B4M36ESW: Efficient software
Lecture 1: Introduction

Michal Sojka
michal.sojka@cvut.cz

February 19, 2019
About the course

Outline

1  About the course
2  Basics
3  Hardware
4  Making the hardware faster
   - Caches
   - Instruction-level parallelism
   - Task parallelism
5  Energy
6  Exercise today
   - C/C++ compiler
   - Profiling
About the course

About this course

Teachers

Michal Sojka  C/C++, embedded systems, operating systems
David Šišlák  Java, servers, …

Scope

- Writing fast programs
- Single (multi-core) computer, no distributed systems/cloud
- Interaction between software and hardware
- How general concepts apply to programs in both C/C++ and Java, i.e. how to use hardware efficiently from C/C++ and Java
- The course is not about comparing C/C++ and Java, but you should be able to make this comparison yourself at the end.
About the course

Grading

- Exercises
  - 7 small tasks
  - semestral work (both C/C++ and Java)
  - Maximum 60 points
  - Minimum 30 points

- Exam
  - Written test: max. 30 points
  - Voluntary oral exam: 10 points
  - Minimum: 20 points
Outline

1 About the course

2 Basics

3 Hardware

4 Making the hardware faster
   - Caches
   - Instruction-level parallelism
   - Task parallelism

5 Energy

6 Exercise today
   - C/C++ compiler
   - Profiling
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
In the end, everything is executed by hardware
- Majority of this course is about how to tailor the code to use the hardware efficiently
- C/C++ source code is transformed into native (machine) code by the compiler
  - Compiler tries to optimize the generated code
  - Optimizations are often only heuristics
- Native code is executed directly or invokes OS services
Layers involved in software execution

- Java source code is also compiled
- Java program can execute
  - interpreted by Java Virtual Machine (JVM) or
  - natively after being just-in-time (JIT) compiled by JVM
- JVM is a native program
- Java program can use native libraries (JNI)
- ... long way from source to HW
Fundamental theorem of software engineering

All problems in computer science can be solved by another level of indirection

... except for the problem of too many layers of indirection.

—David Wheeler

Layers of indirection in today’s systems

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>microcode, ISA</td>
<td>operating system kernel</td>
</tr>
<tr>
<td>virtual memory, MMU</td>
<td>compiler</td>
</tr>
<tr>
<td>buses, arbiters</td>
<td>language runtime</td>
</tr>
<tr>
<td></td>
<td>application frameworks</td>
</tr>
</tbody>
</table>
Outline

1  About the course
2  Basics
3  Hardware
4  Making the hardware faster
   ■ Caches
   ■ Instruction-level parallelism
   ■ Task parallelism
5  Energy
6  Exercise today
   ■ C/C++ compiler
   ■ Profiling
CPU – principle of operation

1. Fetch instruction from memory
2. Fetch data from memory
3. Perform computation
4. Store the result to memory

C code and machine code

```c
int a, b, r;
void func() {
    r = a + b;
}
```

```assembly
mov 0x100,%eax ; load a
add 0x104,%eax ; add b
mov %eax,0x108 ; store r
```
Source of many performance problems in today’s computers

Reason: Memory is slow compared to CPUs!

Solution: Caching $\Rightarrow$ memory hierarchy

- **Capacity**
  - 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

- **Access Time**
  - 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

- **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 µs</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>
What distance travels light in vacuum during one 3 GHz CPU clock cycle?

- 10 cm
- Speed of light in silicon is even slower
- Each gate delays the information a bit
- It’s already difficult to pass information quickly from one side of the chip to another
- Physical distance plays important role in the speed of computation
Example: Intel-based system (single socket, 2009)

Intel’s P55 platform
Source: ArsTechnica

Lynnfield CPU
Source: Intel
Hardware designers intensively optimize their hardware

These optimizations improve performance in common (average) cases

Using the HW in “uncommon” ways can drastically degrade the performance

The layers between source code and hardware complicate understanding how is the hardware actually “used”

What are the features that can be problematic from performance point of view?

We will look at them in more detail in the rest of the lectures.
Caches

- **Principle**
  - Smaller but faster memory
  - Take advantage of spacial and temporal locality of memory accesses performed by the code.

- **Problems**
  - Random Access Memory (RAM) is no longer RAM from performance point of view
  - Management of multiple copies of a single data... (known as cache coherence)
Outline

1 About the course
2 Basics
3 Hardware
4 Making the hardware faster
   ▪ Caches
   ▪ Instruction-level parallelism
   ▪ Task parallelism
5 Energy
6 Exercise today
   ▪ C/C++ compiler
   ▪ Profiling
Making the hardware faster – Instruction-level parallelism

Pipelining, branch prediction

Branch = if/else

Example pipeline stages

1. Fetch instruction
2. Decode instruction
3. Calculate operands
4. Fetch operands
5. Execute instruction
6. Write output (result)

- Branch predictor tries to predict branch target and condition
- If it fails, we pay branch penalty
- Here, branch penalty is a few cycles, but it is much more severe with things like superscalar CPUs are involved.
Superscalar CPUs

Instruction stream

\[ r = a + b \]
\[ s = c + d \]
\[ t = e + f \]
\[ u = g + h \]
\[ v = u + i \]

Superscalar execution

\[ r = a + b; \ s = c + d; \ t = e + f \]
\[ u = g + h \]
\[ v = u + i \]

- Goal: Order instructions in a program to use all execution units (e.g. ALUs) in parallel
- Task for the compiler
- Complicates reading of assembler
Example: AMD Bulldozer CPU
Outline

1  About the course
2  Basics
3  Hardware
4  Making the hardware faster
   ■ Caches
   ■ Instruction-level parallelism
   ■ Task parallelism
5  Energy
6  Exercise today
   ■ C/C++ compiler
   ■ Profiling
Multiple CPUs

- Computers usually run multiple programs simultaneously
- Let’s execute them simultaneously on two CPUs
- The CPUs can be on
  - single chip $\Rightarrow$ multi-core
  - multiple chips $\Rightarrow$ multi-socket

Performance problems: synchronization

- Communication between cores (via shared cache or memory interconnect) is slow
- What is communication?
  - Access to shared data in memory
  - Mutex – e.g. to ensure mutually exclusive access to shared data structure in memory
  - synchronized keyword in Java
“Cheaper variant”

Duplicate just the registers, not the execution engines (ALU)

Add HW scheduler to simulate parallel execution

When one HW thread waits for memory, the other can execute

From SW point of view, it looks like a multi-core CPU

Imperfect instruction-level-parallelism (superscalar CPU) is improved by task-parallelism
Non-Uniform Memory Access (NUMA)

- Multi-socket system
- Each socket has locally connected memory
- Other sockets access the memory via inter-socket interconnects (slower, ca 15%)
- Software sees all memory
- SW (OS) should allocate memory local to where it runs, apps could help

Two possible mappings of memory addresses to memory location
Out-of-order execution

Instruction stream

\[
\begin{align*}
  r &= a + b \\
  s &= c + d \\
  t &= e + f \\
  u &= g + h \\
  v &= u + i \\
\end{align*}
\]

\textbf{a} and \textbf{c} are not cached, the rest is:

Superscalar, out-of-order execution

\[
\begin{align*}
  t &= e + f; u = g + h \\
  r &= a + b; s = c + d; v = u + i \\
\end{align*}
\]

- Complicates synchronization
- Other CPUs can see results of computations in different order

When order matters?

\[
\begin{align*}
  &\text{lock} = 1 \\
  &r = a + b \\
  &s = a - b \\
  &\text{lock} = 0 \\
\end{align*}
\]

The above example will likely not work, because accesses to “lock” may be reordered.
Embedded heterogeneous systems
Different CPUs/GPUs on a single chip

Source: ARM
Outline

1. About the course
2. Basics
3. Hardware
4. Making the hardware faster
   - Caches
   - Instruction-level parallelism
   - Task parallelism
5. Energy
6. Exercise today
   - C/C++ compiler
   - Profiling
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

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

Charging the parasite capacities needs energy

**Power consumption**

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

**Gate delay**

\[ t = \frac{\gamma \cdot C \cdot V_{dd}}{(V_{dd} - V_T)^2} \approx \frac{1}{V_{dd}} \]

Low power \( \Rightarrow \) slow
Methods to reduce power/energy consumption

- use better technology/smaller gates (HW engineers)
- use better placing and routing on the chip (HW engineers)
- reduce power supply $V_{DD}$ and/or frequency = Dynamic voltage and frequency scaling (OS job – apps can help)
  - raising it back takes time (ramp-up latency)
  - deciding optimal sleep state to take requires knowing the future
  - recent Android versions have API for “predicting future”
- reduce activity (clock gating = switch off parts of the chip that are not used) [job for OS and HW, apps can help]
- use better algorithms and/or data structures (SW engineers)
Outline

1. About the course
2. Basics
3. Hardware
4. Making the hardware faster
   - Caches
   - Instruction-level parallelism
   - Task parallelism
5. Energy
6. Exercise today
   - C/C++ compiler
   - Profiling
Exercises example

- Ellipse detection using RANSAC algorithm
Outline

1. About the course
2. Basics
3. Hardware
4. Making the hardware faster
   - Caches
   - Instruction-level parallelism
   - Task parallelism
5. Energy
6. Exercise today
   - C/C++ compiler
   - Profiling
C/C++ compiler

- Generates native code from C/C++ source code
- Popular compilers: GCC, Clang (LLVM), icc, MSVC, ...
- Perform many “optimization passes”
  - Those will be covered in a separate lecture
- For now, very brief overview of what you might need today
Compiler flags (gcc, clang)

- Documentation is your friend:
  - Command (p)info gcc
  - https://gcc.gnu.org/onlinedocs/
  - Clang’s flags are mostly compatible with gcc

- Generate debugging information: `-g`
- Optimization level: `-00, -01, -02, -03, -0s (size)`
  - `-02` is considered “safe”, `-03` may be buggy
  - Individual optimization passes:
    - `-free-ccp, -fast-math, -fomit-frame-pointer, -free-vectorize, ...`
  - Find out which optimizations passes are active for given optimization level:
    - `g++ -Q -02 --help=optimizers`

- Code generation
  - `-fpic, -fpack-struct, -fshort-enums`
  - Machine dependent:
    - Generate instructions for given micro-architecture: `-march=haswell, -march=skylake (will not run on older hardware)`
    - Use only “older” instructions, but schedule them for given µarch:
      - `-mtune=haswell, -mtune=native,`
      - `-m32, -minline-all-stringops, ...`
Outline

1. About the course
2. Basics
3. Hardware
4. Making the hardware faster
   - Caches
   - Instruction-level parallelism
   - Task parallelism
5. Energy
6. Exercise today
   - C/C++ compiler
   - Profiling
Profiling

- Profiling: Identifies where your code is slow
- “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)
Bottlenecks

Sources:
- code
- memory
- network
- disk
- ...
Linux Performance Tools

These tools can observe the state of the system at rest, without load.
# 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 the 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>% cumulative</th>
<th>self</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td>calls</td>
<td>s/call</td>
</tr>
<tr>
<td>33.86</td>
<td>15.52</td>
<td>1</td>
<td>15.52</td>
</tr>
<tr>
<td>33.82</td>
<td>31.02</td>
<td>1</td>
<td>15.50</td>
</tr>
<tr>
<td>33.29</td>
<td>46.27</td>
<td>1</td>
<td>15.26</td>
</tr>
<tr>
<td>0.07</td>
<td>46.30</td>
<td>0.03</td>
<td></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':

```
1,898,543,656   cycles   (79.98%)
  267,572    branch-misses   #   0.08% of all branches   (79.97%)
348,090,074    branches   (79.95%)
 20,232,628    cache-misses   #   75.588 % of all cache refs   (80.51%)
26,767,103    cache-references   (80.09%)
```

0.619472916 seconds time elapsed
perf usage II.

- `perf record -e cycles -e branch-misses ./vecadd`
- `perf report`
Useful resources

- Denis Bakhvalov’s blog: https://dendibakh.github.io/notes/
- https://dendibakh.github.io/blog/2019/02/16/Performance-optimization-contest-1