Link Optimization

This section describes two techniques for improving application performance by optimizing the link order of the Mach-O executable file. These techniquesÐobject file reordering and procedure reorderingÐboth work to improve your application's locality of reference. Either technique can improve an application's response time and memory use. Procedure reordering provides finer granularity and greater optimization, but requires profiling and relinking the application after it is already working. While the two techniques are compatible, procedure reordering takes precedence over object file reordering.

Object file reordering is an Project Builder feature that lets you change the order in which object files are linked. Using this technique, you can position source files used most frequently by your application so that they are linked into the executable first; you can also position files that refer to one another contiguously in the executable.

Procedure reordering is done by specifying to the link editor an exact ordering of the routines in your program. The technique for procedure reordering is discussed in detail in the remainder of this section.

Procedure Reordering

Reordering the procedures in your executable can boost your application's performance substantially, by reducing average memory use and paging activity. Procedure reordering places the blocks of code for individual methods or functions in an optimized order, independent of the source file they came from or their

position in the source file. When procedures are reordered, the operating system can generally keep an active application's most frequently called code paged into active memory. You can also use procedure reordering to place all procedures for discrete tasks into contiguous sequences of the executable.

Note: Procedure reordering is generally performed only after an application is debugged and essentially ready to use, since further development (i.e., adding new routines or changing which routines call what other routines) can affect the optimal procedure order.

Segments, Sections, and Procedure Order

Mach-O executables are organized by segments of memory. Each segment can contain one or more sections. Segments are always page aligned. The sections of your executable are created from object files by the link editor.

There are several standard segments defined by the Mach-O file format (described in *NeXT Development Tools*). The **__TEXT** segment holds the actual code and other read-only portions of your program. The link editor places code from your ª.oº files in the **__text** section of the **__TEXT** segment. As your program runs, pages from the **__text** section are loaded into memory on demand, as routines on these pages are used. Code is linked into the **__text** section in the order in which it appears in the source file, and source files are linked in the order in which they are listed on the link editor command line (or in Project Builder's Files display, if you use Project Builder's object-file reordering feature). Thus, code from the first ª.oº file is linked from start to finish, followed by code from the second and subsequent files.

However, this is rarely the optimal order. For example, say that certain methods or functions in your code are invoked repeatedly, while others are seldom used. Reordering the procedures to place high hit-rate code at the beginning of the **__text** section minimizes the average number of pages that your application uses, and reduces

paging activity. As another example, say that all the objects defined by your Objective C code are initialized at the same time. Because the **initialize** method for each class is defined in a separate source file, the initialization code is ordinarily distributed across the **__text** section. By reordering initialization code for all classes to place it contiguously, you enhance initialization performance: The application requires just the small number of pages containing initialization code, rather than a larger number of pages, each containing a small piece of initialization code.

The Basic Reordering Process

Depending on the size and complexity of your application, there are different strategies you can pursue for finding a procedure ordering that improves your performance. Like most performance tuning, the more time you spend measuring and re-tuning your procedure ordering, the more memory you will save. A good first cut ordering can be easily obtained by running your application and sorting the routines by call frequency. The steps for this simple strategy are:

1. Compile source files with the **-pg** and **-g** options. This generates an executable containing symbols used in profiling and reordering procedures.

2. Run and use the application to create a set of profile data. Perform a series of functional tests, or have someone use the program for a test period.

3. Run **gprof** with the **-S** option to create order files, listing procedures in optimized order. Order files are used by the link editor to reorder procedures in the executable.

4. Run the link editor using the **-sectorder** option and the order file. This creates an executable with

procedures linked into the __**text** section as specified in the order file.

These steps for this basic ordering are described in detail in the following sections. Afterwards, some more advanced strategies are discussed.

Step 1: Compile and Link the Source Files

Compile and link your program using the **-pg** and **-g** options. The **-pg** option adds hooks for profiling the application and for listing procedure calls in the order file (step 2). The **-g** option creates symbol tables with source file references for use with the debugger; this option is used by the **gprof -S** option to add source file names to the order file (step 3).

The easiest way to do this is to make "clean" and then make "profile" from Project Builder's "Build" panel.

If you want to reorder library procedures along with those in object files, use the link editor's **whatsloaded** option to create a file of all loaded procedures in the project directory. This option is described below in the section ªCreating a Default Order File.º

If your application contains assembly language code, there are some additional contraints to how you may reorder your code. See the appendix at the end of this document for more information.

Step 2: Run and Use the Application

By running and using the application, you can generate a profile of procedures called during use. The profiling data is placed in a file named **gmon.out** automatically, each time you run the program. Note that each time you stop the program it creates a new **gmon.out** file, overwriting the old file. Thus you should rename the file before

restarting the program for the next pass of profiling.

The simplest way to profile is an application is to exercise it through a test suite, or to let a user put the program to use on a daily basis for a few days. These techniques will generate large sets of profile data that will be used to generate an procedure ordering file. In more advanced strategies (see below), you might profile particular operations to get many profile data sets, generate a number of procedure orderings from these sets, and then combine those orderings into a final ordering.

Step 3: Run gprof to Create Order Files

An order file contains an ordered sequence of lines, made up of a source file name and a symbol name, separated by a colon (ª:º) with no other whitespace; this format should be followed exactly for the link editor to process the file correctly. If the object file *name*:*symbol* name pair is not exactly the name seen by the link editor, it will try its best to match up the names with the sources being edited. Each line represents a block to be placed in a section of the executable.

The lines in an order file for procedure reordering consist of an object file name and procedure name (function, method, or other symbol). The sequence in which procedures are listed in the order file represents the sequence in which they will be linked into the __**text** section of the executable.

To create an order file from the profiling data generated by using a program, run **gprof** using the **-S** option. For example,

```
gprof -S MyApp.profile/MyApp gmon.out
```

Ths -S option produces two order files:

gmon.order ordering based on a ªclosest is bestº analysis of the profiling call graph.

callf.order ordering based on call frequency.

You may want to try using each of these order files, to see if either provides a performance improvement. See the discussion of **filemem** below for measuring the results of the ordering.

These order files contain only those procedures which were used during profiling. However, the link editor keeps track of missing procedures and links them in their default order after those listed in the order file. Library functions are included only if the project directory contains a file generated by the link editor's **whatsloaded** option; use of this option is described later in this section, under the topic ªCreating a Default Order File.º

The order file will be missing filenames for files not compiled with the **-g** option, assembly files, and stripped executable files. If your order file contains such references, you must either edit the file to add the filenames or delete the references so the procedures can be linked in default order.

Note that the **gprof -S** option doesn't work with executables that have already been reordered.

Linking a Whole Source File. If you don't want to reorder the procedures in a source file, you can link the file as a whole with a simple edit of the order file, replacing all references to the source file with a single entry using the special symbol **.section_all**. For example, if the object file **foo.o** comes from assembly source and you want to be link it as a whole, then delete all references to blocks in **foo.o** and insert the following line in the order file :

```
foo.o:.section_all
```

This option is useful for object files compiled from assembly source, or for which you don't have the source.

Step 4: Relink the Program

Once you've generated an order file, you can relink the program using the **-sectorder** and **-e start** options:

```
cc -o outputfile inputfile.o ... -sectorder __TEXT __text orderfile -e start
```

Note: If any *inputfile* is a library (rather than an object file), you may need to edit the order file before linking to replace all references to the object file with references to the appropriate library file. Again, the link editor does its best to match names in the order file with the sources it is editing.

With these options, the link editor constructs the executable file *outputfile* so that the contents of the **__TEXT** segment's **__text** section is constructed from blocks of the input files' **__text** sections. The link editor arranges the routines in the input files in the order listed in *orderfile*.

As the *orderfile* is processed, procedures whose object-file/symbol-name pair isn't listed in *orderfile* are placed last in *outputfile*'s **__text** section. These symbols are linked in the default order. Object-file/symbol-name pairs listed more than once always generate a warning, and the first occurrence of the pair is used.

By default, the link editor prints a summary of the number of symbol names in the linked objects not in orderfile, the number of symbol names in orderfile not in the linked objects, and the number of symbol names that it tried to match that were ambiguous, if such entries exist. To produce a detailed list of these symbols, use the **-sectorder_detail** option.

The **ld** option **-e start** preserves the executable's entry point. The symbol start (no leading ª_º) is defined in **crt0.o**; it represents the first text address in your program when you link normally. When you reorder, you have to use this option to fix the entry point. Another way to do so is to put the line **/lib/crt0.o:start** or **/lib/crt0.o:section_all** as the first line in your order file.

More Advanced Reordering Strategies

For many applications, the ordering generated by the steps above are a substantial improvement over doing no ordering at all. For a simple application with few features, such an ordering represents most of the gains to be had by procedure reordering. However, larger applications will often benefit greatly from some additional analysis. While the orderings based on call frequency or the call graph are a good start, your knowledge of the structure of your application and the way that it is used can further reduce the working set of your application.

Creating a Default Order File

If you want to reorder an application's procedures using techniques other than those described above, you may want to skip the profiling steps and just start with a default order file that lists all the routines of your application. Once you have a list of the routines in suitable form, you can then rearrange the entries by hand or with a sorting technique of your choice. You can use the resulting order file with the link editor's **-sectorder** option as described in Step 4 above.

To create a default order file, first run the link editor with the **-whatsloaded** option:

```
cc -o outputfile inputfile.o -whatsloaded > loadedfile
```

This creates a file, *loadedfile*, listing the object files loaded in the executable, including any in libraries (the **whatsloaded** option can also be used to make sure that order files generated by **gprof -S** include library procedures).

Using the *loadedfile*, you can run **nm** with the **-onjls** options and the **__TEXT __text** argument:

```
nm -onjls __TEXT __text `cat loadedfile` > orderfile
```

The contents of the file *orderfile* is the symbol table for the text section. Procedures are listed in the symbol table in their default link order. You can rearrange entries in this file to change the order in which you want

procedures to be linked, then run the link editor as described in Step 4 above.

Using filemem and pageSymbols

As with any performance tuning, the real work is performed in a cycle: make a change, measure the result, make a change, measure the result.

The two programs **filemem** and **pageSymbols** help you measure the effectiveness of your procedure ordering. These tools provide a way to learn which pages of the executable file are loaded in memory at a given time. This section briefly describes using these tools; for more information, see their UNIX manual pages.

Note: These utilities work only with single architecture executable files; they don't work with multi-architecture ("fat") files. See the Project Builder documentation and the UNIX manual for **lipo** for more information.

filemem prints the number of resident pages that are paged into memory from a specific file, which of course if the number we are trying to minimize. In addition, the program prints a list of page offsets (from the beginning of the file) that are currently resident. filemem is used as follows:

```
filemem filename
```

filename represents the name of any file in memory; an executable, in this case. The output will be a list of text page numbers that are currently resident in memory from *filename*. **filemem** has one restriction: it won't work on files that are paged over an NFS file system. If this limitation creates a problem, simply copy the file to your local disk and run it again.

To make the data from **filemem** more useful, the program **pageSymbols** prints out the symbols on a particular page of executable text.

```
pageSymbols filename pagenumber
```

The output of **pageSymbols** is a list of procedures contained in *filename* on *pagenumber*. You can use the output from this program to determine if each page associated with the file in memory is optimized. If it isn't, you may rearrange entries in the order file and relink the executable to maximize performance gains, for example, by moving two procedures that are called together so they are linked on the same page. This may require several cycles of linking and tuning to get just right.

Profiling Specific Operations

For reasons described in the next section, it can be useful to gather profiling information for the duration of a specific routine. This provides a list of all routines used during that operation. The easiest way to do this is with **perfctrl**, a little utility program whose source is at the end of this document. **perfctrl** lets you interactively send messages to your application to start and stop profiling, to write out the accumulated profiling data and to reset the buffer of profiling data.

As an example, let's say you wanted to gather the list of routines used in your application named "MyApp" during printing. You would run the application, get ready to print, and then start profiling with the shell command

```
perfctrl MyApp start
```

Next, you would do the print operation. After the application is finished, you execute

```
perfctrl MyApp stop
perfctrl MyApp write /tmp/myProfilingData
perfctrl MyApp reset
```

These commands stop profiling, and write the profiling data out to the named file. The reset clears the profiling data that's accumulated so the next operation you measure starts from a clean slate.

Combining Order Files

Why generate profile data for individual operations of your application? The strategy is based on the assumption that a large application will have three general groups of routines:

- "Hot" routines ±€These routines are used during almost all uses of the application. They are often primitive routines that provide a foundation for the application's features (e.g., routines for accessing a document's data structure) or routines that implement the bread and butter of your application (e.g., routines that implement typing in a word processor). Almost any time a user is in the application, these routines will be used. Therefore we want them clustered on pages together, undiluted by less important routines.
- "Warm" routines ±€These routines are used during only specific uses of the application. They are usually associated with particular features that user perform occasionally in the application (e.g., launching, printing, or importing graphics). Since user's do sometimes use these routines, we want them to be packed together on a small set of pages so we can get them loaded quickly. However, since there will be large stretches of time when user's aren't accessing this functionality, we want these routines out of the "hot" category, so they can page out as a group as well as page in together.
- "Cold" routines ±€These routines are are rarely used in the application. They implement obscure features or cover boundary or error cases. We want these routines all together so that they stay paged out, and do not become dead-wood wasting space on a hot page.

The summary is that at any given time, we expect to have most of the hot pages resident, we would like to have the warm pages resident for the features that the user is using, and only very rarely should a cold page be

brought in.

To achieve this order, you must gather a number of profile data sets. The first one to gather is a list of the "hot" routines. As described above, compile the application for profiling, launch it, and force most "initialization" to happen (e.g., bring up all the panels that are commonly used, cause any lazy data structures to be created, open a couple ofaverage documents). Now you're ready to gather data on common use. Turn profiling on with **profctrl,** use the common features of the application, and dump out the resulting profile data. Generate a frequency sorted order file from this data called **hot.order** using **gprof -S**.

Using the same technique, you then create order files for the various features that users occasionally use. An important example of this is application launch. Printing, opening documents, importing images and using various panels and tools are other examples of features that users use occasionally but not continually, and are good candidates for having their own order files. Call these files ***feature*.order** for each feature measured.

Lastly, you generate a "default" order file **default.order** (as described above) to get a listing of all routines.

Once you have these order files, they are combined with a utility named **unique**, whose source is included in Appendix C. This program simply combines files by removing duplicate lines, retaining the ordering of the original data. In our example you would generate your final order file with

```
unique hot.order feature1.order ... featureN.order default.order > final.order
```

Of course, the real test of the ordering is how many pages it saves. Run your application and use different features, and use **filemem** to examine how well your ordering file is performing under different conditions.

Final Tuning - Finding that one last hot routine

Usually after reordering you will have a region of pages with "cold" routines that you expect to be rarely used, often at the end of your text ordering. However, sometimes one or two "hot" routines will slip through the cracks and land in this cold section. This is a costly mistake, because using one of these hot routines now requires an entire page to be resident, a page that is otherwise filled with cold routines that are not likely to be used.

Use **filemem** to check that the cold pages of your executable are not being paged in unexpectedly. Look for pages that are resident with high page offsets, in the cold region of your application's text segment. If there is an unwanted page, you need to find out what routine on that page is being called. One way to do this is to profile during the particular operation that is touching that page, and grep the routines from the profile for routines that reside on that page. Alternatively, a quick way to identify where a page is being touched is to run the application under **gdb** and use the Mach call **vm_protect** to disallow all access to that page:

```
(gdb) p vm_protect(task_self(), startpage, vm_page_size, FALSE, 0);
```

After clearing the page protections, any access to that page will cause a memory fault, which will break the program in **gdb**. At this point you can simply look at the stack backtrace to learn why the routine was being called.

Reordering Other Sections

The **-sectorder** link editor option can be used to order blocks in any section of the executable. Sections that may occasionally benefit from reordering are literal sections, such as the __**TEXT** segment's __**cstring** section, and the __**DATA** segment's __**data** section.

Reordering Literal Sections

The lines in the *orderfile* for literal sections can most easily be produced with with **ld**(1) and **otool**(1). For literal sections, **otool** creates a specific type of *orderfile* for each type of literal section:

- For C string literal sections, the *orderfile* format is one literal C string per line (with ANSI C escape sequences allowed in the C string). For example, a line might look like:

```
Hello world\n
```

- For 4-byte literal sections, the *orderfile* format is one 32-bit hex number with a leading 0x per line with the rest of the line treated as a comment. For example, a line might look like:

```
0x3f8ccccd (1.10000002384185790000e+00)
```

- For 8-byte literal sections, the *orderfile* is two 32-bit hexadecimal numbers per line separated by white space each with a leading 0x, with the rest of the line treated as a comment. For example, a line might look like:

```
0x3ff00000 0x00000000 (1.00000000000000000000e+00)
```

- For literal pointer sections, the format of the *orderfile* are lines representing the pointers one per line. A literal pointer is represented by the segment name, section name of the literal pointer, and then the literal itself. These are separated by a colon with no extra white space. For example, a line might look like:

```
__OBJC:__selector_strs:new
```

- For all the literal sections, each line in the the *orderfile* is simply entered into the literal section and will appear in the output file in the order of the order file. There is no check to see if the literal is in the loaded objects.

To reorder a literal section, first create a *whatsloaded* file using the **ld -whatsloaded** option as described earlier.

Then, run **otool** with the appropriate options, segment/section names, and filenames. The output of **otool** is a default order file for the specified section. For example, the following command line produces an order file listing the default load order for the __**TEXT** segment's __**cstring** section in the file **cstring_order**:

```
otool -X -v -s __TEXT __cstring `cat whatsloaded` > cstring_order
```

Once you've created the file **cstring_order**, you can edit the file and rearrange its entries to optimize locality of reference. For example, you can place literal strings used most frequently by your program (such as labels that appear in your user interface) at the beginning of the file. To produce the desired load order in the executable, use the following command:

```
cc -o hello hello.o -sectorder __TEXT __cstring  cstring_order
```

Reordering Data Sections

There are currently no tools to measure references to data symbol usage. However you may know a program's data use patterns, and may be able to get some savings by separating data for seldom used features from other data. One way to approach __**data** section reordering is to sort the data by size so that small data items end up on as few pages as possible. For example, if a larger data item is placed across two pages with two small items sharing each of these pages, the larger item must be paged in to access the smaller items; reordering the data by size can minimize this sort of ineffeciency. Since this data is private per-process and would have to be written to the swap file, this could be a major savings in some programs.

To reorder the __**data** section, first create an order file listing source files and symbols in the order in which you want them linked (order file entries are described at the beginning of Step 3 above). Then, relink the program using the **-sectorder** command line option:

cc -o *outputfile inputfile*.o -sectorder __DATA __data *orderfile* -e start

Appendix A: Reordering Assembly Language Code

There are a few extra things to keep in mind when reordering routines coded in assembly language:

- Relocation Entries and Existing Object Files

  Starting with Release 2.0, the NeXT compiler for the Objective C language creates a new type of relocation entry. Before reordering your executable, any object files that were compiled under Release 1.0 must be recompiled. If you attempt to reorder procedures from an object file compiled using the 1.0 compiler, the link editor doesn't detect the problem. The result will be a program that won't execute. By the same token, object files compiled by the current compiler aren't backward compatible with pre-2.0 releases of the link editor. If such an object file were linked using an earlier version of the link editor, the result would be an error message such as follows:

  ```
  ld: hello.o r_address (0xa000005e) field of relocation entry 4 in
  section (__TEXT,__text) out of range
  ```

  The key here is that the high bit of the **r_address** field, bit 0x80000000, is set, meaning it is one of the new relocation entries. For more on these relocation entries, see the comments in the file **/NextDeveloper/Headers/mach-o/reloc.h**.

  Note: You can avoid problems with previously compiled object files by linking them whole, without reordering. The technique for doing so is described in Step 3 below.

- Temporary Labels in Assembly Code

Within hand-coded assembly code, be careful of branches to temporary labels that branch over a non-temporary label. For example, if you use a label that starts with ªLº or a *d* label (where *d* is a digit), as follows:

```
foo: bra 1f
     ...
bar: ...
1:   ...
```

the resulting program won't link or execute correctly, because only the symbols **foo** and **bar** make it into the object file's symbol table. References to the temporary label **1** are compiled as offsets; as a result, no relocation entry is generated for the instruction **bra 1f**. If the link editor does not place the block associated with the symbol bar directly after that associated with **foo**, the branch to **1f** will not go to the correct place; since there is no relocation entry, the link editor doesn't know to fix up the branch. The source code change to fix this problem is to change the label **1** to a non-temporary label (**bar1** for example).

Note: You can avoid problems with object files containing hand-coded assembly code by linking them whole, without reordering. The technique for doing so is described in Step 3 below.

- The Pseudo-Symbol **.section_start**

If the specified section in any input file has a non-zero size and there is not a symbol with the value of the beginning of its section, the pseudo symbol **.section_start** is used by the link editor as the symbol name it associates with the first block in the section. This should never be needed with the 2.0 and later compilers for C code. The main reason for this symbol was to deal with literal constants whose symbols did not persist into the object file. Since literal strings and floating-point constants are now in literal sections this is no longer a problem. For assembly source code and non-NeXT compilers this symbol might be used. However, it is suggested that such code not be reordered and that the file instead be linked whole, without reordering. The

technique for doing so is described in Step 3 below.

Appendix B: Source Code for perfctrl

```
/*
       perfctrl

       A utility for interactively controling profiling.  Compile with:

                     cc -o perfctrl profctrl.m -lNeXT_s
*/

#import <appkit/appkit.h>
#import <stdlib.h>

void Usage() {
    printf ("Usage: perfctrl appname on|off|reset|write [filename]\n");
    exit (5);
}

enum { ON, OFF, RESET, WRITE };

void main(int argc, char *argv[]) {
    port_t port;
    id speaker;
    int cmd = 0;
    int retVal = -1;
```

```
  if (argc < 3) Usage();

  if (!strcmp(argv[2], "on") && argc == 3) cmd = ON;
  else if (!strcmp(argv[2], "off") && argc == 3) cmd = OFF;
  else if (!strcmp(argv[2], "reset") && argc == 3) cmd = RESET;
  else if (!strcmp(argv[2], "write") && argc == 4) cmd = WRITE;
  else Usage();

  if ((port = NXPortFromName (argv[1], NULL)) == PORT_NULL) {
     printf ("Couldn't contact application %s\n", argv[1]);
     Usage();
  }

  [(speaker = [[Speaker alloc] init]) setSendPort:port];

/* NOTE: these Listener messages are for debugging only! */
  if (cmd == ON) {
     retVal = [speaker selectorRPC:"appStartProfiling" paramTypes:""];
  } else if (cmd == OFF) {
     retVal = [speaker selectorRPC:"appStopProfiling" paramTypes:""];
  } else if (cmd == RESET) {
     retVal = [speaker selectorRPC:"appResetProfiling" paramTypes:""];
  } else if (cmd == WRITE) {
     retVal = [speaker selectorRPC:"appWriteProfilingData:" paramTypes:"c", argv[3]];
  }
  if (retVal != 0) {
     printf ("Could not perform operation\n");
     exit (5);
  } else {
```

```
        exit (0);
    }
}
```

Appendix C: Source Code for unique

```
/*
        unique

        A utility for interactively combining files while removing
        duplicate lines.  Compile with:

                cc -O -o unique unique.c
*/


#import <stdio.h>
#import <objc/hashtable.h>

#define MAX_LINE 8*1024

void doFile(FILE *fp) {
    char buf[MAX_LINE];
    static NXHashTable *table = NULL;

    if (!table) {
        table = NXCreateHashTable(NXStrPrototype, 0, NULL);
    }
    while (fgets(buf, MAX_LINE, fp)) {
```

```
        if (!NXHashInsert(table, NXCopyStringBuffer(buf))) {
            fputs(buf, stdout);
        }
    }
}


void main(int argc, char *argv[]) {
    int i;
    FILE *fp;

    if (argc > 1) {
        for (i = 1; i < argc; i++) {
            if (fp = fopen(argv[i], "r")) {
              doFile(fp);
              fclose(fp);
            } else {
              fprintf(stderr, "Could not open %s\n", argv[i]);
            }
        }
    } else {
        doFile(stdin);
    }
}
```