COSC345 Week 16 Notes

The topic is really a little more general: it's "getting insight into the structure and behaviour of a program by using tools that summarise execution traces".

When you are revising this paper for your own use, Java tools have improved dramatically over the last few years. The "Java Platform Debugger Architecture" — what NetBeans and Eclipse use to find out what's going on — uses the "Java Virtual Machine Tool Interface", which is a very powerful tool indeed. Consider spending some time on it, or perhaps the higher level "Java Debug Interface" to this stuff.

On my system at the time of writing, "man java" includes

```
-Xprof
  Profiles the running program, and sends profiling data
  to standard output.  This option is provided as a
  utility that is useful in program development and is not
  intended to be be used in production systems.

-Xrunhprof[:help][:suboption=value,...]

  Enables cpu, heap, or monitor profiling.  This option is
  typically followed by a list of comma-separated
  suboption=value pairs.  Run the command java
  -Xrunhprof:help to obtain a list of suboptions and their
  default values.
```

This stuff used to go through the "Java Virtual Machine Profiler Interface" and/or the "Java Virtual Machine Debug Interface" but in Java 1.5 went through the "JVM Tool Interface" (JVMTI). See http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/index.html for details. (Follow the link in that page to 139 pages of details!) The easiest way to demonstrate building some kind of special-purpose tracer these days is probably to use this stuff.

In Java 1.8 the JVMTI documentation is at http://download.oracle.com/javase/8/docs/technotes/guides/management/index.html

Since Java 1.6 it appears that "Java Visual VM" is the thing to look at. For Java 1.8 see http://download.oracle.com/javase/8/docs/technotes/guides/visualvm/index.html and http://download.oracle.com/javase/8/docs/technotes/guides/management/index.html. You may also be able to use DTrace with Java.

"What happened in a program" is of course precisely what an execution trace records. Recording a complete program trace is possible, but such traces quickly get ridiculously large. It's therefore important to trace selectively; perhaps to record only certain events (like memory allocation and freeing, used in memory profilers), and/or to record at selected times, or to record events from only part of the program.

I mention using an instruction interpreter or code rewriter to get information about cache behaviour. This is an opportunity I take to mention the fact which students have not previously grasped, that main memory is much slower than the CPU. I refer to numbers obtained from lmbench3 (a benchmarking program you'll find on the web) to show that fetching data from random locations in main memory is on the order of 100 times slower than fetching from level 1

cache. Hardware performance counters can tell you **that** cache misses are a problem, but they cannot tell you **where**. A hundred-fold slowdown due to poor use of memory is an important thing to catch, if you can! Indeed, the slogan going around these days is, "memory is the new disc, disc is the new tape."

The "Coding Horror" blog has a nice table in http://blog.codinghorror.com/the-infinite-space-between-words/ taken from "Systems Performance: Enterprise and the Cloud" (http://www.amazon.com/dp/0133390098/?tag=codihorr-20). The idea is to equate one computer cycle with a second (I'd have chosen a tenth of a second, a typical human reaction time, but it's not my table).

1 CPU cycle	0·3 ns	1 s
Level 1 cache access	0·9 ns	3 s
Level 2 cache access	2·8 ns	9 s
Level 3 cache access	12·9 ns	43 s
Main memory access	120 ns	6 min
Solid-state disk I/O	50-150 us	2-6 days
Rotational disk I/O	1-10 ms	1-12 months
Internet: SF to NYC	40 ms	4 years
Internet: SF to UK	81 ms	8 years
Internet: SF to Australia	183 ms	19 years
OS virtualization reboot	4 s	423 years
SCSI command time-out	30 s	3000 years
Hardware virtualization reboot	40 s	4000 years
Physical system reboot	5 m	32 millennia

The same page has an example from Jim Gray. If fetching a datum from a register is like you dredging something up from memory, then getting data from disc is like getting it couriered from Pluto; even using SSD is like getting it couriered from Jupiter.

Talking about scriptable debuggers is a chance to point out that interactive debugging is a slow and painful process and what you really want from a debugger is features that will reduce the amount of information you have to view with your own eyes. How good is your debugger at not showing things? My slide mentions dbx commands, because that's what I use. Gdb is similar. You should adapt this to whatever debugger you use. For the 2006 project, which used OPL on a hand-held device, I wrote a little source to source tracing tool called opltrace. What would be really neat would be to use the extensible compiler for C, 'xoc' (available free from MIT, there is a Masters thesis and a PhD thesis describing it) to plug in some kind of tracing facility in C. Or perhaps Caml's ckit. I provided an instruction editing tracer that takes a .s assembly code file produced by an C compiler on a SPARC machine and rewrites the 'call' instructions to call a monitor procedure. This needs to be rewritten to work on 80*86 assembly code for x86 Solaris, x86 Linux, and intel-Macs. The key idea is that

```
call label
```

turns into

```
call T_label
```

where `T_label` is a generated function that does some tracing and then jumps to label. The SPARC version is only about a page of AWK. The code was

provided as a handout in the class, and a sample trace and a graph produced from it. This year I wanted I want to record not just which function is being called but which it is being called from, so that I could get a real "who-called-whom" graph, but ran out of time. This is a subgraph of the "who might call whom" graph that one can get from say CScout, but it is the subgraph that is relevant to the test cases you have run. Of course this can also be extracted from the output of the gprof command, which has its own place in this lecture.

I can usually illustrate any software engineering topic from something I'm working on at the time. When I was revising this lecture in 2011, the XML parser in my Smalltalk library wasn't working, so I added a crude call-tracing facility to my compiler. Even that revealed some surprises: the most commonly called method, for example, turned out to be involved with hashing. It did reveal where the program was when it reported the error. Further work on the tracer gave me function calls, function returns (including long distance ones), and object allocations, so I can now see not only how much memory is allocated but where it is allocated.

I can usually illustrate any software engineering topic from something I'm working on at the time. When I was revising this lecture in 2015, I ran into a problem with my Smalltalk compiler. The compiler is written in portable C. It is co-developed on Solaris 10 and Mac OS X 10.6.8 with frequent builds on Solaris 11, Mac OS X 10.9.5, and Linux, and occasional builds on Cygwin and OpenBSD. This week, building on OpenBSD 5.6 produced two unexpected problems:
1. 'gcc -m32 ...' produced 32-bit .o files, and then complained that it could not link them to make a 64-bit executable (which I dod not want). I could find no way to make it generate a 32-bit executable.
2. The program *should* work as a 64-bit program, but it got a segmentation violation that had never been seen on any other system.

Enter valgrind! Valgrind has never been available for Solaris, and has had an unfortunate history of being available in OS X 10.x but not OS X 10.(x+1). It wasn't installed on the OpenBSD VM. Fortunately, it turned out to be available for Mac OS X 10.9.

Download valgrind-3.10.1, unpack, configure, make, OOPS! The source code contained Mac OS X-specific annotations that gcc choked on. Unpack again, configure with CC=cc CXX=cc, make, make install, and it worked.

Two issues were found, which I would *never* have found with a debugger.
1. In order to handle numbers with *lots* of digits, the compiler represents numbers as (sign,size,bigit[0..size-1]). There were places where I checked num→bigit[0], but if size=0, bigit[0] did not exist.
2. Expressions are represented as (op,size,child[0..size-1]). *e0* ifNil: *b1* ifNotNil: *b2* is thus represented as (IF_NIL,3,{*e0*,*b1*,*b2*}). After code to handle this was written, a space saving was introduced, where *e0* ifNil: *b1* is represented as (IF_NIL,2,{*e0*,*b1*}). There was still code that loaded exp→child[2] and then happened not to use it when size=2.

The existence of "prof" and "gprof" tends to mislead students into believing that the only kind of information you can get from profiling tools is time usage. That is indeed what prof and gprof record. But you can also do heap profiling. I first became familiar with heap profiling in Haskell and Clean, where it's quite hard for people to predict what a lazy functional language will do, so measuring what really happened is important. It's also useful for C and Java.

It's particularly important for Object-Oriented languages with large libraries where you have no idea how the components you are calling use memory. Fortunately, Instruments in Mac OS X gives you some insight into what is happening with memory, as it is happening. However, Instruments, like many tools, suffers from the crippling defect of not being *scriptable*. Suppose you want to track allocation

The statement that cycle counters give precise CPU times needs to be qualified: on modern chips which dynamically adjust their frequency to adapt to workload it isn't really true any longer. But at least you can get precise *cycle counts*. It's important to explain the limitations of sampling, because sampling based profilers can give surprising results, like attributing some part of the time to functions that were never called (space sampling) or giving anomalously high or low times (time sampling).

Test coverage tools (tcov, gcov, and bprint (which comes with lcc)) are introduced as statement count profilers. The idea of using them to test your tests and the various levels of coverage (statement, branch, path) are introduced.

If you are using Java and you don't have any other coverage program handy, you really should look into EMMA, which is an open source code rewriter. It can give you coverage counts for classes, methods, and lines. You can find examples at its web site, emma.sourceforge.net.

Handouts should include manual pages for prof, gprof, and ctrace, tcov and gcov. There's also the little tracing tools I wrote.

In 2009 I demonstrated, or tried to demonstrate, the Linux tracing programs strace (show system calls) and ltrace (show library calls). In fact the Linux system I tried this on had a version skew problem (gcc was compiled for 64-bit but the libraries were 32-bit) so I was unable to run the program I intended to, and we ended up tracing gcc itself to discover what the problem was.

Information about the Java tracing and monitoring facilities would be good to provide, and a demonstration tool showing how to do profiling by plugging into it would be good. Just such a demonstration comes with the JDK these days. Another good handout would be one contrasting the "may call" graph of a program with the "did call" graph derived from the trace of a test case. The "may call" graph will be huge for even quite a modest program. Object-oriented programming makes this much worse. The "did call" graph may be usable for even quite a large program as long as the test case is modest. Some kind of interactive graph viewing program, that lets you delete nodes, merge nodes, and ideally colour nodes according to some kind of rule, would be a good tool to use. GraphViz is

One minor theme of this paper is that you can build your own tools. Here, for example, is an AWK script that takes the output of gprof and turns it into a graph in 'dot' format:

```
BEGIN {
    skipping = 1
    printf "digraph did_call {\n"
}
skipping {
    if ($2 == "%time") skipping = 0
    next
}
```

```
/^\[/ {
    callee = $6
    for (i = 1; i <= n; i++)
        printf "    \"%s\" -> \"%s\";\n", caller[i], callee
    next
}
NF >= 5 && $3 ~ /^[0-9]*\/[0-9]*$/ && $4 ~ /^[a-zA-Z_]/ && $NF ~ /\[.*\]/ {
    caller[++n] = $4
    next
}
NF >= 4 && $1 ~ /^[0-9]*$/ && $2 ~ /^[a-zA-Z_]/ && $NF ~ /\[.*\]/ {
    caller[++n] = $2
    next
}
/^--/ {
    n = 0
}
END {
    printf "}\n"
}
```

This ignores the call counts in the gprof output, but does show you which functions called which others.

One theme which this lecture is meant to get across, but which is not explicit in the text (sorry), is the need for measurement rather than guesswork. Find some modest C programs. A good one would be the free lcc compiler. Where does the time go when compiling? Compile the program (or programs) with profiling, run them (compile something), and show the profiles. Prepare to be surprised.

In class I tell a few anecdotes about time wasted optimising the wrong thing, about people being very much surprised about where the time (or space) was really going. One of my favourite examples comes from the Smalltalk world, where a consultant was asked into a company to improve a Smalltalk program that was embarrassingly slow. "What did the profile show?" he asked. "What profile?" They asked. So he did

```
MessageTally spyOn: [TheirProgram run]
```

and stared at the results for a few minutes. He then added

```
today
    Today ifNil: [Today := Date today].
    ^Today
```

and the program ran twice as fast. (There is a joke about a man who takes his car to a garage to be repaired. The mechanic listens to it for a while, then hits the engine with a hammer, and it works perfectly. He asks for $100. The main complains. The mechanic then writes out an invoice: "Hitting the engine with a hammer: $1. Knowing *where* to hit: $99.")

This might be a good lecture to explain the concept of profile-driven optimisation. (Compile program with profiling. Run it on realistic test cases. Rerun the compiler, telling the compiler to read the profiles.) Part of the issue here

is that mispredicted branches make modern machines run very poorly thanks to deep pipelines. Point out the limitation: if a new data set is substantially different from the training data the compiler used, the performance may be worse than if it hadn't been optimised that way.) Point out the relevance to object-oriented programming, where the thing to profile is "at this calling site, which of all the thousands of classes that the message receiver might have been was it really?" which permits optimistic inlining.

Point out that you can profile the workflows that your program is responsible for supporting. Rare requests have to be supported, but they may not be the ones you need to put most of your effort into.

noexec is at noexec.sourceforge.net It lets you run a program (rather like time does) that runs as normal *except* that it cannot run anything else. It *can* fork(), but it cannot use any memory of the "exec" family to chain to another program. It uses LD_PRELOAD to slip in a library that redefines all the "exec" functions to dummies that just fail.