COMP0104 Software Development Practice: Profiling and Coverage

Jens Krinke

Centre for Research on Evolution, Search & Testing
Software Systems Engineering Group
Department of Computer Science
University College London

Overview

- Tuning: Improving the Performance
- Profiling: Create Profiles over Program Execution
- Coverage: Measure the Program Execution

Improving performance

- A important part of programming is **tuning**, the systematic performance improvement.
- Systematic does not mean randomly!
- With profiling, one can determine
  - the **locations** where the performance can be improved
  - the **effect** of possible optimisations on the runtime.

System Tuning

- Establish acceptable behaviour
- Measure the current performance – is it acceptable?
- Identify the bottleneck(s)
- Modify the bottleneck
- Measure the performance again – is it now acceptable?

Tools for profiling

- TIME determines the runtime of **processes**.
- GPROF shows how the runtime is split across **functions**.
- GCOV counts how often individual **program lines** are executed.

Runtime measurement using TIME

TIME executes the command passed as an argument, and then outputs the runtime that was used by the process.

```
$ c++ -o huffencode huffencode.C
$ time ./huffencode COPYING > COPYING.huff.C
real    0m0.465s
user    0m0.420s
sys     0m0.010s
$ _
```

Runtime optimisations

```
$ c++ -O -o huffencode huffencode.C
$ time ./huffencode COPYING > COPYING.huff.C
real    0m0.735s
user    0m0.130s
sys     0m0.010s
$ _
```

CPU time for user functions has dropped to a third

However, the real runtime has increased. Why?

- other processes could be running
- it may take a moment for the newly created program to be loaded into memory

Program profiles with GPROF

A program profile shows

- which functions have been called,
- how often, and
- how much time they required.

This is done in three steps:

- During compilation, the program is instrumented
- During execution, a program profile is created.
- The program profile is analysed by GPROF.

Program profiles with GPROF

- Set up the program for profiling:

```
$ c++ -O -pg -o huffencode huffencode.C
$ _
```

- Execute the program

```
$ ./huffencode COPYING > COPYING.huff.C
$ _
```

- Analysis of the profile:

```
$ gprof ./huffencode
...
```

The output from GPROF consists of two parts

- A **flat profile**, which indicates
  how the runtime is divided into individual functions
- A **structured profile** in which
  the functions that have been called
  and the functions that will be called are indicated

The flat profile

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
33.33	0.02	0.02	47110	0.42	0.42	string_Scat
16.67	0.03	0.01	10674	0.94	0.94	bits_to_byte
16.67	0.04	0.01	10673	0.94	0.94	string::at
16.67	0.05	0.01	1	10000.00	42298.03	encode
16.67	0.06	0.01	1	10000.00	17668.44	read_input
0.00	0.06	0.00	10983	0.00	0.00	string_Salloc
0.00	0.06	0.00	256	0.00	0.00	compare
0.00	0.06	0.00	154	0.00	0.00	string::op char *
0.00	0.06	0.00	154	0.00	0.44	_cook
0.00	0.06	0.00	153	0.00	0.00	extract_min
0.00	0.06	0.00	153	0.00	0.00	insert
0.00	0.06	0.00	78	0.00	0.00	string_Scopy
0.00	0.06	0.00	1	0.00	0.00	global destructors
0.00	0.06	0.00	1	0.00	0.00	global constructors
0.00	0.06	0.00	1	0.00	0.00	string::from
0.00	0.06	0.00	1	0.00	0.00	huffman
0.00	0.06	0.00	1	0.00	64.53	init_codes
0.00	0.06	0.00	1	0.00	0.00	initial_queue
0.00	0.06	0.00	1	0.00	0.00	length

Insights

- The `string_Scat` function takes up a third of the runtime itself and with more than 47,000 calls is also the most frequently called function.
- Optimisations in the following functions can also increase the performance of the program:
  - `bits_to_byte`,
  - `string::at`,
  - `encode` and
  - `read_input`.

Structured profiles

For each individual function *f*, a structured profile indicates

- what share *f* and the functions called by *f* have of the entire runtime (`%time`).
- which functions have called *f* and how often, as well as *f*'s individual share of their runtime (these functions are listed above *f*)
- which functions *f* has called and how often, and what share of *f*'s runtime they have (these functions are listed below *f*)

```
index % time    self  children    called     name
                0.00    0.06       1/1           _start [2]
[1]    100.0    0.00    0.06       1         main [1]
                0.01    0.03       1/1           encode [3]
                0.01    0.01       1/1           read_input [5]
                0.00    0.00       1/1           write_huffman [10]
                0.00    0.00       1/1           huffman [17]
                0.00    0.00       1/1           write_encoding [20]
-----------------------------------------------
                                                 <spontaneous>
[2]    100.0    0.00    0.06                 _start [2]
                0.00    0.06       1/1           main [1]
-----------------------------------------------
                0.01    0.03       1/1           main [1]
[3]     70.5    0.01    0.03       1         encode [3]
                0.01    0.00   28737/47110       string_Scat [4]
                0.01    0.00   10674/10674       bits_to_byte [6]
                0.01    0.00   10673/10673       string::at [7]
                0.00    0.00       1/1           init_codes [9]
                0.00    0.00     77/154          _cook [8]
                0.00    0.00   10752/10983       string_Salloc [11]
                0.00    0.00     256/256         compare [12]
                0.00    0.00       1/1           string::from [16]
                0.00    0.00       1/78          string_Scopy [15]
```

```
                      0.00    0.00      152/47110        init_codes [9]
                      0.00    0.00      158/47110        _cook [8]
                      0.01    0.00    18063/47110        read_input [5]
                      0.01    0.00    28737/47110        encode [3]
[4]       33.3        0.02    0.00    47110          string_Scat [4]
-----------------------------------------------
                      0.01    0.01        1/1            main [1]
[5]       29.4        0.01    0.01        1          read_input [5]
                      0.01    0.00    18063/47110        string_Scat
[4]
-----------------------------------------------
```

- `string_Scat` does not call any other functions (at least none that is set up for profiling)
- of the 47,110 calls,
  - 18,063 come from the `read_input` function
  - and 28,737 come from the `encode` function.

Perhaps these are two suitable candidates for optimisation?

Coverage measurement

- GCOV measures the **line coverage**:
  it calculates how often each line of the program was run.
- Coverage is a measurement of how good a test suite is.
- Line coverage is useful for performance analyses.

Setting up a program for coverage measurement

- During compilation, the program is set up for coverage measurement:

```
$ c++ -g -fprofile-arcs -ftest-coverage -o
huffencode huffencode.C
```

- During the execution of the program, coverage measurement information is generated.

```
$ ./huffencode COPYING > COPYING.huff.C
```

- The coverage information is analyzed by GCOV

```
$ gcov huffencode.C
```

GCOV output

```
            // Fill CODES[C] with a [01]+ string denoting
            // the encoding of C in TREE
            static void init_codes(string codes[UCHAR_MAX + 1],
                                   HuffNode *tree,
                                   string prefix = "")
   153      {
   153          if (tree == 0)
######              return;

   153          if (tree->isleaf)
    77              codes[(unsigned char)tree->l.c] = prefix;
                else
    76          {
    76              init_codes(codes, tree->i.left,  prefix + "0");
    76              init_codes(codes, tree->i.right, prefix + "1");
                }
            }
```

```
             // Encode TEXT using the Huffman tree TREE
             static string encode(const string& text, HuffNode *tree)
    1        {
    1            string codes[UCHAR_MAX + 1];

    1            init_codes(codes, tree);
    1            string bit_encoding;
    1            for (int i = 0; i < text.length(); i++)
18063                bit_encoding += codes[(unsigned char)text[i]];

    1            string byte_encoding;
    1            for (int i = 0; i < bit_encoding.length() - BITS_PER_CHAR;
                      i += BITS_PER_CHAR)
                 {
10673                byte_encoding += bits_to_byte(bit_encoding.at(i, BITS_PER_CHAR));
10673            }
    1            byte_encoding += bits_to_byte(bit_encoding.from(i));

    1            return byte_encoding;
             }
```

Insights

Now we can see from where the numerous calls of `string_Scat` come from:

- In two loops, character strings are concatenated.
- The first loop is executed once for every character of the input, precisely 18,063 times
- the second for each character of the output, precisely 10,673 times.

These two critical positions can now be optimised.

```
// Encode TEXT using the Huffman tree TREE
static string encode(const string& text, HuffNode *tree)
{
    string codes[UCHAR_MAX + 1];

    init_codes(codes, tree);
    ostrstream bit_encoding_os;
    for (int i = 0; i < int(text.length()); i++)
        bit_encoding_os << codes[(unsigned char)text[i]];
    string bit_encoding(bit_encoding_os);

    ostrstream byte_encoding;
    for (int i = 0; i < int(bit_encoding.length()) - BITS_PER_CHAR;
         i += BITS_PER_CHAR)
    {
        byte_encoding << bits_to_byte(bit_encoding.at(i, BITS_PER_CHAR));
    }
    byte_encoding << bits_to_byte(bit_encoding.from(i));

    return byte_encoding;
}
```

How GPROF and GCOV work

- GPROF and GCOV have the compiler to produce **instrumented code**, the code is enriched with specific instructions for profiling.
- For GPROF, functions are compiled so that every called function creates an entry showing when and from where it was called.
- For GCOV, each node in the control flow graph is provided with a counter that is incremented during the execution of the program.

Sampling vs. instrumentation

- The run-time profiling of GPROF is not realised via counting, but via **sampling**.
- Instead, a special function analyses the current program counter at specific intervals and marks which functions have just been executed.
  (For example, 100 times per CPU second.)
- Compared to the accurate counting of function calls, sampling is subject to statistical inaccuracies.

Testing with Coverage

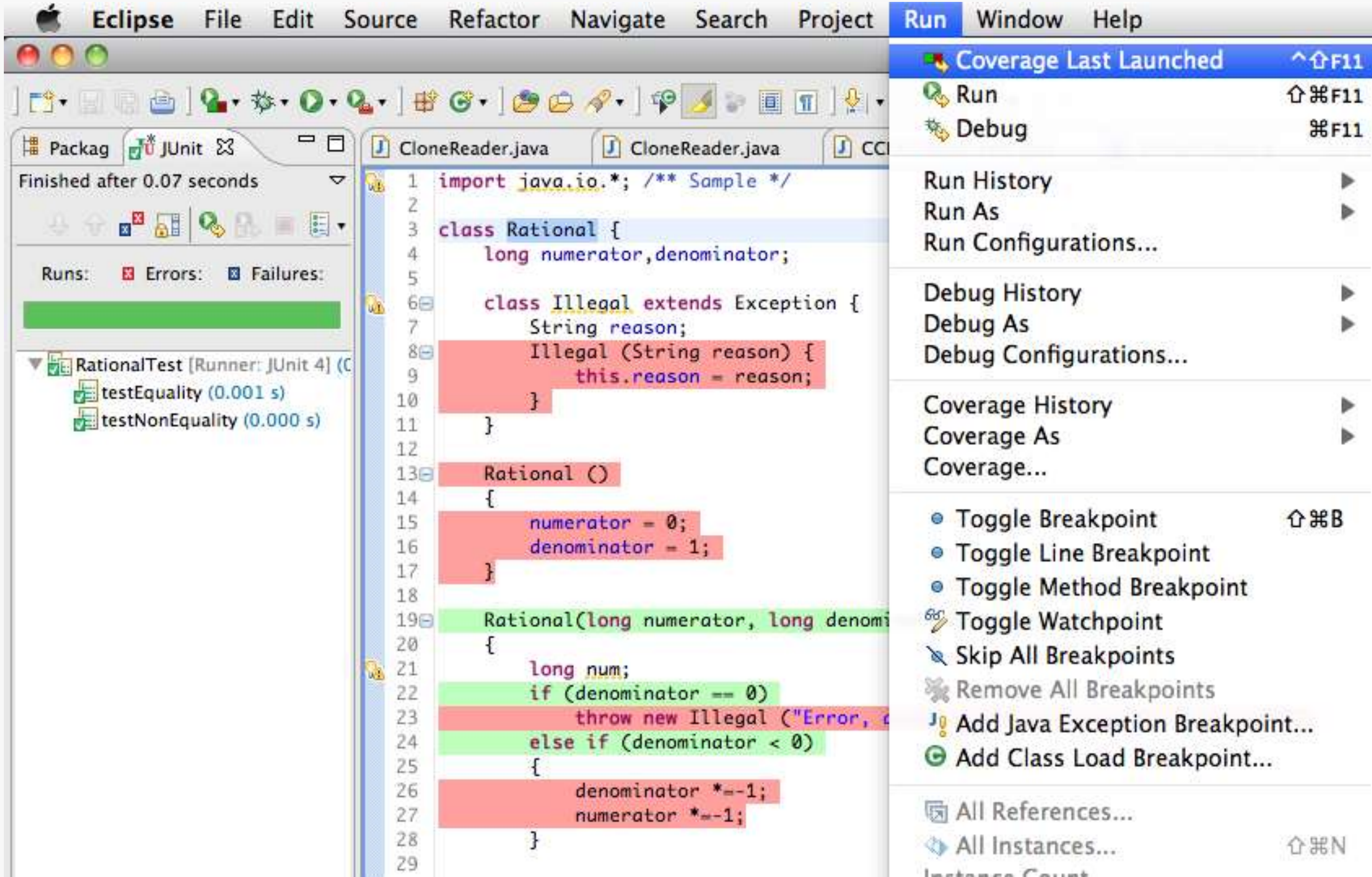
- The data gained in the profiling can also be used for systematic tests.
- Through repeated execution of the program one can strive for a minimum coverage.
- Example: one can ensure that a test executes 95% of all statements at least once, thus making a statement about the quality of the test suite.

Statement Coverage

- Statement coverage:
  % of statements executed by your testing
- Statement coverage = $\frac{\#\ of\ \ executed\ statements}{\#\ of\ \ all\ statements}$
- Variations:
  - Instruction coverage
  - Line coverage

EclEmma

JaCoCo



EclEmma – Coverage View

Problems | @ Javadoc | Declaration | Console | Coverage

RationalTest (Feb 28, 2011 11:32:34 AM)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
src	48.5 %	167	177	344
(default package)	48.5 %	167	177	344
Rational.java	40.1 %	112	167	279
Rational	40.1 %	112	167	279
compareTo(Object)	0.0 %	0	53	53
add(Rational)	0.0 %	0	23	23
subtract(Rational)	0.0 %	0	23	23
divide(Rational)	0.0 %	0	17	17
multiply(Rational)	0.0 %	0	17	17
Rational(long, long)	57.6 %	19	14	33
Illegal	0.0 %	0	9	9
Rational()	0.0 %	0	9	9
toString()	90.0 %	18	2	20
equals(Object)	100.0 %	39	0	39
GCD(long, long)	100.0 %	15	0	15
simplestForm()	100.0 %	21	0	21

Statement and branch coverage

- **Statement coverage** is only a minimal criterion.
- In practice, it is usually required that each branch of the control flow graph is run at least once (so-called **branch coverage**).
- Branch coverage = $\frac{\#\ of\ executed\ branching\ edges}{\#\ of\ all\ branching\ edges}$
- GCOV has the `-b` option which can be used to measure branch coverage.

Branch coverage with GCOV

```
$ gcov -b huffencode.C
 96.69% of 121 source lines executed
 75.00% of 100 branches executed
 69.00% of 100 branches taken at least once
 92.36% of 144 calls executed
Creating huffencode.C.gcov.
$ _
```

```
      153 {
      153     if (tree == 0)
branch 0 taken = 100%
#####           return;
branch 0 never executed

      153     if (tree->isleaf)
branch 0 taken = 50%
       77     {
       77         codes[(unsigned char)tree->l.c] = prefix;
branch 1 taken = 100%
              }
              else
       76     {
       76         init_codes(codes, tree->i.left,  prefix + "0");
       76         init_codes(codes, tree->i.right, prefix + "1");
branch 2 taken = 100%
              }
          }
```

Insights

- The branch of the term `tree == 0` was executed in 100% of all runs.
- The branch from `return` to the end of the function was never executed (since the `return` statement was never executed).
- The branch of the term `tree->isleaf` (branch 0) was executed in 50% of all runs.
- The following branch to the end of the function (branch 1) was executed in 100% of all cases.
- The last branch to the end of the function (branch 2) was also executed in all runs.

Concepts (1/2)

- The aim of **tuning** is the systematic performance increase, where the critical locations are identified and optimised.
- TIME determines the **runtime** of a process and thus the global effect of optimisations.
- The flat profile of GPROF shows how the runtime is divided between the individual functions.

Concepts (2/2)

- The **structured** profile of GPROF shows how the runtime of a function is divided between sub-functions it has called.
- Using GCOV, it is possible to establish **how often** individual lines and branches were executed.
- The GPROF runtime profiling uses **sampling**, and the measured times are subject to statistical **inaccuracy**.
- In contrast, the number of calls and executions reported by GPROF and GCOV are **accurate**.