each address in the PLT is a reference to an entry in another linker-generated table, the Global Offset Table (GOT). The PLT has a relocation section which contains the dynamic relocations for each entry in the PLT. When the program is run, the dynamic linker knows the actual addresses of the dynamic objects, and modifies the entries in the GOT such that, for each dynamic object, that object's entry in the PLT jumps to the object's address.

The PLT is a synthetic section created by the linker. The GNU linker creates the PLT out of C structures like this one:

```
static const bfd_byte elf_i386_plt_entry[PLT_ENTRY_SIZE] =
{
  0xff, 0x25,   /* jmp indirect */
  0, 0, 0, 0,   /* replaced with address of this symbol in .got.  */
  0x68,         /* pushl immediate */
  0, 0, 0, 0,   /* replaced with offset into relocation table.  */
  0xe9,         /* jmp relative */
  0, 0, 0, 0    /* replaced with offset to start of .plt.  */
};
```

2.1.5 Executable and Linkable Format (ELF)

The Executable and Linkable Format was designed by UNIX System Laboratories to serve as a portable object file format for a variety of operating systems and architectures [UNI].

ELF Sections

An ELF object file is broken into a number of sections. Each section has an associated number – an index into a table of section headers – a name, and a type. ELF section names are provided largely for convenience, as sections are never referenced by name and a section's format can be determined by its type. For simplicity, sections are referred to by name here, but bear in mind that section names are not at all deterministic. Table 1 below describes the sections in a typical object file, `elftools.o`, created as part of the build process for the optimizer described in this paper.

Every ELF file has a dummy section, section zero. The `.text` section contains the actual compiled program code generated from `elftools.c`. In the table above, it is relocatable code, and the section containing its relocations is `.rel.text`. An ELF file can contain multiple code sections, e.g. `.text1` and `.text2`, whose relocation sections would be named correspondingly: `.rel.text1` and `.rel.text2`. The `.data` section contains the object file's initalized data, and `.bss` contains the program's uninitialized data; `.bss` uses no space in the file, and when the program is loaded, this section is initalized to all zeroes. The `.note` section stores miscellaneous information used by some program vendors to ensure compatibility between object files. Similarly, the `.comment` section holds version control information – usually a string identifying the compiler which built the object file[3]. Finally, `.symtab` contains the file's symbol table and `.strtab` contains their associated names.

ELF Relocations

[2]Please note that the behavior described here is specific to the Intel x86 PLT behavior; on other architectures, it differs in mechanism but not in intent.

[3]The `.comment` section in `elftools.o` contains the string "GCC: (GNU) 2.8.1"

Number	Name	Type	Offset	Size	Address	Alignment
0	dummy	none	0x00000	0	0x0	0
1	.text	progbits	0x00034	6134	0x0	4
2	.rel.text	rel	0x02cd8	3168	0x0	4
3	.data	progbits	0x0182c	36	0x0	4
4	.bss	nobits	0x01850	0	0x0	4
5	.note	note	0x01850	20	0x0	1
6	.rodata	progbit	0x01864	2630	0x0	1
7	.comment	progbits	0x022aa	20	0x0	1
8	.shstrtab	strtab	0x022be	77	0x0	1
9	.symtab	symtab	0x024c4	1168	0x0	4
10	.strtab	strtab	0x02954	898	0x0	1

Table 1: The sections in a typical ELF object file, `elftools.o`

Each relocation section has a header which specifies the section being relocated; `.rel.text` relocates `.text`, for instance. The section being relocated is specified by its index into the table of section headers. Additionally, each relocation table refers to a symbol table; whenever a symbol is referenced by index, it can be found in the specified symbol table. The symbol table is also specified by its index into the table of section headers. Table 2 contains a representative subset of the relocations in `.rel.text` in our sample object file, `elftools.o`.

A relocation's offset is the offset into the section being relocated – the target section. Relocation

Index	Offset	Type	Addend	Symbol	Symbol Name
0	0x0c	R_386_PC32	-4	10	elf32_getshdr
1	0x1f	R_386_PC32	-4	11	elf_errno
2	0x27	R_386_PC32	-4	12	elf_errmsg
3	0x32	R_386_32	0	6	.rodata
4	0x37	R_386_32	0	13	_IO_stderr_
5	0x3c	R_386_PC32	-4	14	fprintf

Table 2: Sample Relocation Table

5 in the table above, for example, describes a reference to a call to the function `fprintf`. The relocation's offset is at `0x3c`. The assembly code in that region of `.text` is:

```
31:   68 00 00 00 00  pushl  $0x0
36:   68 00 00 00 00  pushl  $0x0
3b:   e8 fc ff ff ff  call   0xfffffffc
```

The five bytes at offset `0x3b` contain an unresolved function call. On the Intel x86 architecture, a function call consists of the CALL opcode, `0xe8`, followed by a four-byte little-endian PC-relative address. The relocation's offset, `0x3c`, is the section-relative offset of the function address. When the linker performs the final link, it walks the list of relocations and calculates each reference's target address by consulting the symbol table entry for the referenced symbol – in this case, `fprintf`. In the case of a PC-relative relocation (R_386_PC32), the new address is the symbol's address plus the relocation's addend minus the address of the data being relocated. For a non-PC-relative relocation, the address is simply the symbol's address plus the relocation's addend. There are two ways of storing a relocation's addend in ELF. The first is to store it in the relocation table along with all of the other fields. Relocation entries whose addends are stored this way are referred to as having explicit addends. The second way to store a relocation's addend is to embed it in the code being relocated. This is known as an implicit addend, and can save a great deal of space in large programs. In this example, the addend is implicitly embedded in the code being relocated; `0xfffffffc` is the twos-complement binary notation for -4. Finally, the two PUSHL calls which precede the function call push the function's arguments onto the stack. Consulting the relocation table reveals that these arguments can be found at the addresses of symbols `.rodata`[4] and `_IO_stderr_` – clearly this section of code prints an error message[5].

2.2 Program Loading and Execution

When a program is executed, the operating system loads parts of it into memory. Generally, each section in the executable is associated with a virtual memory address, or VMA. The contents of the section are loaded into the address specified by that section's VMA, which was determined by the linker during the final link. Not every section is loaded into memory at runtime. For example, if a binary contains bugging information – the information used by a debugger to determine, for instance, which line of source code is being executed – this information is not loaded into memory when the program is run.

Furthermore, an entire section is not necessarily loaded into memory at once. Instead, in some binaries, the data and code in the binary is loaded into memory in blocks. A full block is loaded into memory whenever any code or data in that block is needed; a block is the smallest amount of space that can be loaded into memory at once. These blocks are referred to as pages, and such a binary is referred to as being **demand paged**.[6]

Demand paging is a convenient operating system feature because, during execution, the entire binary does not need to be resident in memory at once. This greatly reduces program start-up times and resource consumption. However, in a demand-paged executable, the smallest unit of data which can be loaded from disk to memory is a page, despite the size of the data type being accessed. On an Intel Pentium running Linux, for example, if a subroutine must be loaded into memory, an entire page – 4096 bytes – will be loaded, no matter the size of the subroutine in question. Each time a page must be loaded, a page fault is generated, and the data is transferred from disk. Clearly, the fewer pages that are loaded, the better, as less time is wasted transferring data between disk and memory. The optimizer described in this paper reduces the number of loaded pages by increasing the density of in-use subroutines on each page. This is described in greater depth in the next chapter.

2.3 GNU binutils

The GNU binutils package is a set of tools which handles the last two stages of program building – assembly and linking. GNU binutils also provides a set of miscellaneous tools for manipulating binaries and libraries. The tools in the binutils package do not read and manipulate binaries directly; this functionality is abstracted in a library called the BFD[7]. The BFD library is supposed

[4]This is a *section symbol* – a symbol which refers to the beginning of a section in an ELF file.

[5]In fact, this code was assembled from the C statement `fprintf(stderr, ``Error getting symbol table header: %s\n'', ...);`

[6]Demand paging does add some complexity to the linker's task. Because data is loaded in from each section one page at a time, and each section has both an associated VMA and file offset, the linker must ensure that section offset modulo the file offset is the same as the target virtual address modulo the section's VMA. Because the linker determines VMA's during the final link, this is usually not difficult, but makes reordering preblematic, as discussed later.

[7]It is an in joke among linker hackers that the acronym "BFD" does not stand for anything in particular, or, if it does, its meaning has been lost in time. **Binary Format DLL** is probably the most appropriate non-obscene

to be generic; that is, it has an internal binary format representation and interchangeable back-ends to support different binary formats. The BFD's internal representation is referred to as "canonical," as it is supposed to be able to represent every binary format for which there is a back-end. In this way, programs which use the BFD are completely portable between architectures. In theory, this is a good idea, even allowing a program to manipulate object files of multiple types at once. In practice, however, the BFD library's internal representation is not flexible enough to handle all the binary formats in existence, and so format-specific code ends up in a lot of places in the GNU linker and the other tools. The result is that the GNU linker and the BFD are inextricably intertwined, and, at times, very difficult to decipher.

2.4 Profiling

When a program is profiled, data is collected about the program's behavior during its execution. Most profilers, notably **gprof**, primarily record data about the program's pattern of subroutine invocation. Specifically, for each subroutine, a **call count** is recorded, as is the total amount of CPU time spent executing that subroutine. When one subroutine invokes another, there is an **arc** between them, and so for each pair of subroutines, an **arc count** is recorded. This is the number of times one of the pair is invoked by the other. Additional information, such as the program's resource usage, can be recorded as well.

With **gprof**, the GNU profiler, profiling is a simple task divided into three steps. The program to be profiled must be recompiled with profiling turned on. The program is then linked with the GNU profiling library routines, known collectively as **gmon**, which contain the measurement functionality. The result is an *instrumented* binary – one with embedded instructions which monitor its operation and performance. At the start of the program's execution, the profiling startup routine, **monstartup()**, is invoked. This routine initializes the data structures that will record the profiling data and sets a timer which issues the **SIGPROF** signal to the program at fixed intervals. Whenever the program receives this signal, it records which subroutine it is executing, and in this manner is able to keep track of how long it spends executing each subroutine. Also, when the program is compiled with the profiling options enabled, the compiler emits calls to the profiling function **mcount()** at the start of each function. Thus, whenever a function is called, **mcount()** determines the calling function and updates the invocation counts and arc invocation counts appropriately. When the program terminates, it writes the profiling buffers to disk, in a file usually called **gmon.out**. A separate program, **gprof**, reads this file and outputs readily-parseable information about the program's run-time behavior.

I have implemented an extension to this scheme which makes it easy to plug new measurement facilities into **gmon**. Different profiling modes can be activated at run-time via an environment variable which **monstartup()** reads when the program starts. The environment variable, **PROFMODES**, is set, and then the program is run, as in the example below:

```
% export PROFLIST="pchist callgraph seqgraph memuse"
% ./instrumented-program
```

This sort of functionality was necessary for me to add routines which gather new types of profiling data which **grope** needed to compute function orderings. This is all discussed in greater detail later.

expansion of the acronym.

3 The Motivation for GROPE

The GNU Rope optimizer reorders the functions in an executable so that those functions which are often in memory together are on the same page. The result is an executable with a higher per-page density of in-use subroutines. This means that fewer subroutines need to be loaded from disk into memory; this will reduce program load times. Furthermore, the number of in-use pages is also reduced. This is a reduction of the program's *working set*. A reduction in a program's working set will result in better cache mapping and reduced memory usage. The reduction in disk I/O and the improved cache hit rate yields much improved performance.

The optimizer's overall design was based upon my idea of how it would be used: Ideally a program vendor could distribute an executable which each user could optimize for his particular usage patterns. The user would instrument the binary, profile it while performing whatever actions he wanted to be particularly fast, and then reorder the executable based on the data gathered during profiling. The result would be an executable tailored to the usage pattern of the profiling run. For example, given a large program such as a web browser, a user may only use a small subset of its functionality. Some users, for example, may use a great deal of Java and others may not. Each user could tune his web browser to his particular style of usage by profiling the program while using the functionality he wants to be fastest.

A user's particular style of usage will affect the pattern of subroutine invocation which characterizes the execution of the program; a frequent user of Java will generate many more calls to the browser's Java routines, and so, for this user, the Java subroutines should be probably be intermingled with the rest of the browser's code, while this would be a suboptimal situation for other users.
It was necessary to design a scheme which would allow the user this sort of flexibility with the least possible negative impact on performance. Two tools were designed. The first is a link-time optimizer which directs the linker to arrange the functions in the executable which it is creating in an optimal pattern. The second, more complex tool is a post-link optimizer which rearranges the functions in an executable after it has been linked. The following goals were set.

1. **It must be possible to insert profiling stubs into an executable post-link.** Ordinarily, when a program is instrumented for profiling, it is rebuilt and relinked using the original source code. Often, however, vendors do not wish to distribute their source code, and so it must be possible to instrument an executable in the absence of its source code and object files. Additionally, the uninstrumented, unoptimized executable must run no slower than the original; that is, whatever is done to make it possible to instrument the executable post-link must not adversley affect performance. It is unavoidable that gathering profiling data will consume system resources, and so it is acceptable for the executable to run slower when it is being profiled.

2. **It must be possible to locate the subroutines in an executable post-link.** In order to gather meaningful profiling data, and to reorder the executable, it is necessary that the executable's subroutines be demarcated in some way, such that they can be found and manipulated.

3. **It must be possible to determine an optimal or near-optimal subroutine ordering.** Once profiling data is gathered, it must be analyzed, and an optimal ordering of subroutines determined. An algorithm must be designed for this task.

4. **It must be possible to reposition the subroutines in an executable post-link.** Finally, to optimize the executable, its subroutines must be positioned according to the ordering determined using the profiling data. This requires updating all function and symbol references – essentially it requires that the executable be relinked in the absence of the object files with which it was built. This is the core of the post-link optimizer.

4 Ordering Algorithms

The GNU Rope optimizer rearranges the functions in an executable to improve the program's speed and reduce its memory use. I have shown that a good ordering of functions is one which requries that the fewest number of pages be in memory at any given time. But, how do we go about producing such an ordering? In this section, I describe a number of the algorithms which **grope** supports. First, I discuss a new type of profiling data which I have devised and which I call a *sequence graph*. Sequence graphs are used by several of the ordering algorithms discussed below. Next, I describe the *node grouping* algorithm, which makes use of this sequence graph data. Then, I describe an ordering algorithm written by Jeff Law which is part of **gprof**. Finally, I discuss simulated annealing and how I have applied it to the problem of determining function orderings.

4.1 Goals

Before an ordering algorithm can be devised, we have to have some idea as to what we want it to accomplish. First, straight off, we know that we want our algorithm to minimize the number of page faults that a program generates. We would like it to do this on machines with ample memory to keep the whole program resident, but also on machines that might have to swap parts of the program out from time to time. That accomplished, it would be extra nice if our ordering algorithm could take into account the behavior of the computer's cache. Naturally, some cache-friendliness comes "for free" by virtue of minimizing page faults, but some special consideration can be applied as well. Finally, some machines have interesting hardware which could be accounted for as a special case. As an example, the Ultrasparc has a one-line quick cache for instruction memory, and so when a heavily-utilized loop straddles a page boundary, this quick cache is mainly useless. Shifting the loop so that it is only on one page would be a great accomplishment for our algorithm.

4.2 Sequence Graphs and Node Grouping

4.2.1 Call Sequence

The standard profiling data which **gprof** gathers consists of a histogram and a call graph. This is a fundamental problem, since it does not provide enough information to produce optimal function orderings. Often times, knowledge about the *sequence* of function calls is necessary to produce an optimal ordering, and normal call graphs do not contain this information. If an ordering algorithm only uses a call graph, functions which are called near the beginning of a program's execution can get intermingled with functions which are only called near the end of the program's execution – a suboptimal situation.

The solution to this problem is to use information about the sequence of function accesses to determine an ordering. For example, consider the following piece of code:

```
void a(void)
{
  int i;

  for (i = 0 ; i < 10 ; i++)
    {
      b(); d();
    }
for (i = 0 ; i < 10 ; i++)
  {
    c(); e();
  }

}
```

This code will generate the following sequence of function accesses:

```
a b a d a b a d a b a d a b a d a b a d a b a d a b a d a b a d a b a
d a b a d a c a e a c a e a c a e a c a e a c a e a c a e a c a e a c
a e a c a e a c a e a
```

Note that these are function accesses, not just function calls; a function must be resident when one of its children returns to it, not just when it is called. It is clear from the sequence listed above that functions **b** and **d** should be placed together, and that they should not be intermingled with **c** and **e**; something that would not be clear from a normal call graph.

Information about the sequence of calls can be embedded into the same graph data structure that callgraphs use if the meaning of the arcs between graph nodes is changed. In typical call graphs, an arc between **a** and **b** indicates that **a** calls **b** or that **b** calls **a**[8] This meaning can be changed such that an arc between functions **a** and **b** indicates that **a** and **b** are accessed near each other in the function sequence. This embeds sequence data into a call graph-like structure.

To generate such a graph, we section off a 'window' of the function access sequence like this:

```
--- start ---
a
b
a
d
--- end ---
a
b
a
d
a
b
a
d
```

You can think of this window as representing the program's working set at a given time. For each window position, increment the weights of the arcs between all functions which are together in a window. In the example above, the following arcs are strengthened, or created if they do not already exist: (**a**,**b**), (**a**,**d**), and (**b**, **d**). Then, as the program executes, slide the window down:

```
a
--- start ---
b
```

[8]This description is specific to *undirected* call graphs, and **gprof** happens to use a directed callgraph, in which calls from **a** to **b** are distinguished from calls from **b** to **a**. The difference is not particularly significant to this discussion, however.

```
a
d
a
--- end ---
b
a
d
a
b
a
d
```

And increment the weight of the arcs between all of the functions in the window: (**a**, **b**), (**b**, **d**), (**a**, **d**). And so on, until the program has finished executing. Applying this method to the above program produces the graph shown in Figure 1.

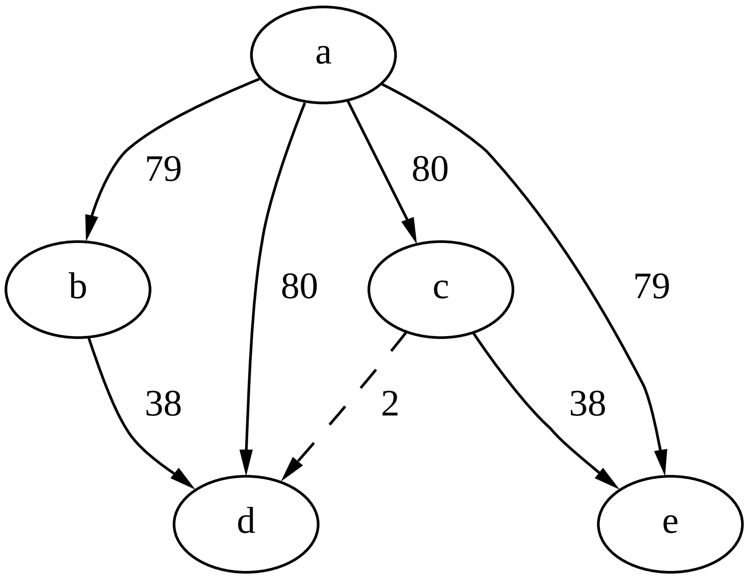

Figure 1: A Simple Sequence Graph

I call this new type of grpah a *sequence graph*, because it contains sequence rather than call data[9]. Using the new profiling framework describe above, I implemented a sequence graph gathering mode for **glibc**, and have made use of this sequence data in several function ordering algorithms as described below.

4.2.2 Node Grouping

Once you have a call graph or a sequence graph, how do you use it to determine a function ordering? One algorithm that answers this question is outlined in a paper by Pettis and Hansen [PH90]. The algorithm which they described, and which I call *node grouping*, applies equally well to call graphs and sequence graphs. It is therefore somewhat useful to think of the weight of an arc between two functions as specifying the *affinity* each function has for the other, since the graph could be either

[9]Note that a call graph is basically the degenerate case of a sequence graph, with a window size of two.

a call graph or a sequence graph, or some type of graph yet to be devised. The node grouping algorithm works as follows.

1. Sort all of the arcs in the graph.
2. Find the arc with the highest weight. Take the two nodes which make up this arc and combine them. If the arc with the highest weight is (**a**, **b**), then the new, combined node is **a,b**. Any other arcs which connected to **a** and **b** should be added together and should now connect to this new node **a,b**.
3. If there is only one node left, we're done. Otherwise, go back to the first step.

Using the sequence graph from the last section as an example, the intermediate steps of the node grouping algorithm are shown in Figures 2 through 6. The final graph, shown in Figure 6, contains only one node, **a,c,e,d,b**, and the name of this node is the function ordering generated by this algorithm.

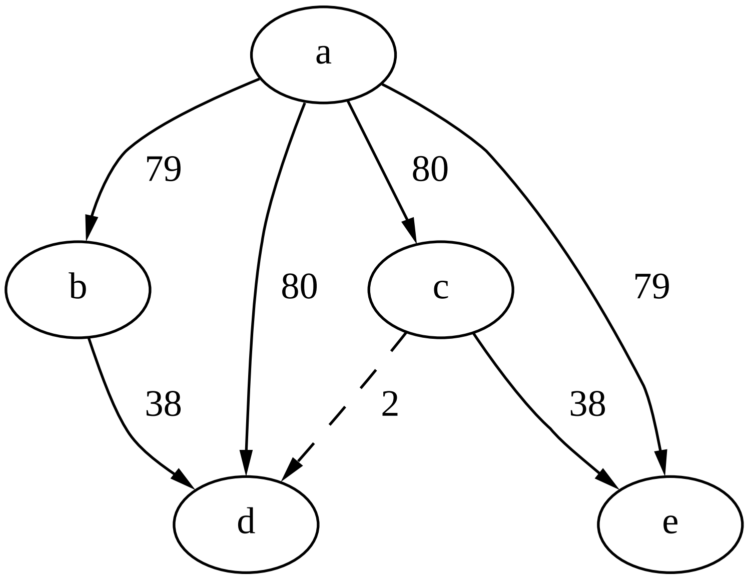

Figure 2: The Initial Graph

4.3 The gprof Algorithm

The GNU profiling tool, gprof, can use the callgraph data in a normal gmon.out file to compute a suggested function ordering. This algorithm was designed by Jeff Law, and it works as follows:

1. All unused functions are grouped together at the end of the function ordering.
2. The arc list is sorted, and the highest-used arcs are extracted. Of the functions in the list of most-used arcs, those which have at least five callers are grouped together, and these are placed at the start of the function order.
3. The functions which occur in the top 99% of the arcs – that is, all of the functions except those which are only very rarely called – are grouped together using a greedy placement algorithm. This is then repeated for all of the remaining, unplaced functions.

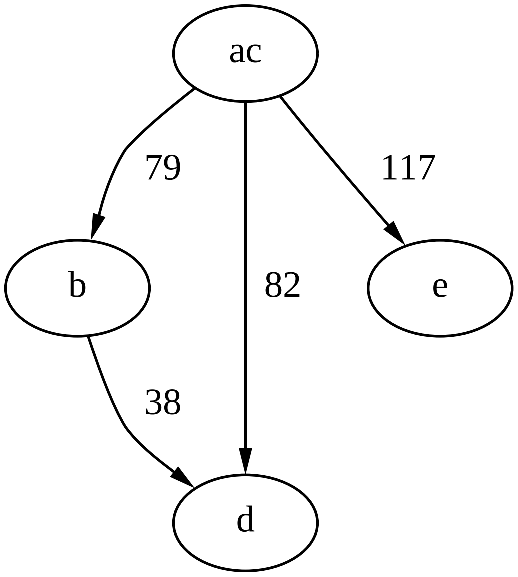

Figure 3: After the First Grouping

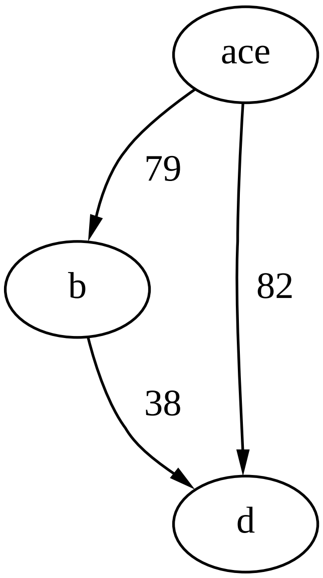

Figure 4: After the Second Grouping

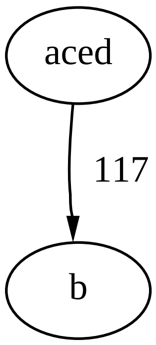

Figure 5: After the Third Grouping

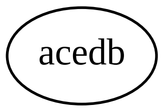

Figure 6: After the Final Grouping

First, by grouping all unused functions separately, this algorithm automatically generates the fewest possible number of page faults on those machines which have enough memory to keep the entire program resident. The other algorithms, like simulated annealing, achieve this effect eventually, but handling this explicitly is a good time-saving trick.

Second, grouping the often-called functions together at the beginning of the executable establishes a set of pages in the executable which are almost always resident. Finally, the greedy algorithm attempts to place chains of functions contiguously, so that if **a** calls **b** and **b** calls **c**, all three functions form a chain which the gprof algorithm will attempt to place together in the final ordering.

This algorithm does a very solid job, largely due to the greedy algorithm which places function chains contiguously in the executable. As discussed later, when simulated annealing is described, this **gprof** algorithm actually has found a local minimum in the space of function orderings. Also, although this algorithm was designed to use call graph data, it also performs quite well using a sequence graph instead of a call graph.

4.4 Simulated Annealing

Simulated Annealing is a powerful optimization algorithm which has become popular relatively recently for optimizing systems with many local minima. It has had particular success with the travelling salesman problem, a very difficult and normally very computationally expensive NP-complete problem of path length minimization.

4.4.1 Annealing

Annealing is the process of heating a material, holding it at a high temperature, and then slowly cooling it. In metallurgy, annealing is often performed on metals to improve their ductility and reduce their brittleness. Metals are composed of small crystals known as “grains,” of varying sizes and shapes, randomly distributed throughout the metal. The place where two grains meet is known as a “grain boundary,” and depending on the configuration of the metal, there can be different levels of stress at this boundary. When the metal is annealed, the grains in the metal shift around. If the metal is cooled slowly enough, the resulting configuration of grains generally has the minimum cumulative stress at the grain boundaries. Annealing can be thought of as nature’s optimization process, minimizing the potential energy of the grain boundaries in a metal.

4.4.2 Simulated Annealing Overview

The simulated annealing algorithm is based upon the details of the way annealing works. It is a randomized algorithm which attempts to minimize a system's energy. For any particular system, its energy can be whatever you define it to be, and in this way simulated annealing can be applied to a wide variety of problems.

The way the algorithm works is this: Simulated annealing operates on a given system. That system may be a configuration of grains in a metal, or a given path in the traveling salesman problem, or, in our case, a particular ordering of functions. The algorithm uses a function which operates on the system and randomly tweaks its parameters slightly. The resulting system is a random nearby system in the system space. Then, the energy of the new system is calculated. If this new, tweaked system is of lower energy than the starting system, the algorithm switches to it. This is analagous to a downhill step.

Clearly, if this were all that there is to simulated annealing, the algorithm would just quickly converge upon the nearest local minimum. If the new system is of higher energy, the algorithm will sometimes switch to it, with a probability dictated by the Boltzmann Probability Distribution,

$$Pr[switch] = e^{\frac{-\Delta E}{T}} \tag{1}$$

In the above equation, ΔE is the difference in energy between the new system and the original system, and T is the system's temperature. This temperature is a variable which starts very high and is slowly lowered as the algorithm runs. The rate at which the temperature changes is dictated by the annealing schedule, and this is one of the algorithm's tunable parameters. At high temperature, there is a higher chance of taking an uphill step than at low temperature.

Simulated annealing is not an algorithm which can be easily analyzed, but the intuition for how it works suggests that, at high temperature, the system does not settle; its relatively high probability of stepping uphill makes it avoid local minima. Later in the algorithm, when the system is at lower temperature, you can think of the algorithm as settling into what should be the global minimum.

4.4.3 Computing Function Orders

Applying simulated annealing to the problem of computing function orders boiled down to solving three problems. First, it was necessary to design a function which would compute the energy of an ordering. This would be the quantity that the algorithm would minimize. Second, the "step" function would have to be devised; this is the function which randomly tweaks a system to produce a nearby system. Finally, an appropriate annealing schedule had to be determined.

4.4.4 The Energy Function

Ideally, to determine the quality of a function ordering, we would apply the ordering to an executable, run the executable, and measure its performance. This is completely impractical, however, since simulated annealing will want to test thousands of candidate orderings before settling on an optimal one. So, instead, the *energy function* uses profiling data to heuristically determine the quality of an ordering.

The energy function is the sexy part about simulated annealing, because it allows you to make the algorithm account for the peculiarities of the machine for which you are optimizing simply by writing them into the algorithm. For example, the Ultrasparc's one-line quick cache can be handled by assigning a higher energy to any ordering which makes a heavily-used loop straddle a page boundary. For machines like the StrongARM, which has a virtually addressed cache, you can write a special energy function which takes into account the mechanisms of the machine's cache.

The energy function which I devised is intended to directly relate the number of in-use pages at any given time during the course of the program's execution. One way to do this is to simulate running the program with a given ordering, count up the average working set size at any given time, and use this as your energy. You can run a simulation like this one using special profiling data, such as a function access sequence described above.

This is, however, very slow, and the energy function is something which has to be called a lot during the optimization process. What I do instead is use the sequence graph, as discussed above, to roughly approximate this quantity. The algorithm is represented in pseudocode below:

```
cost = 0
For each arc <node1, node2, weight> in the sequence graph,
  p = number of unique pages touched by functions node1 and node2
  cost = cost + (p - 1)*weight
```

Recall that the weight of the arc between two nodes is the number of times that those nodes appear together in the sliding window of function calls. That number will therefore be higher if there is a penalty for putting these two functions on separate pages. If the weight of the arc is very high, then when the simulated annealing algorithm step function moves the two functions onto the same page, the energy of the resulting ordering will be much lower. Note that this energy function also takes into account the fact that it is undesirable to have heavily-used functions straddling pages.

4.4.5 The Random Step

The step function which I use is somewhat based on the step function used for the travelling salesman problem. The types of steps that the algorithm can take are the following:

1. Choose a random segment of the ordering and randomize the order of all of the functions in that segment.
2. Choose a random segment of the ordering and move it between two random functions.
3. Choose two functions randomly and swap their positions.
4. Choose two functions randomly and swap their positions, several times in a row.

The one type of step which is conspicuously missing here is one which adds a hole between two functions; that is, advances the position of one function slightly. This could be very useful to keep heavily-used functions from straddling page boundaries.

4.4.6 The Annealing Schedule

The *annealing schedule* is the method used to determine when and by how much to lower the temperature of the simulated annealing algorithm. Typically, trial and error is used to tune the parameters of the annealing schedule. What I found in experimenting with different rates of cooling was that the minima tended to be very deep, and this required a long cooling time. There are a lot of local minima, too, however, and so the annealer needs to run at a moderately high temperature for a while too. If the temperature is too high for too long, however, the algorithm will take too many uphill steps – the downhill steps being so much more scarce – and ends up wasting a lot of time.

It is interesting to note that, when the ordering generated by the **gprof** algorithm is fed into the simulated annealer running at a very low temperature (i.e. no uphill steps. I call the mini-program that generates orderings with this sort of annealing schedule "dive."), the annealer reports that it is at a local minimum and that it cannot take any downhill steps.

Acknowledgments

Many thanks to Miguel de Icaza (miguel@kernel.org) for his constant encouragement and support. Thanks also to Richard Henderson (rth@cygnus.com), Toshi Morita (tm2@best.com), Andi Kleen (ak@muc.de) and Ulrich Drepper (drepper@cygnus.com) for their knowledgable advice. Finally, thanks to Matt Loper (matt@loper.org), Mike Shaver (shaver@netscape.com), Wim Coeaker (wcoekaer@us.oracle.com), and Matthew Wilcox (willy@bofh.ai) for their invaluable help with this paper.

References

[PH90] Karl Pettis and Robert C. Hansen. Profile Guided Code Positioning. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, 1990.

[UNI] UNIX Systems Laboratories Portable Formats Specificatoin. *Elecutable and Linkable Format*, Version 1.1 edition.