PB173 - Tématický vývoj aplikací v C/C++ (podzim 2012)

Skupina: Aplikovaná kryptografie a bezpečné programování

https://minotaur.fi.muni.cz:8443/pb173_crypto

Petr Švenda, *svenda@fi.muni.cz*
Konzultace: G201, Pondělí 16-16:50

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

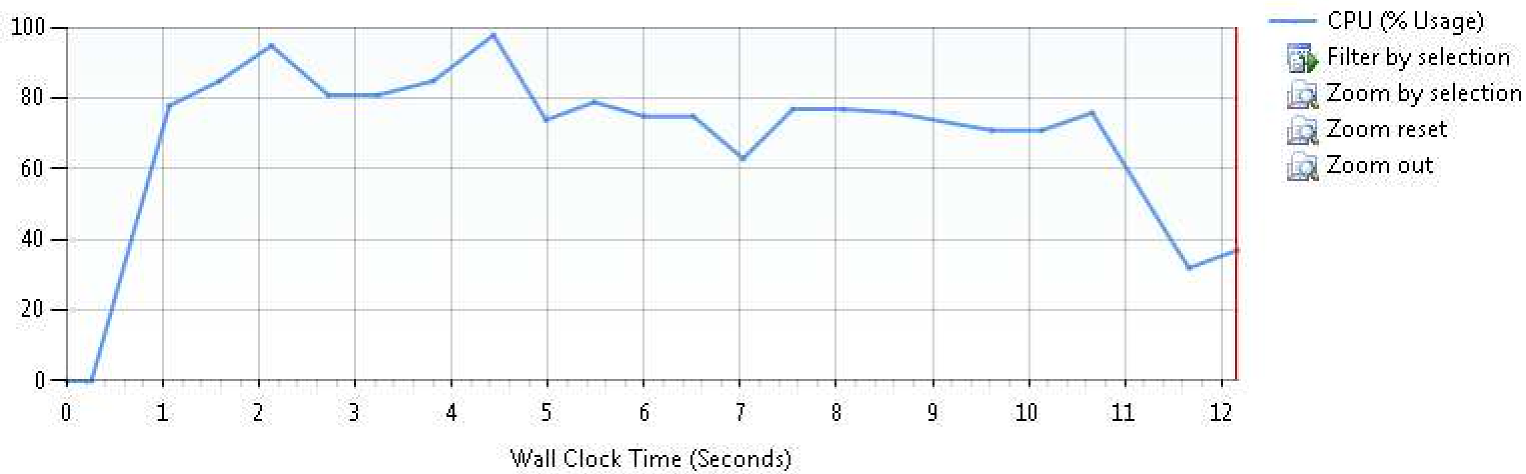
- Automatic tool to measure time and memory used
- “Time” spend in specific function
- How often a function is called
- Call tree
  - what function called actual one
  - based on real code execution (condition jumps)
- Many other statistics, depend on the tools

MS Visual Studio Profiler

- Analyze->Launch Performance Wizard
- 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
- 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	10	10	0.57	0.57
aes_expandEncKey(unsigned	322	1	18.51	0.06
aes_mixColumns(unsigned	26	10	1.49	0.57
aes_shiftRows(unsigned cha	3	3	0.17	0.17
aes_subBytes(unsigned char	1,378	4	79.20	0.23
aes256_encrypt_ecb(struct a	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)	1,694	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

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

 42.0 %       while (x--) {z = atb; atb <<= 1; if (z & 0x80) atb^= 0x1b; atb ^= z;}

< 0.1 %       return atb;
  0.3 %   } /* gf_alog */
```

- 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 ☺

Memory consumption profiling

- MSVS Profiler does not provide for native apps
  - unfortunately
  - available for managed code
- Visual Studio is detecting memory leaks!
  - 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

Optimizing crypto

Optimizing crypto

- Clever tricks both on design and implementation
  - optimization of both algorithm and mode used
  - see aestab.h and aesopt.h for example (Gladman)
- Possibility for pre-computation
  - code itself: macros, templates, static arrays
  - pre-computed tables
    - AES optimized with large tables
    - table lookup only implementation (AES/DES)
    - see http://cr.yp.to/aes-speed.html
  - pre-computed key stream (if mode supports)
    - key stream in advance, then simple xor

Parallelization of operations

- Speedup by parallel execution
- Purpose build hardware
  - cryptographic coprocessors
  - e.g., fast modulo exponentiation
- Using multiple CPU cores
  - multiple threads running
  - http://msdn.microsoft.com/en-us/library/69644x60%28v=VS.80%29.aspx
  - use so-called worker threads

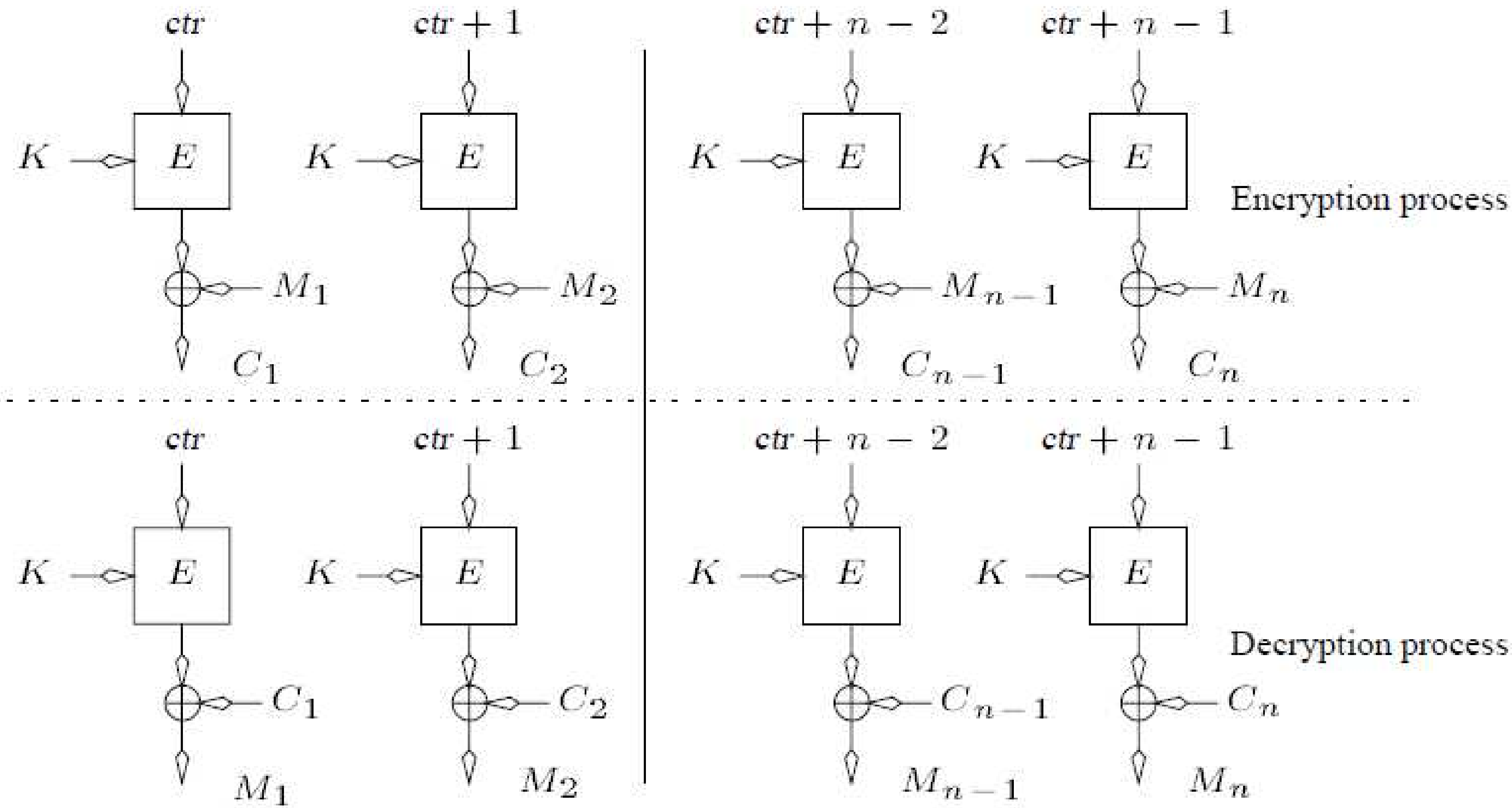
Parallelization of modes

- Assume that algorithm itself is sufficiently optimized
- Algorithm is used in some mode
  - e.g., block encryption modes (ECB, CBC…)
- We need parallelizable modes!
  - CBC encryption is not parallelizable
    - (decryption is – why?)
- Counter (CTR) mode

Counter (CTR) mode for encryption

- Mode approved by NIST (US standardization)
  - http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf
- Designed for confidentiality with parallelization and pre-computation in mind
- Key stream is produced by iterated encryption of the incremental counter
  - counter is incremented for each new block
  - key stream is then xored to message
  - **key stream(== counter) must not repeat with same key**

Counter (CTR) mode for encryption



http://www.mindspring.com/~dmcgrew/ctr-security.pdf

Practical assignment - analysis

- Produce detailed speed estimation for:
  - data package preparation
  - license preparation
  - package access
- Which function(s) is consuming most of the CPU?
  - provide a list with %
- How fast the package with 1MB, 10MB and 100MB can be prepared when required?
  - assume that your program is already running
  - give time in miliseconds

Practical assignment (2)

- Implement encryption of data packets with CTR mode (privacy only, not MAC)
  - pre-compute key stream (e.g., 100MB in RAM array)
  - use parallel threads to prepare key stream
    - number of available cores is parameter for function
    - (at least one thread required ;))
- Document performance gains
  - speed before and after the optimization
  - account correctly for key stream pre-computation