COSC345 Week 16

Profilers and Coverage Analysers

28 July 2015

Richard A. O'Keefe

What happened in my program?

Sometimes we want to find out what happened in a program. There are several reasons for this:

- to help find a bug
- to help understand someone else's code
- to help improve quality (speed, memory, &c.)

What kinds of tools can we use?

- Interpreters or debuggers (dbx, gdb, ddd, ups)
- Tracers (ctrace, own code)
- Profiling (prof, gprof, Instruments)
- Coverage analysers (tcov, gcov, bprint)

Interpreters

We can interpret real hardware instructions (valgrind (PC), atom (Alpha), shade (SPARC), Bochs (x86)).

$\rightarrow$ Good for examining cache use

$\rightarrow$ Can work with any language and library

We can interpret virtual instructions (Cint from CERN; mawk)

$\rightarrow$ More portable, easier to do and modify

We can interpret parse trees (nawk)

$\rightarrow$ much easier to write patterns for events

Debuggers

A scriptable debugger does the work for us.

dbx has **when** *event* { *stmt*;...*stmt* }

event includes **in** *func*, **returns** *func*,

at *line*, **access** *mode variable*

modifiers **-if** *expr*, **-in** *func*

stmt includes **print** *expr*,..., **assign**, **call**

$\rightarrow$ Can have debugger script for many kinds of tracing.

Library interposers

Library interposition slides something in between your program and the real library (handout)

It can trace library calls (Sun's truss, BSD ktrace?)

It can block dangerous calls (noexec)

It can introduce controlled faults

It redirect I/O to other files

Do you see a security issue here?

Tracing I

• Tracing records that certain events (entry to a function, exit from a function, arrival at a statement, raising a signal, handling a signal, using a variable, setting a variable, . . .) have occurred, possibly with details.

• A trace may be written as text, written in binary form, sent as messages to another process, and so on.

• Traces tend to become very large very quickly.

• Traces may be at machine level, source level, or user level.

Tracing II

- User level tracing can be very helpful

— but it usually needs pre-planned hooks.

- Machine level tracing can be very precise

— but you have to understand the machine level.

- Source level tracing suits most programmers

— typically done by source to source translation *e.g.*, ctrace.

Tracing III

- opltools example in class: opltrace
- Can instrument code by editing source file.
- Compiler-independent
- Language-dependent
- Limited by language expressiveness

Tracing IV

- trace example in class
- Can instrument code by editing compiler output
- Compiler generates .s file(s)
- .s file(s) edited by by simple script
- .s files compiled by C compiler
- When run generates trace
- Works for any compiler that generates .s

Profiling I

Profiling measures time or space or counts events.

- Statement counts
- Statement times (wouldn't it be lovely)
- Function calls (simple or contextual)
- Function times (simple or contextual)
- Memory allocation (heap profilers)

Profiling II

Profiling *changes* your code

Added code counts events, reads cycle counters *etc*

- Cycle counters give precise CPU times.

— may not include waits; doesn't match wall clock

- UNIX uses sampling, imprecise two ways:

— sample taken 50, 60, 100, or 120 times/second

— address chopped to "bucket", wrong function possible

- May count VM instructions and estimate time (gauge)

Coverage tools help to test your tests

What do your tests test?

$\rightarrow$ Only the code they have run.

How can you tell?

$\rightarrow$ Partly by design.

$\rightarrow$ Partly by measurement.

Designing your tests

— also known as “glass box”, “structural”.

— derive (more) test cases from source code

— decisions in code may split equivalence partitions

— you must *have* the source code

— read Sommerville about “path testing”

— tends to overlook *missing* code

Levels of Coverage

Statement coverage: every statement is executed in some test. This is the minimum acceptable level.

Branch coverage: every branch alternative is executed in some test. Strictly speaking, it's impossible in a language with exception handling like Lisp, Ada, C++, Java. At least cover the normal branches.

Path coverage: every path through the component is executed in some test. Obviously not achievable with loops, but try to cover 0 trips, 1 trip, a few trips, lots of trips.

Beizer books in bibliography belong on your shelf.

tcov lets you check for statement coverage (gcov)