B4M36ESW: Efficient software
Lecture 1: Introduction, TODO

Michal Sojka
michal.sojka@cvut.cz

February 19, 2018
Outline

1. About the course
2. Basics
3. Hardware
4. Making the hardware faster
   - Caches
   - Instruction-level parallelism
   - Task parallelism
5. Energy
6. Software, C/C++ compiler
7. Profiling
8. Exercise today
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
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. Software, C/C++ compiler
7. Profiling
8. Exercise today
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
Layers involved in software execution

- 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

C/C++ source code

C/C++ compiler

Native program

OS

Hardware (CPUs, buses, memory, caches)
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></th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>microcode, ISA</td>
<td></td>
<td>operating system kernel</td>
</tr>
<tr>
<td>virtual memory, MMU</td>
<td></td>
<td>compiler</td>
</tr>
<tr>
<td>buses, arbiters</td>
<td></td>
<td>language run-time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>application frameworks</td>
</tr>
</tbody>
</table>
1 About the course
2 Basics
3 Hardware
4 Making the hardware faster
   - Caches
   - Instruction-level parallelism
   - Task parallelism
5 Energy
6 Software, C/C++ compiler
7 Profiling
8 Exercise today
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
## 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
Outline

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

... and more tricky to use it efficiently from software

- 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.
Making the hardware faster » Caches

Caches

- **Principle**
  - Smaller but faster memory
  - Take advantage of spatial 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...
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...
Making the hardware faster » Instruction-level parallelism

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 efficiently
- Task for the compiler
- Complicates reading of assembler
Example: AMD Bulldozer CPU
Multiple CPUs

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

Performance problems: synchronization
- Communication between cores (via memory interconnect) is slow
- Mutex – e.g. mutually exclusive access to shared data structure in memory
- synchronized keyword in Java
Hyper-threaded CPU

- “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 looks like multi-core CPU
- Imperfect instruction-level-parallelism (superscalar CPU) is improved by task-parallelism
Non-Uniform Memory Access

- 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
### Out-of-order execution

#### Instruction stream

<table>
<thead>
<tr>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>r = a + b</code></td>
</tr>
<tr>
<td><code>s = c + d</code></td>
</tr>
<tr>
<td><code>t = e + f</code></td>
</tr>
<tr>
<td><code>u = g + h</code></td>
</tr>
<tr>
<td><code>v = u + i</code></td>
</tr>
</tbody>
</table>

- `a` and `c` are not cached, the rest is:

#### Superscalar, out-of-order execution

<table>
<thead>
<tr>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>t = e + f; u = g + h</code></td>
</tr>
<tr>
<td><code>r = a + b; s = c + d; v = u + i</code></td>
</tr>
</tbody>
</table>

- Complicates synchronization
- Other CPUs can see operations in different order

#### When order matters

<table>
<thead>
<tr>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lock = 1</code></td>
</tr>
<tr>
<td><code>r = a + b</code></td>
</tr>
<tr>
<td><code>s = a - b</code></td>
</tr>
<tr>
<td><code>lock = 0</code></td>
</tr>
</tbody>
</table>

From a single CPU point of view, everything is correct.
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. Software, C/C++ compiler
7. Profiling
8. Exercise today
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 $\rightarrow$ 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 ⇒ slow
Methods to reduce power/energy consumption

- use better technology/smaller gates
- use better placing and routing on the chip
- reduce power supply $V_{DD}$ and/or frequency = Dynamic voltage and frequency scaling
  - Raising it back takes time (rump-up latency)
  - Deciding optimal sleep state to take requires knowing the future
  - Recent Android versions have API for “predicting future”
- reduce activity (clock gating)
- use better algorithms and/or data structures
Outline

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

objdump -d veclib.o

```
vecadd:
xor    %eax,%eax
test   %rcx,%rcx
je     29 <vecadd+0x29>
nopw   0x0(%rax,%rax,1)
```

```c
vecadd.c
```

```c
vecadd:
xor    %eax,%eax
test   %rcx,%rcx
je     29 <vecadd+0x29>
nopw   0x0(%rax,%rax,1)
```

```c
vecadd:
xor    %eax,%eax
test   %rcx,%rcx
je     29 <vecadd+0x29>
nopw   0x0(%rax,%rax,1)
```

```c
vecadd:
xor    %eax,%eax
test   %rcx,%rcx
je     29 <vecadd+0x29>
nopw   0x0(%rax,%rax,1)
```

```c
vecadd:
xor    %eax,%eax
test   %rcx,%rcx
je     29 <vecadd+0x29>
nopw   0x0(%rax,%rax,1)
```
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%! 
Outline

1. About the course
2. Basics
3. Hardware
4. Making the hardware faster
   - Caches
   - Instruction-level parallelism
   - Task parallelism
5. Energy
6. Software, C/C++ compiler
7. Profiling
8. Exercise today
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>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td>calls</td>
<td>s/call</td>
<td>s/call</td>
</tr>
<tr>
<td>33.86</td>
<td>15.52</td>
<td>1</td>
<td>15.52</td>
<td>15.52</td>
</tr>
<tr>
<td>33.82</td>
<td>31.02</td>
<td>1</td>
<td>15.50</td>
<td>15.50</td>
</tr>
<tr>
<td>33.29</td>
<td>46.27</td>
<td>1</td>
<td>15.26</td>
<td>30.75</td>
</tr>
<tr>
<td>0.07</td>
<td>46.30</td>
<td>0.03</td>
<td></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':

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,898,543,656</td>
<td>cycles</td>
<td>(79.98%)</td>
</tr>
<tr>
<td>267,572</td>
<td>branch-misses</td>
<td>(79.97%)</td>
</tr>
<tr>
<td>348,090,074</td>
<td>branches</td>
<td>(79.95%)</td>
</tr>
<tr>
<td>20,232,628</td>
<td>cache-misses</td>
<td>(75.588%)</td>
</tr>
<tr>
<td>26,767,103</td>
<td>cache-references</td>
<td>(80.09%)</td>
</tr>
</tbody>
</table>

0.619472916 seconds time elapsed
perf usage II.

- perf record -e cycles -e branch-misses ./vecadd
- perf report
Outline

1. About the course
2. Basics
3. Hardware
4. Making the hardware faster
   - Caches
   - Instruction-level parallelism
   - Task parallelism
5. Energy
6. Software, C/C++ compiler
7. Profiling
8. Exercise today
Ellipse detection using RANSAC algorithm