CSE 303: Concepts and Tools for Software Development

Dan Grossman

Spring 2007

Lecture 19— Profiling (gprof); Linking and Libraries

Where are we

Already started playing with `gprof`; some more to learn about:

- Effective performance tuning
- Limitations of gprof-style profiling

Then: Linking code in multiple files (the .o story and the Java story and more)

Friday: societal implications (electronic voting)

Next week +: Finish linking, A taste of concurrency, A taste of C++

Using `gprof`

- Compile with `-pg` *on the right*.
  - When you create the `.o` (for call counts)
  - When you create the executable (for time samples)
- Run the program (creates (overwrites) `gmon.out`)
- Run `gprof` (on `gmon.out`) to get human-readable results.
- Read the results (takes a little getting used to).

Getting useful info

- The information depends on your inputs! (Always know what you're profiling)
- Statistical sampling requires a reasonable number of samples
  - Probably want at very least a few thousand
  - Can run a program over and over and use `gprof -s` (learn on your own; write a shell-script)
- Make sure performance matters
  - Is 10% faster worth uglier or buggier code?
  - Do you have better things to do (documentation, testing, ...)?

Performance tuning

- Never tune until you know the bottleneck (that's what `gprof` is for, but it doesn't tell you how to tune).
- Rarely overtune to some inputs at the expense of others.
- Always focus on the overall algorithm first.
- Think doubly-hard about making non-modular changes.
- Focus on low-level tricks only if you really need to (< 5 times in your career?)
- See if compiler flags (e.g., `-O`) are enough.

Note: Performance tuning a library is harder because you want to do well for "unknown programs and inputs".

Our example

- Different bottlenecks for large array-size and large max-number!!
  - If you knew max-number could never be more than 10, would you optimize `is_prime`?
- Optimal algorithm for `is_prime` is slower than for `find_largest`, but we did not write the optimal algorithms!
- After fixing time for `find_largest`, we still had a stack overflow.
- Changing the `is_prime` algorithm helped a lot.
- Little things (e.g., reordering tests and loops) generally "lost in the noise".
- Output affects wall-clock time.

Note: For more rigorous comparisons, we should not randomly seed the random-number generator.

Misleading Fact #1

Cumulative times are based on *call estimation*. They can be really, really wrong, but usually aren't.

```
int g = 0;
void c(int i) {
  if(i) return;
  for(; i < 100000000; ++i)
    ++g;
}
void a() { c(0); }
void b() { c(1); }
int main(int argc,char**argv) { a(); b(); return 0; }
```

Conclusion: You *must* understand what your profiler measures and what it presents to you. gprof doesn't lie (if you read the manual)

Misleading Fact #2

Sampling errors (for time samples) can be caused by too few samples, or by *periodic sampling*

```
void a() { /* takes 0.09 s */ }
void b() { /* takes 0.01 s */ }
int main(int argc,char**argv) {
  for(; i < 10000; ++i) {
     a();
     b();
  }
}
```

This probably doesn't happen much and better profilers can use *random intervals* to avoid it.

Related fact: Measurement code changes timing (an uncertainty principle).

Poor man's profiling

The `time` command is more useful because no measurement overhead, but less useful because you get only whole-program numbers.

- real: roughly "wall-clock"
- user: time spent running the code in the program
- system: time the O/S spent doing things on behalf of the program

Not precise for small numbers

Misleading Fact #3: `gprof` does not measure system time?

Effects on real time: Machine load, disk access, I/O

Effects on system time: I/O to screen, file, or `/dev/null`

Compiler Optimization

Compilers must:

- Trade "compile-time" for "code-quality"
- Trade "amount of code" for "specialization of code"
- Make guesses about how code will be used.

You can affect the trade-off via "optimization flags" – definitely easier but less predictable than modifying your code.

`gcc` is not a great optimizer:

- For our initial example, it made a big improvement.
- No promises; it could slow your program down (unlikely, but I have seen it even for the primes program).

Bottom line: Remember to "turn optimizations on" if it matters.

Intro to linking

Linking is just one example of "using stuff in other files"...

In compiling and running code, one constantly needs *other files* and *programs that find them*.

Examples:

- C preprocessor `#include`
- C libraries (where is the code for `printf` and `malloc`)
- Java source files (referring to other source code)
- Java class files (referring to other compiled code)

Usually you're happy with programs "automatically finding what you need" so the complicated rules can be hidden.

Today we will demystify and make generalizations.

Common questions

1. What you are looking for?
2. When are you looking for it?
3. Where are you looking?
4. What problems do cycles cause?
5. How do you change the answers?

our old friends: files, function names, paths, environment variables, command-line flags, scripts, configuration files, ...

Previous example

cpp (invoked implicitly by `gcc` on files ending in `.c`).

What: files named "foo" when encountering `#include <foo>` or `#include "foo"` (note `.h` is just a convention).

When: When the preprocessor is run (making `x.i` from `x.c`).

Where: "include path" current-directory, directories chosen when cpp is installed (e.g., `/usr/include`), directories listed in `INCLUDE` shell variable, directories listed via `-I` flags, ...

The rules on "what overrides what" exist, but tough to remember. Can look at result to see "what really happened".

Example: for nested `#include`, the original current-directory or the header file's current-directory?

Example: Why shouldn't you run cpp on 1 machine and compile the results on another?

javac is similar

If `A.java` defines class `A` to have a field of type `B`, how "does the compiler know what `B` is"?

What: a file named `B.class` (probably the result of compiling `B.java`).

When: When compiling a source file that uses the class `B`.

Where: "class path" current-directory, directories chosen when `javac` was installed, directories listed in `CLASSPATH` shell variables, directories listed via `-classpath` flags, ... (Note: Packages correspond to subdirectories)

The rules on "what overrides what" exist, but tough to remember.

Source code cycles

What if two source files refer to each other?

- C: Can't but don't need to: Put *declarations* in header files and include each header file at most once.
- Java: If `B.class` is not found, but `B.java` is, (implicitly) compile `B.java` (potentially with information the compiler already has about `A`).

IDEs

Fancier development environments provide help with "packages", "projects", etc.

Fundamentally, the questions are the same and their are settings and menu items for controlling your development process.

Compiled code

So far we have talked about finding *source code* to **create** *compiled code* (either `.o` files for C or `.class` files for Java).

These files are *not* whole applications, so we have the same questions for "finding the other code".

The Java story is a bit simpler, so we will do it first.

Java class-loading and execution

Recall `java A` *`args`* runs class `A`'s static `main` method with args.

`java` is just a program that finds `A.class` and knows what to do (*interpretation* and/or *just-in-time compilation*).

But it will probably have to find lots of other classfiles too.

Simple (untrue but doable) version: Recursively find all the class files you need before starting execution:

- What: class files referred to
- When: start of execution
- Where: classpath, etc.

Disadvantages?

Java class-loading continued

Actually, the JVM is much *lazier* (technical word) about class-loading; waiting until a class is actually used (technical definition) during execution.

That is, the *when* is "later" and "more complicated".

So is the *where*:

- `jar` files (lots of classes in one file, retrieved together)
- remote class files (applets with code over the web, etc.)
- different *security* settings for classes found different places

Why use a `jar` ("Java archive") file:

- Keep classes that need each other together
- Faster/simpler remote retrieval

Object code is different

A `.o` file is *not* "runnable" – you have to actually *link* it with the other code to make an *executable*.

Linking (`ld`, or called via `gcc`) is a "when" between compiling and executing.

Again, `gcc` hides this from you (just `-c` or not `-c`), but it helps to know what is going on.

First discuss *static linking*, which is mostly like the untrue version of Java we sketched.

Linking

If a C file uses but does not define a function (or global variable) `foo`, then the `.o` has "unresolved references". *Declarations don't count; only definitions.*

The linker takes multiple `.o` files and "patches them" to include the references. (It literally *moves code* and *changes instructions* like function calls.)

An executable must have no unresolved references (you have seen this error message).

What: Definitions of functions/variables

When: The linker creates an executable

Where: Other `.o` files on the command-line (and much more...)

More about where

The linker and O/S don't know anything about `main` or the C library.

That's why `gcc` "secretly" links in other things.

We can do it ourselves, but we would need to know a lot about how the C library is organized. Get `gcc` to tell us:

- `gcc -v -static hello.c`
- Should be largely understandable soon.
- `-static` (stick with the simple "get all the code you need into `a.out` story)
- the secret `*.o` files: (they do the stuff before `main` gets called, which is why `gcc` gives errors about `main` not being defined).
- `-lc`: complicated story about finding the *library* (a.k.a. "archive") `libc.a` and including any *files* that provide still-unresolved references.