Lecture 1: Introduction, modern computer architecture, compiler, profiling

Michal Sojka
sojkam1@fel.cvut.cz
About this course

- Michal Sojka
  - C/C++, embedded systems, operating systems
- David Šišlák
  - Java, servers, ...
- Scope
  - Writing fast programs
  - Single (multi-core) computer, no distributed systems/clouds
  - Interaction between software and hardware
Lecture outline I.

1. Intro: how to write efficient programs, modern computer architectures, energy consumption

2. Benchmarking, metrics, statistics, WCET, timestamping, profiling (perf, *trace, cachegrind)

3. Program execution – virtual machine, byte-code, Java compiler, JIT compiler, relation to machine code, byte-code analysis, Java byte-code decompilation, compiler optimization, program performance analysis

4. Scalable synchronization – from mutexes to RCU (read-copy-update), transactional memory, scalable API, SIM commutativity

5. JVM concurrency – parallel data accesses, lock monitoring, atomic operations, lock-less/block-free data structures, non-blocking algorithms (fronta, zásobník, množina, slovník)

6. Data serialization – JSON, XML, protobufs, AVRO, cap’n’proto, mmap/shared memory
Lecture outline II.

7. Memory access – cache memory, dynamic memory allocation (malloc, NUMA, …)

8. Efficient servers, C10K problem, non-blocking IO, native cache memory in JVM

9. Representation of objects in JVM – definition loading, materialization of class definition, class initialization, instance initialization, class loader, class finalization, freeing of class definitions

10. JVM memory management – memory organization, data representation, memory management algorithms and their parameters

11. Type of links to instances in Java, efficient cache memory, static and dynamic memory analysis, data structures with reduced memory management overheads, bloom filters

12. Virtualization (IOMMU, SR-IOV, PCI pass-through, virtio, …)

13. Program execution – C compiler (restrict qualifier, optimization), SIMD
Grading

- **Exercise:** 60 points
  - 7× small task
  - semestral work (both C and Java)
  - **Minimum 30 points**

- **Exam:**
  - Written test: 30 points
  - Voluntary oral exam: 10 points
  - **Minimum: 20 points**
Your participation

• There are many techniques how to make your program more efficient
• We will cover only few techniques in this course
• Hardware is still evolving – what was efficient in the past may no longer work today
• We are open to discussion
Efficient software
Efficient software

- There is no theory of how to write efficient software
- Writing efficient software is about:
  - Knowledge of all layers involved
  - Experience in knowing when and how performance can be a problem
  - Skill in detecting and zooming in on the problems
  - A good dose of common sense
- Best practices
  - Patterns that occur regularly
  - Typical mistakes
"All problems in computer science can be solved by another level of indirection"

"...except for the problem of too many layers of indirection."
Layers of indirection in today’s systems

- **Hardware**
  - microcode, ISA
  - virtual memory, MMU
  - buses, arbiters

- **Software**
  - operating system kernel
  - compiler
  - language run-time
  - application frameworks
Hardware optimizations

- **Done by hardware manufacturers**
  - Programmers need to know how to use them properly
- **Instruction-level parallelism**
  - e.g. 2 integer, 2 floating point, 1 MMX/SSE units working in parallel
  - vector instruction (SIMD)
  - Cache hierarchy
  - Prefetching of data
  - Branch prediction

Intel Xeon E5 (Source: extremetech.com)
AMD Bulldozer CPU
Memory hierarchy

Capacity
Access Time
Cost

**CPU Registers**
- 100s Bytes
- 300 – 500 ps (0.3-0.5 ns)

**L1 and L2 Cache**
- 10s-100s K Bytes
- ~1 ns - ~10 ns
- $1000s/ GByte

**Main Memory**
- G Bytes
- 80ns- 200ns
- ~ $100/ GByte

**Disk**
- 10s T Bytes, 10 ms
- (10,000,000 ns)
- ~ $1 / GByte

**Tape**
- infinite
- sec-min
- ~$1 / GByte

**Staging Xfer Unit**
- prog./compiler
  - 1-8 bytes

- cache cntl
  - 32-64 bytes

- cache cntl
  - 64-128 bytes

- OS
  - 4K-8K bytes

- user/operator
  - Mbytes

**Upper Level**
- faster

**Lower Level**
- Larger
## Latencies in computer systems

<table>
<thead>
<tr>
<th>Event</th>
<th>Latency</th>
<th>Scaled</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 CPU cycle</td>
<td>0.3 ns</td>
<td>1 s</td>
</tr>
<tr>
<td>Level 1 cache access</td>
<td>0.9 ns</td>
<td>3 s</td>
</tr>
<tr>
<td>Level 2 cache access</td>
<td>2.8 ns</td>
<td>9 s</td>
</tr>
<tr>
<td>Level 3 cache access</td>
<td>12.9 ns</td>
<td>43 s</td>
</tr>
<tr>
<td>Main memory access (DRAM, from CPU)</td>
<td>120 ns</td>
<td>6 min</td>
</tr>
<tr>
<td>Solid-state disk I/O (flash memory)</td>
<td>50–150 ns</td>
<td>2–6 days</td>
</tr>
<tr>
<td>Rotational disk I/O</td>
<td>1–10 ms</td>
<td>1–12 months</td>
</tr>
<tr>
<td>Internet: San Francisco to New York</td>
<td>40 ms</td>
<td>4 years</td>
</tr>
<tr>
<td>Internet: San Francisco to United Kingdom</td>
<td>81 ms</td>
<td>8 years</td>
</tr>
<tr>
<td>Internet: San Francisco to Australia</td>
<td>183 ms</td>
<td>19 years</td>
</tr>
<tr>
<td>TCP packet retransmit</td>
<td>1–3 s</td>
<td>105–117 years</td>
</tr>
<tr>
<td>OS virtualization (container) system reboot</td>
<td>4 s</td>
<td>423 years</td>
</tr>
<tr>
<td>SCSI command timeout</td>
<td>30 s</td>
<td>3 millennia</td>
</tr>
<tr>
<td>HW virtualization system reboot</td>
<td>40 s</td>
<td>4 millennia</td>
</tr>
<tr>
<td>Physical server system reboot</td>
<td>5 m</td>
<td>32 millennia</td>
</tr>
</tbody>
</table>
Intel-based system (single socket)

Lynnfield CPU. Source: Intel

Intel’s P55 platform. Source: ArsTechnica
Non-Uniform Memory Access (NUMA)
Embedded multi-core system (SoC)

Source: ARM
Nvidia Tegra X1
More detailed diagram of an embedded SoC
Energy is the new speed

- Today, we no longer want just fast software
- We also care about heating and battery life of our mobile phones
- Good news: Fast software is also energy efficient
Power consumption of CMOS circuits

- **Two components:**
  - **Static dissipation**
    - leakage current through P-N junctions etc.
    - higher voltage → higher static dissipation
  - **Dynamic dissipation**
    - charging and discharging of load capacitance (useful + parasitic)
    - short-circuit current

\[
P_{\text{total}} = P_{\text{static}} + P_{\text{dyn}}
\]
Dynamic power consumption, gate delay

\[ P_{\text{dyn}} = a \cdot C \cdot V_{dd}^2 \cdot f \]

- \( a \) – activity factor
- \( f \) – frequency

- **Low power ⇒ slow**
- **Voltage and frequency must be related**
Methods to reduce power/energy consumption

- use better technology/smaller gates
- use better placing and routing on the chip
- reduce power supply $V_{DD}$
- reduce frequency
- reduce activity (clock gating)
- use better algorithms and/or data structures

} dynamic voltage & frequency scaling (DVFS)
Note: ramp-up latency
Software
Bentley's Rules (Writing Efficient Programs)

- **Space-for-Time Rules**
  - Data Structure Augmentation
  - Store Precomputed Results
  - Caching

- **Time-for-Space Rules**
  - Packing
  - Interpreters

**C/C++ compiler**

- gcc, clang (LLVM), icc, ...
- Parsing → syntax tree
- Intermediate representation
- High-level optimizations – HW independent
- Low-level optimizations – HW dependent
- Code generation: IR → machine code
GCC high-level optimizations

- Dead code elimination (if (0))
- Elimination of unused variables
- Constant propagation
  - void func(int i) { if (i!=0) { ... } }
  - func(0); // Nothing happens
- Variable propagation to expressions
  - x = a + const1;
  - if (x == const2) goto ... else goto ...
  - if (a == (const2 - const1)) goto ... else goto ...
- Elimination of subsequent stores (a=1; a=2)
- Loop optimization (operations are replaced by SIMD instructions (MMX, SSE) etc.)
- Simplification of built-in functions (e.g. memcpy).
- Tail call (at the end of a function) can be replaced by a jump.
GCC low-level optimizations

- Common subexpression elimination – intermediate values are stored in temporary variables/registers.
- Selections of addressing modes with respect to their “price”
- Loop optimization (unrolling, modulo scheduling, …)
- Combining multiple operations to one instruction
- Allocation of correct registers for operands and variables, decision of what will be stored on the stack and what in the registers.
  - Variables can be moved between stack and registers during execution
- Instruction reordering for faster execution (optimal use of multiple ALU units in the CPU)
Compiler flags (gcc, clang)

- Optimization level: -O0, -O1, -O2, -O3, -Os (size)
  - -O2 is considered “safe”, -O3 may be buggy
  - Individual optimization passes:
    - -ftree-ccp, -ffast-math, -fomit-frame-pointer, -ftree-vectorize

- Code generation
  - -fpic, -fpack-struct, -fshort-enums
  - Machine dependent
    - -march=core2, -mtune=native, -m32, -minline-all-stringops, ...

- Debugging: -g

- “(p)info gcc” is your friend
Do not trust the compiler :-)

- `gcc -save-temps` – saves intermediate files (assembler)
- `objdump -d` – disassembler
- `objdump -dS` – disassembler + source (needs `gcc -g`)
Example

```c
void vecadd(int * a, int * b, int * c, size_t n) {
    for (size_t i = 0; i < n; ++i) {
        a[i] += c[i];
        b[i] += c[i];
    }
}
```

```c
unsigned a[MM], b[MM], c[MM];
int main() {
    clock_t start,end;
    for (size_t i = 0; i < MM; ++i)
        a[i] = b[i] = c[i] = i;
    start = clock();
    vecadd(a, b, c, MM);
    end = clock();
    printf("time = %Lf\n", (end – start)/ (double)CLOCKS_PER_SEC);
    return 0;
}
```

gcc -Wall -g -O0 -march=core2 -o vecadd *.c ./vecadd       # time = 0.37
gcc -g -O2 -march=core2 -o veclib.o veclib.c ./vecadd       # time = 0.12 ~ 300% speedup

```
vecadd:
    xor    %eax,%eax
    test   %rcx,%rcx
    je     29 <vecadd+0x29>
    nopw   0x0(%rax,%rax,1)
    mov    (%rdx,%rax,4),%r8d
    add    %r8d,(%rdi,%rax,4)
    mov    (%rdx,%rax,4),%r8d
    add    %r8d,(%rsi,%rax,4)
    add    $0x1,%rax
    cmp    %rax,%rcx
    jne    10 <vecadd+0x10>
    retq
```

```c
objdump -d veclib.o
```
veclib.c vecadd.c
Pointer aliasing

- vecadd must work also when called as vecadd(a, a, a, MM)
- Pointer aliasing = multiple pointers of the same type can point to the same memory
  - prevents certain optimizations
- restrict qualifier = promise that pointer parameters of the same type can never alias

```c
void vecadd(int * restrict a, int * restrict b, int * restrict c, size_t n) {
    ...
}
```
- ./vecadd  # time = 0.10, speedup 12%!
Profiling the code
Profiling the code

- “Premature optimization is the root of all evil”
  — D. Knuth
- Software is complex!
- We want to optimize the bottlenecks, not all code
- Real world codebases are big: Reading all the code is a waste of time (for optimizing)
- Profiling: Identifies where your code is slow
Bottlenecks

• Sources
  – code
  – memory
  – network
  – disk
  – ...
  – ...
### Profiling tools

<table>
<thead>
<tr>
<th>In order to do:</th>
<th>You can use:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual instrumentation</td>
<td>printf and similar</td>
</tr>
<tr>
<td>Static instrumentation</td>
<td>gprof</td>
</tr>
<tr>
<td>Dynamic instrumentation</td>
<td>callgrind, cachegrind</td>
</tr>
<tr>
<td>Performance counters</td>
<td>oprofile, perf</td>
</tr>
<tr>
<td>Heap profiling</td>
<td>massif, google-perftools</td>
</tr>
</tbody>
</table>

- **Instrumentation** = modifying the code to perform measurements
Static instrumentation: gprof

- `gcc -pg ... -o program`
  - Adds profiling code to every function/basic block
- `./program`
  - Generates gmon.out
- `gprof program`

Flat profile:

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>%</th>
<th>cumulative</th>
<th>self</th>
<th>seconds</th>
<th>calls</th>
<th>s/call</th>
<th>s/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33.86</td>
<td>15.52</td>
<td>15.52</td>
<td>1</td>
<td>15.52</td>
<td>15.52</td>
<td>15.52</td>
<td>func2</td>
</tr>
<tr>
<td>33.82</td>
<td>31.02</td>
<td>15.50</td>
<td>1</td>
<td>15.50</td>
<td>15.50</td>
<td>15.50</td>
<td>new_func1</td>
</tr>
<tr>
<td>33.29</td>
<td>46.27</td>
<td>15.26</td>
<td>1</td>
<td>15.26</td>
<td>30.75</td>
<td></td>
<td>func1</td>
</tr>
<tr>
<td>0.07</td>
<td>46.30</td>
<td>0.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>main</td>
</tr>
</tbody>
</table>
Event sampling

- **Basic idea**
  - when an interesting event occurs, look at where program executes
  - result is histogram of addresses and event counts

- **Events**
  - time, cache miss, branch-prediction miss, page fault

- **Implementation**
  - timer interrupt → upon entry, program address is stored on stack
  - each event has counting register
    - when threshold is reached, an interrupt is generated
Performance counters

- Hardware inside the CPU (Intel, ARM, ...)
- Software can configure which events to count and when/whether to generate interrupts
- In many cases can be accessed from application code
- Documentation:
  - Intel® 64 and IA-32 Architectures Optimization Reference Manual
  - ARM® Architecture Reference Manual ARMv8, for ARMv8-A architecture profile
perf

- linux-tools package
- Can monitor both HW and SW events
- Can analyze:
  - single application
  - whole system
  - ...
- https://perf.wiki.kernel.org/
perf usage

- perf list
- perf stat -e cycles -e branch-misses -e branches -e cache-misses -e cache-references ./vecadd

Performance counter stats for './vecadd':

<table>
<thead>
<tr>
<th>Event</th>
<th>Value</th>
<th>Percentage (CPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cycles</td>
<td>1,898,543,656</td>
<td>(79.98%)</td>
</tr>
<tr>
<td>branch-misses</td>
<td>267,572</td>
<td>(79.97%)</td>
</tr>
<tr>
<td>branches</td>
<td>348,090,074</td>
<td>(79.95%)</td>
</tr>
<tr>
<td>cache-misses</td>
<td>20,232,628</td>
<td>(79.94%)</td>
</tr>
<tr>
<td>cache-references</td>
<td>26,767,103</td>
<td>(79.93%)</td>
</tr>
</tbody>
</table>

# 0.08% of all branches         (79.97%)
# 75.588 % of all cache refs   (80.51%)

0.619472916 seconds time elapsed
perf usage II.

- perf record -e cycles -e branch-misses ./vecadd
- perf report
Profiler-guided compilation
Excercise – ellipse detection

• Passes
  – scale the image
  – convert to gray
  – blur to have less details
  – find edges
  – find continuous components
    • if component looks roughly like ellipse
      – run RANSAC algorithm to fit the ellipse precisely
Excercise – ellipse detection

- RANSAC algorithm (Random sample consensus)