

B4M36ESW: Efficient software

Lecture 1: Introduction

Michal Sojka

`michal.sojka@cvut.cz`



February 19, 2019

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 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.

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**

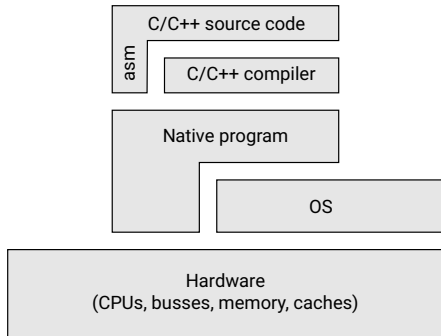
Outline

- 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

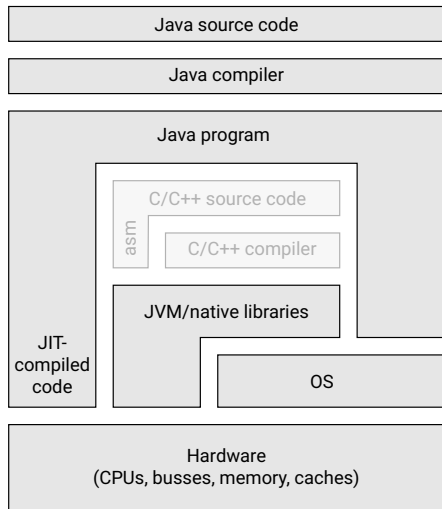
- There is no theory of how to write efficient software
- Writing efficient software is about:
 - Knowledge of all layers involved
 - Experience in knowing when and how performance can be a problem
 - Skill in detecting and zooming in on the problems
 - A good dose of common sense
- Best practices
 -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

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

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-it-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

Hardware

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

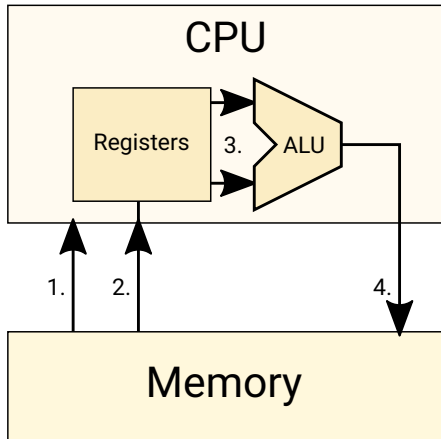
Software

- operating system kernel
- compiler
- language runtime
- application frameworks

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

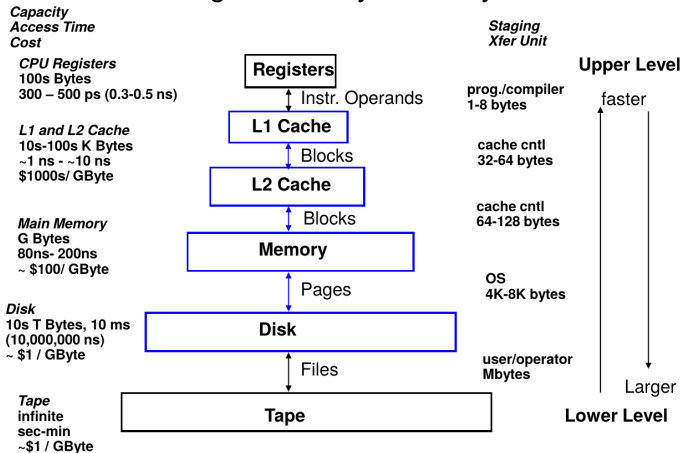
C code and machine code

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

mov 0x100,%eax ; load a
add 0x104,%eax ; add b
mov %eax,0x108 ; store r
```

Memory

- 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

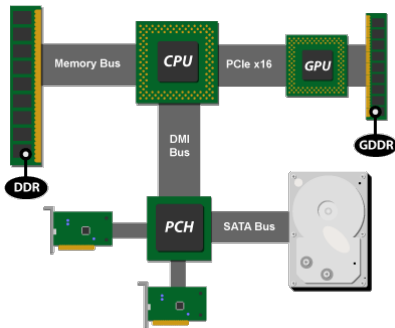
Event	Latency	Scaled
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access (DRAM, from CPU)	120 ns	6 min
Solid-state disk I/O (flash memory)	50–150 μ s	2–6 days
Rotational disk I/O	1–10 ms	1–12 months
Internet: San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

Computer performance and laws of physics

What distance travels light in vacuum during one 3 GHz CPU clock cycle?

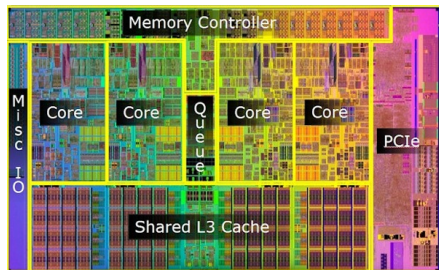
- 10 cm
- Speed of light in silicon is even slower
- Each gate delays the information a bit
- It's already difficult to pass information quickly from one side of the chip to another
- Physical distance plays important role in the speed of computation

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 Exercise today
 - C/C++ compiler
 - Profiling

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.

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

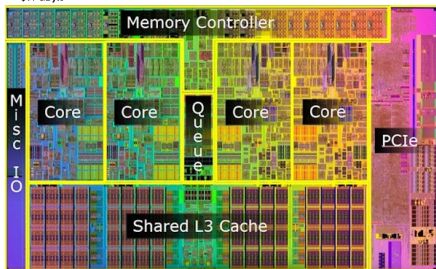
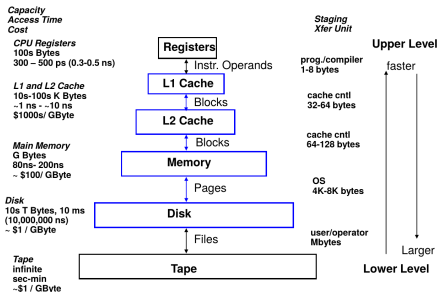
Caches

■ 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**)



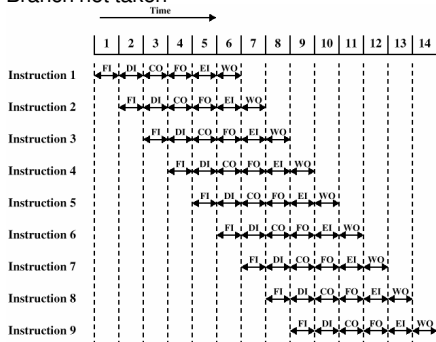
Outline

- 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

Pipelining, branch prediction

Branch = if/else

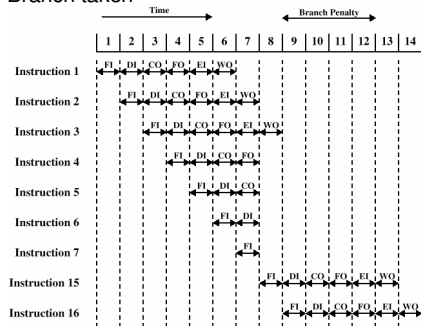
Branch not taken



Example pipeline stages

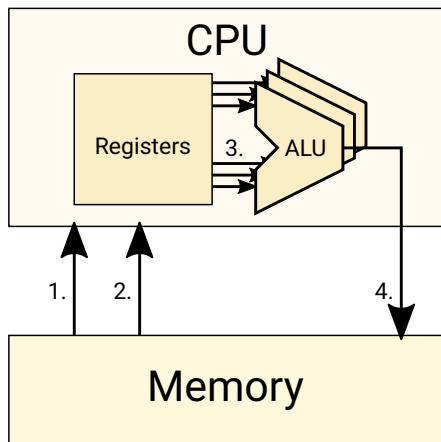
- 1 Fetch instruction
- 2 Decode instruction
- 3 Calculate operands
- 4 Fetch operands
- 5 Execute instruction
- 6 Write output (result)

Branch taken



- 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
  
```

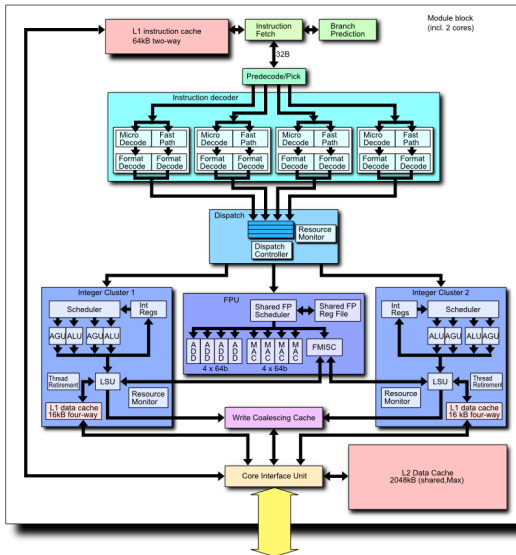
Superscalar execution

```

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

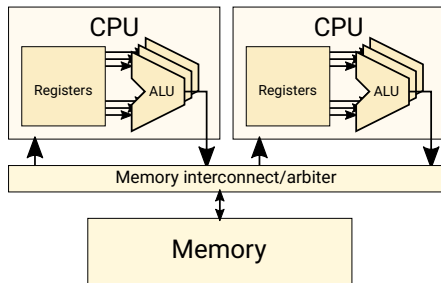
Example: AMD Bulldozer CPU



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

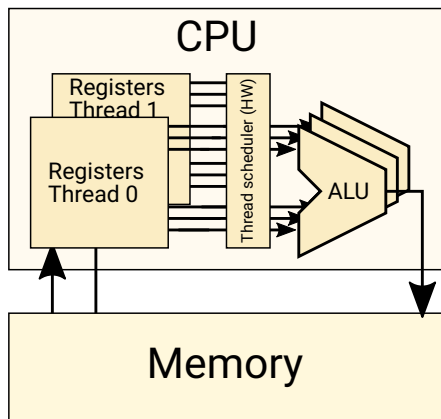


- Computers usually run multiple programs simultaneously
- Let's execute them simultaneously on two CPUs
- The CPUs can be on
 - single chip \Rightarrow multi-core
 - multiple chips \Rightarrow multi-socket

■ Performance problems: synchronization

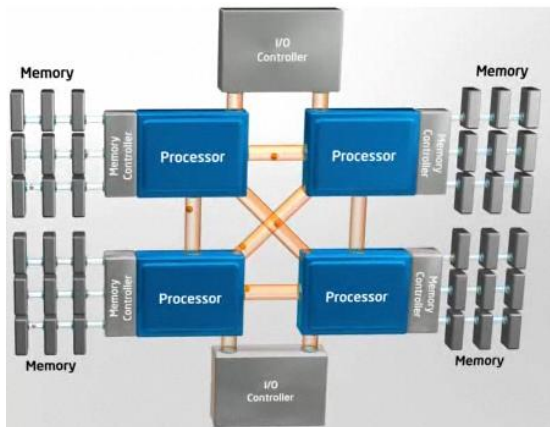
- Communication between cores (via shared cache or memory interconnect) is slow
- What is communication?
 - Access to shared data in memory
 - Mutex – e.g. to ensure mutually exclusive access to shared data structure in memory
 - synchronized keyword in Java

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, it looks like a multi-core CPU
- Imperfect instruction-level-parallelism (superscalar CPU) is improved by task-parallelism

Non-Uniform Memory Access (NUMA)



0 GB 8 GB 16 GB 24 GB 32 GB



- 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

```

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

```

a and **c** are not cached, the rest is:

Superscalar, out-of-order execution

```

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

```

From a single CPU point of view,
everything is correct

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

When order matters?

```

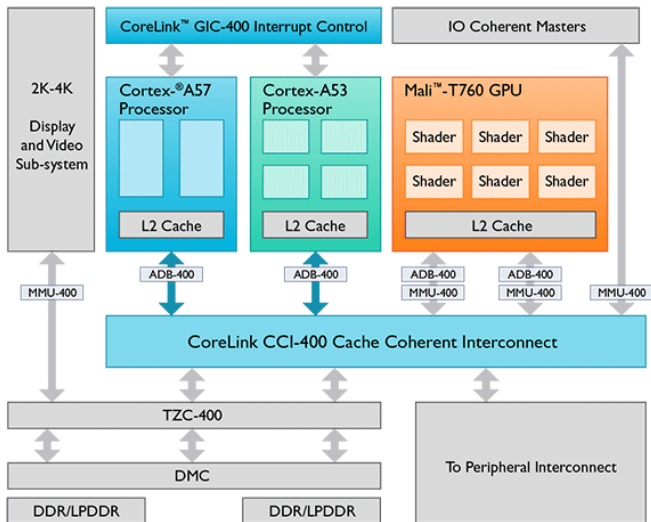
lock = 1
r = a + b
s = a - b
lock = 0

```

The above example will likely not work, because accesses to “lock” may be reordered.

Embedded heterogeneous systems

Different CPUs/GPUs on a single chip



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



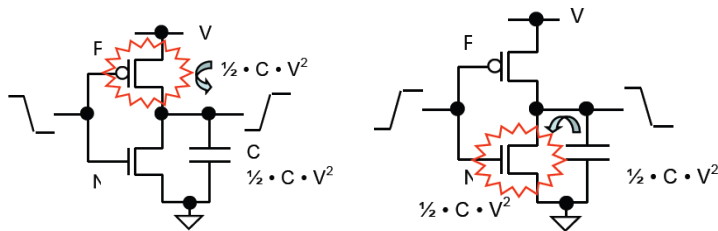
Power consumption of CMOS circuits

Two components:

- Static dissipation
 - leakage current through P-N junctions etc.
 - higher voltage \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

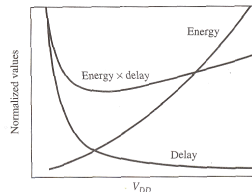
Power consumption

$$P_{dyn} = a \cdot C \cdot V_{dd}^2 \cdot f$$

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

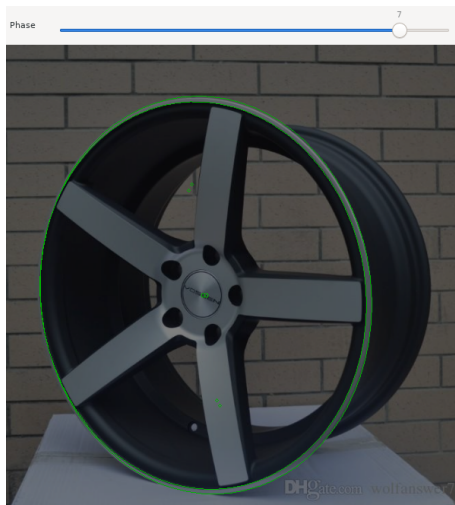
- use better technology/smaller gates (HW engineers)
- use better placing and routing on the chip (HW engineers)
- reduce power supply V_{DD} and/or frequency = Dynamic voltage and frequency scaling (OS job – apps can help)
 - raising it back takes time (rump-up latency)
 - deciding optimal sleep state to take requires knowing the future
 - recent Android versions have API for “predicting future”
- reduce activity (clock gating = 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: `-O0`, `-O1`, `-O2`, `-O3`, `-Os` (size)
 - `-O2` is considered “safe”, `-O3` 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 -O2 --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 for given march:
 - `-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)

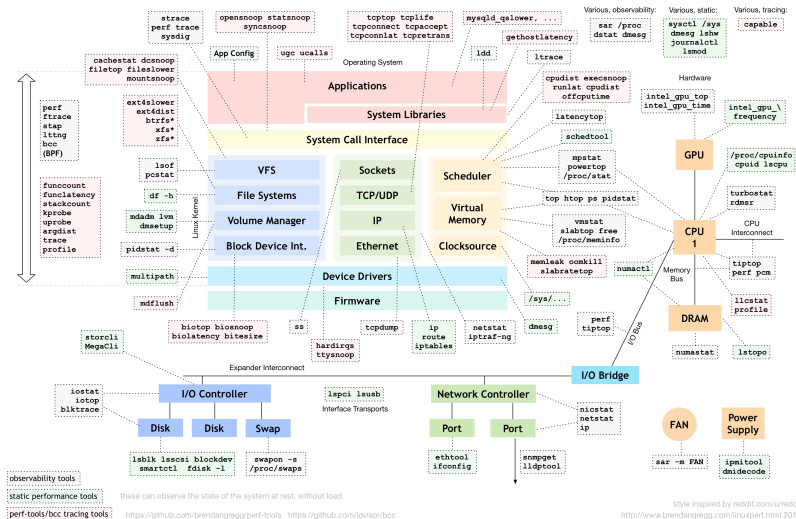
Bottlenecks

Sources:

- code
- memory
- network
- disk
- ...

Linux Performance Tools

Linux Performance Tools



Profiling tools

In order to do:	You can use:
Manual instrumentation	printf and similar
Static instrumentation	gprof
Dynamic instrumentation	callgrind, cachegrind
Performance counters	oprofile, perf
Heap profiling	massif, google-perftools

- **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.

%	cumulative	self		self	total	
time	seconds	seconds	calls	s/call	s/call	name
33.86	15.52	15.52	1	15.52	15.52	func2
33.82	31.02	15.50	1	15.50	15.50	new_func1
33.29	46.27	15.26	1	15.26	30.75	func1
0.07	46.30	0.03				main



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 Software Developer's Manual, Volume 3: System Programming Guide
 - 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>