Performance-optimised computing – Week 4.

Tasks, System calls, PMU & Exercises

Dr. Bakay Árpád – Ericcson

- Branch pred example?
    - In java: https://www.kodewerk.com/java/2023/09/04/code_without_branching.html
- **High context switch and interrupt rates - tools to analyz CS**
    - Vmstat
- CPU affinity of running on another CPU
- Explain the top command
-

- Elhunyt Béládi László | MTA
- Linux kernel & likely/unlikely

Summary of topics discussed so far

- Central components of CPU Core: instructions, registers, ALU, FPU, and vector execution unit.
  - Performance tricks: instruction pipelineing with branch prediction, execution reordering, parallel execution units, complex ISA with domain targeted instructions + hyperthreading.
  - Things we shall care about: avoid slow instructions, make branches predictable, or do it in branch-less way. Let the compiler optimize, but you may also analyze and improve the hotspots (e.g. with avx512 vectors).
- Caches and access times to memory (and to storage)
  - Performance tricks: 3 level caching, prefetch, coherence mechanism for multi-core and multi-socket
  - What we should care about: temporal & spatial locality, avoid frequently used shared data, assign communicating processes to same CPU (or core), use smaller datatypes, packed into 64 bits.
- Virtual addressing & Memory Paging
  - Performance tricks: Translation Lookaside Buffer (+ only 48 bits used to save level count)
  - What we should care about: avoid swapping, use shared libraries

A few things to add...

- Branch prediction vs. ‚branch likelyness prefix'
- Page faults

Missing from last class 1.

- **Minor an major page faults** - the Kernel has to intervene in all cases, after an MMU exception [„trap"]
  - Minor (soft) -> no disk operations: „administration" (and memset/copy) only
    - A new empty (=zeroed) page is requested
    - mmapped data (backed by file) is accessed for read
      - Page is already in RAM
      - Not yet loaded
    - A new empty page is first written - „zero-on-write"
    - A mmapped (=non-zero) data is first written - „copy-on-write"
  - Major -> mapping exists, but not in physical RAM
    - First read access a new page in a mmapped file.
    - Reading a swapped memory page, 💀
    - Reading a swapped memory page with eviction of some clean page
    - Reading a swapped memory page with eviction of a dirty page to cache 💀💀💀 -> thrashing

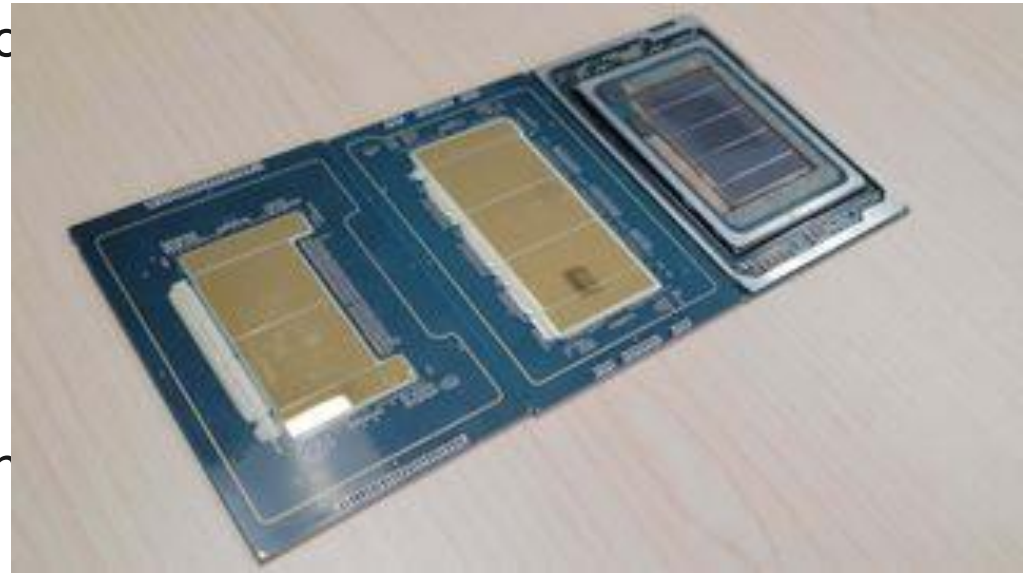
Full configuration with Modern Xeon-s

(e.g. „Granite Rapids" Intel® Xeon® 6980P)

- 128 Cores

  AMD equivalent: 5th Generation AMD EPYC™ Processors

  - 256 threads through hyperthreading
  - 6-path Ultra-Path Interconnect for multi-socket machines, 24 GT/sec
  - 504 Mbytes of L3 cache
- 4 memory controllers, w. 3 channels each, 2 DRAM slo
- Rated thermal power: 500W!!



- For now, up to 4 CPU sockets per computer
  - … But 8-socket Xeon 6 variants are coming!
- Memory is socket-bound: NUMA or Distr.Shared.Mem
  - Cf. „Simmetric MultiProcessing"

Linux Task scheduling

- Make sure tasks (processes, threads) get a **fair share of CPU time (a.k.a. multitasking)**
  - Process -> address space
  - Thread -> execution and scheduling. But: threads of a process are grouped to run from a common „time budget"
- **Context switch** is the manouver to assign a new thread to a „virtual CPU" (a core's hyper-thread)
  - Initiated by:
    - Thread voluntary releasing the CPU (IO or timer wait)
    - Timer interrupts the thread.
  - It requires storage and refresh of all process state (from kernel memory): registers (all kinds of), paging table master pointer, etc. + TLB flush , typically also affects memory caches -> **C.S. is expensive**
  - How frequently does that happen?
    - /proc/sys/kernel/sched_latency_ns: the maximum time a thread should wait for cpu -> typical: 10-30 ms
    - /proc/sys/kernel/sched_min_granularity_ns: the min time allocated for each thread started -> typical: 2-4 ms
  - Interrupt handling also implies a C.S.
- Performance considerations: optimize allocation of processes to CPU-s (be consistent)
  - ‚cpu_affinity' - set with **taskset** command
  - Especially important on multi-socket machines with „non-uniform memory architecture" [NUMA]

System calls & privilege levels

- **System calls** are essential to execute principal operations in kernel on behalf user processes syscalls(2) - Linux manual page (man7.org)
  - Create new processes (fork + exec), wait for children (wait)
  - Terminate current process (exit), or other process (kill)
  - File, device, network access (open / [connect] / read/write / close)
  - File modes and directory operations, user, group identity (geteuid, getegid, seteuid)
  - Get/set system time(s)
    - Inter-process communication (shared memory, semaphores, mutexes, etc)ű
  - Allocate additional virtual memory: mmap
  - Etc.
  - **Exception:** malloc – allocations are not handled by the kernel, only when heap needs an extension: sbrk

- Methods are implemented in **kernel's address space**
- Execution requires CPU in „**Privilege level 0"** mode, i.e. **„user mode"** to **„kernel mode"** transition implied
  - SYSCALL and SYSRET X64 instructions make a call and also elevate into (or leave) privileged mode SYSCALL — (felixcloutier.com)
  - This is not a context switch, as we remain in the same process!!!

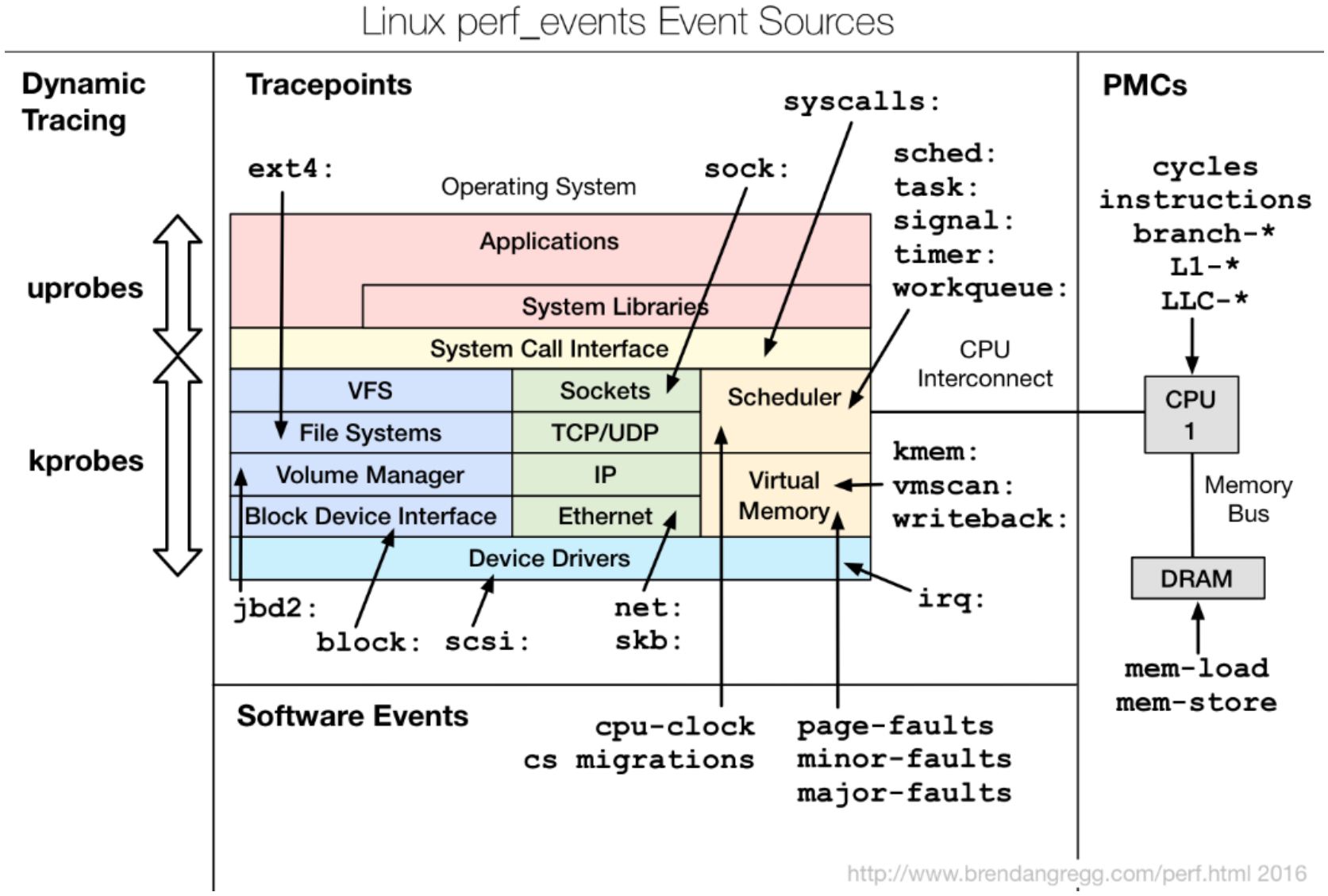
X64 CPU Performance Monitoring Unit (PMU)

- Hundreds of counters maintaned by the X64 CPU, for external perf. monitoring.
  - instructions, cycles,
  - caches' use and miss,
  - TLB use and misses,
  - page faults, context swiches,
  - sys calls count
  - various clocks
  - etc, etc...

- For processor-specific details, see dedicated Intel website: https://perfmon-events.intel.com/

- These can be accessed from special „internal registers", through the RDPMC instruction: RDPMC (felixcloutier.com)
  - This again requires **privilege level 0** mode in CPU, i.e. accessible through a syscall

Perf: generic tool for performance avaluation in Linux



perf

- **perf list** - list supported events – not all are supported by all kernels
  - Software events -> e.g. faults, ctx.swithces
  - Hardware events cpu cycles, instrs, branch hits/misses + cache hits/misses, power, etc.
  - MMU/IMC read / write accesses (e.g. uncore_imc_1/cas_count_read/)
  - Kernel tracepoints
- **perf stat** <command ...> - run <command> and report 1 or several counters on termination
  - Selectable for thread, process, CPU or system (-a)
    - Also can include children threads and processes
- **perf record** <command> - start command and save counters periodically
    - All info is stored in a file (**perf.data** by default), read with perf report
    - Possible **to trace individual functions**
- **perf report** - create a report of recorded perf data.

Exercise: create a small C program and use perf to analyze

- main() shall call 3-4 functions in a loop 1-10 million times or 2-3 secs), with each function processing the result of the previous one (i.e. a „data pipeline”). Example: https://gitlab.inf.elte.hu/-/snippets/32
- Most functions shall call some 1-2 simple library methods like rand, atoi, itoa, strcpy, memcpy, strchr, memchr, malloc, printf, sscanf.
- Compile and test your code.

- Install **perf** on your linux (package depens on distribution, on Debian: **linux-tools-generic**).
- Use the following perf variants to analyze your program:
  - `perf stat -- ./myprog` - execution times, instructions, cycles, branch prediction, mode info with –d -d
  - `perf stat -d -- ./myprog` - cache access / miss statistics - hopefully
  - `perf record -- ./myprog` - create call stack snapshots
  - `perf report --stdio` - show all invoked methods, with percentages of CPU time.

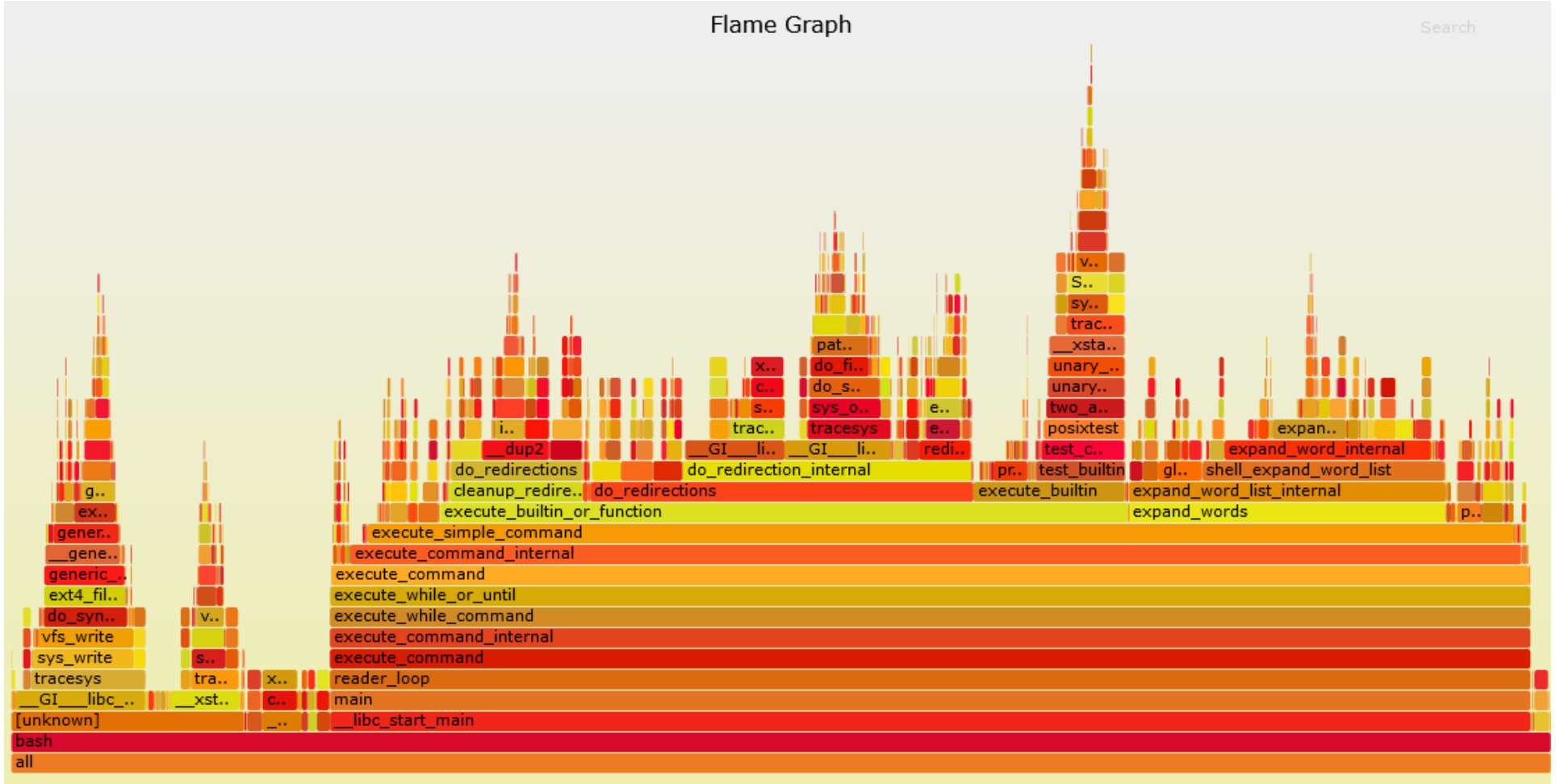
Perf – advanced recording with call tree info

- Default recording
  - perf record – <mypog>
  - Displays all functions called + share of the that function within total time,
- Recording with call tree:
  - perf record –call-graph dwarf -- <myprog>
  - Perf report displays call tree of methods:

```
87.83%     0.50%  snippet32  snippet32          [.] main
           |--87.33%--main
           |          |--51.51%--scan_fun
           |          |           --50.14%--__GI___isoc99_sscanf (inlined)
           |          |                     |--42.96%--__vfscanf_internal
           |           --35.82%--print_fun
           |                     --35.03%--__sprintf (inlined)
           |                               --33.55%--__vsprintf_internal
```

Advanced visualization of CPU usage: Flamegraphs

https://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html#FlameGraph



PAPI - Performance Application Programming Interface

- A „standardized“ interface for consistent, uniform access to various performance counters in
  - CPU and GPU
  - Also: Memory, IO, Interconnects

Think about YOUR performance project
Deadline for ideas: 20 March 2025

Option 1:

- **Benchmark** „anything to anything"
- One of them shall be your code (compared to another existing solution).
- Evaluate the relative performance in various scenarios.
- Create a nice document which describes your findings
- Propose your project by class #6 on 10.14 – we will analyze and discuss.
- Can be done in pairs, in case scope is larger and individual responsibilities are defined

- E.g.
  - I will create a TCP proxy in my favourite language, which receives TCP messages from „producers" and forwards it to a number of registered „subscribers"
  - I will compare this to 2 open source solutions, e.g. Mosquito Eclipse Mosquitto, Kafka

Think about YOUR performance project

<u>Option 2:</u>

- **Deep performance analysis of a tricky computing task.**
- Primarily for C programmers (may be higher level languages, but let we need to discuss)
- Optimize your code based on experience, submit various versions

- E.g.
  - I will create a solution for the **1Billion Record Challenge.**
  - First I will create a solution with „decent performance"
  - Then I will analyze and improve this

That's it for today!