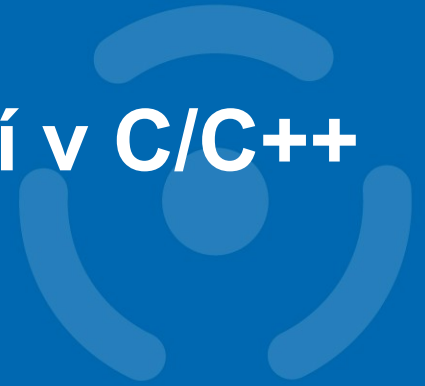


PB173 - Tématický vývoj aplikací v C/C++ (jaro 2016)



Skupina: [Aplikovaná kryptografie a bezpečné programování](#)

Petr Švenda svenda@fi.muni.cz

Konzultace: A406, Pondělí 15-15:50

CRCS

Centre for Research on
Cryptography and Security

Optimization

Optimization steps

1. Do not optimize prematurely - write clean and correct code first!
2. When code works, find performance bottleneck and remove it
3. Document optimization and test it thoroughly

Performance measurement - manual

- Manual speed measure
 1. Measure time **before** target operation
 2. Execute operation
 3. Measure time **after** target operation
 4. Compute and print difference

```
clock_t elapsed = -clock();  
aes256_encrypt_ecb(&ctx, buf);  
elapsed += clock();
```

Manual measurement – possible problems

- It is time consuming
 - additional code, manually inserted
 - less readable, error prone (use DEBUG macro)
- Precision
 - some function returns time in seconds (e.g., `time()`)
 - short operations will take 0
 - prefer functions returning result in ms or CPU ticks
 - e.g., `clock()`
 - check documentation for real precision
 - run operation multiple times (e.g., 1000x)
 - and divide the resulting time by that factor

Manual measurement – possible problems

- Additional unintended overhead may screw the results
 - one-time initialization of objects
 - cache usage, disk swap
 - garbage collection (not in C/C++)
- Need to know the probable bottleneck in advance
 - timing code is inserted manually
 - you are selecting what you like to measure
 - time consuming to localize bottleneck

Automatic measurement - profiling

- Automatic tool to measure time and memory used
- “Time” spend in specific function
- How often a function is called
- Call tree
 - what function called actual one
 - based on real code execution (condition jumps)
- Many other statistics, depend on the tools

Profiling methods

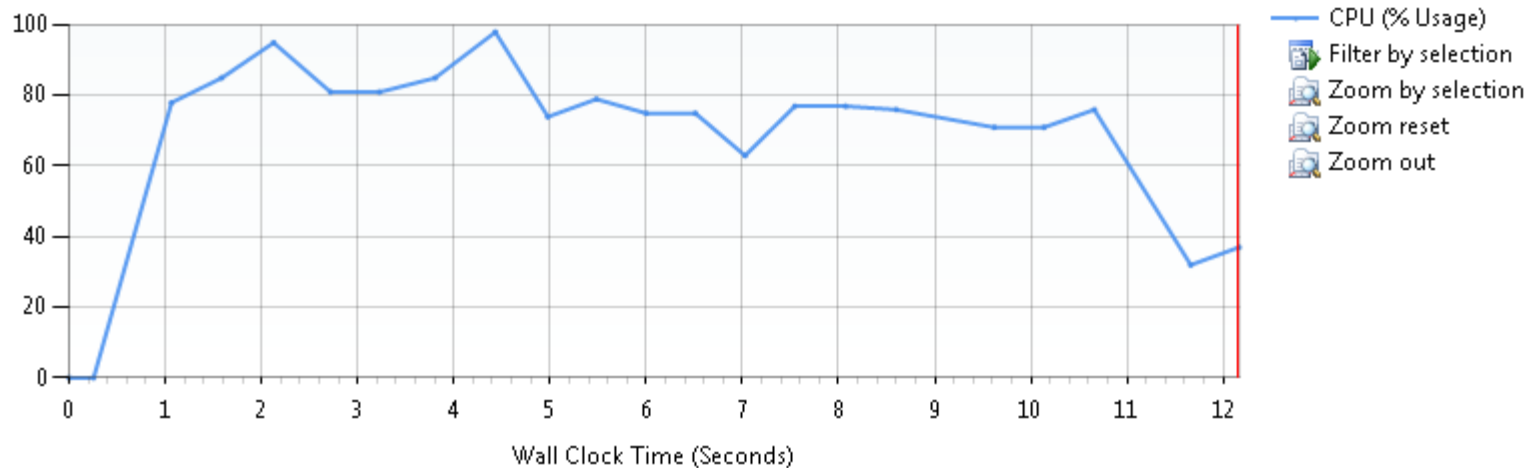
- Profiling method: **CPU Sampling**
 - check periodically what is executed on CPU
 - accurate, low overhead
- Profiling method: **Instrumentation**
 - automatically inserts special accounting code
 - will return exact function call counter
 - (may affect performance timings a bit)
 - additional code present

MS Visual Studio Profiler

- Visual Studio 2013 or earlier
 - Analyze → Launch Performance Wizard
- Visual Studio 2015
 - Debug → Profiler → Start diagnostic tools
- May require admin privileges (will ask)

MS VS Profiler – results (Summary)

- Where to start the optimization work?



Hot Path

The most expensive call path based on sample counts

Name	Inclusive %	Exclusive %
↘ aes_subBytes(unsigned char *)	79.20	0.23
↘ rj_sbox(unsigned char)	78.97	1.26
↘ gf_mulinv(unsigned char)	77.59	0.75
🔥 gf_log(unsigned char)	39.43	39.43
🔥 gf_alog(unsigned char)	37.30	37.30

MS VS Profiler – results (Functions)

- Result given in number of sampling hits
 - meaningful result is % of total time spend in function
- **Inclusive** sampling
 - samples hit in function or its children
 - aggregate over call stack for given function
- **Exclusive** sampling
 - samples hit in exclusively in given function
 - usually what you want
 - fraction of time spend in function code (not in subfunctions)

MS VS Profiler – results (Functions)

pb173_aes101115.vsp × time.h aes32.h pb173_aes.cpp

Current View: Functions

Function Name	Inclusive Samples	Exclusive Samples	Inclusive Samples %	Exclusive Samples %
[pb173_aes.exe]	5	5	0.29	0.29
__RTC_CheckEsp	1	1	0.06	0.06
__tmainCRTStartup	1,740	0	100.00	0.00
_main	1,740	0	100.00	0.00
_mainCRTStartup	1,740	0	100.00	0.00
aes_addRoundKey(unsigned char)	10	10	0.57	0.57
aes_expandEncKey(unsigned char)	322	1	18.51	0.06
aes_mixColumns(unsigned char)	26	10	1.49	0.57
aes_shiftRows(unsigned char)	3	3	0.17	0.17
aes_subBytes(unsigned char)	1,378	4	79.20	0.23
aes256_encrypt_ecb(struct aes256_key)	1,740	1	100.00	0.06
gf_alog(unsigned char)	806	806	46.32	46.32
gf_log(unsigned char)	846	846	48.62	48.62
gf_mulinv(unsigned char)	1,668	14	95.86	0.80
rj_sbox(unsigned char)	24	24	97.36	1.38
rj_xtime(unsigned char)	15	15	0.86	0.86
testProfile(void)	1,740	0	100.00	0.00

Doubleclick to move into Function Details view

46 % of time spend in gf_alog function

Function Code View

d:\documents\develop\pb173\pb173_aes\pb173_aes\aes32.cpp

```
1.4 %      /* ----- */
< 0.1 %    uint8_t gf_alog(uint8_t x) // calculate anti-logarithm gen 3
           {
           uint8_t atb = 1, z;

           while (x--) {z = atb; atb <<= 1; if (z & 0x80) atb^= 0x1b; atb ^= z;}

           return atb;
           } /* gf_alog */
0.3 %
```

- How to speed up `gf_alog` function?

aestab.c

```
AES_RETURN aes_init(void)
{
    uint_32t  i, w;

    #if defined(FF_TABLES)

        uint_8t  pow[512], log[256];

        if(init)
            return EXIT_SUCCESS;
        /* log and power tables for GF(2^8) finite field with
           WPOLY as modular polynomial - the simplest primitive
           root is 0x03, used here to generate the tables
        */

        i = 0; w = 1;
        do
        {
            pow[i] = (uint_8t)w;
            pow[i + 255] = (uint_8t)w;
            log[w] = (uint_8t)i++;
            w ^= (w << 1) ^ (w & 0x80 ? WPOLY : 0);
        }
        while (w != 1);
    // ...

```

MS VS Profiler – save results

- You can save results and compare later
- To check the real impact of your optimization
- Don't forget to eventually stop the optimization 😊

GCC gcov tool

- <http://gcc.gnu.org/onlinedocs/gcc/Gcov.html#Gcov>
- 1. Compile program by GCC with additional flags
 - `gcc -Wall -fprofile-arcs -ftest-coverage main.c`
 - `gcc -Wall --coverage main.c`
 - additional monitoring code is added to binary
- 2. Execute program
 - files with “.bb” “.bbg” and “.da” extension are created
- 3. Analyze resulting files with **gcov**
 - `gcov main.c`
 - annotated source code is created
- Lcov - graphical front-end for gcov
 - <http://ltp.sourceforge.net/coverage/lcov.php>

LCOV - code coverage report

Current view: [top level](#) - [example/methods](#) - [iterate.c](#) ([source](#) / [functions](#))

Test: [Basic example](#) ([view descriptions](#))

Date: 2012-10-12

Legend: Lines: hit not hit | Branches: + taken - not taken # not executed

	Hit	Total	Coverage
Lines:	8	8	100.0 %
Functions:	1	1	100.0 %
Branches:	4	4	100.0 %

Branch data	Line data	Source code
1	:	: /*
2	:	: * methods/iterate.c
3	:	: *
4	:	: * Calculate the sum of a given range of integer numbers.
5	:	: *
6	:	: * This particular method of implementation works by way of brute force,
7	:	: * i.e. it iterates over the entire range while adding the numbers to finally
8	:	: * get the total sum. As a positive side effect, we're able to easily detect
9	:	: * overflows, i.e. situations in which the sum would exceed the capacity
10	:	: * of an integer variable.
11	:	: *
12	:	: */
13	:	:
14	:	: #include <stdio.h>
15	:	: #include <stdlib.h>
16	:	: #include "iterate.h"
17	:	:
18	:	:
19	:	3 : int iterate_get_sum (int min, int max)
20	:	: {
21	:	: int i, total;
22	:	:
23	:	3 : total = 0;
24	:	:
25	:	: /* This is where we loop over each number in the range, including
26	:	: both the minimum and the maximum number. */
27	:	:
28	[+ +]:	67548 : for (i = min; i <= max; i++)
29	:	: {
30	:	: /* We can detect an overflow by checking whether the new
31	:	: sum would become negative. */
32	:	:
33	[+ +]:	67546 : if (total + i < total)
34	:	: {
35	:	1 : printf ("Error: sum too large!\n");
36	:	1 : exit (1);
37	:	: }
38	:	:
39	:	: /* Everything seems to fit into an int, so continue adding. */
40	:	:
41	:	67545 : total += i;

Taken from <http://ltp.sourceforge.net/coverage/lcov/output/example/methods/iterate.c.gcov.html>

Memory consumption profiling

- MSVS Profiler does not provide for native apps
 - unfortunately, available only for managed code
- Visual Studio is detecting memory leaks!
 - `_CrtDumpMemoryLeaks()`
 - run program in debug mode (possibly without any breakpoint)
 - let it finish and watch Output pane
- `Valgrind -v --leak-check=full`
- Write your own new and delete
 - and log the allocated/freed memory

Assignment – performance analysis

- Produce detailed speed estimation for:
 - New user registration
 - User authentication to server
 - Obtain list of other users
 - Prepare protected message for another user
 - Unprotect message from another user
- Which function(s) is consuming most of the CPU?
 - provide a list with %, discuss possible improvements
- Improve performance if significant bottleneck is found

Submissions, deadlines

- Upload application source codes as single zip file into IS Homework vault (Crypto - 6. homework (Performance))
 - Max. 2xA4 with performance analysis
- DEADLINE 11.4. 12:00
 - 0-10 points assigned