CPU Hardware Architecture and Performance Optimization

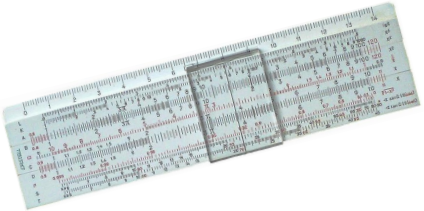
G. Amadio (CERN)

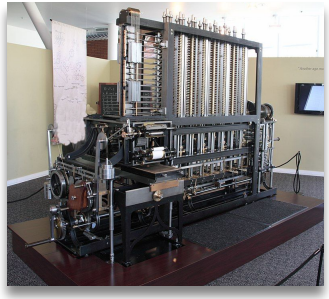
Our Main Goals

- Understand the architecture of modern CPU hardware
  - Hardware evolution
  - Main features of modern hardware
- Understand how to analyze the performance of our code
  - How to identify performance bottlenecks
  - What to measure and how to measure it
- Combine architectural knowledge and performance analysis
  - How to interpret performance measurements
  - What changes to make to the software

CPU Hardware Architecture and Evolution

Early Computing Devices





2700 – 2300 BC

Abacus

Used since ancient times, until Arabic numerals became the norm. Still in use as an educational tool.

1620– 1630

Slide Rule

Uses logarithm scales to help with multiplications and also computing other functions. Extensively used by engineers in the last century, before computers became powerful.

1642

Pascaline

Mechanical calculator invented by Blaise Pascal to help his father with tax calculations. Could add and subtract.

1820s

Difference Engine

Automatic mechanical calculator designed to tabulate polynomial functions. Designed and first created by Charles Babbage.

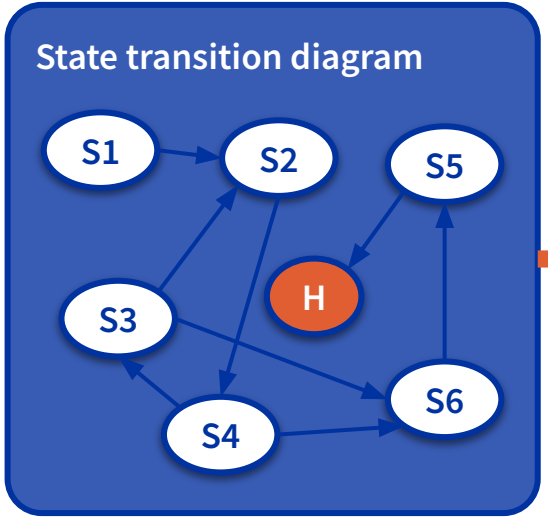
Images: Wikipedia

Ada Lovelace, the first computer programmer

Augusta Ada King, Countess of Lovelace (10 December 1815 – 27 November 1852) was an English mathematician and writer, known for her work on Charles Babbage's proposed mechanical general-purpose computer, the Analytical Engine. She was the first to recognize that the new machine had applications beyond simple calculations. She arguably wrote the first “computer program”. In her article entitled “note G” on the Analytical Engine, she described in detail an algorithm to compute a sequence of Bernoulli numbers using it.

Source: Wikipedia

The Turing Machine: concept of first generic computer



Universal Turing Machine

A Turing machine is capable of computing any computable sequence.

HEAD: READ/WRITE/MOVE

1 0 1 1 0 0 1 0 0 1 0 0 0 0 1 1 0 1 1 0 0 1

Infinite tape

doi:10.1112/plms/s2-42.1.230

From Turing Machine to Stored-Program Computer

1936

Turing Machine

Conceptually the first general computing machine.

1937

Harvard IBM Mark I

Inspired on the Analytical Engine. One of the earliest general-purpose electromechanical computers. First computer bug discovered on it by Grace Hopper.

1942-1945

ABC, Colossus, ENIAC

First truly digital computers, based on boolean logic and vacuum tubes.

1947

Solid-State Transistor

The first solid-state transistor was a based on a point-contact connection to a crystal by closely spaced thin gold foils.

1949

Assembly Language

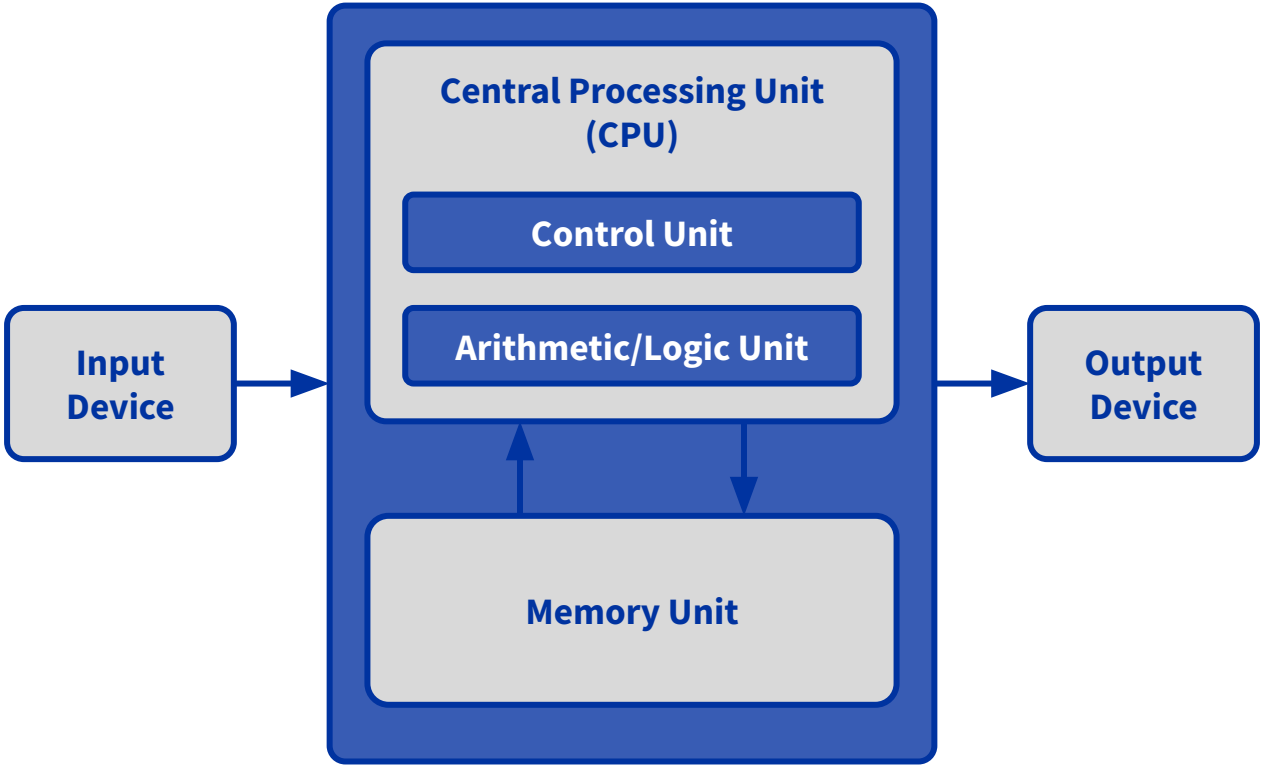
Beginning of standardization of how to program computer with abstract instruction sets.

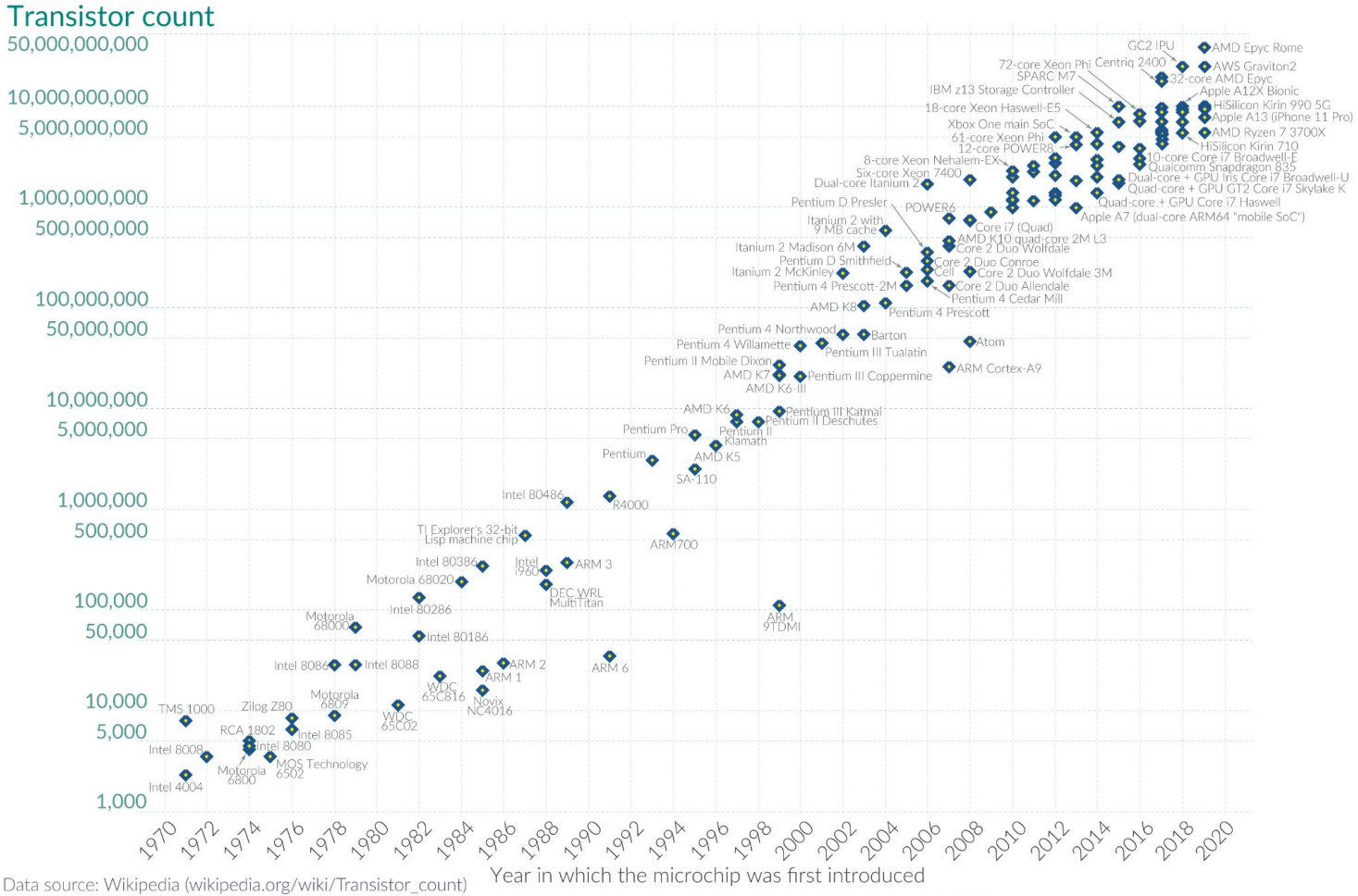
1950

EDVAC

First stored-program computer, based on John von Neumann's architecture concept from 1945.

John von Neumann Architecture



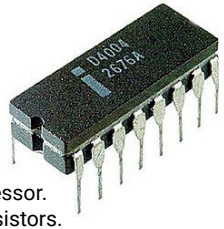


Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

OurWorldinData.org – Research and data to make progress against the world's largest problems.

Integrated Circuit-Based Microprocessors

Intel 4004



First Intel microprocessor.
2,300 Transistors.

MOS 6502

Powered many popular devices, such as the Apple II, Atari 2600, Commodore 64, and the NES.
3,510 Transistors.

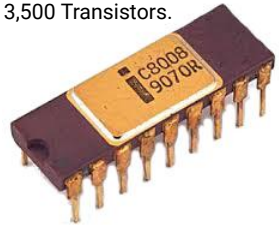
Intel 8086

First 16-bit microprocessor. 29,000 Transistors.
Its successor, the Intel 8088, a slightly modified version, powered first IBM PC.

1970 | 1972 | 1975 | 1976 | 1978 | 1985

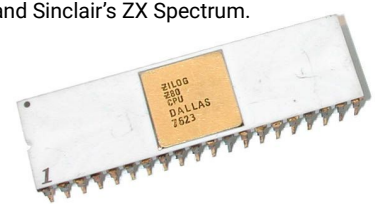
Intel 8008

First 8 bit microprocessor.
3,500 Transistors.



Zilog Z80

8-bit microprocessor.
Powered devices such as
Sega Master System and Mega Drive,
and Sinclair's ZX Spectrum.



Intel 80386

32-bit microprocessor.
275,000 Transistors, 33MHz.
Cemented Intel's PC market dominance.

Intel’s 8086 Registers and Assembly

$^{1}_{9}$ $^{1}_{8}$ $^{1}_{7}$ $^{1}_{6}$ $^{1}_{5}$ $^{1}_{4}$ $^{1}_{3}$ $^{1}_{2}$ $^{1}_{1}$ $^{1}_{0}$ $^{0}_{9}$ $^{0}_{8}$ $^{0}_{7}$ $^{0}_{6}$ $^{0}_{5}$ $^{0}_{4}$ $^{0}_{3}$ $^{0}_{2}$ $^{0}_{1}$ $^{0}_{0}$			*(bit position)*
Main registers			
	AH	AL	**AX** (primary accumulator)
0000	BH	BL	**BX** (base, accumulator)
	CH	CL	**CX** (counter, accumulator)
	DH	DL	**DX** (accumulator, extended acc)
Index registers			
0000	SI		**S**ource **I**ndex
0000	DI		**D**estination **I**ndex
0000	BP		**B**ase **P**ointer
0000	SP		**S**tack **P**ointer
Program counter			
0000	IP		**I**nstruction **P**ointer
Segment registers			
CS		0000	**C**ode **S**egment
DS		0000	**D**ata **S**egment
ES		0000	**E**xtra **S**egment
SS		0000	**S**tack **S**egment
Status register			
- - - - O D I T S Z - A - P - C			Flags

```
; _strtolower:
; Copy a null-terminated ASCII string, converting
; all alphabetic characters to lower case.
; ES=DS
; Entry stack parameters
;   [SP+4] = src, Address of source string
;   [SP+2] = dst, Address of target string
;   [SP+0] = Return address
;
_strtolower proc
            push    bp              ;Set up the call frame
            mov     bp,sp
            push    si
            push    di
            mov     si,[bp+6]        ;Set SI = src (+2 due to push bp)
            mov     di,[bp+4]        ;Set DI = dst
            cld                      ;string direction ascending

loop:       lodsb                    ;Load AL from [si], inc si
            cmp     al,'A'           ;If AL < 'A',
            jl      copy             ; Skip conversion
            cmp     al,'Z'           ;If AL > 'Z',
            jg      copy             ; Skip conversion
            add     al,'a'-'A'       ;Convert AL to lowercase
copy:       stosb                    ;Store AL to [di], inc di
            or      al,al            ;If AL <> 0,
            jne     loop             ; Repeat the loop

done:       pop     di               ; restore di and si
            pop     si
            pop     bp               ;Restore the prev call frame
            ret                      ;Return to caller
            end     proc
```

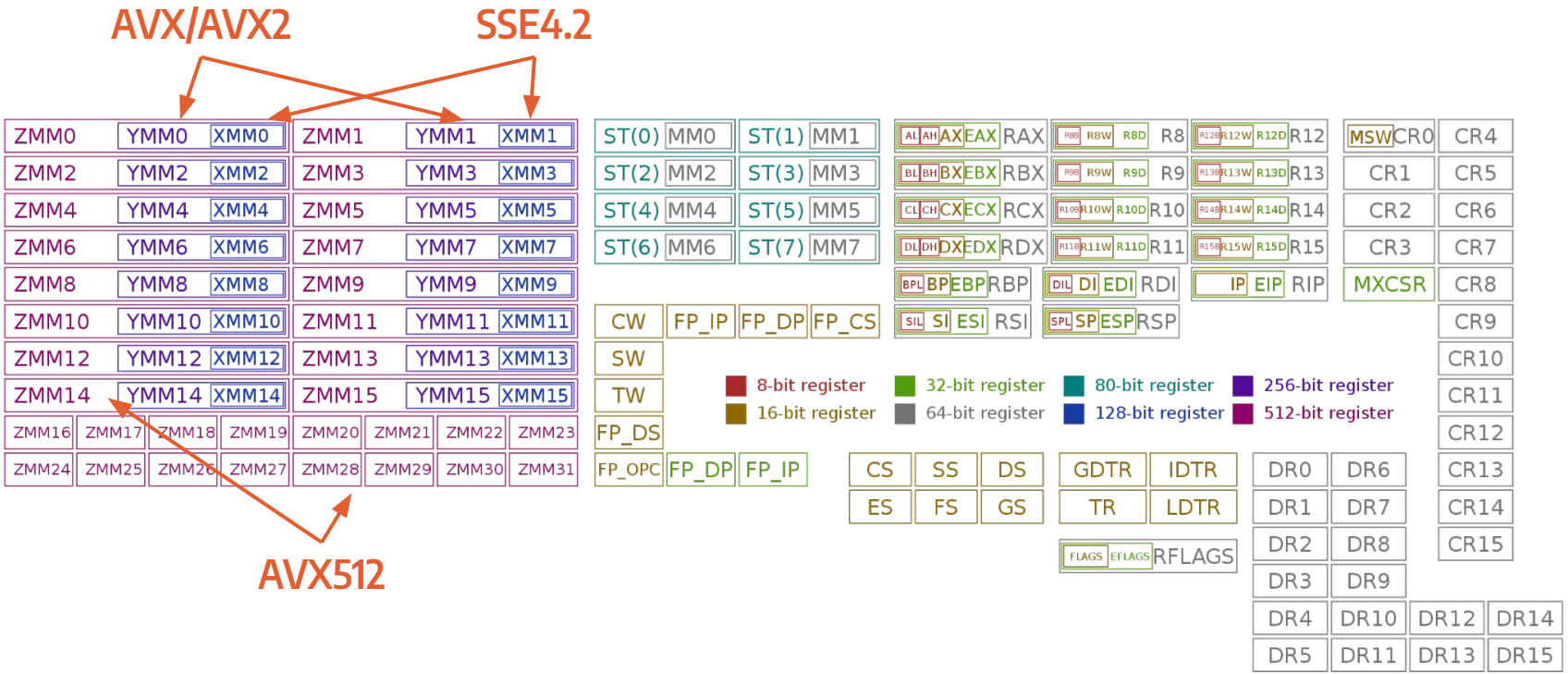
Source: Wikipedia

Intel x86 Assembly

```c
__attribute__((noinline))
int is_odd(unsigned long long n)
{
    return n & 1;
}

unsigned int collatz_count(unsigned long long n)
{
    unsigned int count = 0;

    while (n != 1)
    {
        if (is_odd(n))
            n = 3 * n + 1;
        else
            n = n / 2;

        ++count;
    }

    return count;
}
```

```asm
is_odd:
        mov     eax, edi
        and     eax, 1
        ret
collatz_count:
        xor     edx, edx
        cmp     rdi, 1
        jne     .L7
        jmp     .L3
.L10:
        lea     rdi, [rdi+1+rdi*2]
        add     edx, 1
        cmp     rdi, 1
        je      .L3
.L7:
        call    is_odd
        test    eax, eax
        jne     .L10
        shr     rdi
        add     edx, 1
        cmp     rdi, 1
        jne     .L7
.L3:
        mov     eax, edx
        ret
```

Registers available in the x86-64 instruction set



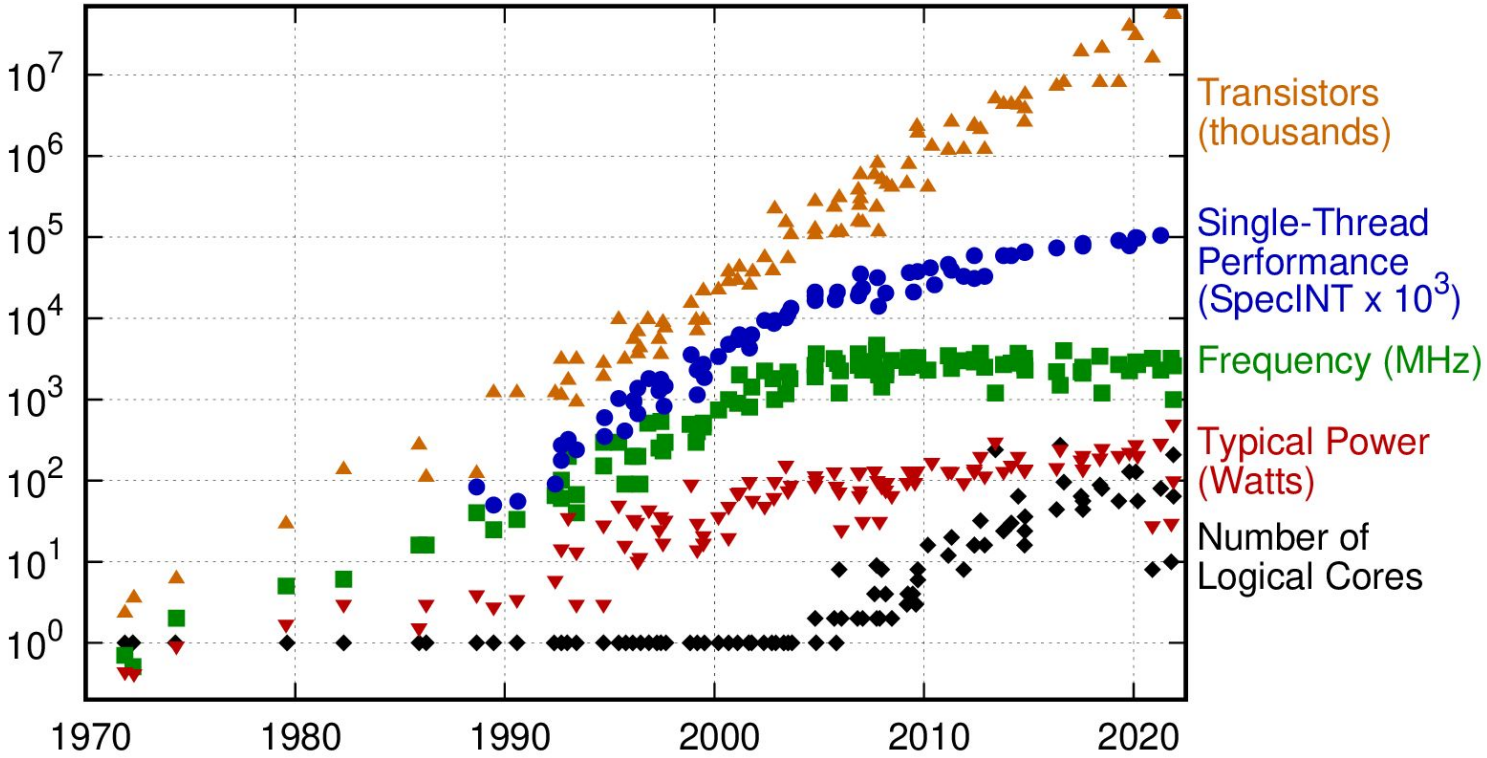
Instruction Sets

- **CISC** (Complex Instruction Set Computer)
  - Intel x86 and AMD64
    - Most laptop and desktop PCs, Playstation 5, Xbox One
  - IBM System z (mainframe computers)
- **RISC** (Reduced Instruction Set Computer)
  - ARM
    - Amazon Graviton (AWS VMs)
    - Apple M1–M4 (iPhone, iPad, iMacs)
    - Ampere Altra, Fujitsu A64FX, etc
    - Qualcomm (mobile phones, tablets)
    - Nintendo Game Boy Advance, DS, 3DS and Switch, Raspberry Pi, etc
  - IBM's PowerPC
    - Apple Macintosh (1994–2005), Nintendo GameCube and Wii, Playstation 3, Xbox 360
  - DEC Alpha, MIPS, Motorola 68000, RISC-V, SPARC, SuperH
    - Apple II (M68k), Nintendo 64, PlayStation 1 and 2 (MIPS), Sega Saturn and Dreamcast (SuperH)

Programming Language Evolution

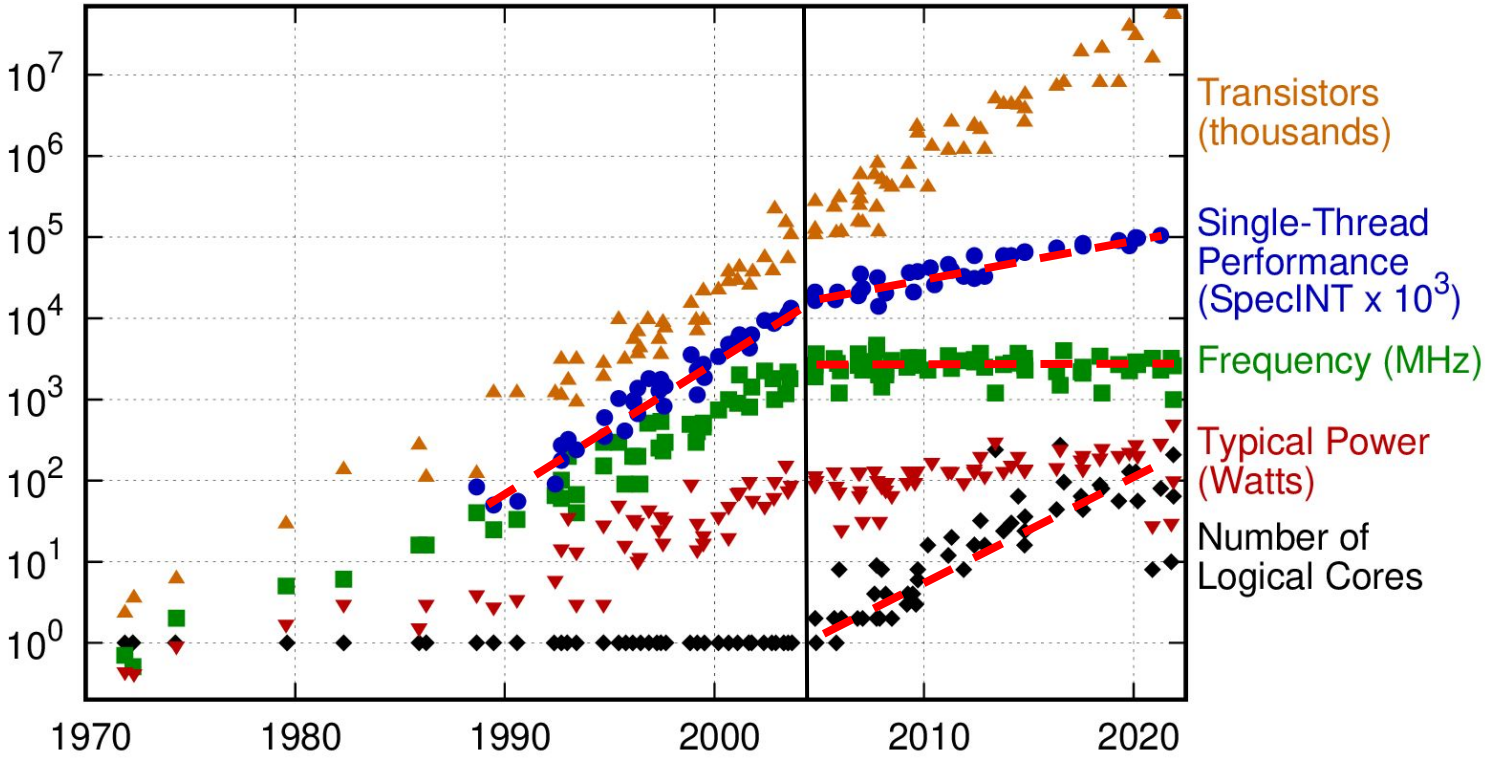
1947 Assembly	1954 Fortran	1963 BASIC	1972 C	1979 C++
Low-level language. High correspondence between language and hardware instructions.	One of the earliest high-level imperative programming languages.	**B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode is a family of high-level languages.	Originally developed to implement many of the utilities for UNIX OSs. Still in wide use today.	C++ was designed with systems programming, embedded software, and efficiency in mind.
Code written in assembly is converted to machine code using an *assembler*, which was a big upgrade over previous forms of programming.	Introduced procedural programming, double precision, and complex numbers. Still popular in HPC, including in GPU application programming.	BASIC became popular during the 8-bit era, but declined in popularity in the 90s, when more advanced languages like C were the norm.	C is a portable, imperative language with a static type system and which supports structured programming.	Although many think of C++ as a superset of C or C with classes, their latest versions are not fully compatible. Used extensively in HEP and HPC nowadays.

50 Years of Microprocessor Trend Data



source: https://github.com/karlrupp/microprocessor-trend-data

50 Years of Microprocessor Trend Data

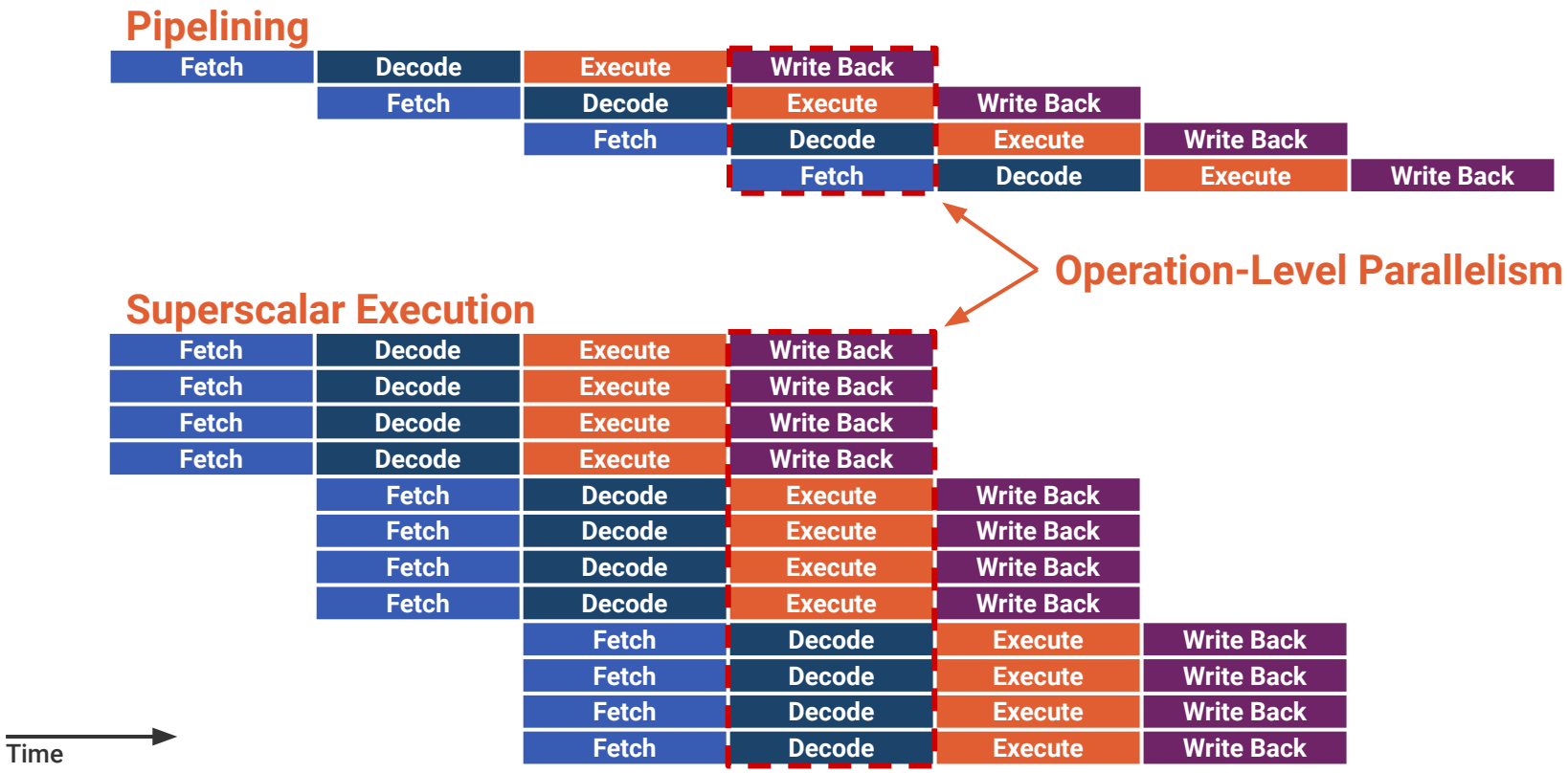


source: https://github.com/karlrupp/microprocessor-trend-data

Breakdown of Dennard's Scaling

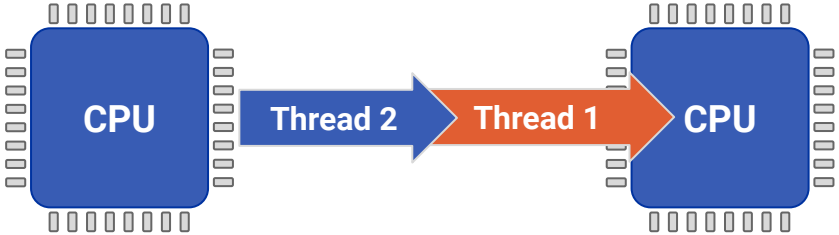
- Power density per unit area stopped decreasing
- Frequency could no longer keep increasing after each die shrink
  - But the transistor numbers kept growing
- Single-thread performance gains continued, albeit at a slower pace
  - More complexity: pipelining, superscalar, out-of-order execution, SIMD
- AMD and Intel bring 64-bit CPUs to the mainstream market
  - Intel with IA-64, and AMD with amd64 (x86_64), announced in 1999
- From symmetric multiprocessing (SMP) to multithreading (SMT)
  - In the '90s, dual socket high-end servers became popular
  - First SMT capable CPU was the Intel Pentium 4, released in 2002
- First dual core processors began to appear in mid 2000s
- The era of parallelism is born

Instruction-Level Parallelism

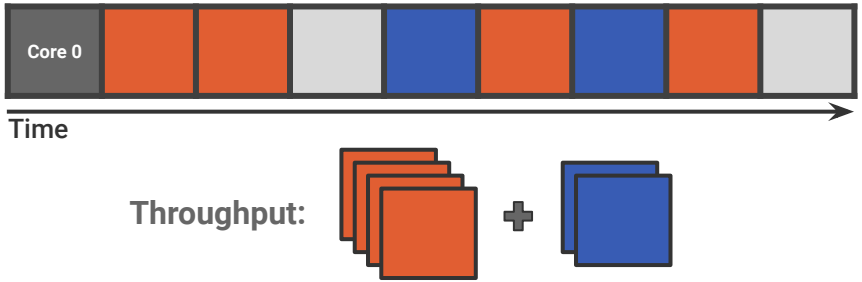


Symmetric multithreading (SMT)

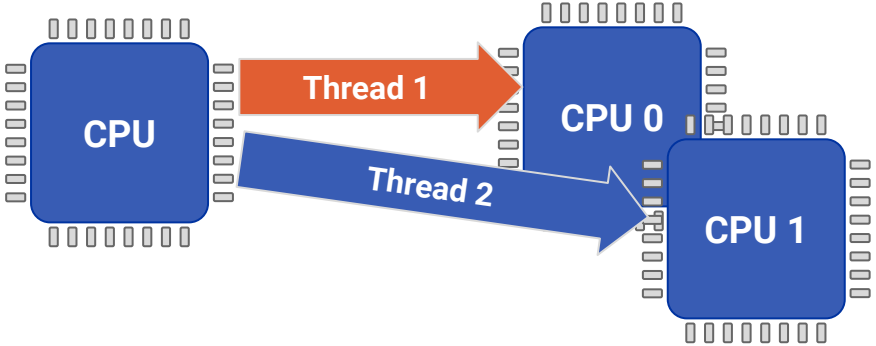
Without SMT



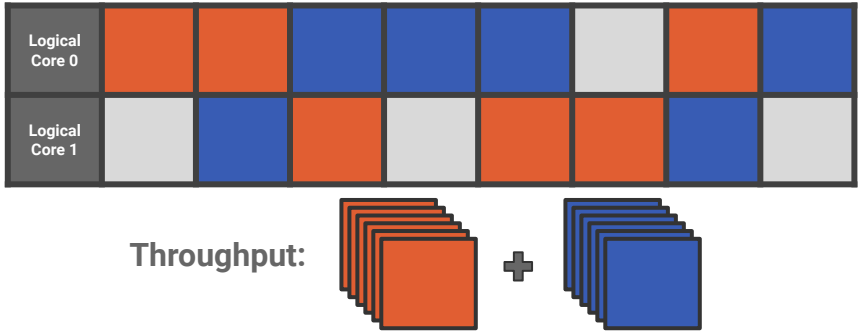
Threads scheduled one at a time on each physical core



With SMT



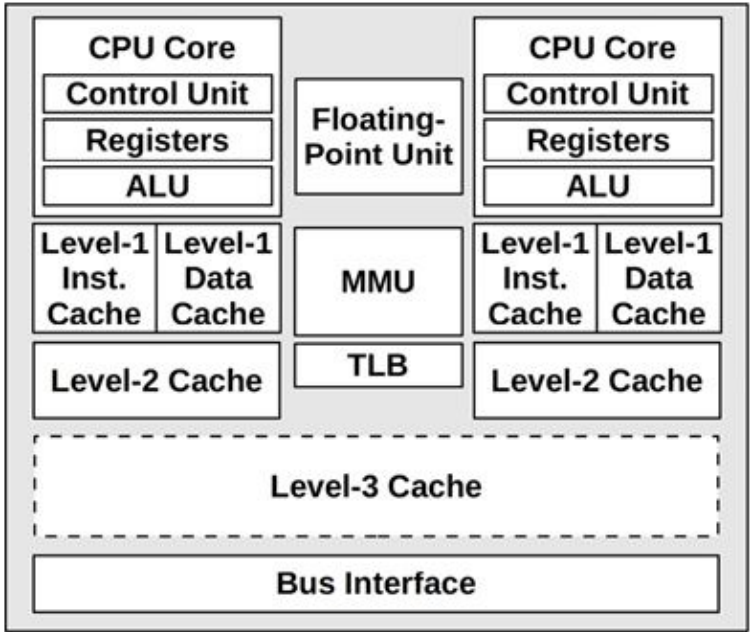
Threads run simultaneously on two logical cores



CPU Architecture

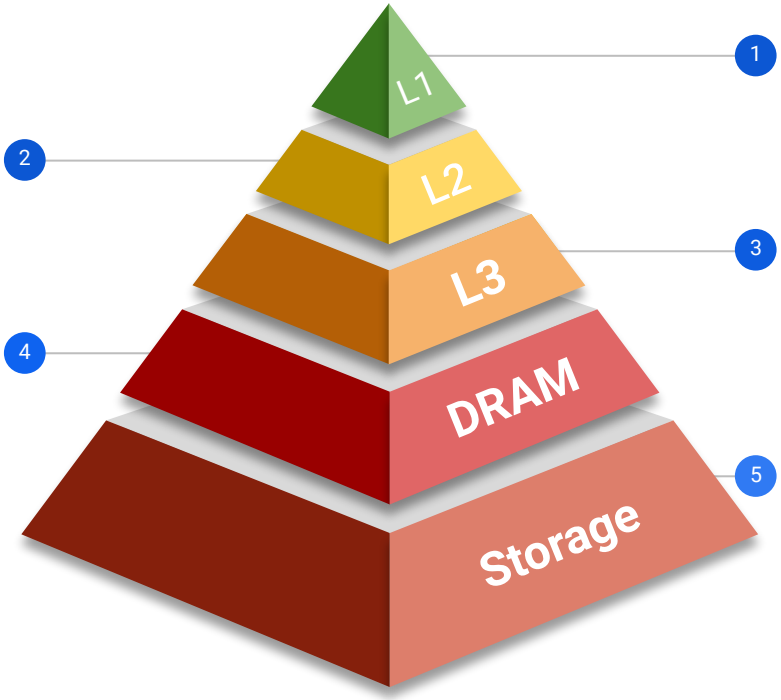
- Logical Components
  - Control Unit
  - Arithmetic Logic Units (ALUs)
  - Floating Point Unit (FPU)
  - Branch Predictor Unit (BPU)
  - Memory Management Unit (MMU)
  - Translation Lookaside Buffer (TLB)
- Memory Subsystem
  - L1(~32–512KB per core)
    - L1 Instruction Cache
    - L1 Data Cache
  - L2 (~1–8MB per core)
    - Instruction/Data Shared Cache
  - L3 (up to ~8MB–1.1GB per socket)
    - Last level cache (LLC)

Generic Dual Core CPU



Source: Systems Performance 2nd Edition, Brendan Gregg

Memory Hierarchy



CPU Registers & L1 Cache
1
L1 Latency: 1–5 cycles
L1 Size: 32–512K / core

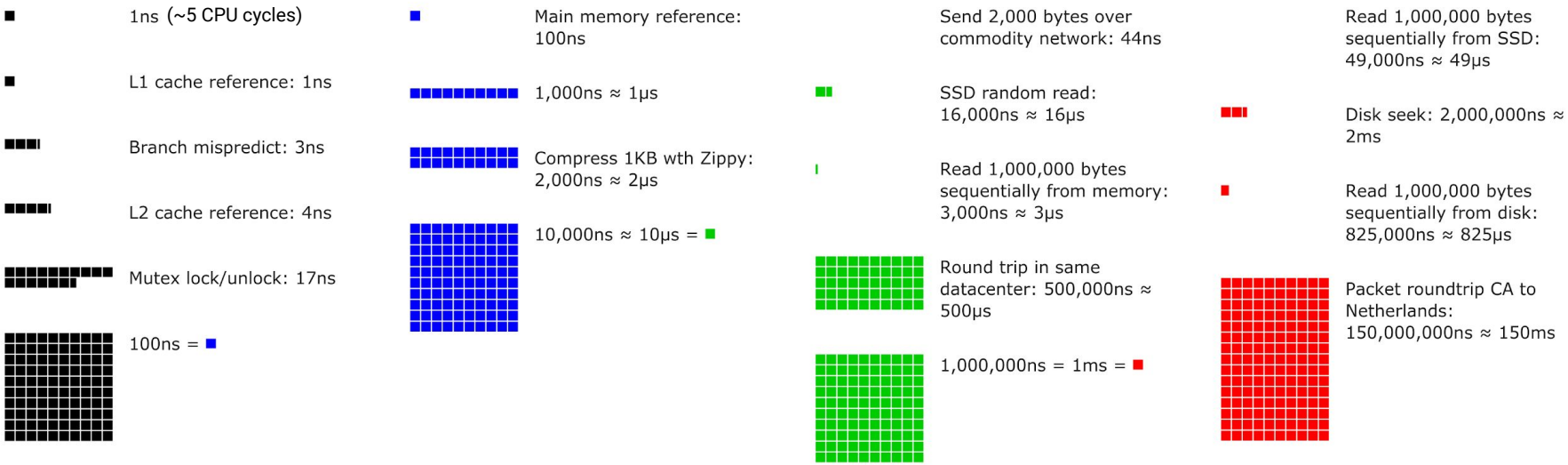
L2 Cache
2
Latency: ~12 cycles
Size: ~1M / core

L3 Cache
3
Latency: ~40 cycles
Size: 8MB – 1.1GB

Main Memory
4
Latency: 100–400 cycles
Size: 10s of GB to few TB

SSD, HDD, Tape Storage
5
SSD Latency: 10–70 µs
HDD Latency: 10–15 ms
Tape Latency: varies, minutes to hours, depends on if tape is already mounted

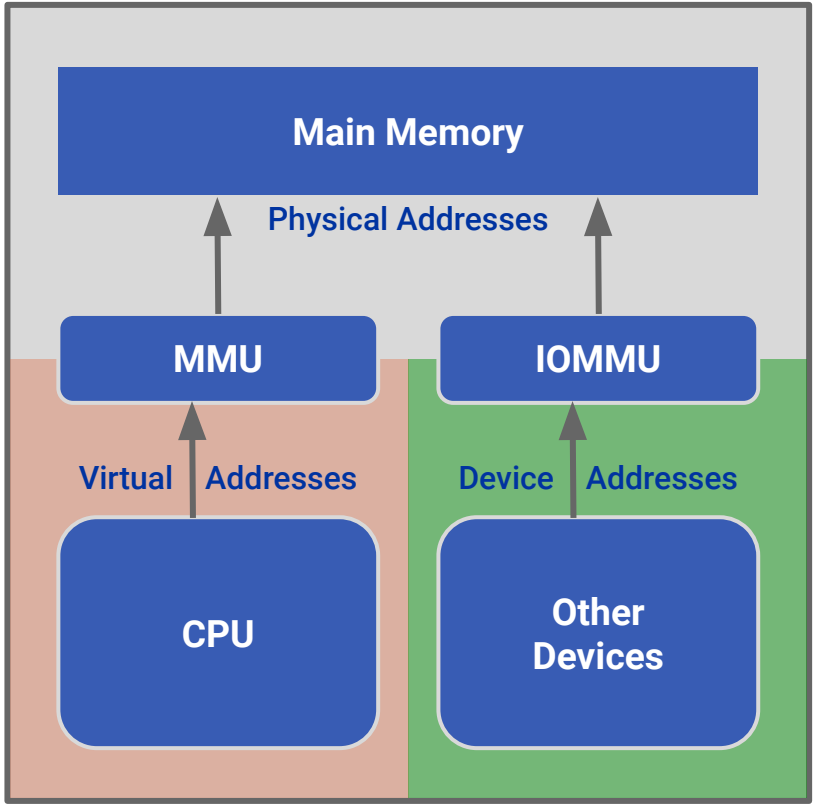
Latency Numbers Every Programmer Should Know



Source: https://colin-scott.github.io/personal_website/research/interactive_latency.html

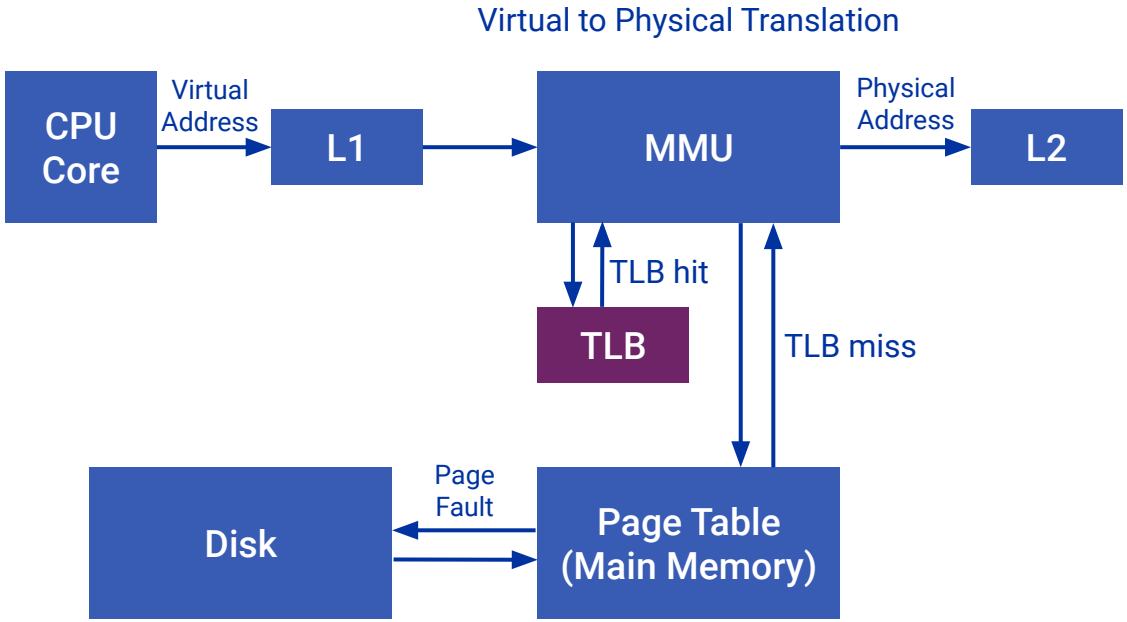
Virtual Memory

- First appeared in the Atlas computer in 1962
- Memory Management Unit (MMU)
- Memory managed in pages
  - Page sizes are usually 4K, 16K, 64K
  - May also support “huge pages” of 2MB, 1GB
- Hides fragmentation of physical memory
- Memory hierarchy managed by the kernel
- Makes application programming easier
  - Memory looks contiguous
  - No need to worry about fragmentation
  - Seems to own whole address space
  - Enabled timesharing features



The Translation Lookaside Buffer

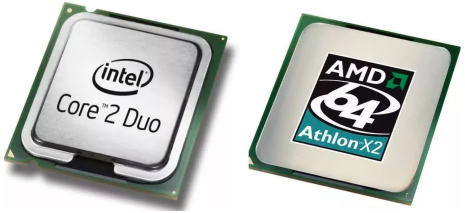
"A translation lookaside buffer (TLB) is a memory cache that is used to reduce the time taken to access a user memory location.
It is a part of the chip's memory management unit (MMU). The TLB stores the recent translations of virtual memory to physical memory and can be called an address-translation cache."

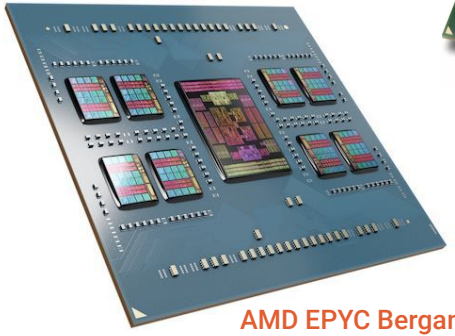


https://en.wikipedia.org/wiki/Translation_lookaside_buffer

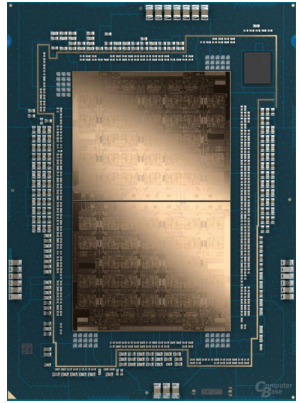
The Parallel Era

- First dual core CPUs debut in 2004
  - Pentium D, based on Pentium 4
  - AMD Athlon X2
- Quickly evolved from 2 to 4 cores
  - Stagnated at 4 cores for several years
- Ryzen brought AMD back in the game
  - Offered more cores, forced Intel to do the same
- ARM finally begins move from phones to servers
  - Amazon Graviton, Fujitsu A64FX, Ampere Altra
- Innovations in packaging led to multi-chip CPUs
  - AMD EPYC, Intel Sapphire and Emerald Rapids

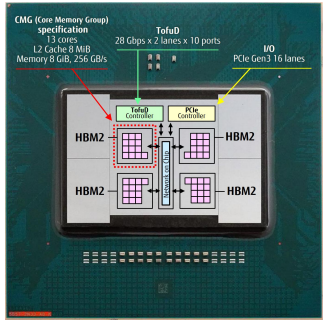




AMD EPYC Bergamo

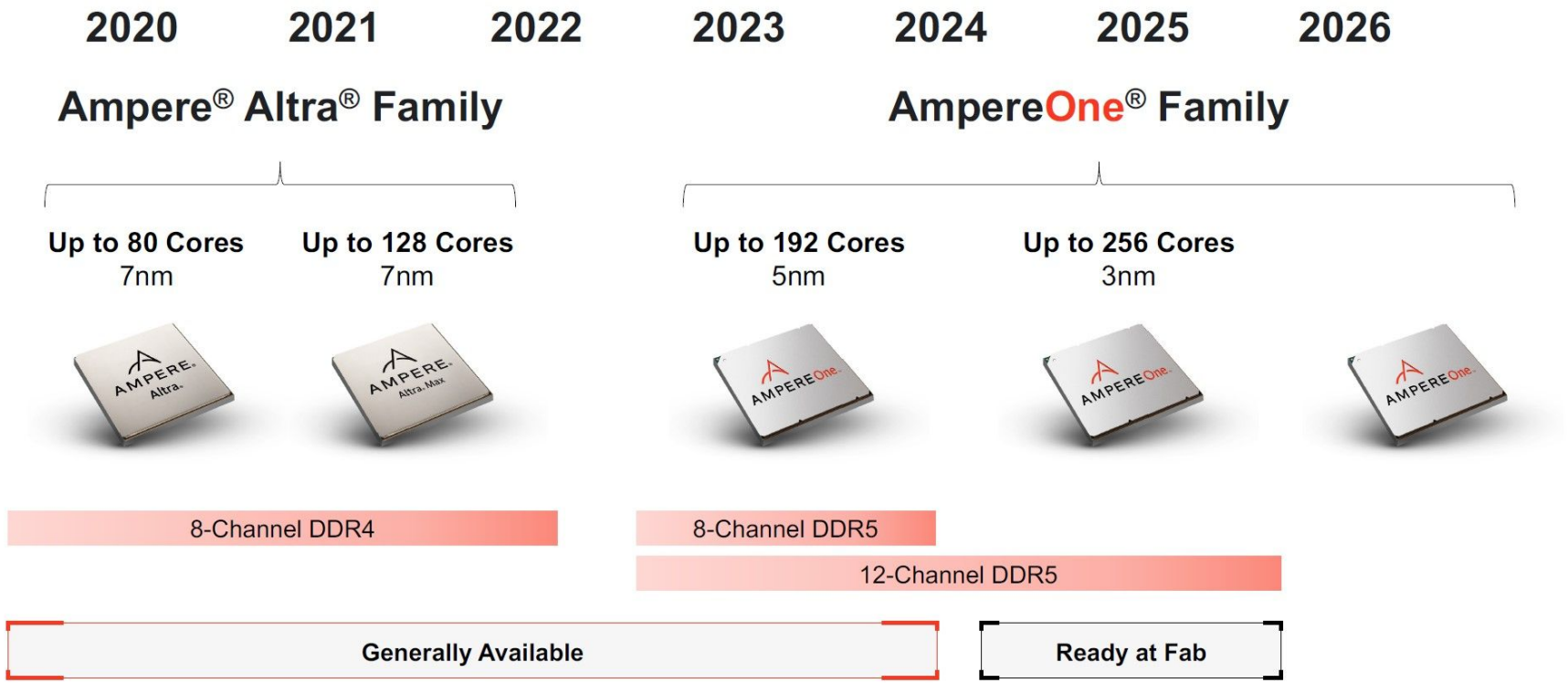


Intel Emerald Rapids



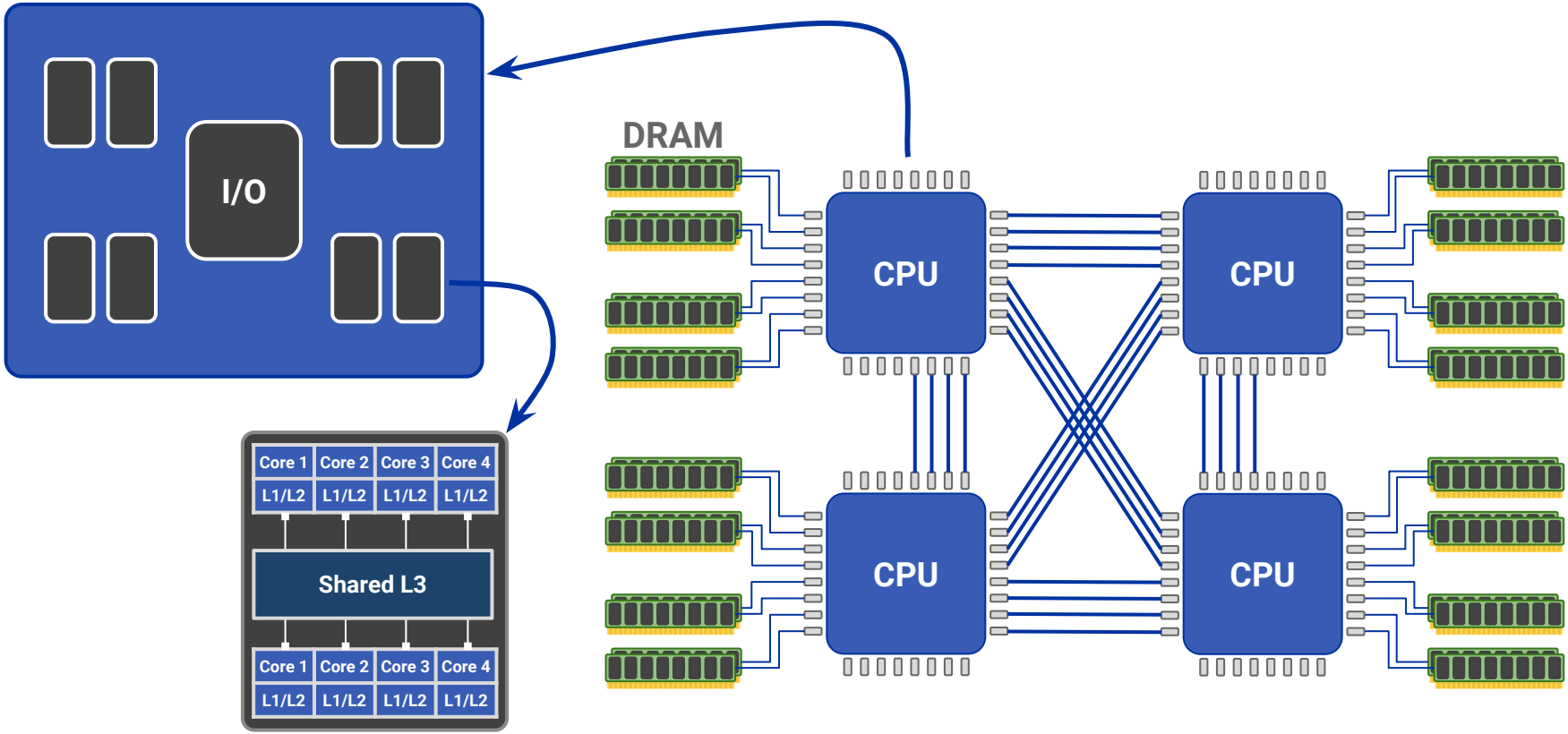
Fujitsu A64FX

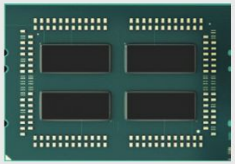
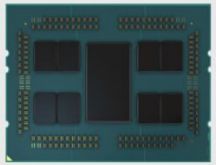
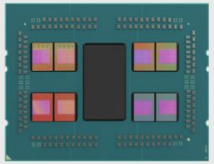
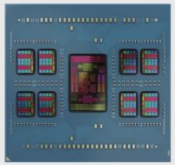
Ampere Roadmap 2020 – 2026



Source: Ampere

Non-Uniform Memory Architecture (NUMA)



	AMD EPYC 7001 'NAPLES'	AMD EPYC 7002 'ROME'	AMD EPYC 7003 'MILAN'	AMD EPYC 9004, 8004 'GENOA', 'SIENA'
				
Core Architecture	'Zen'	'Zen 2'	'Zen 3'	'Zen 4' and 'Zen 4c'
Cores	8 to 32	8 to 64	8 to 64	8 to 128
IPC Improvement Over Prior Generation	N/A	~24%[ROM-236]	~19% [MLN-003]	~14%[EPYC-038]
Max L3 Cache	Up to 64 MB	Up to 256 MB	Up to 256 MB	Up to 384 MB (EPYC 9004) Up to 128 MB (EPYC 8004)
Max L3 Cache with 3D V-Cache™ technology			768 MB	Up to 1152 MB
PCIe® Lanes	Up to 128 Gen 3	Up to 128 Gen 3	Up to 128 Gen 4	Up to 128 Gen 5 8 bonus lanes Gen 3
CPU Process Technology	14nm	7nm	7nm	5nm
I/O Die Process Technology	N/A	14nm	14nm	6nm
Power (Configurable TDP [cTDP])	120-200W	120-280W	155-280W	70-400W
Max Memory Capacity	2 TB DDR3-2400/2666	4 TB DDR4-3200	4 TB DDR4-3200	6 TB DDR5-4800

Source: AMD

AMD Zen4 Architecture in detail

COMPUTE

- AMD "Zen4" x86 cores (Up to 12 CCDs / 96 cores / 192 threads)
- 1MB L2/Core, 96MB L3/CCD / Total up to 1,152MB L3
- ISA updates: BFLOAT16, VNNI, AVX-512 (256b data path)
- Memory addressability with 57b/52b Virtual/Physical Address

- Updated IOD and internal AMD Gen3 Infinity Fabric™ architecture with increased die-to-die bandwidth

- Target TDP range: Up to 400W (cTDP)

- Updated RAS

Memory

- 12 channel DDR5 with ECC up to 4800 MHz
- Option for 2, 4, 6, 8, 10, 12 channel memory interleaving1

- RDIMM, 3DS RDIMM

- Up to 2 DIMMs/channel capacity with up to 12TB in a 2 socket system (256GB 3DS RDIMMs)1

Source: AMD

ORANGE indicates difference from General Purpose

SP5 Platform

- New socket, increased power delivery and VR
- Up to 4 links of Gen3 AMD Infinity Fabric™ with speeds of up to 32Gbps
- Flexible topology options

- Server Controller Hub (USB, UART, SPI, I2C, etc.)

Integrated I/O – No Chipset

Up to 160 IO lanes (2P) of PCIe® Gen5

- Speeds up to 32Gbps, bifurcations supported down to x1
- Up to 12 bonus PCIe® Gen3 lanes in 2P config (8 lanes–1P)
- Up to 32 IO lanes for SATA

- 64 IO Lanes support for CXL1.1+ w/bifurcations supported down to x4

Security Features

Dedicated Security Subsystem with enhancements

Secure Boot, Hardware Root-of-Trust

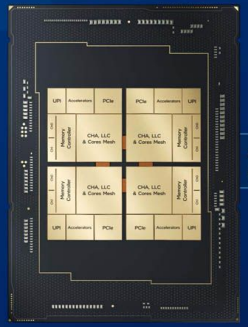
SME (Secure Memory Encryption)

SEV-ES (Secure Encrypted Virtualization & Register Encryption)

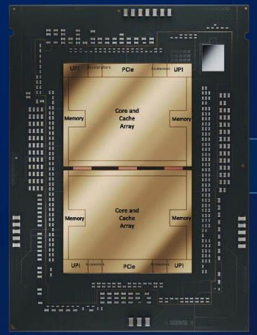
SEV-SNP (Secure Nested Paging), AES-256-XTS with more encrypted VMs

Intel® Xeon® Die Package Enhancements

Scalable, Balanced Architecture



4th Gen Intel Xeon
(Four-Tile Architecture)



5th Gen Intel Xeon
(Two-Tile Architecture)

intel.

5th Gen Intel® Xeon® Processors Turbo Frequencies

Introducing Improved 5 Turbo Ratio Levels

- Improves Turbo Frequencies for Intel® AVX heavy and Intel® AMX light workloads including HPC and AI
- ~2 bins Turbo Frequency Upside on Intel® AVX-512 Heavy usage
- ~9% performance improvement on low load (4T or 8T) LINPACK AVX512
- ~5% performance improvement on low instance (4 or 32) Resnet50 amx_int8, amx_bfloat16 and avx_fp32
- Lowers the Turbo frequency penalty for using AVX512 or AMX, broadening usability of these instruction sets

4th Gen Intel® Xeon® CPU

Instruction Class	Cdyn Class			
	0	1	2	3
SSE	128 Light	128 Heavy		
AVX2	256 Light	256 Moderate	256 Heavy	
AVX512	512 Ultra-Light	512 Light	512 Moderate	512 Heavy
AMX		AMX Light	AMX Moderate	AMX Heavy
Turbo Frequency	SSE	AVX2	AVX512	AMX

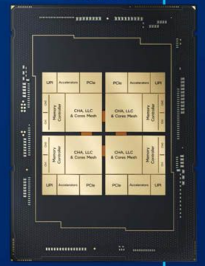
5th Gen Intel® Xeon® CPU

Instruction Class	Cdyn Class				
	0	1	2	3	4
SSE	128 Light	128 Heavy			
AVX2	256 Light	256 Moderate	256 Heavy		
AVX512	512 Ultra-Light	512 Light	512 Moderate	512 Heavy	
AMX ENHANCED		AMX Ultra-Light	AMX Light	AMX Moderate	AMX Heavy
Turbo Frequency	SSE	AVX2	AVX512	AVX512 H NEW	AMX

5th Gen Intel® Xeon® Processors

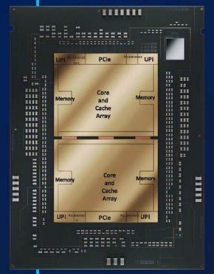
Platform enhancements delivering significant gains at the same power envelope[1]

4th Gen Xeon



- **Drop-in compatible** with 4th Gen Xeon processors
- Up to **64** cores per CPU
- Up to **5600** MT/s memory speed
- Up to **3x** shared Last Level Cache (up to **320 MB** total)
- Up to **20** GT/s UPI 2.0 Speed
- **Type 3** memory support with Compute Express Link® 1.1*
- Intel® Trust Domain Extension **broad support**

5th Gen Xeon



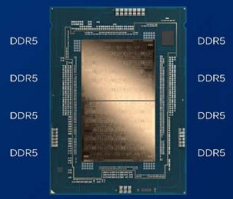
intel.

* Type 3 CXL targeted deployment
See backup for configuration details. Results may vary.

Compute Express Link® 1.1 Enhancements

Type 3 memory support with 5th Gen Intel® Xeon® processors

CXL Memory Buffer



CXL Memory Buffer

2 Tier Memory Support Example

2–Tier Memory Support

Type 3 Memory Expansion Devices:

Capacity Expansion

- Tier 1 memory = native DDR, Tier 2 memory = CXL® attached memory
- Supports up to 4 channels of CXL memory across two CXL type 3 devices
- Supports CXL memory latency QoS distress signaling

Increased transactions per second for In-Memory databases (e.g. Redis)

Single Tier Memory Support

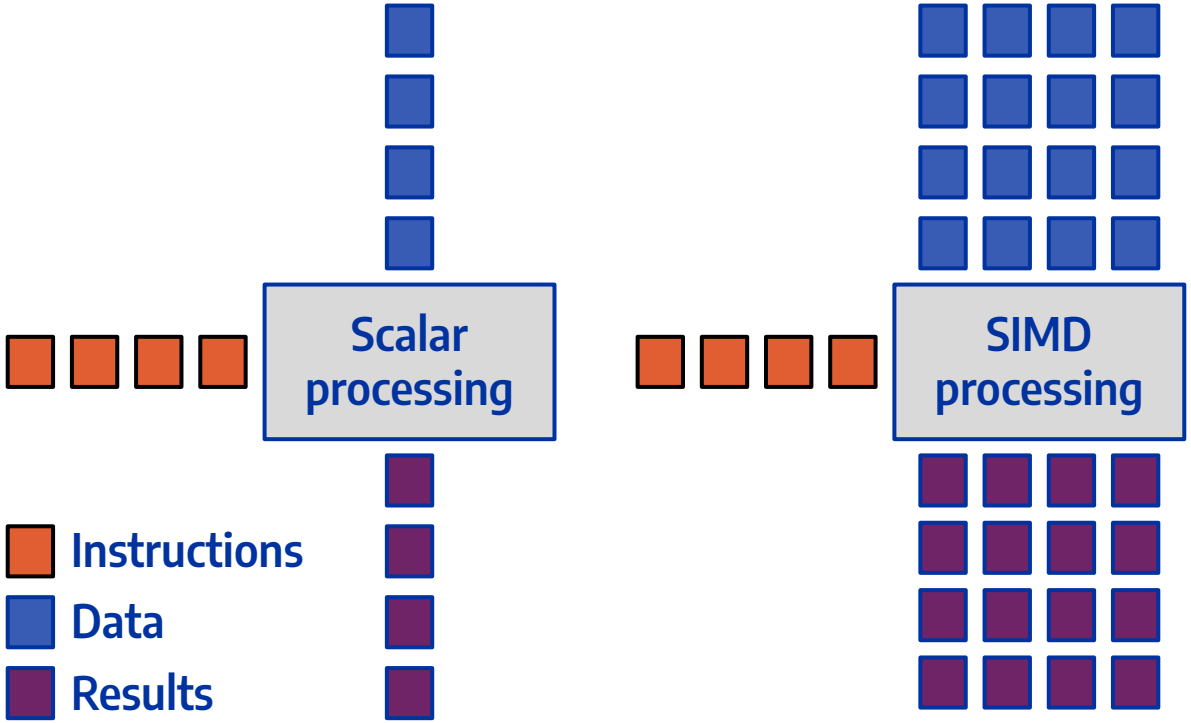
12 channel DDR+CXL interleaved memory

- Either for capacity or bandwidth expansion

intel.

Source: Intel

SIMD Vectorization



History of Intel® SIMD ISA Extensions

- Intel® Pentium Processor (1993)
  32bit
- Multimedia Extensions (MMX in 1997)
  64bit integer support only
- Streaming SIMD Extensions (SSE in 1999 to SSE4.2 in 2008)
  32bit/64bit integer and floating point, no masking
- Advanced Vector Extensions (AVX in 2011 and AVX2 in 2013)
  Fused multiply-add (FMA), HW gather support (AVX2)
- Many Integrated Core Architecture (Xeon Phi™ Knights Corner in 2013)
  HW gather/scatter, exponential
- AVX512 on Knights Landing, Skylake Xeon, and Core X-series (2016/2017)
  Conflict detection instructions

= 32 bit word

Evolution of Intel® SIMD ISA Extensions

- AVX 10
  - Supported on both P-cores and E-cores
  - Brings benefits of AVX512 to smaller registers
- Advanced Matrix Extensions (AMX)
  - Targeted at AI applications
  - SIMD for small matrix operations
  - Available on 4th and 5th generation Xeon
- Advanced Performance Extensions (APX)
  - Adds new features that improve general-purpose performance
  - Expands x86 instruction set with more general-purpose registers (from 16 to 32)
  - New REX2 prefix provides uniform access to the new registers
  - Adds conditional forms of load, store, and compare/test instructions
  - New prefix increase average instruction length, but there are less instructions overall

Intel® AVX	Intel® AVX2
128/256-bit FP	Float16
16 registers	128/256-bit FP FMA
NDS (and AVX128)	256-bit int
Improved blend	PERMD
MASKMOV	Gather
Implicit unaligned	

Intel® AVX-512
128/256/512-bit FP/Int
32 vector registers
8 mask registers
512-bit embedded rounding
Embedded broadcast
Scalar/SSE/AVX "promotions"
Native media additions
HPC additions
Transcendental support
Gather/Scatter
Flag-based enumeration
Intel® Xeon P-core only

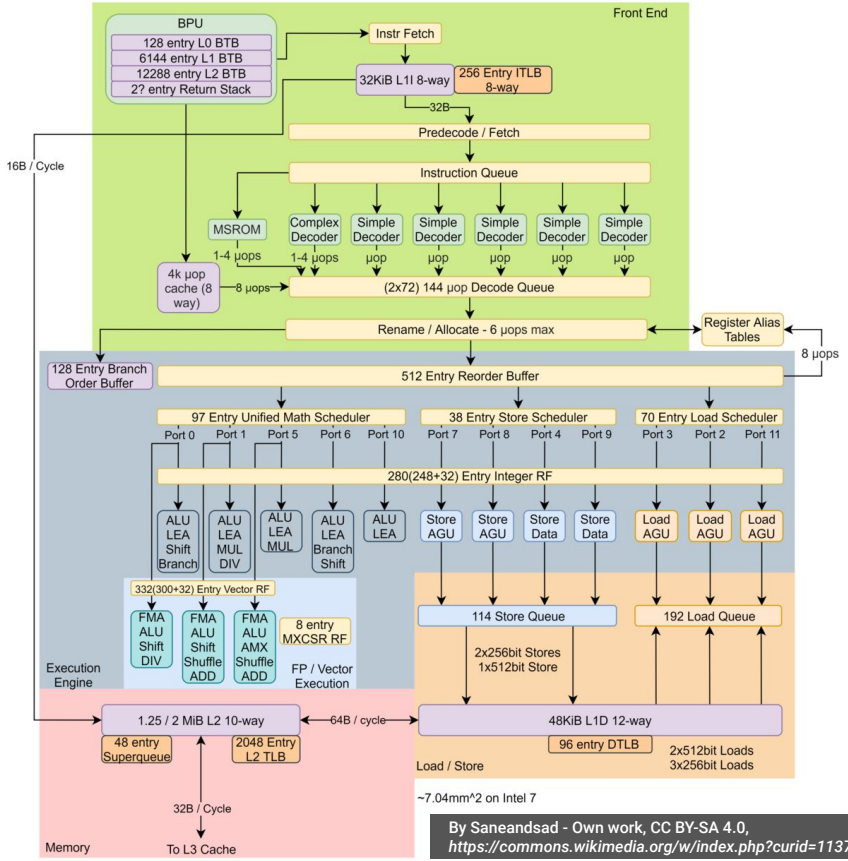
Intel® AVX10.1 (pre-enabling)
Optional 512-bit FP/Int
128/256-bit FP/Int
32 vector registers
8 mask registers
512-bit embedded rounding
Embedded broadcast
Scalar/SSE/AVX "promotions"
Native media additions
HPC additions
Transcendental support
Gather/Scatter
Version-based enumeration
Intel® Xeon P-core only

Intel® AVX10.2
New data movement, transforms and type instructions
Optional 512-bit FP/Int
128/256-bit FP/Int
32 vector registers
8 mask registers
256/512-bit embedded rounding
Embedded broadcast
Scalar/SSE/AVX "promotions"
Native media additions
HPC additions
Transcendental support
Gather/Scatter
Version-based enumeration
Supported on P-cores, E-cores

Microarchitecture of a Modern Intel Core

- Front End
  - Instruction Fetch and Decode
  - Branch Predictor Unit (BPU)
  - L1 Instruction Cache
  - Instruction TLB
- Back End
  - Execution Engine
    - Scheduler
    - Register File
    - Execution Units (EUs)
  - Memory Subsystem
    - Load/Store Units (LSU)
    - L1 / L2 Data Cache
    - Data TLB

Intel Golden Cove Core



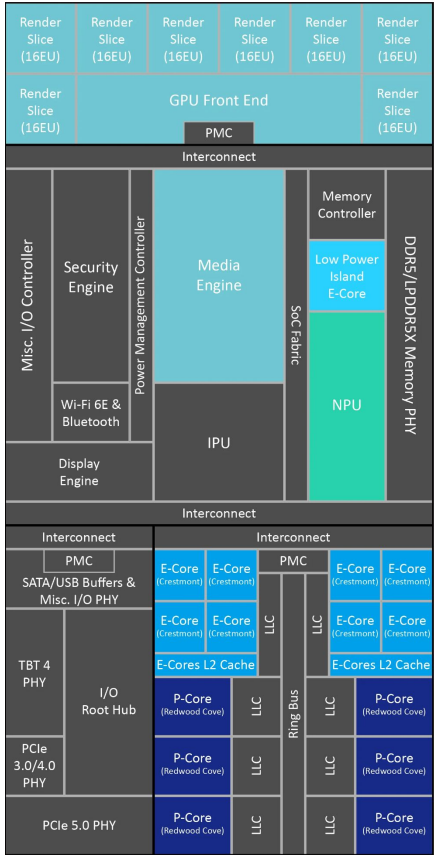
Meteor Lake Hybrid Architecture

Ultra Mobile

Mobile

Desktop

Meteor Lake Block Diagram



Source: Intel

REDWOOD COVE

New P-core

Targeted for efficient performance

Improved performance efficiency*

Increased BW per core package*

Improved Performance Monitoring Unit

Improved feedback Intel Thread Director

Source: Intel

*Architectural simulation vs. Golden Cove architecture. Results may vary across workloads.

CRESTMONT

New E-core

Significant improvements over prior E-core

IPC gains
over prior E-cores*

AI acceleration
VNNI, ISA improvements*

Enhanced
branch prediction

Enhanced Feedback
Intel Thread Director

*Architectural simulation vs. Gracemont architecture across a broad set of workloads. VNNI improvements based on doubling the number of VNNI ports. Results may vary.

Source: Intel

DCAI Architecture Evolution

CPU P-Core

- 4th Gen Intel® Xeon® Scalable processors
- Intel® Xeon® CPU Max Series
- 5th Gen Intel® Xeon® codenamed Emerald Rapids
- Intel® Xeon® Processors codenamed Granite Rapids

CPU E-Core

- Intel® Xeon® Processor codenamed Sierra Forest
- Intel® Xeon® Processor codenamed Clearwater Forest

GPU

- Intel® Data Center GPU Flex Series codenamed Arctic Sound-M
- Intel® Data Center GPU Max Series codenamed Ponte Vecchio
- Intel® Data Center GPU Flex Series codenamed Melville Sound
- Next-Generation Accelerator Architecture Codename: Falcon Shores

Dedicated AI

- Habana® Gaudi® 2
- Habana® Gaudi® 3
- Next-Generation Accelerator Architecture

FPGA

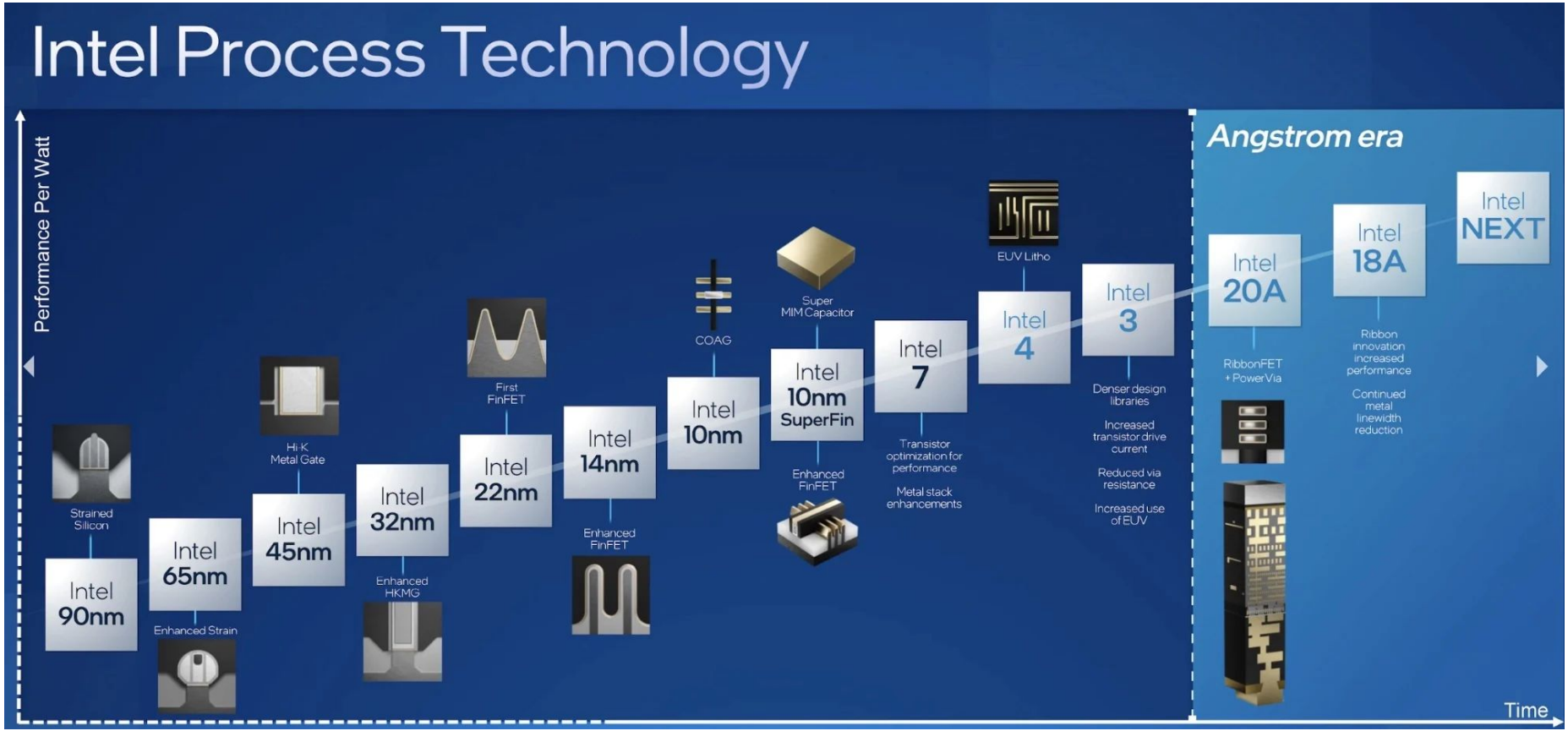
- intel. STRATIX 10
- intel. eASIC
- intel. AGILEX
- 15 new FPGAs on schedule to PRQ in 2023
- intel. eASIC
- intel. AGILEX
- Next Gen FPGAs

Source: Intel

Roadmap: 2023-2025

DCAI Investor Webinar March 2023

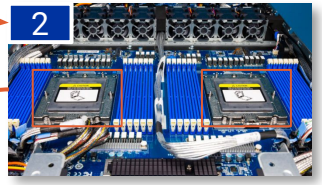
intel.

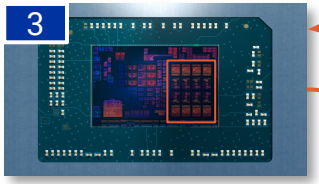


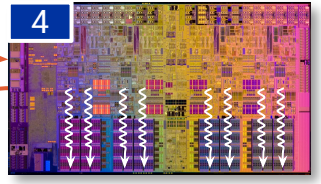
Source: Intel

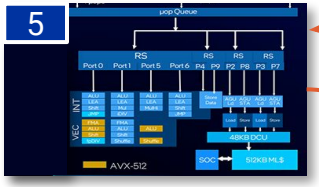
Modern Hardware

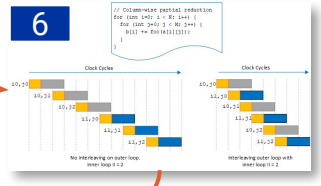
1	Systems	• Has always been there • Run copies of your code on each node
2	Sockets	• Modern machines have up to 8 sockets • Trend towards reduction back to 2 only
3	Cores	• Higher frequency no longer possible • 128 cores in each socket now common
4	Threads	• Hardware has N physical cores • OS sees 2N logical cores, shared exec.
5	Ports	• Superscalar execution • Multiple instructions executed at once
6	Pipelining	• Parallel instruction execution steps • Fetch / Decode / Execute / Write Back
7	SIMD Vectors	• SSE4.2, AVX2, AVX512, AVX10 • PowerPC Altivec, ARM SVE, etc

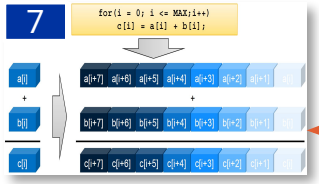












Summary

- We've come a long way, modern hardware is quite complex
  - NUMA Architecture (multi socket)
  - High parallelism (multicore, superscalar)
  - Advanced Packaging (chiplets)
  - Hybrid Architectures (performance/efficiency)
  - Variable CPU frequency scaling (turbo boost, thermal throttling)
  - Accelerators and Heterogeneity (GPUs, NPUs, FPGAs, ASICs)
- Performance does not come for free, we needed to adapt our software
  - Concurrency and Parallelism (processes, threads, SIMD)
  - Memory alignment, access patterns, fragmentation
  - Code layout, compiler optimizations, data structures, software design
  - Need the right tools to guide us: profilers, static analysis, etc
  - Need the right methodology: identify causes of bottlenecks, address the right issue

Performance Analysis on Modern CPUs

Performance is challenging

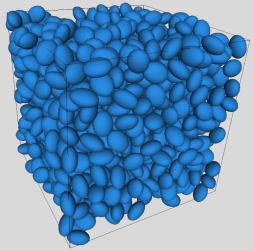
- Measuring Performance
  - Instrumentation and measurement has some overhead
  - Sophisticated hardware architecture (out of order, superscalar)
  - Variable CPU frequency scaling (turbo boost, thermal throttling)
  - Often missing symbols (JIT, interpreted languages, stripped binaries)
  - Unreliable stack unwinding (deep call stacks, inlining, missing frame pointers)
- Optimization and Tuning
  - Floating-point arithmetics is complicated (denormals)
  - Memory access patterns, fragmentation, (mis-)alignment
  - Concurrency issues (shared resources with hyperthreading, contention)
  - Reliance on compiler optimizations (exceptions vs vectorization, dead code)

Instrumentation-Based Profiling

- Use a timer and print out how long a section of code takes to run
  - Simplest form of instrumentation
  - Make changes and measure again
- Use an instrumentation-based profiler
  - May need to compile application with profiling information (`-g -pg`)
  - Run the application and analyze the output file
  - Examples: **gprof**, **valgrind**, **uftrace**
  - Yields number of calls for each function, unlike sampling
  - Usually suffers from high overhead
  - Cannot use in production systems

Flat profile example using gprof

```
$ pack -f 0.5 examples/ellipsoids # compiled with -O2 -g -pg, simulates a packing of ellipsoids, as shown below
100.00% 0.5000 0.0000/min 2.1e-01 ev/s 4.9 s
$ file gmon.out
gmon.out: GNU prof performance data - version 1
$ gprof --no-graph pack | head -n 20
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 34.05      0.95     0.95  7677145    0.00     0.00  HGrid::find_neighbors(Particle const*, std::vector<Particle*>&)
 24.73      1.64     0.69 66007037    0.00     0.00  intersect(Particle const&, Particle const&, float)
  7.89      1.86     0.22 31828514    0.00     0.00  Ellipsoid::support(Vector const&) const
  5.38      2.01     0.15  6685781    0.00     0.00  Particle::world_transform(float) const
  4.30      2.13     0.12 140620355   0.00     0.00  Ellipsoid::bounding_radius() const
  3.94      2.24     0.11 10459271    0.00     0.00  closest_point_triangle(Point&, Point&, Point&, result&)
  3.23      2.33     0.09 13858812    0.00     0.00  Simplex::add_vertex(Vector const&, Point const&, Point const&)
  2.15      2.39     0.06 13858812    0.00     0.00  Simplex::update()
  2.15      2.45     0.06  4288132    0.00     0.00  closest_point_tetrahedron(Point&, Point&, Point&, Point& result&)
  1.79      2.50     0.05 13732871    0.00     0.00  Simplex::reduce()
  1.79      2.55     0.05   481841    0.00     0.00  check_overlap(Particle&)
  1.79      2.60     0.05     1000    0.00     0.00  Ellipsoid::name() const
  1.43      2.64     0.04  6784500    0.00     0.00  time_of_impact(Particle const&, Particle const&, float, float)
  1.43      2.68     0.04  3342851    0.00     0.00  Simplex::reset()
  1.43      2.72     0.04                             _init
```



Flat profile example using valgrind

```
$ valgrind --tool=callgrind -- pack -f 0.5 examples/ellipsoids # no need for -pg
==2140677== Callgrind, a call-graph generating cache profiler
==2140677== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==2140677== Using Valgrind-3.23.0 and LibVEX; rerun with -h for copyright info
==2140677== Command: pack -f 0.5 examples/ellipsoids
==2140677==
==2140677== For interactive control, run 'callgrind_control -h'.
100.00% 0.5000 0.0000/min 6.7e-03 ev/s 150.0 s
==2140677==
==2140677== Events    : Ir
==2140677== Collected : 29183525425
==2140677==
==2140677== I   refs:      29,183,525,425
$ kcachegrind callgrind.out.2140677
```

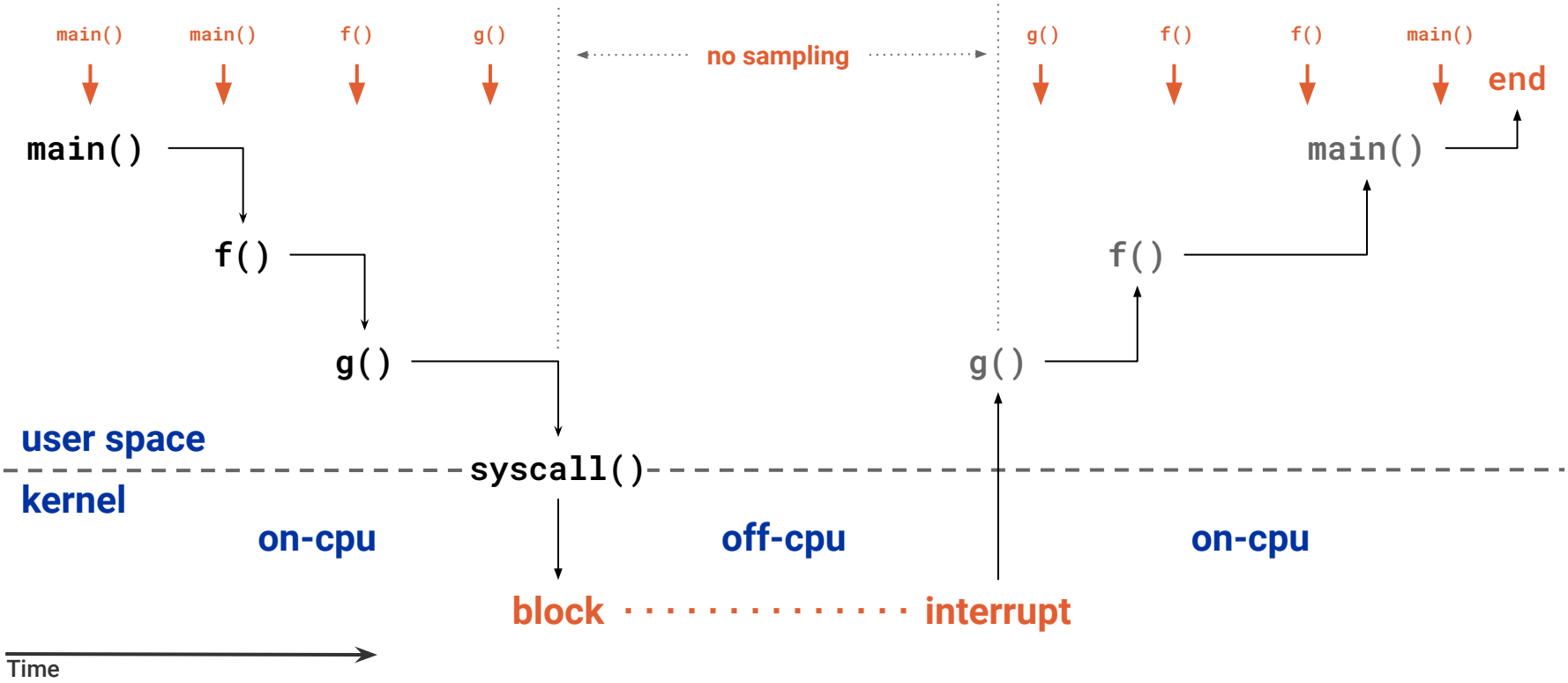
Incl.	Self	Distance	Calling	Callee
42.94	42.85	4-6 (6)	8 695 851	HGrid::find_neighbors(Particle const*, std::vector<Particle*, std::allocator<Particle*> >&) (pack: hgrid.cc, ...)
48.98	23.14	5-8 (6)	82 179 113	intersect(Particle const&, Particle const&, float) (pack: gjk.cc, ...)
69.30	4.11	5	556 101	check_overlap(Particle&) (pack: collision.cc, ...)
4.10	4.10	6-7 (7)	35 193 894	Ellipsoid::support(Vector const&) const (pack: ellipsoid.h, ...)
11.55	3.94	7-10 (8)	15 268 580	Simplex::update() (pack: simplex.cc, ...)
5.02	3.81	6-7 (7)	7 513 640	Particle::world_transform(float) const (pack: particle.h, ...)
2.56	2.56	8-10 (10)	11 495 780	closest_point_triangle(Point const&, Point const&, Point const&, closest_result&) (pack: simplex.cc, ...)
2.54	2.54	8-9 (9)	15 129 509	Simplex::reduce() (pack: simplex.cc)
3.84	2.49	8-9 (9)	4 717 916	closest_point_tetrahedron(Point const&, Point const&, Point const&, Point const&, closest_result&) (pack: simplex.cc, ...)
1.77	1.77	6-7 (7)	174 081 520	Ellipsoid::bounding_radius() const (pack: ellipsoid.h)
1.63	1.63	6-7 (7)	15 268 580	Simplex::contains(Vector const&) (pack: simplex.cc, ...)
1.17	1.17	6-9 (9)	7 542 131	sincos (libm.so.6: s_sincos.c, ...)
0.94	0.94	6-7 (7)	15 268 580	Simplex::add_vertex(Vector const&, Point const&, Point const&) (pack: simplex.cc)
23.91	0.88	4-7 (5)	7 313 206	time_of_impact(Particle const&, Particle const&, float, float) (pack: collision.cc)
12.33	0.79	6-9 (7)	15 268 580	Simplex::closest(Vector&) (pack: simplex.cc)

Parts | Callees | Call Graph | All Callees | Caller Map | Machine Code

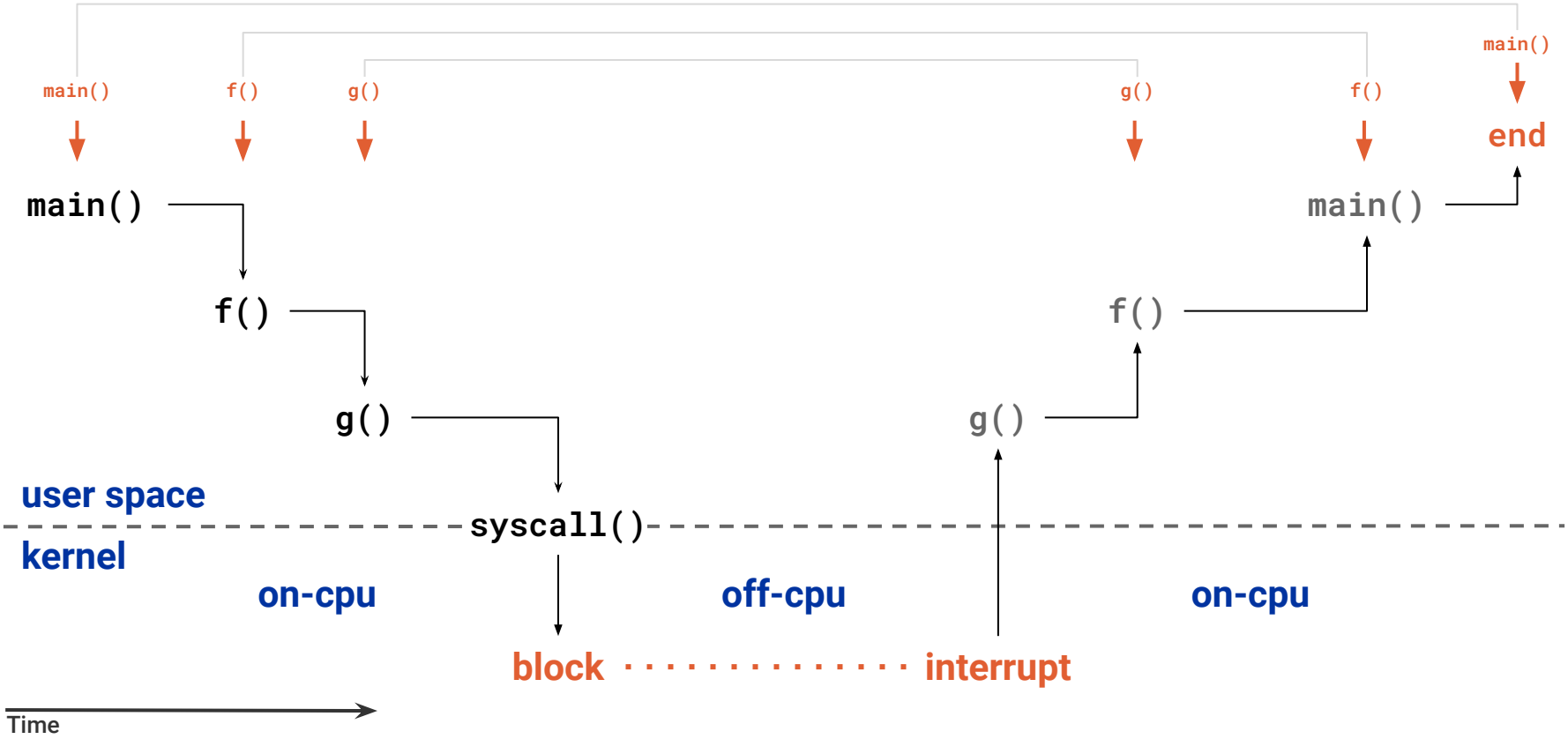
perf – Performance analysis tools for Linux

- Official Linux profiler (source code is part of the kernel itself)
- Both hardware and software based performance monitoring
- Much lower overhead compared with instrumentation-based profiling
- Kernel and user space
- Counting and Sampling
  - Counting — count occurrences of a given event (e.g. cache misses)
  - Event-based Sampling — a sample is recorded when a threshold of events has occurred
  - Time-based Sampling — samples are recorded at a given fixed frequency
  - Instruction-based Sampling — processor follows instructions and samples events they create
- Static and Dynamic Tracing
  - Static — pre-defined tracepoints in software
  - Dynamic — tracepoints created using uprobes (user) or kprobes (kernel)

Sampling


main()
main()
f()
g()
no sampling
g()
f()
f()
main()
end
main()
f()
g()
f()
main()
g()
user space
syscall()
kernel
on-cpu
off-cpu
on-cpu
block
interrupt
Time

Tracing


main()
f()
g()
g()
f()
main()
end
main()
f()
g()
g()
f()
main()
user space
syscall()
kernel
on-cpu
off-cpu
on-cpu
block
interrupt
Time

perf – subcommands

```
bash ~ $ perf

 usage: perf [--version] [--help] [OPTIONS] COMMAND [ARGS]

 The most commonly used perf commands are:
   annotate        Read perf.data (created by perf record) and display annotated code
   archive         Create archive with object files with build-ids found in perf.data file
   c2c             Shared Data C2C/HITM Analyzer.
   config          Get and set variables in a configuration file.
   data            Data file related processing
   diff            Read perf.data files and display the differential profile
   evlist          List the event names in a perf.data file
   list            List all symbolic event types
   mem             Profile memory accesses
   record          Run a command and record its profile into perf.data
   report          Read perf.data (created by perf record) and display the profile
   sched           Tool to trace/measure scheduler properties (latencies)
   script          Read perf.data (created by perf record) and display trace output
   stat            Run a command and gather performance counter statistics
   timechart       Tool to visualize total system behavior during a workload
   top             System profiling tool.
   version         display the version of perf binary
   probe           Define new dynamic tracepoints
   trace           strace inspired tool

 See 'perf help COMMAND' for more information on a specific command.
```

Flat profile example using perf

```
$ pack -f 0.5 examples/ellipsoids # compiled with -O2 -g
100.00% 0.5000 0.0001/min 3.2e-01 ev/s 3.1 s
$ perf record -F 1000 -e cycles -- pack -f 0.5 examples/ellipsoids
perf record -F 1000 -e cycles -- pack -f 0.5 examples/ellipsoids
100.00% 0.5000 0.0002/min 2.9e-01 ev/s 3.5 s
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.138 MB perf.data (3431 samples) ]
$ perf report --stdio | sed -ne /Overhead/,25p
# Overhead  Command  Shared Object      Symbol
# ........  .......  .................  ......................................
#
    34.07%  pack     pack               [.] HGrid::find_neighbors
    29.13%  pack     pack               [.] intersect
     8.23%  pack     pack               [.] Ellipsoid::support
     5.74%  pack     pack               [.] Particle::world_transform
     3.72%  pack     pack               [.] closest_point_tetrahedron
     3.65%  pack     pack               [.] closest_point_triangle
     2.42%  pack     pack               [.] Simplex::update
     2.27%  pack     pack               [.] Ellipsoid::bounding_radius
     2.25%  pack     pack               [.] Simplex::contains
     2.19%  pack     pack               [.] check_overlap
     0.95%  pack     libm.so.6          [.] __sincos
     0.61%  pack     pack               [.] HGrid::insert
     0.51%  pack     pack               [.] Simplex::reduce
     0.37%  pack     pack               [.] Simplex::closest
```

CPU Features for Performance Analysis

- Performance Monitoring Unit (PMU)
  - Performance monitoring counters (PMC)
    - Hardware: cycles, instructions, branches, stalled cycles in frontend/backend, etc
    - PMUs have several slots (usually 4–6) for counting hardware events together
    - Core PMU (CPU related events) and Uncore PMUs (I/O, caches, memory, interconnect)
    - If more events need to be measured than fit in a PMU, this needs to be done via multiplexing
- Varies depending on hardware vendor/model
  - Basic events have equivalents in most hardware
  - More specific events may only be available on certain hardware models
  - Some events have the same name, but count different things (e.g. cache misses)
- Profilers make use of hardware/software events
  - Software events: page faults, context switches, migrations, etc
- Intel VTune, AMD µprof, macOS Instruments, Linux perf, etc

perf – hardware and software events

```
bash ~ $ perf list hw cache

List of pre-defined events (to be used in -e):

  branch-instructions OR branches                    [Hardware event]
  branch-misses                                      [Hardware event]
  cache-misses                                       [Hardware event]
  cache-references                                   [Hardware event]
  cpu-cycles OR cycles                               [Hardware event]
  instructions                                       [Hardware event]
  stalled-cycles-backend OR idle-cycles-backend      [Hardware event]
  stalled-cycles-frontend OR idle-cycles-frontend    [Hardware event]

  L1-dcache-load-misses                              [Hardware cache event]
  L1-dcache-loads                                    [Hardware cache event]
  L1-dcache-prefetches                               [Hardware cache event]
  L1-icache-load-misses                              [Hardware cache event]
  L1-icache-loads                                    [Hardware cache event]
  branch-load-misses                                 [Hardware cache event]
  branch-loads                                       [Hardware cache event]
  dTLB-load-misses                                   [Hardware cache event]
  dTLB-loads                                         [Hardware cache event]
  iTLB-load-misses                                   [Hardware cache event]
  iTLB-loads                                         [Hardware cache event]
```

```
bash ~ $ perf list sw

List of pre-defined events (to be used in -e):

  alignment-faults                                   [Software event]
  bpf-output                                         [Software event]
  context-switches OR cs                             [Software event]
  cpu-clock                                          [Software event]
  cpu-migrations OR migrations                       [Software event]
  dummy                                              [Software event]
  emulation-faults                                   [Software event]
  major-faults                                       [Software event]
  minor-faults                                       [Software event]
  page-faults OR faults                              [Software event]
  task-clock                                         [Software event]

  duration_time                                      [Tool event]
```

perf – Intel Skylake events

```
bash ~ $ perf list pipeline

List of pre-defined events (to be used in -e):

pipeline:
  arith.divider_active
       [Cycles when divide unit is busy executing divide or square root operations. Accounts for integer and floating-point operations]
  baclears.any
       [Counts the total number when the front end is resteered, mainly when the BPU cannot provide a correct prediction]
  br_inst_retired.all_branches
       [All (macro) branch instructions retired Spec update: SKL091]
  br_inst_retired.all_branches_pebs
       [All (macro) branch instructions retired Spec update: SKL091 (Must be precise)]
  br_inst_retired.conditional
       [Conditional branch instructions retired Spec update: SKL091 (Precise event)]
  br_inst_retired.far_branch
       [Counts the number of far branch instructions retired Spec update: SKL091 (Precise event)]
  br_inst_retired.near_call
       [Direct and indirect near call instructions retired Spec update: SKL091 (Precise event)]
  br_inst_retired.near_return
       [Return instructions retired Spec update: SKL091 (Precise event)]
  br_inst_retired.near_taken
       [Taken branch instructions retired Spec update: SKL091 (Precise event)]
  br_inst_retired.not_taken
       [Counts all not taken macro branch instructions retired Spec update: SKL091 (Precise event)]
  br_misp_retired.all_branches
       [All mispredicted macro branch instructions retired]
  ...
```

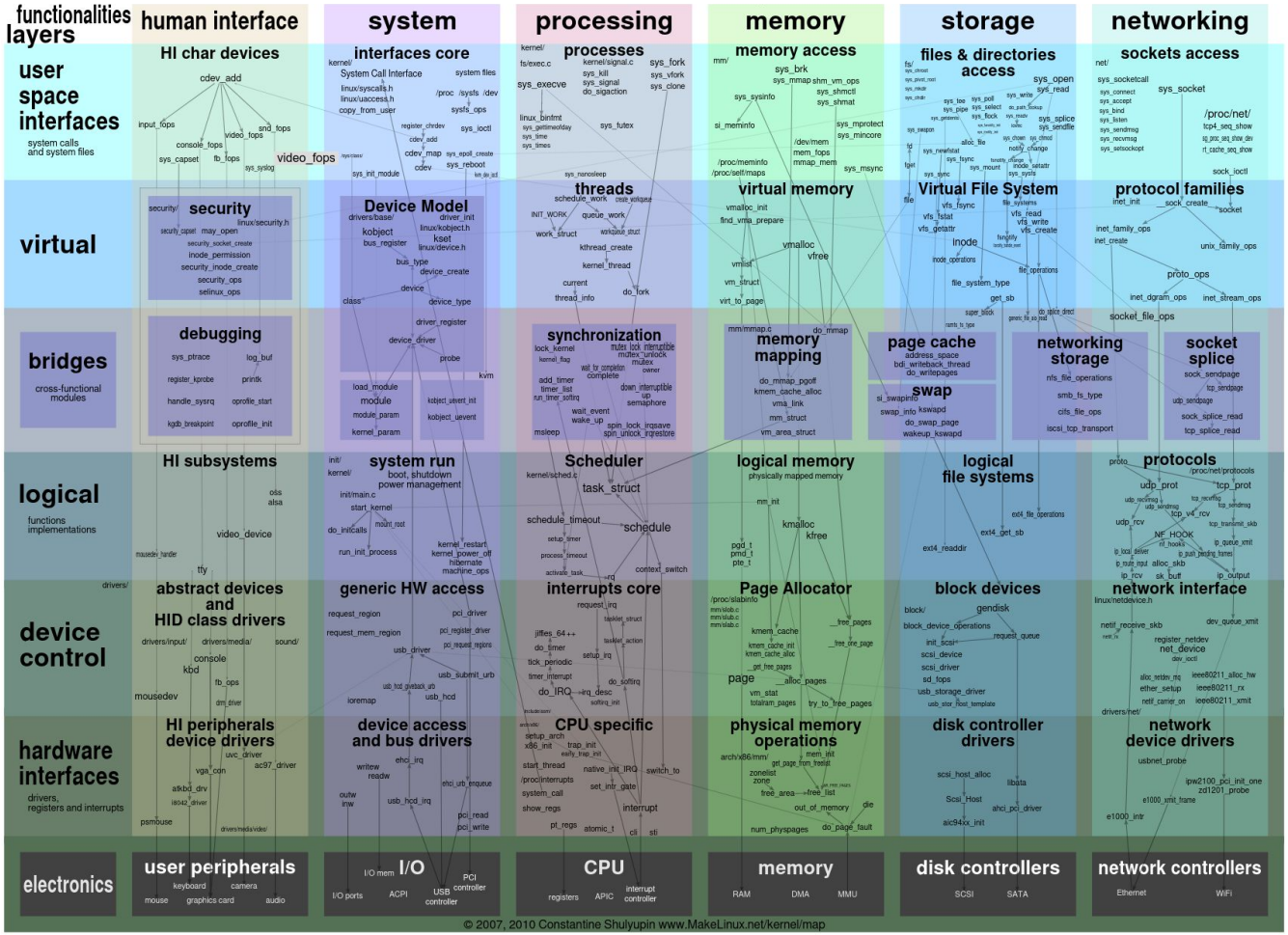
perf – AMD Ryzen events

```
bash ~ $ perf list core

List of pre-defined events (to be used in -e):

core:
  ex_div_busy
       [Div Cycles Busy count]
  ex_div_count
       [Div Op Count]
  ex_ret_brn
       [Retired Branch Instructions]
  ex_ret_brn_far
       [Retired Far Control Transfers]
  ex_ret_brn_ind_misp
       [Retired Indirect Branch Instructions Mispredicted]
  ex_ret_brn_misp
       [Retired Branch Instructions Mispredicted]
  ex_ret_brn_resync
       [Retired Branch Resyncs]
  ex_ret_brn_tkn
       [Retired Taken Branch Instructions]
  ex_ret_brn_tkn_misp
       [Retired Taken Branch Instructions Mispredicted]
  ex_ret_cond
       [Retired Conditional Branch Instructions]
  ex_ret_cond_misp
       [Retired Conditional Branch Instructions Mispredicted]
  ...
```

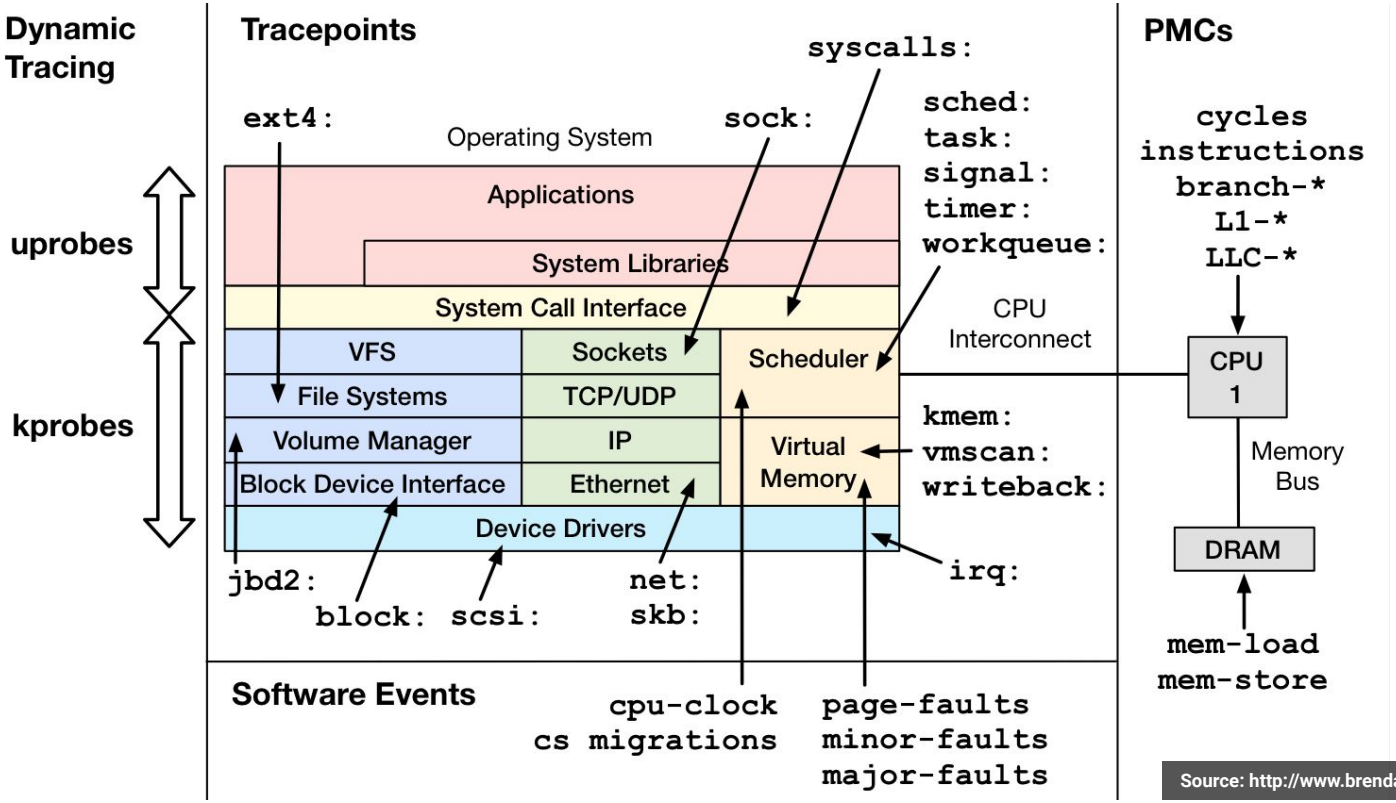
perf – static tracepoint events

```
bash ~ $ sudo perf list 'sched:*'

List of pre-defined events (to be used in -e):

  sched:sched_kthread_stop                           [Tracepoint event]
  sched:sched_kthread_stop_ret                       [Tracepoint event]
  sched:sched_migrate_task                           [Tracepoint event]
  sched:sched_move_numa                              [Tracepoint event]
  sched:sched_pi_setprio                             [Tracepoint event]
  sched:sched_process_exec                           [Tracepoint event]
  sched:sched_process_exit                           [Tracepoint event]
  sched:sched_process_fork                           [Tracepoint event]
  sched:sched_process_free                           [Tracepoint event]
  sched:sched_process_wait                           [Tracepoint event]
  sched:sched_stat_runtime                           [Tracepoint event]
  sched:sched_stick_numa                             [Tracepoint event]
  sched:sched_swap_numa                              [Tracepoint event]
  sched:sched_switch                                 [Tracepoint event]
  sched:sched_wait_task                              [Tracepoint event]
  sched:sched_wake_idle_without_ipi                  [Tracepoint event]
  sched:sched_wakeup                                 [Tracepoint event]
  sched:sched_wakeup_new                             [Tracepoint event]
  sched:sched_waking                                 [Tracepoint event]
```

Map of the Linux Kernel

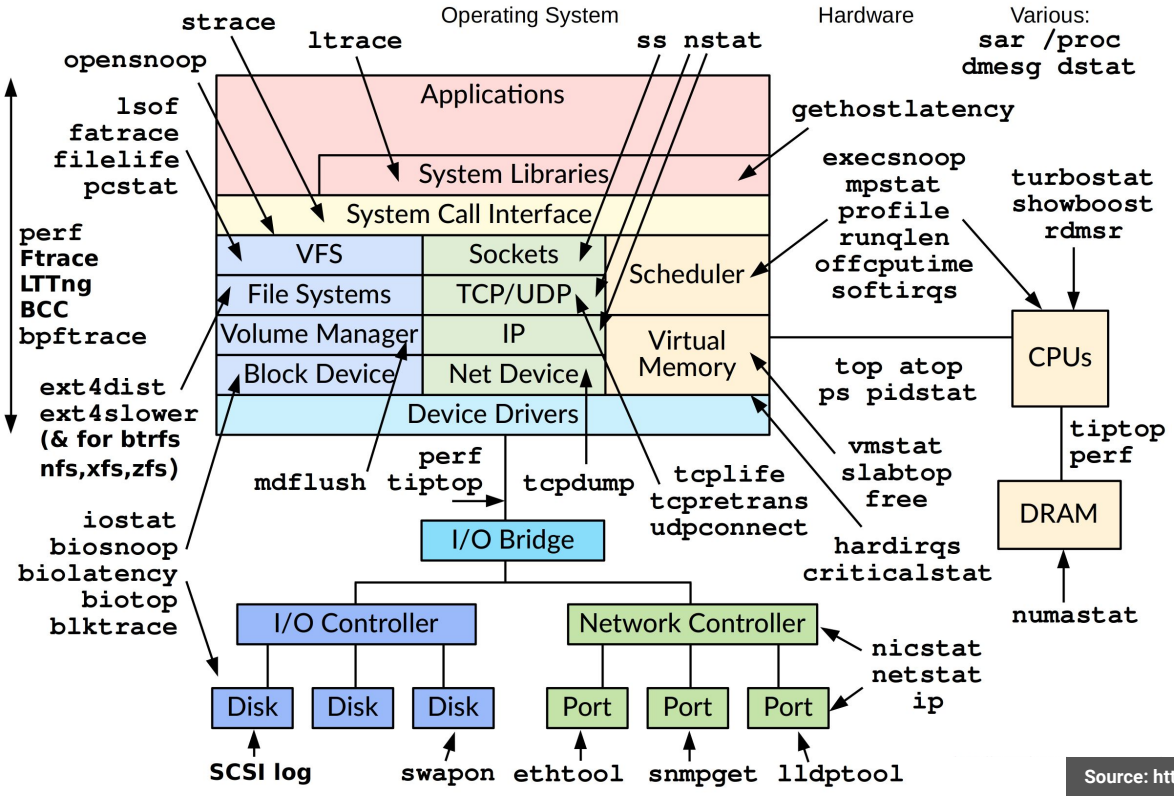


perf – event sources



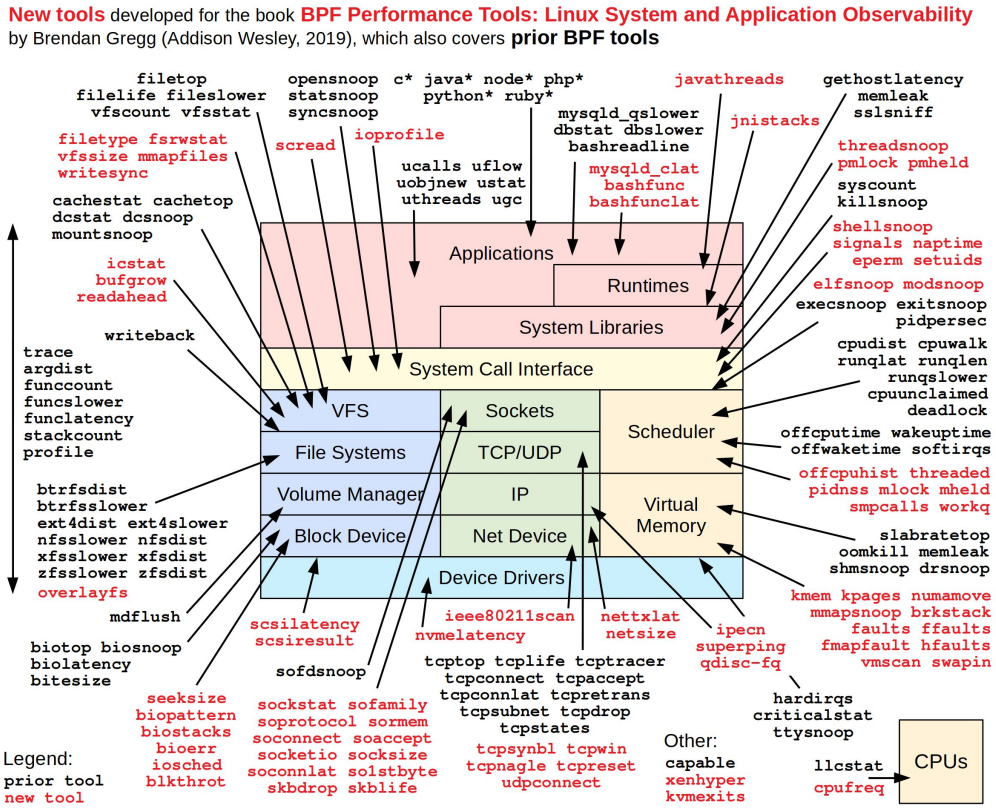
Source: http://www.brendangregg.com/perf.html

Linux Observability Tools



Source: http://www.brendangregg.com/perf.html

Linux eBPF-based Observability Tools

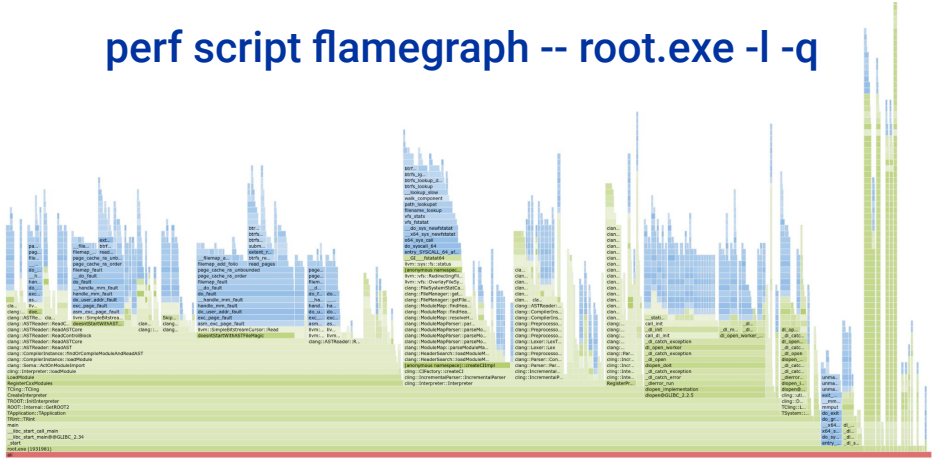


Source: http://www.brendangregg.com/perf.html

Flamegraphs

- Visualization tool by Brendan Gregg
  - https://www.brendangregg.com/flamegraphs.html
- Call stacks on the vertical axis
- Number of samples as width
- Easy to identify where time is spent
- Not very good for in-depth analysis
- Built-in support now exists in perf
- Creates browseable HTML file

perf script flamegraph -- root.exe -l -q



Avoid broken stack traces and missing symbols

- Compile code with debugging information (`-g`)
- Add `-fno-omit-frame-pointer` to compile options to keep frame pointer
- Install system packages with debugging info for the kernel and system libs

When recording data:

- Use `--call-graph=fp/dwarf` + DWARF debugging information
- Use precise events to avoid skidding (`cycles:pp` instead of just `cycles`)
- Adjust sampling rate to avoid large amounts of data and high overhead
- Sample events in a group if computing derived metrics (e.g. instr. per cycle)
- See `man perf-list` for more information on events and their modifiers

Frame Pointer

- Saved/restored on each function call
- Lightweight and accurate backtraces
- DWARF backtraces not as accurate
- High overhead for very short functions

Simple square and cube functions

```
float square(float x)
{
    return x * x;
}

float cube(float x)
{
    return x * square(x);
}
```

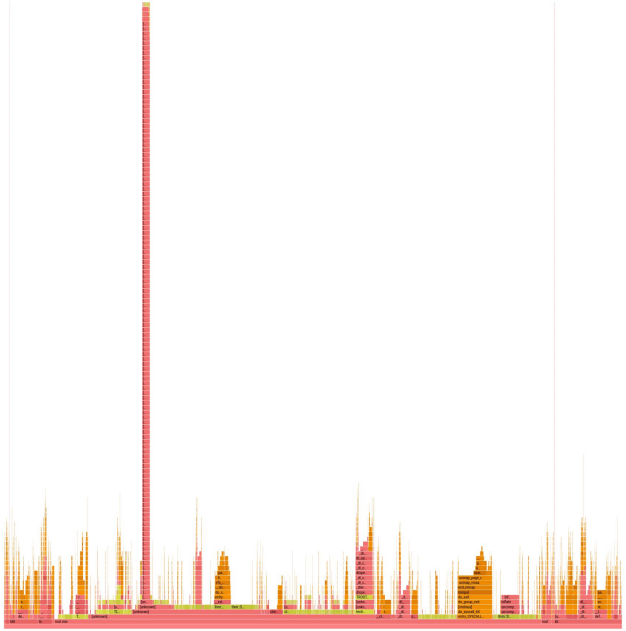
Without frame pointer

```
0000000000000000 <square>:
   0:   c5 fa 59 c0             vmulss %xmm0,%xmm0,%xmm0
   4:   c3                      ret
   5:   66 66 2e 0f 1f 84 00    data16 cs nopw 0x0(%rax,%rax,1)
   c:   00 00 00 00

0000000000000010 <cube>:
  10:   c5 f8 28 c8             vmovaps %xmm0,%xmm1
  14:   e8 00 00 00 00          call   19 <cube+0x9>
  19:   c5 fa 59 c1             vmulss %xmm1,%xmm0,%xmm0
  1d:   c3                      ret
```

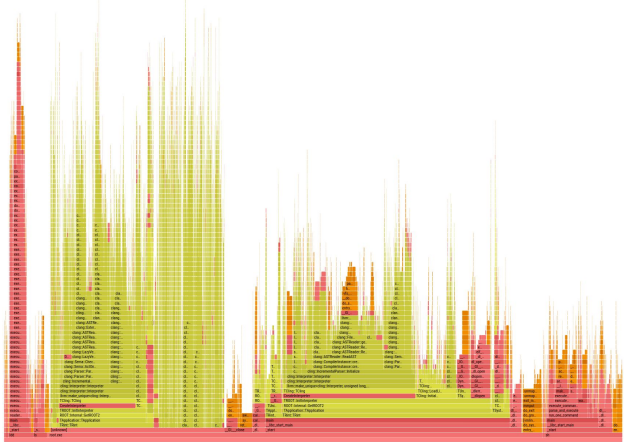
With frame pointer

```
0000000000000000 <square>:
   0:   c5 fa 59 c0             vmulss %xmm0,%xmm0,%xmm0
   4:   c3                      ret
   5:   66 66 2e 0f 1f 84 00    data16 cs nopw 0x0(%rax,%rax,1)

0000000000000010 <cube>:
  10:   55                      push   %rbp
  11:   c5 f8 28 c8             vmovaps %xmm0,%xmm1
  15:   48 89 e5                mov    %rsp,%rbp
  18:   e8 00 00 00 00          call   1d <cube+0xd>
  1d:   5d                      pop    %rbp
  1e:   c5 fa 59 c1             vmulss %xmm1,%xmm0,%xmm0
  22:   c3                      ret
```

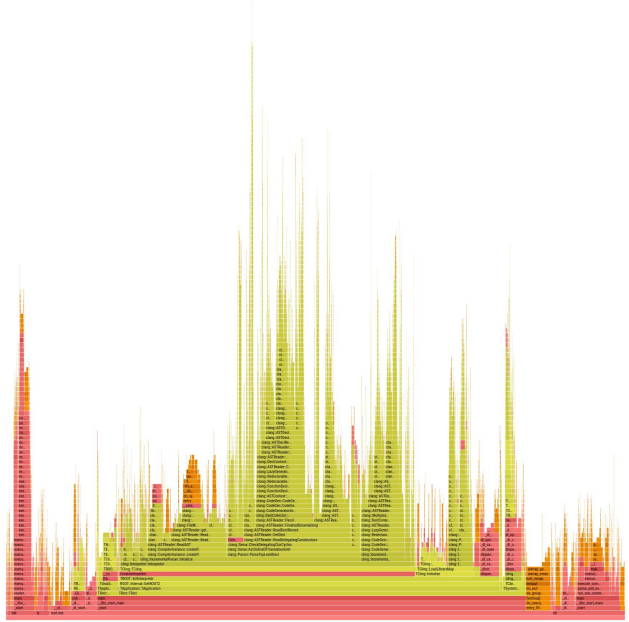
ROOT startup flamegraph for various configurations



perf record --call-graph=fp
(debugging info not available)

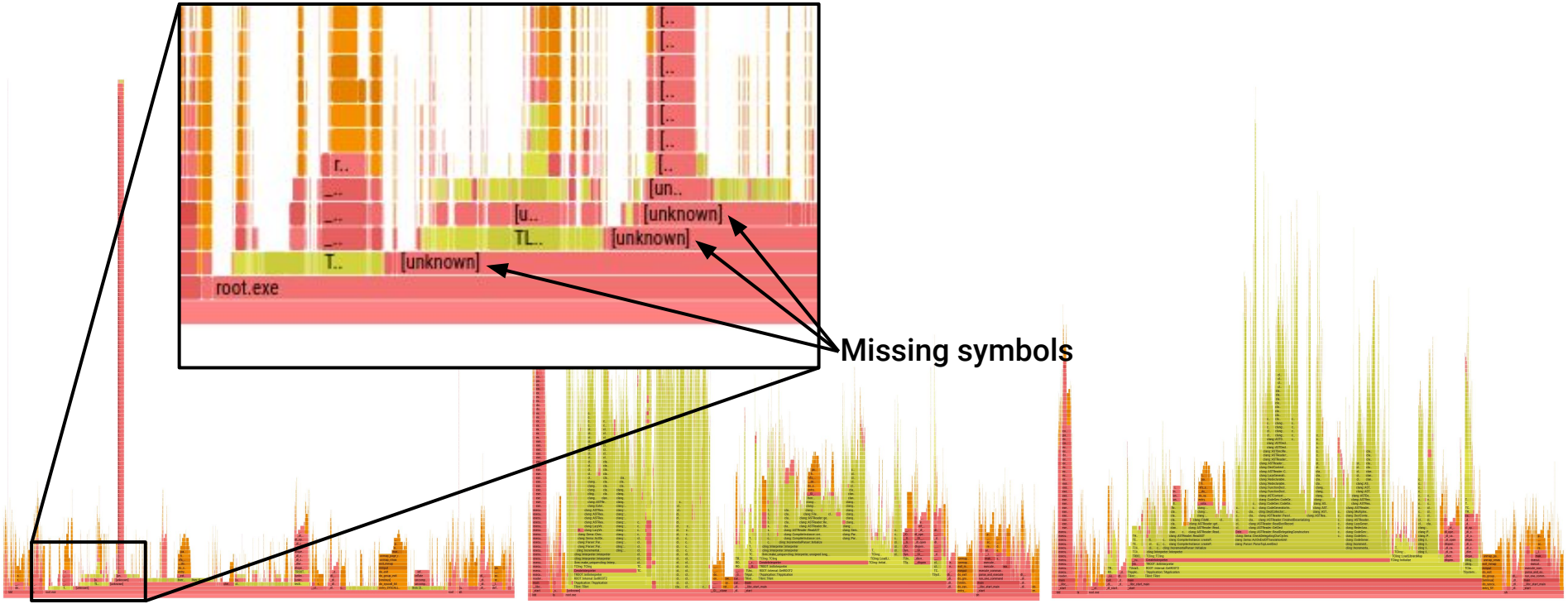


perf record --call-graph=dwarf
(frame pointer not available)



perf record --call-graph=fp
(frame pointer and debugging info)

ROOT startup flamegraph for various configurations



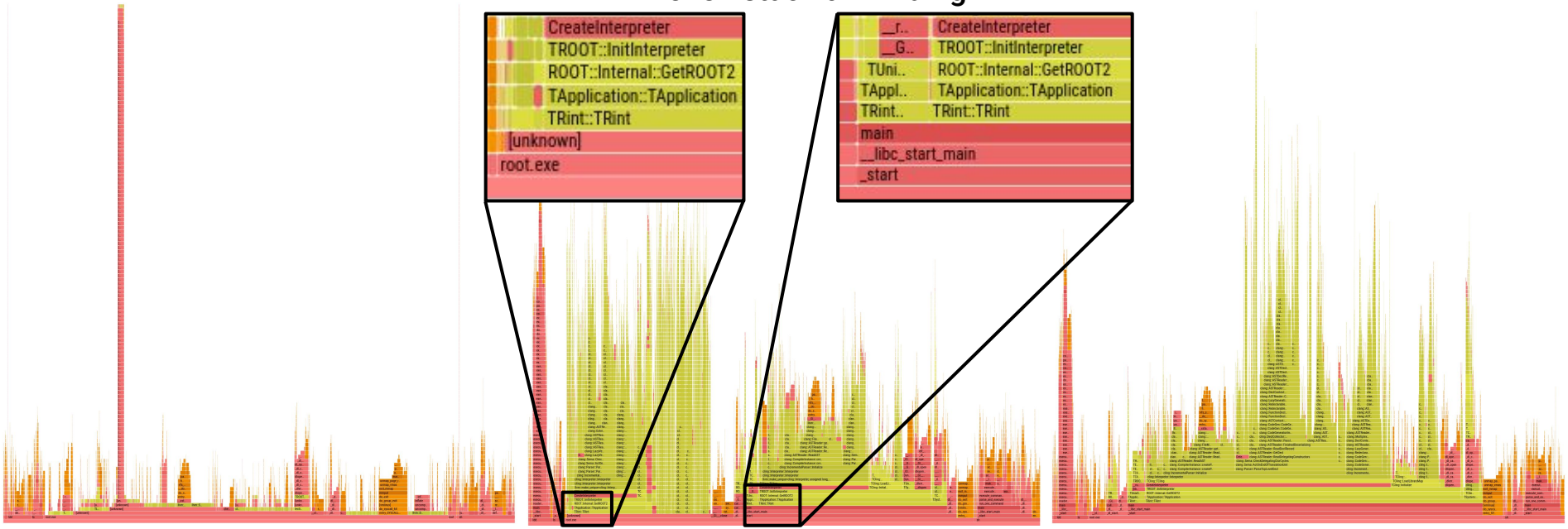
perf record --call-graph=fp
(debugging info not available)

perf record --call-graph=dwarf
(frame pointer not available)

perf record --call-graph=fp
(frame pointer and debugging info)

ROOT startup flamegraph for various configurations

Broken stack unwinding



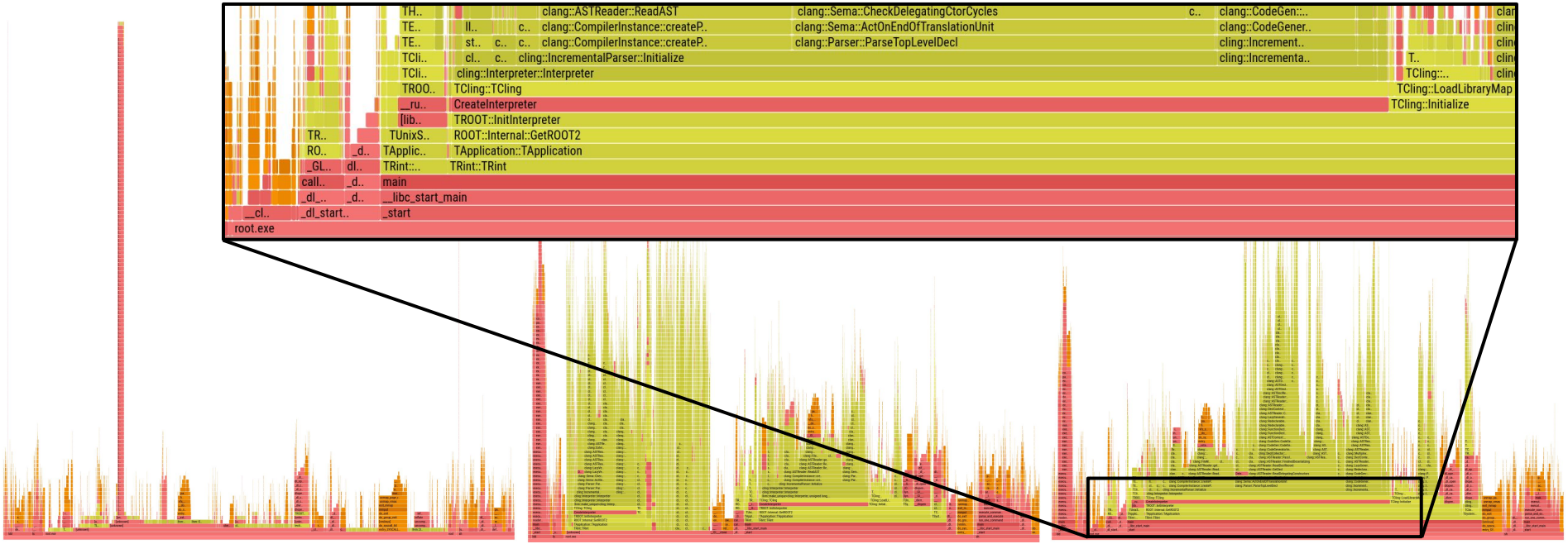
perf record --call-graph=fp
(debugging info not available)

perf record --call-graph=dwarf
(frame pointer not available)

perf record --call-graph=fp
(frame pointer and debugging info)

ROOT startup flamegraph for various configurations

Correctly merged stacks



perf record --call-graph=fp
(debugging info not available)

perf record --call-graph=dwarf
(frame pointer not available)

perf record --call-graph=fp
(frame pointer and debugging info)

perf stat – counting cycles vs instructions vs wall time

```
# measure ROOT startup 20 times and print stats with averages and deviations

$ perf stat -d -r 20 -- root.exe -l -q >/dev/null

Performance counter stats for 'root.exe -l -q' (20 runs):

          119.72 msec task-clock                #    0.442 CPUs utilized           ( +-  0.76% )
             579      context-switches          #    0.005 M/sec                   ( +-  4.34% )
              13      cpu-migrations            #    0.109 K/sec                   ( +-  6.94% )
           11260      page-faults               #    0.094 M/sec                   ( +-  0.49% )
       493768274      cycles                    #    4.125 GHz                     ( +-  0.75% )  (66.72%)
        33420383      stalled-cycles-frontend   #    6.77% frontend cycles idle    ( +-  1.56% )  (75.75%)
       177325752      stalled-cycles-backend    #   35.91% backend cycles idle     ( +-  1.87% )  (79.76%)
       532310517      instructions              #    1.08  insn per cycle
                                                #    0.33  stalled cycles per insn ( +-  0.35% )  (82.16%)
       107905661      branches                  #  901.351 M/sec                   ( +-  0.26% )  (82.38%)
         2282743      branch-misses             #    2.12% of all branches         ( +-  0.77% )  (78.52%)
       246528817      L1-dcache-loads           # 2059.290 M/sec                   ( +-  1.12% )  (71.14%)
         5628008      L1-dcache-load-misses     #    2.28% of all L1-dcache hits   ( +-  1.30% )  (63.57%)
<not supported>       LLC-loads
<not supported>       LLC-load-misses

          0.2709 +- 0.0205 seconds time elapsed  ( +-  7.58% )
```

```
# same measurements again, to show difference in noise for wall time, cycles, instructions

$ perf stat -d -r 20 -- root.exe -l -q >/dev/null

Performance counter stats for 'root.exe -l -q' (20 runs):

          118.38 msec task-clock                #    0.565 CPUs utilized           ( +-  0.73% )
             433      context-switches          #    0.004 M/sec                   ( +- 12.62% )
              12      cpu-migrations            #    0.103 K/sec                   ( +-  5.57% )
           11267      page-faults               #    0.095 M/sec                   ( +-  0.50% )
       488189557      cycles                    #    4.124 GHz                     ( +-  0.73% )  (60.32%)
        32509432      stalled-cycles-frontend   #    6.66% frontend cycles idle    ( +-  1.70% )  (78.43%)
       175081210      stalled-cycles-backend    #   35.86% backend cycles idle     ( +-  1.45% )  (83.54%)
       533538019      instructions              #    1.09  insn per cycle
                                                #    0.33  stalled cycles per insn ( +-  0.35% )  (84.97%)
       108436560      branches                  #  915.999 M/sec                   ( +-  0.29% )  (84.34%)
         2279445      branch-misses             #    2.10% of all branches         ( +-  1.05% )  (81.41%)
       244414949      L1-dcache-loads           # 2064.653 M/sec                   ( +-  0.94% )  (71.80%)
         5720566      L1-dcache-load-misses     #    2.34% of all L1-dcache hits   ( +-  1.35% )  (55.19%)
<not supported>       LLC-loads
<not supported>       LLC-load-misses

          0.2093 +- 0.0220 seconds time elapsed  ( +- 10.53% )


# (ratio of wall clock durations)
$ bc -l <<< "0.2709 / 0.2093"
1.29431438127090301003

# (ratio of cycles measurements)
$ bc -l <<< "493768274 / 488189557"
1.01142735832835522944

# (ratio of instructions measurements)
$ bc -l <<< "532310517 / 533538019"
0.99769931671917086006
```

Intel’s Last Branch Record

- Useful when frame pointers are not available
- Use with `perf record -b` or `perf record --call-graph=lbr`
- Hardware registers on Intel CPUs that allow sampling branches
- Registers hold a ring buffer of the most recent branch decisions
- Useful to analyze branching behavior (branching probabilities, mispredictions)
- Available on AMD Zen4 or later CPUs
  - On older CPUs, some events provide similar functionality
- Articles describing LBR on LWN.net
  - An introduction to last branch records [LWN.net]
  - Advanced usage of last branch records [LWN.net]

Precise CPU Events for Sampling

- PMU counts events on a per-core basis
  - Sample is taken when counter reaches threshold
  - Fixed frequency sampling achieved by predicting/adjusting the threshold
  - Instruction-level parallelism and speculative execution introduce noise and skidding
    - Only one base pointer per thread
    - Many instructions in flight on the core at the same time
    - Shared resources mean mixed counting when using hyperthreading
- Intel Processor Event-Based Sampling (PEBS)
  - Instruction pointer (and auxiliary information) stored in a designated area
  - No interrupts during sampling, reduced or no skidding
- AMD Instruction-Based Sampling (IBS)
  - Tracks instructions rather than events, marks every Nth instruction to be tracked
  - Two forms: IBS Fetch sampling (front-end) and IBS Op sampling (back-end)

"Precise Event Sampling on AMD Versus Intel: Quantitative and Qualitative Comparison" doi: 10.1109/TPDS.2023.3257105

Instructions vs Micro-operations (µops)

Instructions from a CISC instruction set are usually broken into one or more RISC-like operations in hardware. For example, an addition of two values from memory may be broken into memory loads into registers, the addition itself, then memory stores.
These operations are usually called **micro-ops** and abbreviated as **µops**. Some PMUs have hardware events that allow counting separately µops issued, executed, and retired.

While instructions are usually split into simpler µops, the µops can instead be fused together when instructions are decoded in the front-end of the processor. **Microfusion** is when µops from the same machine instruction are fused together, and **macrofusion** is when µops from distinct instructions are fused.

Instructions Retired vs Executed

Instructions executed refers to any instructions that have been processed by the CPU. For example, a multiplication of two numbers that has loaded the inputs, calculated the results and stored it somewhere. This metric includes speculatively executed instructions on branches that may have been discarded later on.

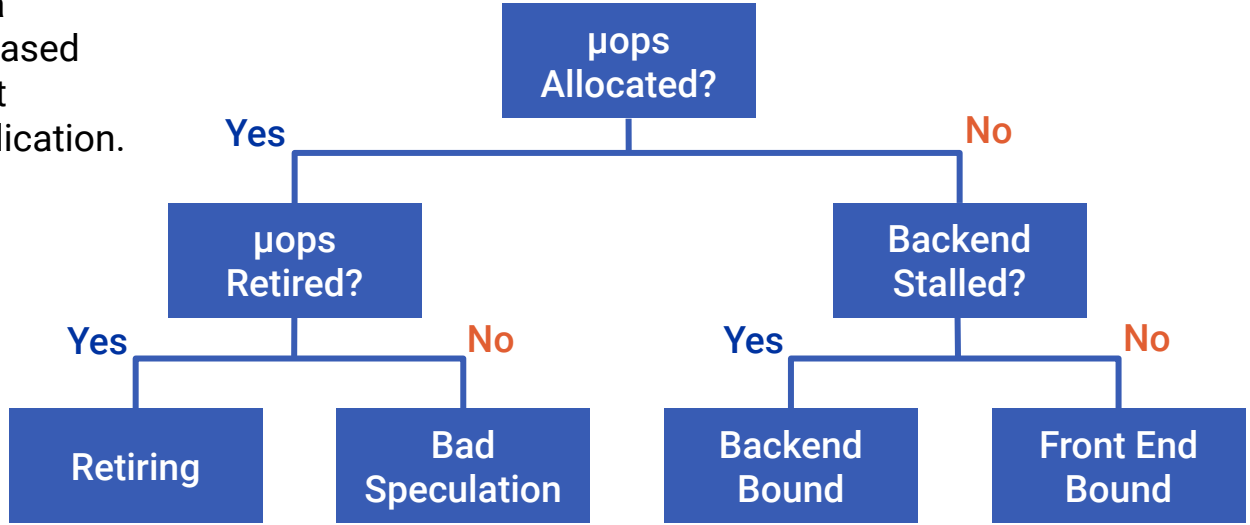
Instructions retired refers to executed instructions that have actually contributed to the main line of execution of a program, that is, that has not been discarded as speculatively executed.

Instructions per cycle (IPC) is a measure of the instruction-level parallelism, or how many instructions were retired on average in each CPU cycle. CPI (cycles per instruction) is also common. Typically up to 4 instructions per cycle can be executed on AMD/Intel CPUs.

Top-Down Microarchitecture Analysis

The Top-Down Characterization is a hierarchical organization of event-based metrics that identifies the dominant performance bottlenecks in an application.

Its aim is to show, on average, how well the CPU's pipelines are being utilized while running an application.

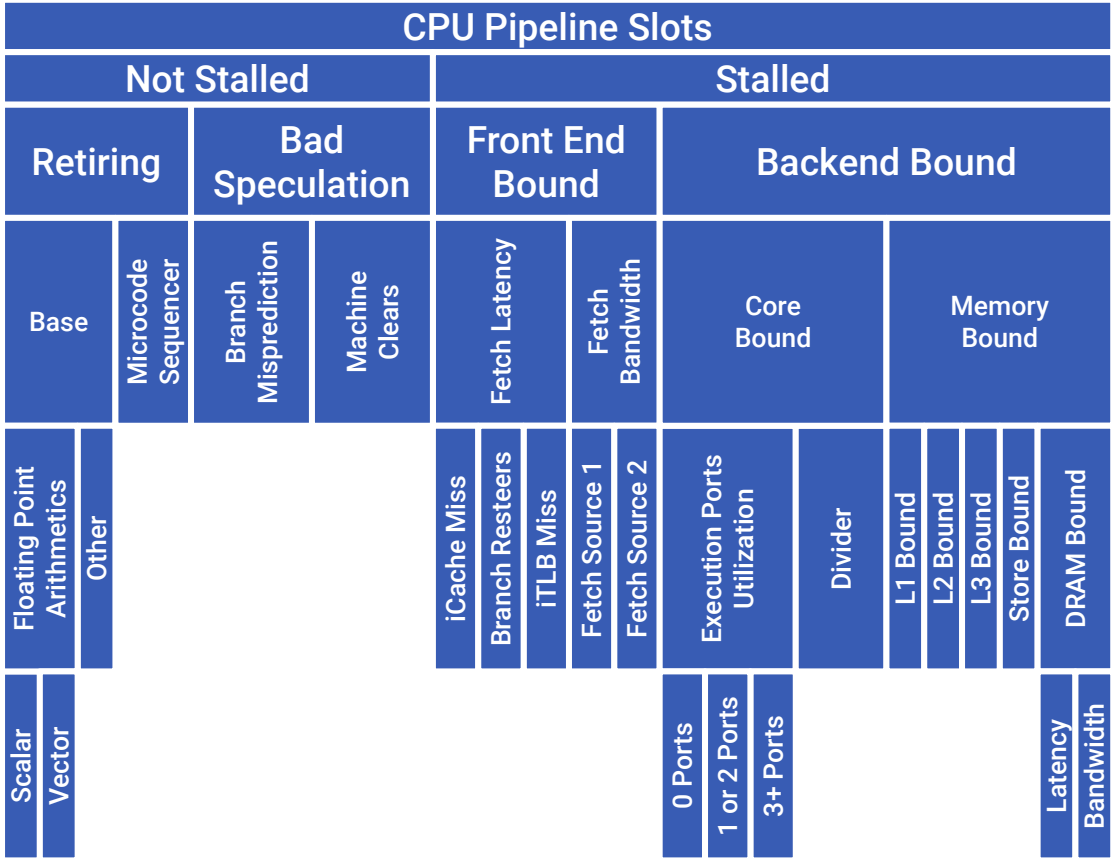


Ahmad Yasin, "A Top-Down method for performance analysis and counters architecture," *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Monterey, CA, 2014, pp. 35-44, doi: 10.1109/ISPASS.2014.6844459.

https://www.intel.com/content/www/us/en/develop/documentation/vtune-cookbook/top/methodologies/top-down-microarchitecture-analysis-method.html

Top-Down Microarchitecture Analysis

- Retiring
  - Useful Work
- Bad Speculation
  - Branching Issues
- Front End Bound
  - Instruction Fetch Issues
- Back End Bound
  - Core Bound
    - Port Utilization
    - Execution Latency
  - Memory Bound
    - Cache misses
    - Memory Bandwidth



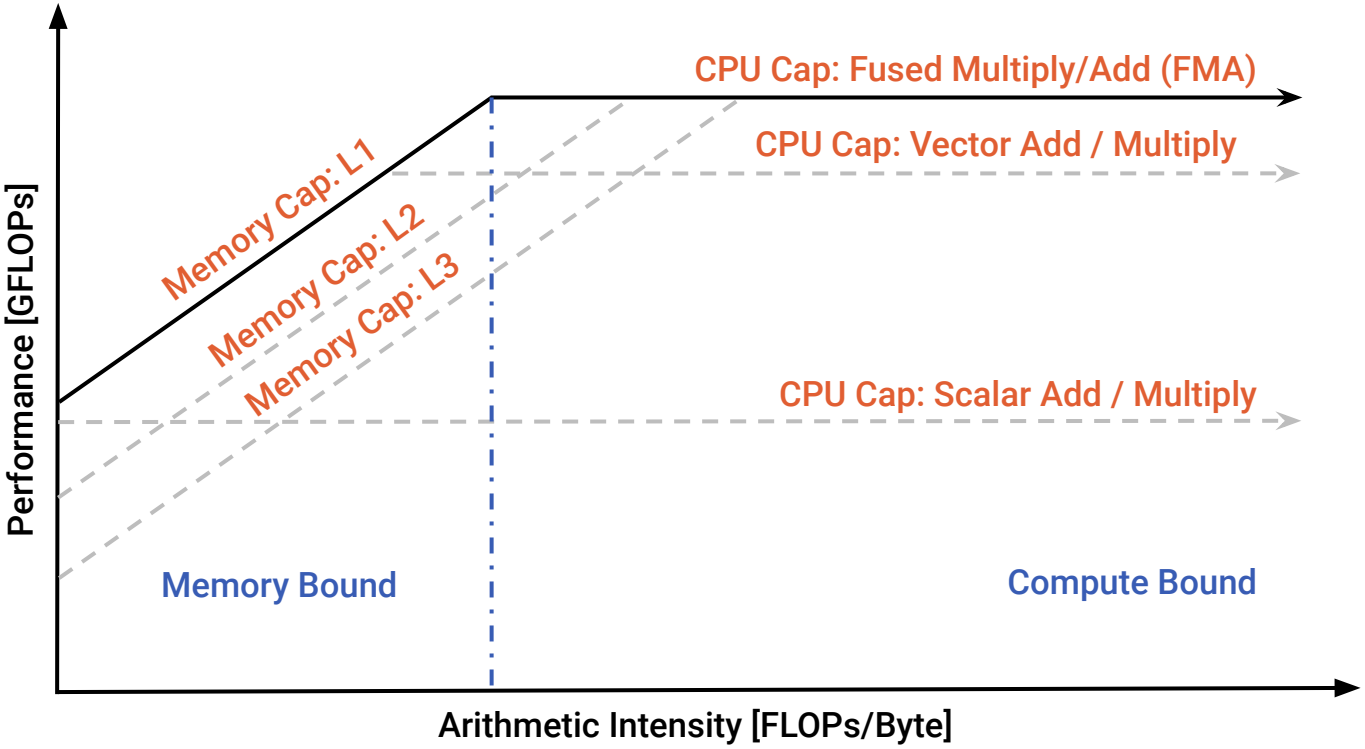
Ahmad Yasin, "A Top-Down method for performance analysis and counters architecture," *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Monterey, CA, 2014, pp. 35-44, doi: 10.1109/ISPASS.2014.6844459

Expected Ranges of Pipeline Slots for Each Category

Category	Client/Desktop Application	Server/Database Distributed Application	High Performance Computing (HPC) Application
Retiring	20 – 50%	10 – 30%	30 – 70%
Back-End Bound	20 – 40%	20 – 60%	20 – 40%
Front-End Bound	5 – 10%	10 – 25%	5 – 10%
Bad Speculation	5 – 10%	5 – 10%	1 – 5%

https://www.intel.com/content/www/us/en/develop/documentation/vtune-cookbook/top/methodologies/top-down-microarchitecture-analysis-method.html

Roofline Performance Model



https://www.intel.com/content/www/us/en/developer/articles/guide/intel-advisor-roofline.html

perf – recording and reporting data

```
bash ~ $ perf record -g -F max -- root.exe -l -q
info: Using a maximum frequency rate of 32500 Hz

[ perf record: Woken up 6 times to write data ]
[ perf record: Captured and wrote 2.003 MB perf.data (7035 samples) ]
bash ~ $ perf report -q --stdio -c root.exe | head -n 20
# comm: root.exe
    82.13%     0.00%  root.exe                 [.] _start
            |
            ---_start
               __libc_start_main@@GLIBC_2.34
               __libc_start_call_main
               main
               |
               |--79.94%--TRint::TRint
               |          |
               |          |--76.22%--TApplication::TApplication
               |          |          |
               |          |           --76.14%--ROOT::Internal::GetROOT2
               |          |                     TROOT::InitInterpreter
               |          |                     |
               |          |                     |--69.43%--CreateInterpreter
               |          |                     |          |
               |          |                     |           --69.41%--TCling::TCling
               |          |                     |                     |
               |          |                     |                     |--32.52%--RegisterCxxModules
```

perf – flat profile report

```
bash ~ $ perf report -q --stdio --call-graph=none -c root.exe | head -n 25
# comm: root.exe
    82.13%     0.01%  root.exe                 [.] main
    82.13%     0.00%  libc.so.6                [.] __libc_start_call_main
    82.13%     0.00%  libc.so.6                [.] __libc_start_main@@GLIBC_2.34
    82.13%     0.00%  root.exe                 [.] _start
    79.94%     0.00%  libRint.so.6.30.06       [.] TRint::TRint
    76.22%     0.00%  libCore.so.6.30.06       [.] TApplication::TApplication
    76.14%     0.00%  libCore.so.6.30.06       [.] ROOT::Internal::GetROOT2
    76.14%     0.00%  libCore.so.6.30.06       [.] TROOT::InitInterpreter
    69.43%     0.00%  libCling.so.6.30.06      [.] CreateInterpreter
    69.41%     0.00%  libCling.so.6.30.06      [.] TCling::TCling
    38.76%     0.00%  libCling.so.6.30.06      [.] clang::CompilerInstance::loadModule
    38.52%     0.00%  libCling.so.6.30.06      [.] clang::CompilerInstance::findOrCompileModuleAndReadAST
    38.32%     0.21%  libCling.so.6.30.06      [.] clang::ASTReader::ReadAST
    32.52%     0.00%  libCling.so.6.30.06      [.] RegisterCxxModules
    32.26%     0.00%  libCling.so.6.30.06      [.] LoadModule
    31.87%     0.00%  libCling.so.6.30.06      [.] cling::Interpreter::loadModule
    31.75%     0.01%  libCling.so.6.30.06      [.] clang::Sema::ActOnModuleImport
    26.63%     0.00%  libCling.so.6.30.06      [.] cling::Interpreter::Interpreter
    22.80%     0.28%  [kernel.kallsyms]        [k] entry_SYSCALL_64
    22.51%     0.29%  [kernel.kallsyms]        [k] asm_exc_page_fault
    22.14%     0.36%  [kernel.kallsyms]        [k] do_syscall_64
    21.68%     0.31%  [kernel.kallsyms]        [k] exc_page_fault
    19.17%     0.38%  [kernel.kallsyms]        [k] do_user_addr_fault
    19.08%     0.00%  libCling.so.6.30.06      [.] cling::IncrementalParser::ParseInternal
```

perf – flat profile report by self-time

```
bash ~ $ perf report -q --stdio --call-graph=none --no-children --percent-limit 0.75 -c root.exe
# comm: root.exe
     5.85%  libz.so.1.3.1           [.] inflate_fast
     4.11%  libCling.so.6.30.06     [.] llvm::SimpleBitstreamCursor::Read
     2.63%  [kernel.kallsyms]       [k] unmap_page_range
     2.27%  libCling.so.6.30.06     [.] llvm::BitstreamCursor::readRecord
     1.89%  [kernel.kallsyms]       [k] __mod_lruvec_state
     1.78%  [kernel.kallsyms]       [k] srso_untrain_ret
     1.77%  [kernel.kallsyms]       [k] srso_return_thunk
     1.53%  [kernel.kallsyms]       [k] trace_hardirqs_off
     1.43%  libz.so.1.3.1           [.] adler32_z
     1.32%  [kernel.kallsyms]       [k] __lruvec_stat_mod_folio
     1.31%  [kernel.kallsyms]       [k] clear_page_rep
     1.31%  ld-linux-x86-64.so.2    [.] _dl_lookup_symbol_x
     1.26%  libCling.so.6.30.06     [.] llvm::StringMapImpl::LookupBucketFor
     1.10%  [kernel.kallsyms]       [k] preempt_count_add
     1.04%  [kernel.kallsyms]       [k] __mod_memcg_lruvec_state
     0.97%  [kernel.kallsyms]       [k] link_path_walk
     0.94%  [kernel.kallsyms]       [k] preempt_count_sub
     0.92%  libz.so.1.3.1           [.] inflate_table
     0.85%  [kernel.kallsyms]       [k] percpu_counter_add_batch
     0.81%  libc.so.6               [.] _int_malloc
     0.79%  libz.so.1.3.1           [.] inflate
     0.76%  ld-linux-x86-64.so.2    [.] do_lookup_x
```

perf – hierarchical profile report

```
bash ~ $ perf report -q --stdio --call-graph=none --hierarchy --percent-limit 1 --comm root.exe
# comm: root.exe
    92.93%        root.exe
       47.60%        [kernel.kallsyms]
           2.63%        [k] unmap_page_range
           1.89%        [k] __mod_lruvec_state
           1.78%        [k] srso_untrain_ret
           1.77%        [k] srso_return_thunk
           1.53%        [k] trace_hardirqs_off
           1.32%        [k] __lruvec_stat_mod_folio
           1.31%        [k] clear_page_rep
           1.10%        [k] preempt_count_add
           1.04%        [k] __mod_memcg_lruvec_state
       26.75%        libCling.so.6.30.06
           4.11%        [.] llvm::SimpleBitstreamCursor::Read
           2.27%        [.] llvm::BitstreamCursor::readRecord
           1.26%        [.] llvm::StringMapImpl::LookupBucketFor
        9.10%        libz.so.1.3.1
           5.85%        [.] inflate_fast
           1.43%        [.] adler32_z
        4.56%        libc.so.6
                        no entry >= 1.00%
        2.96%        ld-linux-x86-64.so.2
           1.31%        [.] _dl_lookup_symbol_x
        1.12%        libCore.so.6.30.06
                        no entry >= 1.00%
```

perf – pre-packaged metrics (Intel CPU)

```
bash ~ $ perf list metrics

Metrics:

  Backend_Bound
       [This category represents fraction of slots where no uops are delivered due to a lack of required resources for accepting new uops in the Backend]
  Bad_Speculation
       [This category represents fraction of slots wasted due to incorrect speculations]
  BpTB
       [Branch instructions per taken branch]
  CLKS
       [Per-Logical Processor actual clocks when the Logical Processor is active]
  CPI
       [Cycles Per Instruction (per Logical Processor)]
  CPU_Utilization
       [Average CPU Utilization]
  CoreIPC
       [Instructions Per Cycle (per physical core)]
  Frontend_Bound
       [This category represents fraction of slots where the processor's Frontend undersupplies its Backend]
  ILP
       [Instruction-Level-Parallelism (average number of uops executed when there is at least 1 uop executed)]
  IPC
       [Instructions Per Cycle (per Logical Processor)]
  Instructions
       [Total number of retired Instructions]
  IpB
       [Instructions per Branch (lower number means higher occurance rate)]
  IpCall
       [Instruction per (near) call (lower number means higher occurance rate)]
  IpL
       [Instructions per Load (lower number means higher occurance rate)]
```

perf – pre-packaged metrics (Intel CPU)

```
bash ~ $ perf stat -M Frontend_Bound,Backend_Bound,Bad_Speculation,Retiring -- root -l -q


 Performance counter stats for 'root -l -q':

        535853293      cycles
                                                  #     0.32 Frontend_Bound          (50.07%)
        676507752      idq_uops_not_delivered.core                                    (50.07%)
        803157447      uops_issued.any            #     0.10 Bad_Speculation
                                                  #     0.28 Backend_Bound           (49.93%)
        540449552      cycles
                                                  #     0.31 Retiring                (49.93%)
        676523326      idq_uops_not_delivered.core                                    (49.93%)
         19393734      int_misc.recovery_cycles                                       (49.93%)
        667220596      uops_retired.retire_slots                                      (49.93%)

      0.243072802 seconds time elapsed

      0.158384000 seconds user
      0.088028000 seconds sys

bash ~ $
```

Example – using perf + awk to get percent retiring

```
bash df102_NanoAODDimuonAnalysis $ perf record -F max -e '{cpu_clk_unhalted.thread,uops_retired.retire_slots}' -- df102_NanoAODD
imuonAnalysis 8 Run2012B_DoubleMuParked.root Run2012C_DoubleMuParked.root
info: Using a maximum frequency rate of 8,000 Hz
Couldn't synthesize cgroup events.
[ perf record: Woken up 57 times to write data ]
[ perf record: Captured and wrote 15.548 MB perf.data (406080 samples) ]
bash df102_NanoAODDimuonAnalysis $ perf report -q --stdio --group -F period,symbol -w 0,90 | head
   104728157092  9748014622  [.] ROOT::Detail::RDF::RFilter<bool (*)(ROOT::VecOps::RVec<int> const&), ROOT::Detail::RDF
   94152141228 10108015162  [.] ROOT::Detail::RDF::RFilter<bool (*)(unsigned int), ROOT::Detail::RDF::RLoopManager>::C
   79494119241  3454005181  [.] TTree::LoadTree
   51302076953 92238138357  [.] inflate_fast
   35698053547 14764022146  [.] TBranch::GetEntry
   24610036915  2248003372  [.] TLeafI::GetMaximum
   14942022413 13312019968  [.] tbb::internal::custom_scheduler<tbb::internal::IntelSchedulerTraits>::receive_or_steal
    8372012558  2912004368  [k] sysret_check
    7632011448  1702002553  [.] ROOT::Detail::RDF::RCustomColumn<float (*)(ROOT::VecOps::RVec<float> const&, ROOT::Vec
    7476011214  6442009663  [.] ROOT::Internal::RDF::RColumnValue<ROOT::VecOps::RVec<float> >::Get<ROOT::VecOps::RVec<
bash df102_NanoAODDimuonAnalysis $
```

Example – using perf + awk to get percent retiring

```
bash df102_NanoAODDimuonAnalysis $ echo "Retiring  Symbol"; perf report -q -F period,symbol --percent-limit 1 | awk '/^$/{next}
{ symbol = gensub(".*\\[.\\] ","","g"); slots = 4*$1; retiring = 100*$2/slots; printf("%7.2f%%  %s\n", retiring, symbol) | "sort
 -nr"; }' | cut -b -128
Retiring  Symbol
  70.18%  adler32_z
  44.95%  inflate_fast
  37.48%  ROOT::Internal::TTreeReaderValueBase::ProxyReadTemplate<&ROOT::Detail::TBranchProxy::ReadNoParentNoBranchCountNoCollec
  27.16%  __expm1f
  22.27%  tbb::internal::custom_scheduler<tbb::internal::IntelSchedulerTraits>::receive_or_steal_task
  21.54%  ROOT::Internal::RDF::RColumnValue<ROOT::VecOps::RVec<float> >::Get<ROOT::VecOps::RVec<float>, 0>
  10.36%  ROOT::Detail::RDF::RLoopManager::RunAndCheckFilters
  10.34%  TBranch::GetEntry
   8.70%  sysret_check
   5.58%  ROOT::Detail::RDF::RCustomColumn<float (*)(ROOT::VecOps::RVec<float> const&, ROOT::VecOps::RVec<float> const&, ROOT::V
   2.68%  ROOT::Detail::RDF::RFilter<bool (*)(unsigned int), ROOT::Detail::RDF::RLoopManager>::CheckFilters
   2.33%  ROOT::Detail::RDF::RFilter<bool (*)(ROOT::VecOps::RVec<int> const&), ROOT::Detail::RDF::RFilter<bool (*)(unsigned int)
   2.28%  TLeafI::GetMaximum
   1.09%  TTree::LoadTree
bash df102_NanoAODDimuonAnalysis $ _
```

Matrix Multiplication

```c
#include <stdio.h>
#include <stdlib.h>

// This version has minor modifications applied, the
// original version is linked at the bottom of the slide

#define SIZE 1024
#define LENGTH 32

int **mkmatrix(int rows, int cols);
void zeromatrix(int rows, int cols, int **m);
void freematrix(int rows, int **m);

int **mmult(int rows, int cols,
            int **m1, int **m2, int **m3) {
    int i, j, k;

    for (i=0; i<rows; i++) {
        for (j=0; j<cols; j++) {
            m3[i][j] = 0;
            for (k=0; k<cols; k++)
                m3[i][j] += m1[i][k] * m2[k][j];
        }
    }
    return(m3);
}
```

https://github.com/llvm-mirror/test-suite/blob/master/SingleSource/Benchmarks/Shootout/matrix.c

```c
int main(int argc, char *argv[]) {

    int i, n = ((argc == 2) ? atoi(argv[1]) : LENGTH);

    int **m1 = mkmatrix(SIZE, SIZE);
    int **m2 = mkmatrix(SIZE, SIZE);
    int **mm = mkmatrix(SIZE, SIZE);

    zeromatrix(SIZE, SIZE, mm);

    for (i=0; i<n; i++)
       mm = mmult(SIZE, SIZE, m1, m2, mm);

    printf("%d %d %d %d\n",
           mm[0][0], mm[2][3], mm[3][2], mm[4][4]);

    freematrix(SIZE, m1);
    freematrix(SIZE, m2);
    freematrix(SIZE, mm);
    return(0);
}
```

Simple Top-Down Analysis with perf

```
bash ~ $ perf stat -M Retiring,Bad_Speculation,Frontend_Bound,Backend_Bound a.out
1431831040 368052224 -168294912 -692581888

 Performance counter stats for 'a.out':

        2686289661      IDQ_UOPS_NOT_DELIVERED.CORE #     0.00 Frontend_Bound
                                                    #     0.55 Backend_Bound          (50.01%)
         200034632      INT_MISC.RECOVERY_CYCLES                                      (50.01%)
      135846388590      CPU_CLK_UNHALTED.THREAD                                       (50.01%)
      241410802284      UOPS_ISSUED.ANY                                               (50.01%)
       30384549081 ns   duration_time
         199807164      INT_MISC.RECOVERY_CYCLES  #     0.00 Bad_Speculation          (49.99%)
      135871753474      CPU_CLK_UNHALTED.THREAD   #     0.44 Retiring                 (49.99%)
      240760535477      UOPS_RETIRED.RETIRE_SLOTS                                     (49.99%)
      241407738202      UOPS_ISSUED.ANY                                               (49.99%)
       30384549081 ns   duration_time

      30.384549081 seconds time elapsed

      30.356367000 seconds user
       0.009971000 seconds sys


bash ~ $
```

Annotated Source

```
Samples: 30K of event 'cycles', 1000 Hz, Event count (approx.): 135123213483
main  /home/amadio/a.out [Percent: local period]
  0.00 | 58:   mov  (%r12,%r9,1),%rdi
           m3[i][j] += m1[i][k] * m2[k][j];
  0.00         mov  0x0(%r13,%r9,1),%r8
               xor  %edx,%edx
               nop
           m3[i][j] = 0;
  0.02 | 68:   movl $0x0,(%rdi,%rdx,1)
  0.00         xor  %eax,%eax
  0.01         xor  %esi,%esi
           m3[i][j] += m1[i][k] * m2[k][j];
  5.37 | 73:   mov  0x0(%rbp,%rax,8),%rcx
 35.56         mov  (%rcx,%rdx,1),%ecx
 20.16         imul (%r8,%rax,4),%ecx
           for (k = 0; k < cols; k++)
  5.62         add  $0x1,%rax
           m3[i][j] += m1[i][k] * m2[k][j];
  9.61         add  %ecx,%esi
 17.01         mov  %esi,(%rdi,%rdx,1)
           for (k = 0; k < cols; k++)
  0.01         cmp  $0x400,%rax
  6.63       ↑ jne  73
Press 'h' for help on key bindings
```

Load m1[i][k] and m2[k][j] into memory and multiply

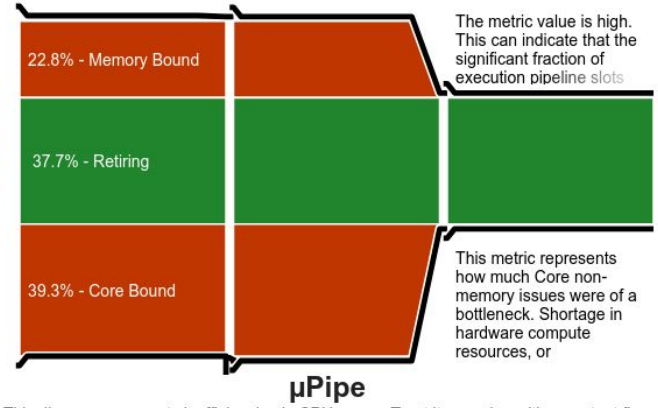
Add result into m3[i][j]

Loading m2 matrix elements in column major order is causing backend stalls.

Top-Down Analysis with Intel VTune Profiler

Elapsed Time : 30.547s

Clockticks:	134,784,000,000
Instructions Retired:	275,184,000,000
CPI Rate:	0.490
Retiring:	37.7%
Front-End Bound:	0.3%
Bad Speculation:	0.0%
Back-End Bound:	62.1%
Memory Bound:	22.8%
L1 Bound:	0.0%
L2 Bound:	2.6%
L3 Bound:	12.2%
Contested Accesses:	0.0%
Data Sharing:	0.0%
L3 Latency:	100.0%
SQ Full:	0.0%
DRAM Bound:	0.0%
Store Bound:	0.0%
Core Bound:	39.3%
Divider:	0.0%
Port Utilization:	24.8%
Cycles of 0 Ports Utilized:	6.2%
Cycles of 1 Port Utilized:	6.9%
Cycles of 2 Ports Utilized:	10.0%
Cycles of 3+ Ports Utilized:	20.0%
Vector Capacity Usage (FPU):	0.0%
Average CPU Frequency:	4.4 GHz
Total Thread Count:	2
Paused Time:	0s



As shown by the red arrows, the loop is being performed in column major order, which in C/C++ is not optimal, because the memory layout is row major. Therefore, we need to perform a loop inversion for the indices j and k to improve performance.

Source	Clockticks	Instructions Retired	CPI Rate	Locators: Retiring	Front-End Bound	Bad Speculation	Back-End Bound: Memory Bound: L1 Bound	L2 Bound	L3 Bound / L3 Latency	Core Bound
`int **mmult(int rows, int cols, int **m1, int **m2, int **m3) {`										
`int i, j, k;`										
`for (i = 0; i < rows; i++) {`										
`for (j = 0; j < cols; j++) {`	0.0%	0.0%	0.000	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
`m3[i][j] = 0;`	0.0%	0.0%	0.000	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
`for (k = 0; k < cols; k++)`	35.8%	42.9%	0.414	16.1%	0.0%	0.0%	0.0%	0.0%	0.0%	13.5%
`m3[i][j] += m1[i][k] * m2[k][j];`	64.1%	57.0%	0.558	22.1%	0.0%	0.6%	0.0%	2.5%	100.0%	24.6%
`}`										
`}`										
`return(m3);`										
`}`										

Loop inversion solves the problem

```
bash ~ $ perf stat -M Retiring,Bad_Speculation,Frontend_Bound,Backend_Bound a.out
1431831040 368052224 -168294912 -692581888

 Performance counter stats for 'a.out':

        297649292      IDQ_UOPS_NOT_DELIVERED.CORE #     0.00 Frontend_Bound
                                                   #     0.03 Backend_Bound          (50.00%)
        212555840      INT_MISC.RECOVERY_CYCLES                                      (50.00%)
      71499685017      CPU_CLK_UNHALTED.THREAD                                       (50.00%)
     276063180566      UOPS_ISSUED.ANY                                               (50.00%)
      16081241308 ns   duration_time
        212615678      INT_MISC.RECOVERY_CYCLES  #     0.01 Bad_Speculation          (50.00%)
      71533499469      CPU_CLK_UNHALTED.THREAD   #     0.96 Retiring                 (50.00%)
     275204536376      UOPS_RETIRED.RETIRE_SLOTS                                     (50.00%)
     276178541892      UOPS_ISSUED.ANY                                               (50.00%)
      16081241308 ns   duration_time

     16.081241308 seconds time elapsed

     16.061082000 seconds user
      0.009992000 seconds sys


bash ~ $ _
```

Now we are no longer bound by the backend. The speedup obtained was ≈2x with this change. Can we improve this result?
We can parallelize the code with OpenMP, for example.

Parallel code with OpenMP gains more performance

```
bash ~ $ perf stat -M Retiring,Bad_Speculation,Frontend_Bound,Backend_Bound env OMP_NUM_THREADS=8 a.out
1431831040 368052224 -168294912 -692581888

 Performance counter stats for 'env OMP_NUM_THREADS=8 a.out':

        1164303702      IDQ_UOPS_NOT_DELIVERED.CORE #     0.00 Frontend_Bound
                                                    #     0.03 Backend_Bound            (49.99%)
         204914876      INT_MISC.RECOVERY_CYCLES                                        (49.99%)
       71650959743      CPU_CLK_UNHALTED.THREAD                                         (49.99%)
      275645117900      UOPS_ISSUED.ANY                                                 (49.99%)
        2277867268 ns   duration_time
         205160954      INT_MISC.RECOVERY_CYCLES    #     0.01 Bad_Speculation          (50.07%)
       71630170542      CPU_CLK_UNHALTED.THREAD     #     0.96 Retiring                 (50.07%)
      275250371320      UOPS_RETIRED.RETIRE_SLOTS                                       (50.07%)
      276156990337      UOPS_ISSUED.ANY                                                 (50.07%)
        2277867268 ns   duration_time

       2.277867268 seconds time elapsed

      18.073005000 seconds user
       0.009998000 seconds sys


bash ~ $ _
```

The percentage of time spent retiring is too high. This is also indicative of a problem. Let's look again at the annotated source.

Annotated Source with perf annotate

```
Samples: 35K of event 'cycles', 1000 Hz, Event count (approx.): 142392966330
mmult._omp_fn.0  /home/amadio/a.out [Percent: local period]
Percent │       mov     %rax,%rcx
        │     for (k = 0; k < cols; k++)
        │       lea     (%rsi,%rbp,1),%r11
        │       nop
        │     for (j = 0; j < cols; j++)
        │     m3[i][j] += m1[i][k] * m2[k][j];
   0.18 │ b0:   mov     (%r10),%rdi
        │       xor     %eax,%eax
        │       nop
  24.57 │ b8:   mov     (%rsi),%edx
  11.31 │       imul    (%rdi,%rax,4),%edx
  63.80 │       add     %edx,(%rcx,%rax,4)
        │     for (j = 0; j < cols; j++)
        │       mov     %rax,%rdx
        │       add     $0x1,%rax
        │       cmp     %rdx,%r14
        │     ↑ jne     b8
        │     for (k = 0; k < cols; k++)
   0.09 │       add     $0x4,%rsi
        │       add     $0x8,%r10
        │       cmp     %rsi,%r11
Press 'h' for help on key bindings
```

The loop is still using scalar instructions.
We can further improve performance with vectorization.

Vectorization significantly improves performance

```
bash ~ $ # Baseline
bash ~ $ gcc -w -O2 -g matrix.c && time a.out # Using -w to avoid warning about unused pragma
1431831040 368052224 -168294912 -692581888
16.02
bash ~ $ # Parallel code with OpenMP
bash ~ $ gcc -Wall -fopenmp -O2 -g matrix.c && time a.out
1431831040 368052224 -168294912 -692581888
2.26
bash ~ $ # Parallel code with OpenMP and vectorization using AVX2
bash ~ $ gcc -Wall -fopenmp -O2 -ftree-vectorize -mavx2 -g matrix.c && time a.out
1431831040 368052224 -168294912 -692581888
0.54
bash ~ $
```

We've improved performance from ~30s down to 0.54s, not bad!
That's a speedup of about 56.3x.

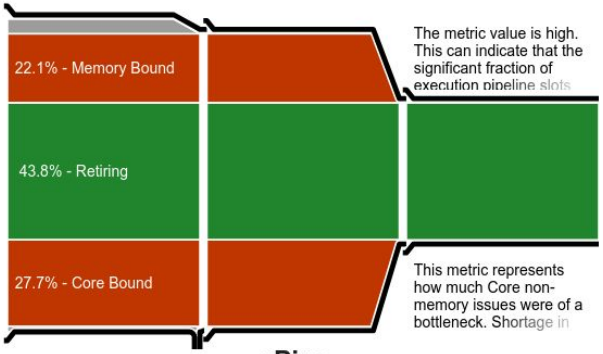
Comparison between initial and final versions

```
bash ~ $ diff -u matrix.orig.c matrix.c
--- matrix.orig.c       2022-08-15 15:12:15.457813585 +0200
+++ matrix.c    2022-08-15 15:50:32.247841744 +0200
@@ -28,14 +28,15 @@
     free(m);
 }

-int **mmult(int rows, int cols, int **m1, int **m2, int **m3) {
+int **mmult(int rows, int cols, int ** restrict m1, int ** restrict m2, int ** restrict m3) {
     int i, j, k;
+#pragma omp parallel for
     for (i = 0; i < rows; i++) {
-        for (j = 0; j < cols; j++) {
+        for (j = 0; j < cols; j++)
             m3[i][j] = 0;
-            for (k = 0; k < cols; k++)
+        for (k = 0; k < cols; k++)
+            for (j = 0; j < cols; j++)
                 m3[i][j] += m1[i][k] * m2[k][j];
-        }
     }
     return(m3);
 }
bash ~ $ _
```

Final performance summary in VTune (10x runtime)

Elapsed Time : 5.700s

Clockticks:	330,449,056,717	
Instructions Retired:	267,867,099,936	
CPI Rate:	1.234	
MUX Reliability:	0.780	
Retiring:	43.8%	of Pipeline Slots
Light Operations:	33.7%	of Pipeline Slots
Heavy Operations:	10.0%	of Pipeline Slots
Front-End Bound:	4.7%	of Pipeline Slots
Bad Speculation:	1.8%	of Pipeline Slots
Back-End Bound:	49.8%	of Pipeline Slots
Memory Bound:	22.1%	of Pipeline Slots
L1 Bound:	13.0%	of Clockticks
DTLB Overhead:	100.0%	of Clockticks
Loads Blocked by Store Forwarding:	0.0%	of Clockticks
Lock Latency:	0.0%	of Clockticks
Split Loads:	5.3%	of Clockticks
4K Aliasing:	0.9%	of Clockticks
FB Full:	0.1%	of Clockticks
L2 Bound:	5.8%	of Clockticks
L3 Bound:	6.6%	of Clockticks
DRAM Bound:	0.1%	of Clockticks
Store Bound:	0.1%	of Clockticks
Core Bound:	27.7%	of Pipeline Slots
Divider:	0.0%	of Clockticks
Port Utilization:	32.0%	of Clockticks
Cycles of 0 Ports Utilized:	25.8%	of Clockticks
Serializing Operations:	5.2%	of Clockticks
Mixing Vectors:	100.0%	of Clockticks
Cycles of 1 Port Utilized:	12.6%	of Clockticks
Cycles of 2 Ports Utilized:	14.3%	of Clockticks
Cycles of 3+ Ports Utilized:	47.7%	of Clockticks
Vector Capacity Usage (FPU):	6.2%	
Average CPU Frequency:	3.7 GHz	
Total Thread Count:	N/A*	
Paused Time:	0s	

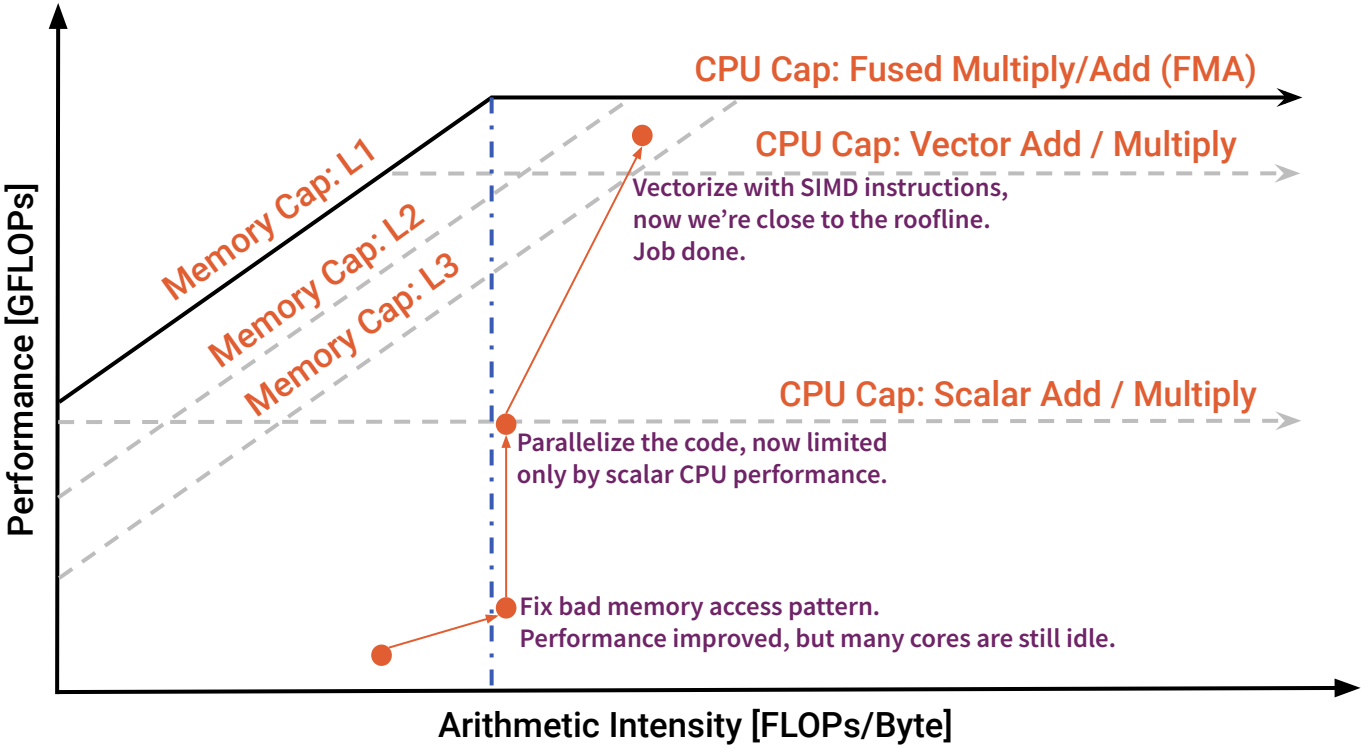


µPipe

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

Category	Client/Desktop Application	Server/Database Distributed Application	High Performance Computing (HPC) Application
Retiring	20 – 50%	10 – 30%	30 – 70%
Back-End Bound	20 – 40%	20 – 60%	20 – 40%
Front-End Bound	5 – 10%	10 – 25%	5 – 10%
Bad Speculation	5 – 10%	5 – 10%	1 – 5%

Matrix Multiplication Roofline Performance

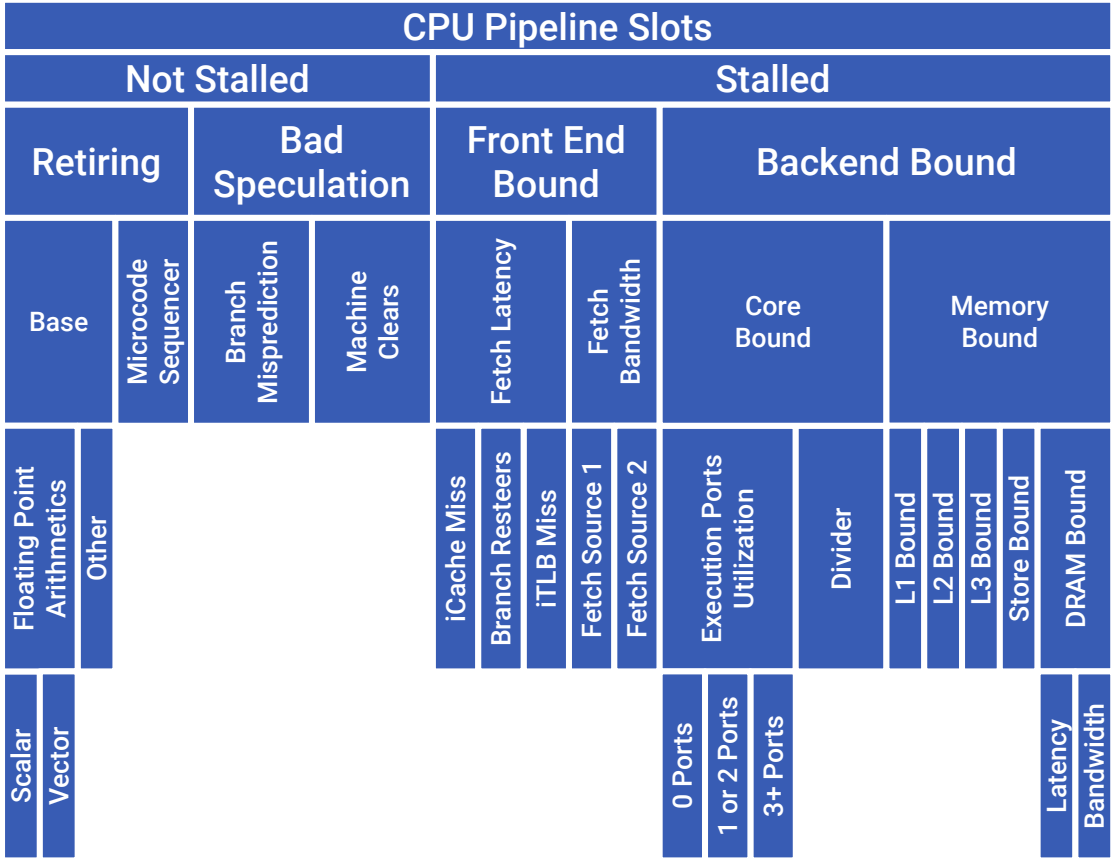


https://www.intel.com/content/www/us/en/developer/articles/guide/intel-advisor-roofline.html

Low Level Performance Optimization Guidelines

Top-Down Microarchitecture Analysis

- Bad Speculation
  - Unpredictable branches
  - Virtual Inheritance
- Front End Bound
  - Code Layout and Bloat
  - Loops with large body
- Core Bound
  - Loops with short body
  - Arithmetics/Data Dependencies
  - Divisions and Special Functions
- Memory Bound
  - Cache Misses
  - True and False Sharing
  - Bad Memory Access Patterns



Ahmad Yasin, "A Top-Down method for performance analysis and counters architecture," *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Monterey, CA, 2014, pp. 35-44, doi: 10.1109/ISPASS.2014.6844459

Bad Speculation Performance Tuning

Main Causes of Bad Speculation

- Sources of unpredictable branches
  - Even distribution of choices
  - Loops over virtual objects
  - Stochastic processes
- Conditional code in loops
- Long conditional expressions
  - Latency until branch can be taken
  - Stronger penalty if predictor is wrong
- Function calls are also branches!
  - Worse if they are in another library

Optimization Techniques

- Branchless code
  - Replace branches with arithmetics
  - Replace branches with predication
  - Replace branches with table lookup
- Annotate likely/unlikely branches
  - Improve branch prediction statistics
  - Helps with code layout optimization
- Ordering of branch conditionals
  - Most/Least likely first
  - Shortcircuit to decision early
- Loop splitting, statically linking

Example of Bad Speculation

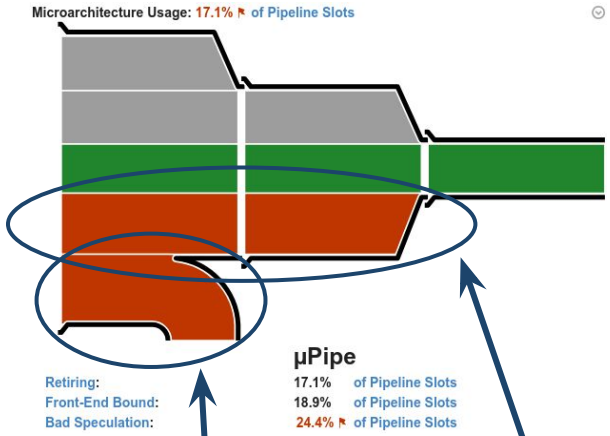
Be careful with what you assume the compiler can optimize for you.

```
class vector {
    public:
    double operator[](int i) {
        switch(i) {
            case 0: return x;
            case 1: return y;
            case 2: return z;
        }
    }

    double operator()(int i) { return (&x)[i]; }

    private:
    double x, y, z;
};
```

```
vector::operator[](int):
  cmp esi, 1
  je .L2
  cmp esi, 2
  jne .L6
  vmovsd xmm0, QWORD PTR [rdi+16]
  ret
.L2:
  vmovsd xmm0, QWORD PTR [rdi+8]
  ret
.L6:
  vmovsd xmm0, QWORD PTR [rdi]
  ret
vector::operator()(int):
  movsx rsi, esi
  vmovsd xmm0, QWORD PTR [rdi+rsi*8]
  ret
```

Example of Bad Speculation

Be careful with what you assume the compiler can optimize for you.



Source Line ▲	Source	Clockticks	Instructions Retired	CPI Rate	Locators					
					Retiring	Front-End Bound	Bad Speculation	Back-End Bound		
								Memory Bound	Core Bound	
									Divider	Port Utilization
40										
41	inline double Hep3Vector::operator () (int i) const {									
42	switch(i) {	20,232,000,000	18,900,000,000	1.070	16.7%	18.9%	23.3%	18.5%	0.0%	20.0%
43	case X:									
44	return x();									
45	case Y:									
46	return y();	72,000,000	36,000,000	2.000	0.4%	0.0%	0.0%	0.0%	0.0%	0.0%
47	case Z:									
48	return z();	144,000,000	0		0.0%	0.0%	1.5%	0.0%	0.0%	1.3%
49	}									
50	return 0.;									
51	}									

Effective Ordering of Conditionals

```
    Reorder condition in G4CrossSectionDataStore::ComputeCrossSection()

diff --git a/source/processes/hadronic/cross_sections/src/G4CrossSectionDataStore.cc b/source/processes/hadronic/cross_sections/src/G4CrossSectionData
Store.cc
index d81d3239ba..06e446f192 100644
--- a/source/processes/hadronic/cross_sections/src/G4CrossSectionDataStore.cc
+++ b/source/processes/hadronic/cross_sections/src/G4CrossSectionDataStore.cc
@@ -271,10 +271,10 @@ G4double
 G4CrossSectionDataStore::ComputeCrossSection(const G4DynamicParticle* part,
                                              const G4Material* mat)
 {
-  if(mat == currentMaterial && part->GetDefinition() == matParticle
-     && part->GetKineticEnergy() == matKinEnergy) {
+  if(part->GetKineticEnergy() == matKinEnergy && mat == currentMaterial &&
+     part->GetDefinition() == matParticle)
     return matCrossSection;
-  }
+
   currentMaterial = mat;
   matParticle = part->GetDefinition();
   matKinEnergy = part->GetKineticEnergy();

commit 3a2c4714fddc62e6ea31b6c5f074f60461ac05f4
Author: Guilherme Amadio <amadio@cern.ch>

    Update History file for G4CrossSectionDataStore

diff --git a/source/processes/hadronic/cross_sections/History b/source/processes/hadronic/cross_sections/History
lines 3391-3418/3673 93%
```

When a branching condition has several terms, order it from the most discriminant term to the least. Here, the energy is different much more frequently than material or particle type, so this order leads to more early decisions and better performance.

Probabilities:

mat == currentMaterial ⇒ 98.9%
matParticle == part->GetDefinition ⇒ 71.6%
matKinEnergy == part->GetKineticEnergy() ⇒ 32.1%

Note that these are independent. They are equal together only about 6.9% of the time.

Front-End Performance Tuning

Sources of Front-End Performance Issues

- Front-End Latency
  - Function inlining
  - Code bloat, duplication
    - More libraries ⇒ more pages to load
  - Frequent function calls
    - More instructions, call overhead
- Front-End Bandwidth
  - Loops with large body
  - Scalar-only arithmetics
  - Overloaded microop cache
  - Instructions need to be re-decoded

Optimization Techniques

- Function inlining
- Basic block reordering
- Basic block placement (alignment)
- Reduce code size, duplication
- Enable link-time optimizations
- Use profile-guided optimization
  - Code layout matching control flow
- Optimize iTLB usage (huge pages)
- SIMD Vectorization (less instructions)

SIMD Programming Models

- Auto-vectorization
- OpenMP 4.1
- Compiler Pragmas
- SIMD Library
- Compiler Intrinsics
- Assembly

```
float a[N], b[N], c[N];

for (int i = 0; i < N; i++)
  a[i] = b[i] * c[i];
```

```
float a[N], b[N], c[N];

#pragma omp simd
#pragma ivdep
for (int i = 0; i < N; i++)
  a[i] = b[i] * c[i];
```

```
#include <Vc/Vc>
Vc::SimdArray<float, N> a, b, c;

a = b * c;
```

```
#include <x86intrin.h>
__m256 a, b, c;

a = _mm256_mul_ps(b, c);
```

```
asm volatile("vmulps %ymm1, %ymm0");
```

Vectorization of Linear Algebra

Many computer simulation codes make extensive use of points, vectors, and affine coordinate transformations. Calculations using these objects can be sped up by using internal and external vectorization. Simple arithmetics (+ – × ÷) can be auto-vectorized by the compiler. Other operations, such as vector cross products and rotations are more complicated, but can still be vectorized manually.

Example: vector cross product (source)

```
void cross(const double * __restrict__ a, const double * __restrict__ b, double *result)
{
    result[0] = a[1]*b[2] - a[2]*b[1];
    result[1] = a[2]*b[0] - a[0]*b[2];
    result[2] = a[0]*b[1] - a[1]*b[0];
    return;
}

void cross_avx2(const double * __restrict__ a, const double * __restrict__ b, double *result)
{
    __m256d a012 = _mm256_load_pd(a);
    __m256d b012 = _mm256_load_pd(b);
    __m256d a201 = _mm256_permute4x64_pd(a012, _MM_SHUFFLE(3,1,0,2));
    __m256d b201 = _mm256_permute4x64_pd(b012, _MM_SHUFFLE(3,1,0,2));
    __m256d  tmp = _mm256_fmsub_pd(b012, a201, _mm256_mul_pd(a012, b201));
    tmp = _mm256_permute4x64_pd(tmp, _MM_SHUFFLE(3,1,0,2));
    tmp = _mm256_blend_pd(_mm256_setzero_pd(), tmp, 0x7); // put zero on 4th position
    _mm256_store_pd(result, tmp);
    return;
}
```

Vectorization of Linear Algebra

Many computer simulation codes make extensive use of points, vectors, and affine coordinate transformations. Calculations using these objects can be sped up by using internal and external vectorization. Simple arithmetics (+ – × ÷) can be auto-vectorized by the compiler. Other operations, such as vector cross products and rotations are more complicated, but can still be vectorized manually.

Example: vector cross product (assembly)

```
cross(double const*, double const*, double*):
 1 vmovsd 0x10(%rdi),%xmm0
 2 vmulsd 0x8(%rsi),%xmm0,%xmm0
 3 vmovsd 0x8(%rdi),%xmm1
 4 vfmsub231sd 0x10(%rsi),%xmm1,%xmm0
 5 vmovsd %xmm0,(%rdx)
 6 vmovsd (%rdi),%xmm0
 7 vmulsd 0x10(%rsi),%xmm0,%xmm0
 8 vmovsd 0x10(%rdi),%xmm2
 9 vfmsub231sd (%rsi),%xmm2,%xmm0
10 vmovsd %xmm0,0x8(%rdx)
11 vmovsd 0x8(%rdi),%xmm0
12 vmulsd (%rsi),%xmm0,%xmm0
13 vmovsd (%rdi),%xmm3
15 vfmsub231sd 0x8(%rsi),%xmm3,%xmm0
16 vmovsd %xmm0,0x10(%rdx)
17 retq
```

```
cross_avx2(double const*, double const*, double*):
  1 vmovapd (%rdi),%ymm2
  2 vmovapd (%rsi),%ymm0
  3 vpermpd $0xd2,%ymm2,%ymm1
  4 vpermpd $0xd2,%ymm0,%ymm3
  5 vmulpd %ymm3,%ymm2,%ymm2
  6 vfmsub132pd %ymm1,%ymm2,%ymm0
  7 vxorpd %xmm1,%xmm1,%xmm1
  8 vpermpd $0xd2,%ymm0,%ymm0
  9 vblendpd $0x7,%ymm0,%ymm1,%ymm0
 10 vmovapd %ymm0,(%rdx)
 11 retq
```

Advantages with AVX2: less memory moves, smaller number of instructions (therefore smaller cost to inline), 2.5x faster, and in this form it is transparent to the caller (same interface as generic code). However, code is more complex, needs to care about memory alignment.

Vectorization of Linear Algebra

Many computer simulation codes make extensive use of points, vectors, and affine coordinate transformations. Calculations using these objects can be sped up by using internal and external vectorization. Simple arithmetics (+ – × ÷) can be auto-vectorized by the compiler. Other operations, such as vector cross products and rotations are more complicated, but can still be vectorized manually.

Example: vector cross product (assembly)

```
cross(double const*, double const*, double*):
 1 vmovsd 0x10(%rdi),%xmm0
 2 vmulsd 0x8(%rsi),%xmm0,%xmm0
 3 vmovsd 0x8(%rdi),%xmm1
 4 vfmsub231sd 0x10(%rsi),%xmm1,%xmm0
 5 vmovsd %xmm0,(%rdx)
 6 vmovsd (%rdi),%xmm0
 7 vmulsd 0x10(%rsi),%xmm0,%xmm0
 8 vmovsd 0x10(%rdi),%xmm2
 9 vfmsub231sd (%rsi),%xmm2,%xmm0
10 vmovsd %xmm0,0x8(%rdx)
11 vmovsd 0x8(%rdi),%xmm0
12 vmulsd (%rsi),%xmm0,%xmm0
13 vmovsd (%rdi),%xmm3
15 vfmsub231sd 0x8(%rsi),%xmm3,%xmm0
16 vmovsd %xmm0,0x10(%rdx)
17 retq
```

```
cross_avx2(double const*, double const*, double*):
  1 vmovapd (%rdi),%ymm2
  2 vmovapd (%rsi),%ymm0
  3 vpermpd $0xd2,%ymm2,%ymm1
  4 vpermpd $0xd2,%ymm0,%ymm3
  5 vmulpd %ymm3,%ymm2,%ymm2
  6 vfmsub132pd %ymm1,%ymm2,%ymm0
  7 vxorpd %xmm1,%xmm1,%xmm1
  8 vpermpd $0xd2,%ymm0,%ymm0
  9 vblendpd $0x7,%ymm0,%ymm1,%ymm0
 10 vmovapd %ymm0,(%rdx)
 11 retq
```

Advantages with AVX2: less memory moves, smaller number of instructions (therefore smaller cost to inline), 2.5x faster, and in this form it is transparent to the caller (same interface as generic code). However, code is more complex, needs to care about memory alignment.

Core Bound Performance Tuning

Sources of Execution Bottlenecks

- Function call overhead
  - Short, frequently called functions
- Virtual inheritance, prevents inlining
- Poor port utilization
  - Chains of load/store instructions
- Arithmetics with data dependencies
  - True dependency (read-after-write)
  - Anti-dependency (write-after-read)
  - Output dependency (write-after-write)
- Divisions and square roots

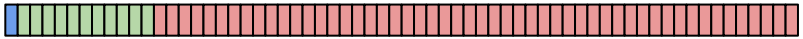
Optimization Techniques

- Function inlining
- Loop unrolling
- Replace recursion with iteration
- Avoid data access indirections
- Regular data members vs pointers
- Split long loops with dependencies
- Use the parenthesis in arithmetics
- Factor out common expressions
- Hide latency from divisions

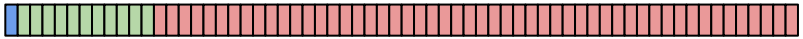
IEEE 754 Floating-Point Representation

- Floating point numbers are represented with a base β and a precision p
- For β = 2 and p = 4, the number ¼ = 0.25 can be written exactly as 1.000×2^{-2}
- Not all numbers can be represented
- For example, 0.04 is approximated

```
$ root -l
root [0] float x = 0.04;
root [1] printf("%.28f\n", x);
0.0399999991059303283691406250
root [2] *reinterpret_cast<int*>(&x)
(int) 1025758986
root [3] .q
```

- With base β = 2 and precision p = 24:
  $0.04 \cong 1.01000111101011100001010_2 \times 2^{-5} \cong 0.0399999991059303283691406250$
- Float:  sign bit, 7 bit exponent, 23 bit mantissa
- Double:  sign bit, 11 bit exponent, 52 bit mantissa
- Note: mantissa is stored as $1.NNNNN\ldots N_2$ such that effectively we have 24 and 53 bit mantissas

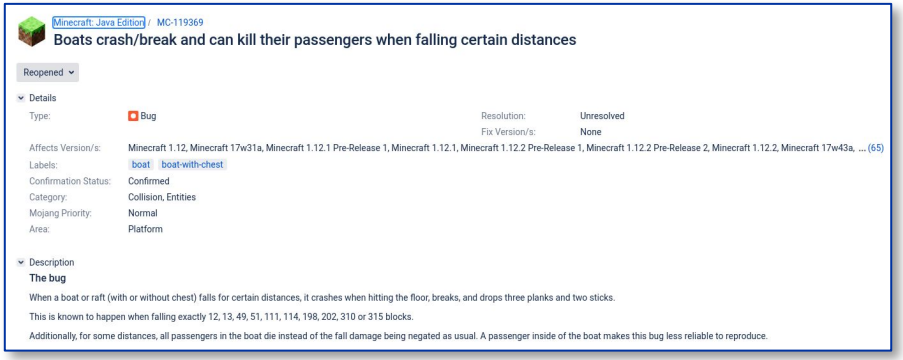
IEEE 754 Floating-Point Representation

- Floating point numbers are represented with a base β and a precision p
- For β = 2 and p = 4, the number ¼ = 0.25 can be written exactly as 1.000×2^{-2}
- Not all numbers can be represented
- For example, 0.04 is approximated
- With base β = 2 and precision p = 24: $0.04 \cong 1.01000111101011100001010_2 \times 2^{-5} \cong 0.03999999910593032836914062 5$
- Float:  sign bit, 7 bit exponent, 23 bit mantissa
- Double:  sign bit, 11 bit exponent, 52 bit mantissa
- Note: mantissa is stored as $1.NNNNN\ldots N_2$ such that effectively we have 24 and 53 bit mantissas

```
$ python
Python 3.12.3 (main, Apr 10 2024, 10:27:02) on linux
>>> bin(1025758986)
'0b111101001000111101011100001010'
>>> 2 * 0b101000111101011100001010 / 2**24 * 2**-5
0.03999999910593033
```

Often Source of Amusing Bugs

- In Minecraft boats normally protect passengers from fall damage
  - But not when falling from some heights
- In Minecraft, g = 0.04 blocks / tick²
  - Well, almost!
- $H_n = (1 + 2 + 3 + \ldots) \times 0.04 = 0.04\, n(n+1)/2$
- So when k(k+1) is a multiple of 1/0.04=25, something interesting happens
  - **status = IN_AIR → status = ON_LAND**
  - Code that checks fall damage is tricked, fails to update status to **onGround = true**
- Deciphered by Matt Parker (linked video)



Minecraft: Java Edition / MC-119369

Boats crash/break and can kill their passengers when falling certain distances

Reopened

Details

Type:	Bug	Resolution:	Unresolved
		Fix Version/s:	None
Affects Version/s:	Minecraft 1.12, Minecraft 17w31a, Minecraft 1.12.1 Pre-Release 1, Minecraft 1.12.1, Minecraft 1.12.2 Pre-Release 1, Minecraft 1.12.2 Pre-Release 2, Minecraft 1.12.2, Minecraft 17w43a, ... (65)		
Labels:	boat boat-with-chest		
Confirmation Status:	Confirmed		
Category:	Collision, Entities		
Mojang Priority:	Normal		
Area:	Platform		

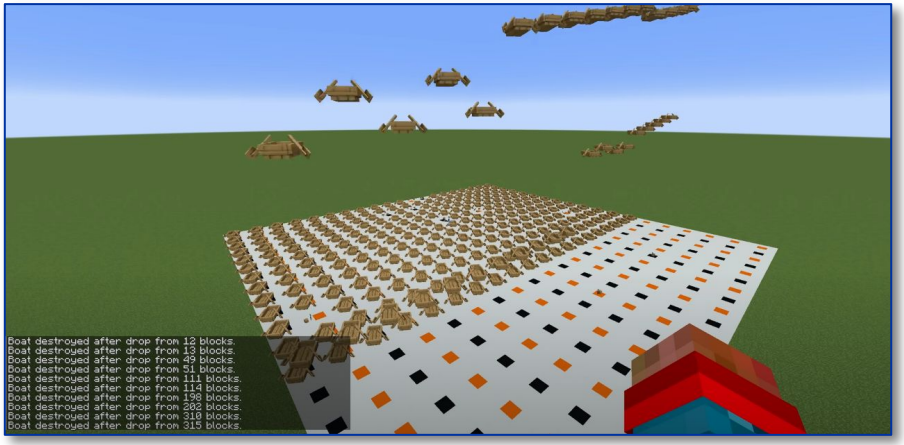
Description

The bug

When a boat or raft (with or without chest) falls for certain distances, it crashes when hitting the floor, breaks, and drops three planks and two sticks.

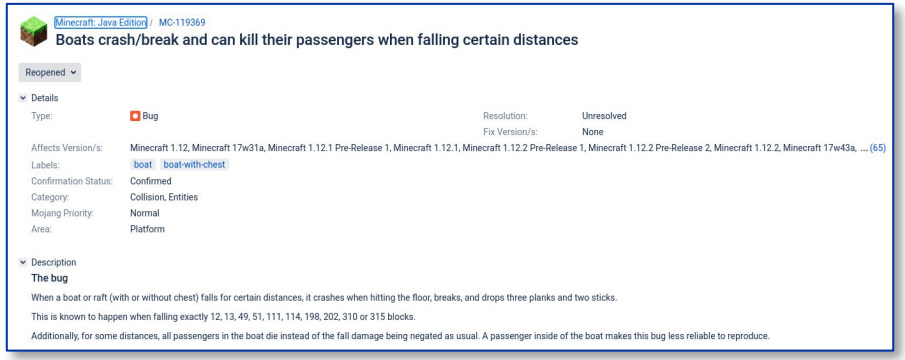
This is known to happen when falling exactly 12, 13, 49, 51, 111, 114, 198, 202, 310 or 315 blocks.

Additionally, for some distances, all passengers in the boat die instead of the fall damage being negated as usual. A passenger inside of the boat makes this bug less reliable to reproduce.



Often Source of Amusing Bugs

- In Minecraft boats normally protect passengers from fall damage
  - But not when falling from some heights
- In Minecraft, g = 0.04 blocks / tick²
  - Well, almost!
- $H_n = (1 + 2 + 3 + \ldots) \times 0.04 = 0.04\, n(n+1)/2$
- So when k(k+1) is a multiple of 1/0.04=25, something interesting happens
  - **status = IN_AIR → status = ON_LAND**
  - Code that checks fall damage is tricked, fails to update status to **onGround = true**
- Deciphered by Matt Parker (linked video)



Minecraft: Java Edition / MC-119369

Boats crash/break and can kill their passengers when falling certain distances

Reopened

Details

Type:	Bug	Resolution:	Unresolved
		Fix Version/s:	None
Affects Version/s:	Minecraft 1.12, Minecraft 17w31a, Minecraft 1.12.1 Pre-Release 1, Minecraft 1.12.1, Minecraft 1.12.2 Pre-Release 1, Minecraft 1.12.2 Pre-Release 2, Minecraft 1.12.2, Minecraft 17w43a, ... (65)		
Labels:	boat boat-with-chest		
Confirmation Status:	Confirmed		
Category:	Collision, Entities		
Mojang Priority:	Normal		
Area:	Platform		

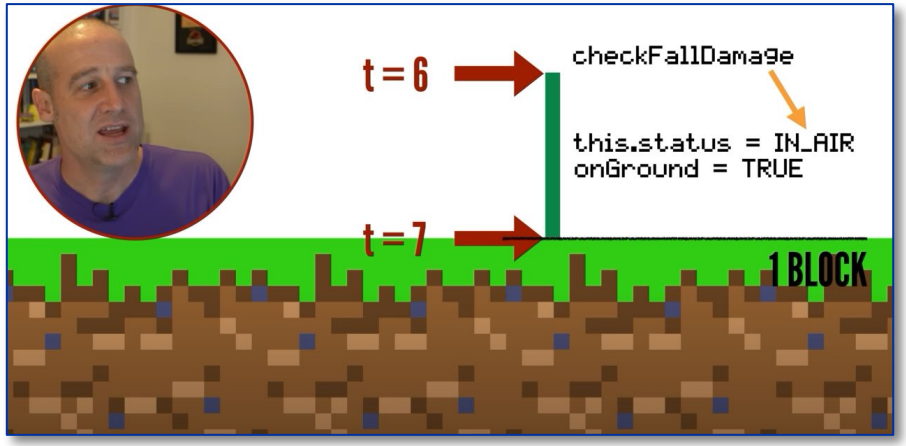
Description

The bug

When a boat or raft (with or without chest) falls for certain distances, it crashes when hitting the floor, breaks, and drops three planks and two sticks.

This is known to happen when falling exactly 12, 13, 49, 51, 111, 114, 198, 202, 310 or 315 blocks.

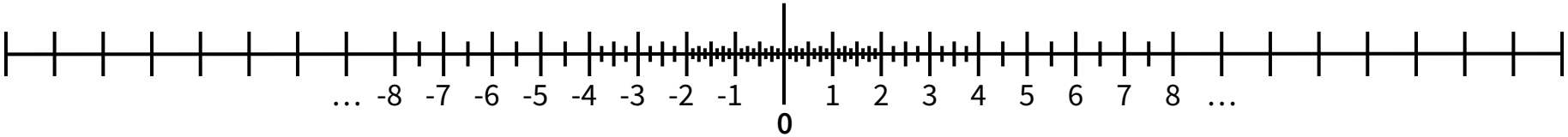
Additionally, for some distances, all passengers in the boat die instead of the fall damage being negated as usual. A passenger inside of the boat makes this bug less reliable to reproduce.



Floating-Point Numbers are Denser Towards Zero

One important aspect of floating-point numbers is that they are denser near the origin. Therefore, one needs to be careful when choosing units for physical quantities such that numerical calculations do not stray too far from 1.0. When numbers become too big, the rounding errors can become quite significant. On the right, we demonstrate that rounding when crossing a density boundary at a power of 2.

```
$ root -l
root [0] printf("%.16f\n", 0.1f);
0.1000000014901161
root [1] printf("%.16f\n", 0.1); /* double, not float */
0.1000000000000000
root [2] FLT_EPSILON
(float) 1.19209e-07f
root [3] printf("%.16f\n", 1.0f + 0.75f * FLT_EPSILON);
1.0000001192092896
root [4] printf("%.16f\n", 1.0f + 1.00f * FLT_EPSILON);
1.0000001192092896
root [5] printf("%.16f\n", 1.0f + 1.25f * FLT_EPSILON);
1.0000001192092896

root [6] printf("%.16f\n", 1.0f + 1.50f * FLT_EPSILON);
1.0000002384185791
root [7] printf("%.16f\n", 1.0f + 1.75f * FLT_EPSILON);
1.0000002384185791
root [8] printf("%.16f\n", 1.0f + 2.00f * FLT_EPSILON);
1.0000002384185791
root [9] printf("%.16f\n", 1.0f + 2.50f * FLT_EPSILON);
1.0000002384185791
```



Examples

0 01111111111 0000000000000000000000000000000000000000000000000000 ≙ 3FF0 0000 0000 0000 ≙ $+2^0 \times 1 = 1$

0 10000000000 0000000000000000000000000000000000000000000000000000 ≙ 4000 0000 0000 0000 ≙ $+2^1 \times 1 = 2$

1 10000000000 0000000000000000000000000000000000000000000000000000 ≙ C000 0000 0000 0000 ≙ $-2^1 \times 1 = -2$

0 00000000000 0000000000000000000000000000000000000000000000000000 ≙ 0000 0000 0000 0000 ≙ +0

1 00000000000 0000000000000000000000000000000000000000000000000000 ≙ 8000 0000 0000 0000 ≙ -0

0 11111111111 0000000000000000000000000000000000000000000000000000 ≙ 7FF0 0000 0000 0000 ≙ +∞ (positive infinity)

1 11111111111 0000000000000000000000000000000000000000000000000000 ≙ FFF0 0000 0000 0000 ≙ -∞ (negative infinity)

0 11111111111 0000000000000000000000000000000000000000000000000001 ≙ 7FF0 0000 0000 0001 ≙ NaN (sNaN on most processors)

0 11111111111 1000000000000000000000000000000000000000000000000001 ≙ 7FF8 0000 0000 0001 ≙ NaN (qNaN on most processors)

0 11111111111 1111111111111111111111111111111111111111111111111111 ≙ 7FFF FFFF FFFF FFFF ≙ NaN (alternative encoding)

0 01111111111 0000000000000000000000000000000000000000000000000001 ≙ 3FF0 0000 0000 0001

≙ $+2^0 \times (1 + 2^{-52}) \approx 1.0000000000000002$

Try it in *https://www.h-schmidt.net/FloatConverter/IEEE754.html*

G4PhysicsVector::Interpolation()

Source Line ▲	Source	Clockticks	Instructions Retired	CPI Rate	Locators				
					Retiring	Front-End Bound	Bad Speculation	Back-End Bound	
								Memory Bound	Core Bound
184	`// ----------`								
185									
186	`inline G4double G4PhysicsVector::Interpolation(const std::size_t idx,`								
187	`const G4double e) const`								
188	`{`								
189	`// perform the interpolation`								
190	`const G4double x1 = binVector[idx];`	0.2%	0.1%	1.011	3.8%	6.0%	3.4%	3.9%	3.8%
191	`const G4double dl = binVector[idx + 1] - x1;`								
192	`// note: all corner cases of the previous methods are covered and eventually`								
193	`// gives b=0/1 that results in y=y0\y_{N-1} if e<=x[0]/e>=x[N-1] or`								
194	`// y=y_i/y_{i+1} if e<x[i]/e>=x[i+1] due to small numerical errors`								
195	`const G4double b = std::max(0., std::min(1., (e - x1) / dl));`	0.0%	0.0%	0.199	0.3%	0.0%	0.0%	0.0%	0.0%
196	`G4double res;`								
197	`if(useSpline) // spline interpolation`	0.0%	0.0%	1.000	0.4%	0.0%		0.4%	
198	`{`								
199	`const G4double os = 0.166666666667; // 1./6.`								
200	`const G4double a = 1.0 - b;`	0.1%	0.0%	3.851	1.4%	0.0%	0.7%	2.3%	2.3%
201	`const G4double c0 = (a * a * a - a) * secDerivative[idx];`	0.1%	0.0%	4.105	0.5%	0.0%	0.9%	1.9%	1.1%
202	`const G4double c1 = (b * b * b - b) * secDerivative[idx + 1];`	0.1%	0.0%	5.371	1.3%	0.0%	2.0%	3.6%	3.8%
203	`res =`								
204	`a * dataVector[idx] + b * dataVector[idx + 1] + (c0 + c1) * dl * dl * os;`	0.6%	0.3%	1.240	9.7%	0.0%	3.7%	17.4%	23.9%
205	`}`								
206	`else // linear interpolation`								
207	`{`								
208	`const G4double y1 = dataVector[idx];`								
209	`const G4double y2 = dataVector[idx + 1];`								
210	`res = y1 + b * (y2 - y1);`	0.0%	0.0%	7.667	0.0%	1.5%	0.0%	0.1%	0.0%
211	`}`								
212	`return res;`								
213	`}`								

Arithmetics: Instruction Level Parallelism

```diff
+++ b/source/global/management/include/G4PhysicsVector.icc
@@ -129,10 +129,10 @@ inline G4double G4PhysicsVector::Interpolation(const std::size_t idx,
 {
   // perform the interpolation
   const G4double x1 = binVector[idx];
-  const G4double dl = binVector[idx + 1] - binVector[idx];
+  const G4double dl = binVector[idx + 1] - x1;

   const G4double y1 = dataVector[idx];
-  const G4double dy = dataVector[idx + 1] - dataVector[idx];
+  const G4double dy = dataVector[idx + 1] - y1;

   // note: all corner cases of the previous methods are covered and eventually
   //       gives b=0/1 that results in y=y0\y_{N-1} if e<=x[0]/e>=x[N-1] or
@@ -143,10 +143,9 @@ inline G4double G4PhysicsVector::Interpolation(const std::size_t idx,

   if(useSpline)  // spline interpolation
   {
-    const G4double a  = 1.0 - b;
-    const G4double c0 = (a * a * a - a) * secDerivative[idx];
-    const G4double c1 = (b * b * b - b) * secDerivative[idx + 1];
-    res += (c0 + c1) * dl * dl * (1.0/6.0);
+    const G4double c0 = (2.0 - b) * secDerivative[idx];
+    const G4double c1 = (1.0 + b) * secDerivative[idx + 1];
+    res += (b * (b - 1.0)) * (c0 + c1) * (dl * dl * (1.0/6.0));
   }

   return res;
lines 3626-3654/3673 100%
```

Refactoring terms saves some multiplications, but note also the parenthesis. Floating point arithmetics is not associative. Parenthesizing ensures that each of the independent multiplications can be performed in parallel. This alone reduces estimated execution from 60 to 51 cycles (llvm-mca).

Without parenthesis

```
vmovsd xmm5, QWORD PTR .LC0[rip]
vmovsd xmm4, QWORD PTR .LC2[rip]
vsubsd xmm2, xmm1, xmm5
vsubsd xmm4, xmm4, xmm1
vmulsd xmm2, xmm2, xmm1
vaddsd xmm1, xmm1, xmm5
vmulsd xmm1, xmm1, QWORD PTR [rcx+8+rdi]
vfmadd231sd xmm1, xmm4, QWORD PTR [rcx+rax*8]
vmulsd xmm1, xmm2, xmm1
vmulsd xmm1, xmm1, xmm3
vmulsd xmm1, xmm1, xmm3
vfmadd231sd xmm0, xmm1, QWORD PTR .LC1[rip]
```

same register ⇒ sequential

With parenthesis

```
vmovsd xmm5, QWORD PTR .LC0[rip]
vmovsd xmm4, QWORD PTR .LC2[rip]
vsubsd xmm3, xmm1, xmm5
vsubsd xmm4, xmm4, xmm1
vmulsd xmm2, xmm2, xmm2
vmulsd xmm3, xmm3, xmm1
vaddsd xmm1, xmm1, xmm5
vmulsd xmm1, xmm1, QWORD PTR [rcx+8+rdi]
vmulsd xmm2, xmm2, QWORD PTR .LC1[rip]
vfmadd132sd xmm4, xmm1, QWORD PTR [rcx+rax*8]
vmulsd xmm3, xmm3, xmm4
vfmadd231sd xmm0, xmm3, xmm2
```

different registers ⇒ parallel

Arithmetics: Instruction Level Parallelism

```
                    0123456789          0123456789          0123456789
Index     0123456789          0123456789          0123456789

[0,0]     DeER .    .    .    .    .    .    .    .    .    .    .    .   movslq %edi, %rax
[0,1]     D=eER.    .    .    .    .    .    .    .    .    .    .    .   leaq   (,%rax,8), %rdi
[0,2]     D=eeeeeER .    .    .    .    .    .    .    .    .    .    .   vmovsd (%rsi,%rax,8), %xmm1
[0,3]     D==eeeeeER.    .    .    .    .    .    .    .    .    .    .   vmovsd 8(%rsi,%rdi), %xmm3
[0,4]     D=eeeeeE-R.    .    .    .    .    .    .    .    .    .    .   vmovsd (%rdx,%rax,8), %xmm2
[0,5]     D======eeeeER .    .    .    .    .    .    .    .    .    .   vsubsd %xmm1, %xmm3, %xmm3
[0,6]     .D=====eeeeE-R .    .    .    .    .    .    .    .    .    .   vsubsd %xmm1, %xmm0, %xmm1
[0,7]     .D=eeeeeE----R .    .    .    .    .    .    .    .    .    .   vmovsd 8(%rdx,%rdi), %xmm0
[0,8]     .D==========eeeeeeeeeeeeeER  .    .    .    .    .    .    .   vdivsd %xmm3, %xmm1, %xmm1
[0,9]     .D======eeeeE--------------R  .    .    .    .    .    .    .   vsubsd %xmm2, %xmm0, %xmm0
[0,10]    .D========================eeeeER   .    .    .    .    .    .   vfmadd132sd    %xmm1, %xmm2, %xmm0
[0,11]    .DeE---------------------------R   .    .    .    .    .    .   testb  %r8b, %r8b
[0,12]    . DeE--------------------------R   .    .    .    .    .    .   je     .L1
[0,13]    . D=eeeeeE---------------------R   .    .    .    .    .    .   vmovsd .LC0(%rip), %xmm5
[0,14]    . D=eeeeeE---------------------R   .    .    .    .    .    .   vmovsd .LC1(%rip), %xmm4
[0,15]    . D=======================eeeeER   .    .    .    .    .    .   vsubsd %xmm5, %xmm1, %xmm2
[0,16]    . D==========================eeeeER  .    .    .    .    .    .   vsubsd %xmm1, %xmm4, %xmm4
[0,17]    . D============================eeeeER    .    .    .    .    .   vmulsd %xmm1, %xmm2, %xmm2
[0,18]    .  D=======================eeeeE---R    .    .    .    .    .   vaddsd %xmm5, %xmm1, %xmm1
[0,19]    .  D=========================eeeeeeeeER .    .    .    .    .   vmulsd 8(%rcx,%rdi), %xmm1, %xmm1
[0,20]    .  D============================eeeeeeeeER  .    .    .    .   vfmadd231sd    (%rcx,%rax,8), %xmm4, %xmm1
[0,21]    .  D=====================================eeeeER   .    .    .   vmulsd %xmm1, %xmm2, %xmm1
[0,22]    .   D=========================================eeeeER    .    .   vmulsd %xmm3, %xmm1, %xmm1
[0,23]    .   D===============================================eeeeER.    .   vmulsd %xmm3, %xmm1, %xmm1
[0,24]    .   D=================================================eeeeeeeeER   vfmadd231sd    .LC2(%rip), %xmm1, %xmm0
[0,25]    .    DeeeeeeeE-----------------------------------------------R   retq
titanx ~ $ g++-11.2.0 -march=native -O3 -S test.cc -DV2=1 -o - | llvm-mca -iterations=1 -timeline 2>/dev/null | sed -n 97,125p
```

Legend

D : Instruction dispatched.

e : Instruction executing.

E : Instruction executed.

R : Instruction retired.

= : Instruction already dispatched, waiting to be executed.

- : Instruction executed, waiting to be retired.

Arithmetics: Instruction Level Parallelism

```
                    0123456789          0123456789          0
Index     0123456789          0123456789          0123456789

[0,0]     DeER .    .    .    .    .    .    .    .    .    .   movslq  %edi, %rax
[0,1]     D=eER.    .    .    .    .    .    .    .    .    .   leaq    (,%rax,8), %rdi
[0,2]     D=eeeeeER .    .    .    .    .    .    .    .    .   vmovsd  (%rsi,%rax,8), %xmm1
[0,3]     D==eeeeeER.    .    .    .    .    .    .    .    .   vmovsd  8(%rsi,%rdi), %xmm2
[0,4]     D=eeeeeE-R.    .    .    .    .    .    .    .    .   vmovsd  (%rdx,%rax,8), %xmm3
[0,5]     D======eeeeER .    .    .    .    .    .    .    .   vsubsd  %xmm1, %xmm2, %xmm2
[0,6]     .D=====eeeeE-R .    .    .    .    .    .    .    .   vsubsd  %xmm1, %xmm0, %xmm1
[0,7]     .D=eeeeeE----R .    .    .    .    .    .    .    .   vmovsd  8(%rdx,%rdi), %xmm0
[0,8]     .D==========eeeeeeeeeeeeeER  .    .    .    .    .   vdivsd  %xmm2, %xmm1, %xmm1
[0,9]     .D======eeeeE--------------R  .    .    .    .    .   vsubsd  %xmm3, %xmm0, %xmm0
[0,10]    .D=========================eeeeER   .    .    .    .   vfmadd132sd     %xmm1, %xmm3, %xmm0
[0,11]    .DeE---------------------------R   .    .    .    .   testb   %r8b, %r8b
[0,12]    . DeE--------------------------R   .    .    .    .   je      .L1
[0,13]    . D=eeeeeE---------------------R   .    .    .    .   vmovsd  .LC0(%rip), %xmm5
[0,14]    . D=eeeeeE---------------------R   .    .    .    .   vmovsd  .LC1(%rip), %xmm4
[0,15]    . D=======================eeeeER   .    .    .    .   vsubsd  %xmm5, %xmm1, %xmm3
[0,16]    . D==========================eeeeER  .    .    .    .   vsubsd  %xmm1, %xmm4, %xmm4
[0,17]    . D=========eeeeE---------------R  .    .    .    .   vmulsd  %xmm2, %xmm2, %xmm2
[0,18]    .  D===========================eeeeER    .    .    .   vmulsd  %xmm1, %xmm3, %xmm3
[0,19]    .  D=======================eeeeE---R    .    .    .   vaddsd  %xmm5, %xmm1, %xmm1
[0,20]    .  D========================eeeeeeeeER .    .    .   vmulsd  8(%rcx,%rdi), %xmm1, %xmm1
[0,21]    .  D=========eeeeeeeeE---------------R .    .    .   vmulsd  .LC2(%rip), %xmm2, %xmm2
[0,22]    .   D==========================eeeeeeeeER  .    .   vfmadd132sd     (%rcx,%rax,8), %xmm1, %xmm4
[0,23]    .   D=====================================eeeeER   .   vmulsd  %xmm4, %xmm3, %xmm3
[0,24]    .   D========================================eeeeER   vfmadd231sd     %xmm2, %xmm3, %xmm0
[0,25]    .    DeeeeeeeE----------------------------------R   retq
titanx ~ $ g++-11.2.0 -march=native -O3 -S test.cc -DV3=1 -o - | llvm-mca -iterations=1 -timeline 2>/dev/null | sed -n 97,125p_
```

Legend

D : Instruction dispatched.

e : Instruction executing.

E : Instruction executed.

R : Instruction retired.

= : Instruction already dispatched, waiting to be executed.

- : Instruction executed, waiting to be retired.

Top 20 classes in Geant4 Before Optimizations

Class / Source Function / Call Stack	CPU Time ▼	Instructions Retired	Microarchitecture Usage	
			Microarchitecture Usage	CPI Rate
CLHEP::Hep3Vector	9.7%	10.7%	37.3%	0.589
LArWheelCalculator_Impl::DistanceCalculatorSaggingOff	8.2%	11.3%	55.8%	0.473
G4PhysicsVector	5.4%	2.9%	21.3%	1.219
[Not part of any known object class]	5.2%	4.4%	33.5%	0.777
G4PolyconeSide	4.6%	7.1%	57.8%	0.431
G4VoxelNavigation	4.0%	1.6%	14.2%	1.661
MagField::AtlasFieldSvc	3.6%	3.7%	40.6%	0.633
G4SteppingManager	2.8%	3.0%	39.7%	0.611
LArWheelSolid	2.2%	1.7%	31.2%	0.860
LArWheelCalculator	2.2%	3.6%	51.4%	0.394
G4ProductionCutsTable	1.6%	0.7%	14.8%	1.555
CLHEP::MixMaxRng	1.6%	2.1%	50.0%	0.493
BFieldCache	1.5%	2.5%	60.3%	0.385
G4CrossSectionDataStore	1.4%	1.6%	39.7%	0.548
G4LogicalVolume	1.4%	1.1%	28.3%	0.893
G4Navigator	1.4%	1.4%	32.7%	0.649
G4Tubs	1.4%	1.2%	34.3%	0.751
G4UrbanMscModel	1.3%	0.8%	24.1%	1.023
G4VProcess	1.3%	0.7%	26.2%	1.159
G4VCSGfaceted	1.3%	1.4%	49.4%	0.607

Top 20 classes in Geant4 After Optimizations

Class / Source Function / Call Stack	CPU Time ▼	Instructions Retired	Microarchitecture Usage	
			Microarchitecture Usage	CPI Rate
CLHEP::Hep3Vector	9.6%	10.9%	40.4%	0.565
LArWheelCalculator_Impl::DistanceCalculatorSaggingOff	8.9%	11.8%	55.7%	0.485
G4PolyconeSide	5.2%	7.5%	55.0%	0.439
G4VoxelNavigation	4.6%	1.7%	14.7%	1.706
[Not part of any known object class]	4.3%	3.5%	31.0%	0.796
MagField::AtlasFieldSvc	3.9%	3.9%	40.7%	0.649
G4PhysicsVector	2.9%	2.0%	27.3%	0.944
G4SteppingManager	2.7%	3.1%	41.4%	0.554
LArWheelSolid	2.6%	1.8%	28.3%	0.892
LArWheelCalculator	2.4%	3.9%	51.1%	0.399
G4Navigator	1.8%	1.8%	35.3%	0.647
G4ProductionCutsTable	1.7%	0.7%	14.6%	1.596
BFieldCache	1.7%	2.7%	61.4%	0.398
G4UrbanMscModel	1.6%	0.9%	20.5%	1.115
G4VEmProcess	1.6%	1.1%	27.9%	0.883
G4LogicalVolume	1.5%	1.1%	29.5%	0.902
G4Tubs	1.5%	1.2%	33.9%	0.785
G4VCSGfaceted	1.5%	1.4%	46.7%	0.650
CLHEP::MixMaxRng	1.3%	1.6%	47.6%	0.527
G4AffineTransform	1.3%	1.3%	37.2%	0.645
G4CrossSectionDataStore	1.3%	1.4%	41.7%	0.580

Floating-Point Performance Tips

- Keep numbers small for better accuracy
- Be careful with direct comparisons
- If you can, use integer arithmetics instead
- Divisons are very slow, use them only when necessary
- Beware of unwanted conversions between single and double precision
- You can optimize for speed or accuracy, but rarely both at the same time
- Forget `-Ofast`, it will often break your code by assuming associativity
- Avoid `long double` at all costs, it uses the FP87 unit and is **very slow**
- Watch out for denormals and NaNs, they can significantly affect performance

Memory Bound Performance Tuning

Sources of Memory Issues

- Bad cache locality
  - Spatial locality
  - Temporal locality
- Bad cache coherence
  - Cache invalidation
  - True and false sharing
- Bad data structures
  - Padding
  - Low information density
- Low information density

Optimization Techniques

- Avoid data access indirections
  - Chains of loads rarely satisfy locality
- Group reads/writes to same struct
- Regular data members vs pointers
- Avoid contiguous per-thread data
- Try to pack holes in data structures
- Replace bools with bit flags
- Use smaller data types (int, float)
- Optimize dTLB usage (huge pages)
- Minimize number of heap allocations

Data Access Patterns: Avoid Indirections

```
Author: Guilherme Amadio <amadio@cern.ch>

    G4SteppingManager: reuse value of fCurrentVolume to find current region

diff --git a/source/tracking/src/G4SteppingManager.cc b/source/tracking/src/G4SteppingManager.cc
index 0e00aca0d3..1108ef957d 100644
--- a/source/tracking/src/G4SteppingManager.cc
+++ b/source/tracking/src/G4SteppingManager.cc
@@ -257,9 +257,10 @@ G4StepStatus G4SteppingManager::Stepping()
   {
     fUserSteppingAction->UserSteppingAction(fStep);
   }
-  G4UserSteppingAction* regionalAction = fStep->GetPreStepPoint()
-        ->GetPhysicalVolume()->GetLogicalVolume()
-        ->GetRegion()->GetRegionalSteppingAction();
+
+  G4UserSteppingAction* regionalAction =
+    fCurrentVolume->GetLogicalVolume()->GetRegion()->GetRegionalSteppingAction();
+
   if(regionalAction)
     regionalAction->UserSteppingAction(fStep);


commit 3c70a7e614d885364905bb5208f6cefbff94c2d9
Author: Guilherme Amadio <amadio@cern.ch>
```

Chained accessors via pointers require several memory accesses to retrieve a single piece of data, with similar cost to traversing a linked list. Here we can avoid 3 access indirections by reusing the value of fCurrentVolume.

Data Access Patterns: Avoid Indirections

```
  // Initialize G4StepPoint attributes.
  // To avoid the circular dependency between G4Track, G4Step
  // and G4StepPoint, G4Step has to manage the copy actions.
  fpPreStepPoint->SetPosition(fpTrack->GetPosition());
  fpPreStepPoint->SetGlobalTime(fpTrack->GetGlobalTime());
  fpPreStepPoint->SetLocalTime(fpTrack->GetLocalTime());
  fpPreStepPoint->SetProperTime(fpTrack->GetProperTime());
  fpPreStepPoint->SetMomentumDirection(fpTrack->GetMomentumDirection());
  fpPreStepPoint->SetKineticEnergy(fpTrack->GetKineticEnergy());
  fpPreStepPoint->SetTouchableHandle(fpTrack->GetTouchableHandle());
  fpPreStepPoint->SetMaterial(
    fpTrack->GetTouchable()->GetVolume()->GetLogicalVolume()->GetMaterial());
  fpPreStepPoint->SetMaterialCutsCouple(fpTrack->GetTouchable()
                                         ->GetVolume()
                                         ->GetLogicalVolume()
                                         ->GetMaterialCutsCouple());
  fpPreStepPoint->SetSensitiveDetector(fpTrack->GetTouchable()
                                        ->GetVolume()
                                        ->GetLogicalVolume()
                                        ->GetSensitiveDetector());
  fpPreStepPoint->SetPolarization(fpTrack->GetPolarization());
  fpPreStepPoint->SetSafety(0.);
  fpPreStepPoint->SetStepStatus(fUndefined);
  fpPreStepPoint->SetProcessDefinedStep(0);
  fpPreStepPoint->SetMass(fpTrack->GetDynamicParticle()->GetMass());
  fpPreStepPoint->SetCharge(fpTrack->GetDynamicParticle()->GetCharge());
  fpPreStepPoint->SetWeight(fpTrack->GetWeight());

  // Set Velocity
"track/include/G4Step.icc" 306 lines --54%--                                  166,1          59%
```

These actually return fDynamicParticle->Get...()!

Not good! Traverses pointer chains multiple times, and G4TouchableHandle is actually reference counted, so this has branches and is incrementing and decrementing counters multiple times too!

Data Access Patterns: Avoid Indirections

Source Line ▲	Source
166	// Initialize G4StepPoint attributes.
167	// To avoid the circular dependency between G4Track, G4Step
168	// and G4StepPoint, G4Step has to manage the copy actions.
169	fpPreStepPoint->SetPosition(fpTrack->GetPosition());
170	fpPreStepPoint->SetGlobalTime(fpTrack->GetGlobalTime());
171	fpPreStepPoint->SetLocalTime(fpTrack->GetLocalTime());
172	fpPreStepPoint->SetProperTime(fpTrack->GetProperTime());
173	fpPreStepPoint->SetMomentumDirection(fpTrack->GetMomentumDirection());
174	fpPreStepPoint->SetKineticEnergy(fpTrack->GetKineticEnergy());
175	fpPreStepPoint->SetTouchableHandle(fpTrack->GetTouchableHandle());
176	fpPreStepPoint->SetMaterial(
177	fpTrack->GetTouchable()->GetVolume()->GetLogicalVolume()->GetMaterial());
178	fpPreStepPoint->SetMaterialCutsCouple(fpTrack->GetTouchable()
179	->GetVolume()
180	->GetLogicalVolume()
181	->GetMaterialCutsCouple());
182	fpPreStepPoint->SetSensitiveDetector(fpTrack->GetTouchable()
183	->GetVolume()
184	->GetLogicalVolume()
185	->GetSensitiveDetector());
186	fpPreStepPoint->SetPolarization(fpTrack->GetPolarization());
187	fpPreStepPoint->SetSafety(0.);
188	fpPreStepPoint->SetStepStatus(fUndefined);
189	fpPreStepPoint->SetProcessDefinedStep(0);
190	fpPreStepPoint->SetMass(fpTrack->GetDynamicParticle()->GetMass());
191	fpPreStepPoint->SetCharge(fpTrack->GetDynamicParticle()->GetCharge());
192	fpPreStepPoint->SetWeight(fpTrack->GetWeight());
193	
194	// Set Velocity
195	// should be placed after SetMaterial for preStep point
196	fpPreStepPoint->SetVelocity(fpTrack->CalculateVelocity());
197	
198	(*fpPostStepPoint) = (*fpPreStepPoint);
199	}
200	

Address ▲	Source Line	Assembly	Clockticks	Instructions Retired	CPI Rate	Retiring
0x2bd92	176	callq 0xf580	1,674,000,000	1,872,000,000	0.894	12.6%
0x2bd97		**Block 8:**				
0x2bd97	176	movq %rax, 0x60(%r15)				
0x2bd9b	178	movq 0x28(%rbx), %rax				
0x2bd9f	178	movq 0x10(%rbx), %r12				
0x2bda3	178	movq 0x38(%rax), %rax				
0x2bda7	178	test %rax, %rax				
0x2bdaa	178	jz 0x2bfb3 <Block 40>				
0x2bdb0		**Block 9:**				
0x2bdb0	178	movq 0x8(%rax), %rdi				
0x2bdb4	178	xor %esi, %esi	0	18,000,000	0.000	0.0%
0x2bdb6	179	movq (%rdi), %rax				
0x2bdb9	178	callq 0x20(%rax)	0	54,000,000	0.000	0.0%
0x2bdbc		**Block 10:**				
0x2bdbc	178	movq 0x10(%rax), %rdi	36,000,000	0		0.0%
0x2bdc0	178	callq 0xf690	684,000,000	504,000,000	1.357	3.1%
0x2bdc5		**Block 11:**				
0x2bdc5	178	movq %rax, 0x68(%r12)				
0x2bdca	182	movq 0x28(%rbx), %rax				
0x2bdce	182	movq 0x10(%rbx), %r12				
0x2bdd2	182	movq 0x38(%rax), %rax				
0x2bdd6	182	test %rax, %rax				
0x2bdd9	182	jz 0x2bfb3 <Block 40>				
0x2bddf		**Block 12:**				
0x2bddf	182	movq 0x8(%rax), %rdi				
0x2bde3	182	xor %esi, %esi				
0x2bde5	183	movq (%rdi), %rax				
0x2bde8	182	callq 0x20(%rax)	54,000,000	36,000,000	1.500	0.0%
0x2bdeb		**Block 13:**				
0x2bdeb	182	movq 0x10(%rax), %rdi	0	0	0.000	0.0%
0x2bdef	182	callq 0xf9e0	720,000,000	540,000,000	1.333	0.0%
0x2bdf4		**Block 14:**				
0x2bdf4	182	movq 0x28(%rbx), %rdi	0	0	0.000	0.0%
0x2bdf8	182	movq %rax, 0x70(%r12)				
0x2bdfd	186	movq 0x50(%rdi), %r12				

Avoid Distant Data Accesses

```
   // Check if the particle has a force, EM or gravitational, exerted on it
   //
-  G4FieldManager* fieldMgr = 0;
-  G4bool fieldExertsForce  = false;

-  fieldMgr = fFieldPropagator->FindAndSetFieldManager(track.GetVolume());
   G4bool eligibleEM =
-    (particleCharge != 0.0) || (fUseMagneticMoment && (magneticMoment != 0.0));
-  G4bool eligibleGrav = fUseGravity && (restMass != 0.0);
+    (particleCharge != 0.0) || ((magneticMoment != 0.0) && fUseMagneticMoment);
+  G4bool eligibleGrav = (restMass != 0.0) && fUseGravity;

-  if((fieldMgr != nullptr) && (eligibleEM || eligibleGrav))
+  fFieldExertedForce = false;
+
+  if(eligibleEM || eligibleGrav)
   {
-    // User can configure the field Manager for this track
-    fieldMgr->ConfigureForTrack(&track);
-    // Called here to allow a transition from no-field pointer
-    // to finite field (non-zero pointer).
-
-    // If the field manager has no field ptr, the field is zero
-    //   by definition ( = there is no field ! )
-    const G4Field* ptrField = fieldMgr->GetDetectorField();
-    if(ptrField)
+    if(G4FieldManager* fieldMgr =
+         fFieldPropagator->FindAndSetFieldManager(track.GetVolume()))
     {
lines 2502-2530/3673 69%
```

Finding the field manager is expensive. It requires accessing distant pieces of data like the fFieldPropagator class member to call its method, and the track's current volume. However, we can avoid checking the field for neutral and/or massless particles, as the field has no effect on them.

Data Access Patterns: Group Nearby Reads & Writes

```
    G4Transportation: Move changes to fParticleChange closer together

diff --git a/source/processes/transportation/src/G4Transportation.cc b/source/processes/transportation/src/G4Transportation.cc
index 1fc97e1595..ba7a621299 100644
--- a/source/processes/transportation/src/G4Transportation.cc
+++ b/source/processes/transportation/src/G4Transportation.cc
@@ -195,12 +195,6 @@ G4double G4Transportation::AlongStepGetPhysicalInteractionLength(
   //
   *selection = CandidateForSelection;

-  fFirstStepInVolume = fNewTrack || fLastStepInVolume;
-  fLastStepInVolume  = false;
-  fNewTrack          = false;
-
-  fParticleChange.ProposeFirstStepInVolume(fFirstStepInVolume);
-
   // Get initial Energy/Momentum of the track
   //
   const G4DynamicParticle* pParticle       = track.GetDynamicParticle();
@@ -510,6 +504,11 @@ G4double G4Transportation::AlongStepGetPhysicalInteractionLength(
     }
   }

+  fFirstStepInVolume = fNewTrack || fLastStepInVolume;
+  fLastStepInVolume  = false;
+  fNewTrack          = false;
+
+  fParticleChange.ProposeFirstStepInVolume(fFirstStepInVolume);
   fParticleChange.ProposeTrueStepLength(geometryStepLength);
lines 2361-2389/3673 65%
```

If a class member needs to be accessed multiple times inside a function or method, prefer keeping these accesses close together to avoid unnecessary cache misses.

Unnecessary Work: Caching Data

```
@@ -297,38 +300,30 @@ G4CrossSectionDataStore::GetCrossSection(const G4DynamicParticle* part,
                                          const G4Element* elm,
                                         const G4Material* mat)
 {
-  if(mat == elmMaterial && elm == currentElement &&
-     part->GetDefinition() == elmParticle &&
-     part->GetKineticEnergy() == elmKinEnergy)
-    { return elmCrossSection; }
-
-  elmMaterial = mat;
-  currentElement = elm;
-  elmParticle = part->GetDefinition();
-  elmKinEnergy = part->GetKineticEnergy();
-  elmCrossSection = 0.0;
-
-  G4int i = nDataSetList-1;
+  G4int i = nDataSetList-1;
   G4int Z = elm->GetZasInt();
+
   if (elm->GetNaturalAbundanceFlag() &&
       dataSetList[i]->IsElementApplicable(part, Z, mat)) {

     // element wise cross section
-    elmCrossSection = dataSetList[i]->GetElementCrossSection(part, Z, mat);
-  } else {
-    // isotope wise cross section
-    size_t nIso = elm->GetNumberOfIsotopes();
-
-    // user-defined isotope abundances
lines 3282-3310/3673 91%
```

This method is always called with each element of each material in a loop, so the element is never the same, and the cache was missed 100% of the time.

Loop Optimizations: Invariant Expressions

```
for (int i = 0; i < N; ++i) {
  for (int j = 0; j < N; ++j)
    a[j] = b[j] * c[i];
}
```

Expressions that do not change with the loop iteration are called loop invariants.

Loop invariants should be hoisted out of the loop to avoid unnecessary computations.

```
for (int i = 0; i < N; ++i) {
  auto ci = c[i];
  for (int j = 0; j < N; ++j)
    a[j] = b[j] * c[i];
}
```

Loop Optimizations: Invariant Conditionals

The body of a loop may contain conditional expressions that may cause poor performance due to branch mispredictions.

The solution is to write two versions of the loop body, and move the condition outside.

Loops with if statements inside

```
for (int i = 0; i < N; ++i) {
  a[i] += b[i];
  if (reset)
     b[i] = 0.0;
}
```

…can sometimes be replaced with separate loops

```
for (int i = 0; i < N; i++)
    a[i] += b[i];
if (reset)
  for (int i = 0; i<N; i++)
    b[i] = 0.0;
```

Loop Optimizations: Loop Unrolling

Loop unrolling can improve performance by reducing the trip count (how many times the body of the loop is run).

Loop unrolling can also help with SIMD vectorization. When it is also vectorized, the unrolled loop puts less pressure in the instruction cache, as there are less instructions overall in the body of the loop.

No loop unrolling

```
for (int i = 0; i < N; ++i)
    a[i] = b[i] * c[i];
```

Loop unrolled once

```
auto n = N/2;
for (int i = 0; i < n; i += 2) {
  a[ i ] = b[ i ] * c[ i ];
  a[i+1] = b[i+1] * c[i+1];
}
```

Loop Optimizations: Strength Reduction

The body of a loop may contain expensive operations or similar operations that can be replaced with cheaper ones.

Here, we replace a multiplication with additions, and avoid computing it twice.

Modern compilers likely already optimize this specific example, however, by computing `3*i` only once and reusing it.

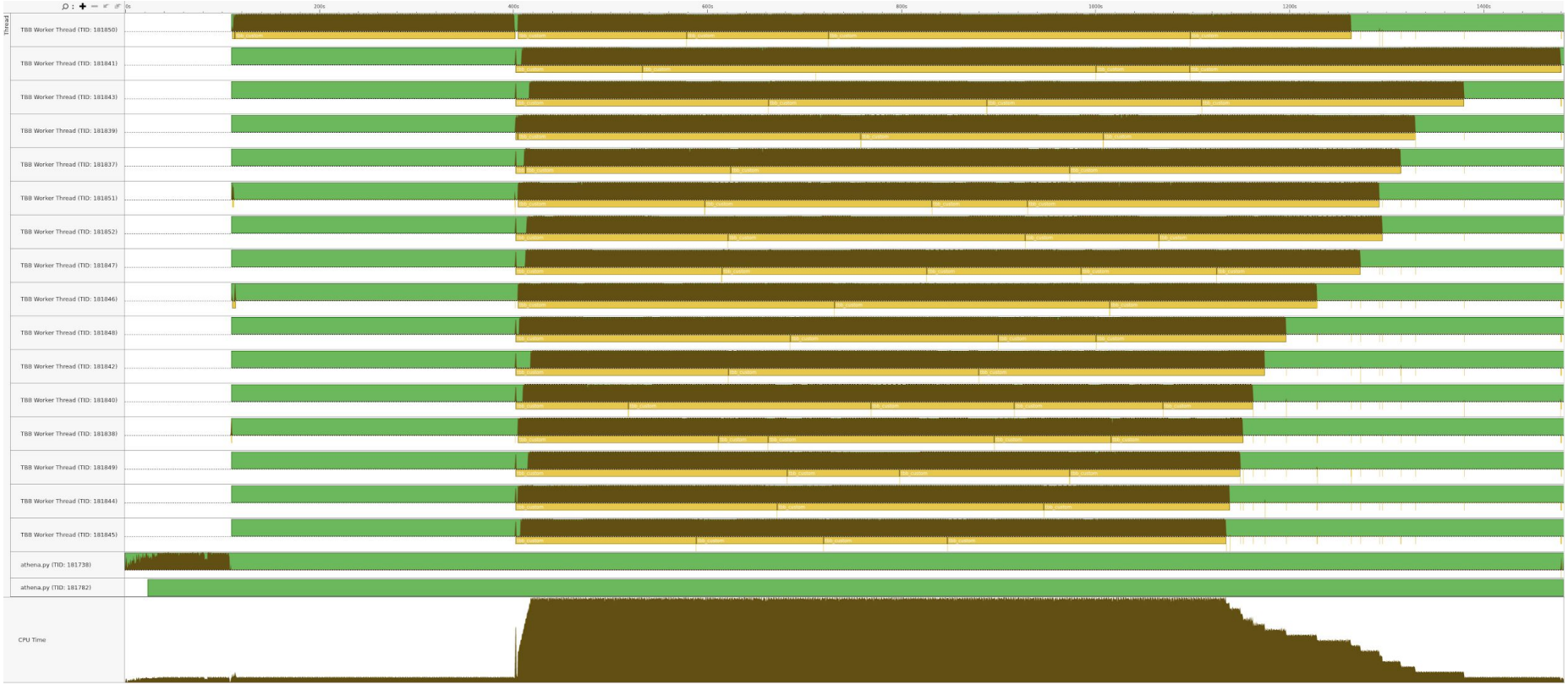
Multiplication in index

```
for (int i = 0; i < N; ++i) {
    a[i] = b[3 * i] * c[i];
    d[i] = f[3 * i] * g[i];
}
```

...can be replaced with additions

```
int j = 0;
for (int i = 0; i<N; i++,j+=3) {
    a[i] = b[j] * c[i];
    d[i] = f[j] * g[i];
}
```

Geant4 Simulation Timeline



perf stat -d – overview of Geant4 initialization

```
bash ~ $ perf stat -r 3 -d -- taskset -c 0 /srv/geant4/install/gcc-10.2.0/10.6.r9-MT/bin/g4run -g ~/src/g4run/CMS.gdml -p pythia
:ttbar -e 0

 Performance counter stats for 'taskset -c 0 /srv/geant4/install/gcc-10.2.0/10.6.r9-MT/bin/g4run -g /home/amadio/src/g4run/CMS.g
dml -p pythia:ttbar -e 0' (3 runs):

         21454.06 msec task-clock                #    0.953 CPUs utilized           ( +-  0.07% )
             1520      context-switches          #    0.071 K/sec                   ( +- 48.20% )
                1      cpu-migrations            #    0.000 K/sec
           110280      page-faults               #    0.005 M/sec                   ( +-  0.06% )
      93708948749      cycles                    #    4.368 GHz                     ( +-  0.02% )  (74.96%)
        428488171      stalled-cycles-frontend   #    0.46% frontend cycles idle    ( +-  2.14% )  (74.95%)
      62140664026      stalled-cycles-backend    #   66.31% backend cycles idle     ( +-  0.04% )  (74.97%)
     129389101781      instructions              #    1.38  insn per cycle
                                                #    0.48  stalled cycles per insn ( +-  0.04% )  (75.02%)
      16731397508      branches                  #  779.871 M/sec                   ( +-  0.06% )  (75.06%)
        156166747      branch-misses             #    0.93% of all branches         ( +-  0.17% )  (75.09%)
      58140887925      L1-dcache-loads           # 2710.018 M/sec                   ( +-  0.10% )  (75.02%)
        685016614      L1-dcache-load-misses     #    1.18% of all L1-dcache accesses  ( +-  1.28% )  (74.93%)
   <not supported>      LLC-loads
   <not supported>      LLC-load-misses

           22.513 +- 0.637 seconds time elapsed  ( +-  2.83% )
```

Whoa, that's a lot of backend cycles idle!

Record Geant4 initialization for further analysis

```
bash ~ $ perf record -e cycles --call-graph=dwarf -F max -- taskset -c 0 /srv/geant4/install/gcc-10.2.0/10.6.r9-MT/bin/g4run --s
tats -g /srv/geant4/gdml/CMS.gdml -p pythia:ttbar -e 0 # record zero event run (initialization only) 2>/dev/null
info: Using a maximum frequency rate of 100000 Hz
       Throughput [events/min]  0
       Initialization Cost [%]  100
       Initialization Time [s]  31.0642
       Event Loop Run Time [s]  2.404e-06
       Init + Ev.Loop Time [s]  31.0642
       Max RSS Before Init [M]  38.7227
       Max RSS  After Init [M]  272.641
       Max RSS  After Loop [M]  272.641
[ perf record: Woken up 76218 times to write data ]
Warning:
Processed 3176007 events and lost 4 chunks!

Check IO/CPU overload!

[ perf record: Captured and wrote 21387.152 MB perf.data (2656290 samples) ]
bash ~ $
bash ~ $ # That's 20GB for ~30s run, dwarf generates a *lot* of data!
bash ~ $ _
```

G4{h,Mu}PairProd. account for ~40% of initialization

```
bash ~ $ perf report -q --stdio --no-children -g none --percent-limit 1 -F overhead,dso,symbol
Warning:
Processed 3176007 events and lost 4 chunks!

Check IO/CPU overload!

    29.82%  libG4processes.so          [.] G4hPairProductionModel::ComputeDMicroscopicCrossSection
    10.38%  libG4processes.so          [.] G4MuPairProductionModel::ComputeDMicroscopicCrossSection
    10.30%  libG4processes.so          [.] G4ElasticHadrNucleusHE::HadrNucDifferCrSec
     3.76%  libG4processes.so          [.] G4eBremsstrahlungRelModel::ComputeLPMfunctions
     3.72%  libG4processes.so          [.] G4hBremsstrahlungModel::ComputeDMicroscopicCrossSection
     3.27%  libG4global.so             [.] G4PhysicsVector::Value
     1.92%  libG4geometry.so           [.] G4Region::BelongsTo
     1.87%  libG4processes.so          [.] G4MuBremsstrahlungModel::ComputeDMicroscopicCrossSection
     1.45%  libG4processes.so          [.] G4SeltzerBergerModel::ComputeDXSectionPerAtom
     1.45%  libG4processes.so          [.] G4ProductionCutsTable::ScanAndSetCouple
     1.33%  libpythia8.so              [.] Pythia8::NNPDF::polint
     1.10%  libG4processes.so          [.] G4eBremsstrahlungRelModel::ComputeXSectionPerAtom
     1.05%  libG4geometry.so           [.] G4LogicalVolume::GetMaterial


bash ~ $  # Looks like ~40% of the time is spent in G4{Mu,h}PairProductionModel::ComputeDMicroscopicCrossSection
bash ~ $  # G4hPairProductionModel::ComputeDMicroscopicCrossSection actually is the same as in G4MuPairProductionModel
bash ~ $  # Note that no inlined functions are shown above, as we asked for no callchain information_
```

Top 3 models account for ~60% of backend stalls

```
bash ~ $ # Same as previous report, but for stalled-cycles-backend
bash ~ $ perf report -q --stdio --no-children -g none --percent-limit 1 -F overhead,dso,symbol
Warning:
Processed 3080687 events and lost 104 chunks!

Check IO/CPU overload!

    33.59%  libG4processes.so          [.] G4hPairProductionModel::ComputeDMicroscopicCrossSection
    14.26%  libG4processes.so          [.] G4ElasticHadrNucleusHE::HadrNucDifferCrSec
    11.97%  libG4processes.so          [.] G4MuPairProductionModel::ComputeDMicroscopicCrossSection
     4.98%  libG4processes.so          [.] G4eBremsstrahlungRelModel::ComputeLPMfunctions
     4.90%  libG4processes.so          [.] G4hBremsstrahlungModel::ComputeDMicroscopicCrossSection
     3.80%  libG4global.so             [.] G4PhysicsVector::Value
     2.46%  libG4processes.so          [.] G4MuBremsstrahlungModel::ComputeDMicroscopicCrossSection
     2.09%  libG4geometry.so           [.] G4Region::BelongsTo
     1.65%  libG4processes.so          [.] G4SeltzerBergerModel::ComputeDXSectionPerAtom
     1.36%  libG4processes.so          [.] G4eBremsstrahlungRelModel::ComputeXSectionPerAtom
     1.07%  libG4processes.so          [.] G4LossTableBuilder::BuildRangeTable
     1.01%  libpythia8.so              [.] Pythia8::NNPDF::polint


bash ~ $ # The top 3 models are responsible for ~60% of all stalled cycles
bash ~ $ # However, L1 cache misses are about ~1.2%, so this is likely *not* a memory access problem_
```

How to improve performance?

- Look for pair production model in Geant4 Physics Manual

Formulae

The differential cross section for electron pair production by muons $\sigma(Z, A, E, \epsilon)$ can be written as :

$$\sigma(Z, A, E, \epsilon) = \frac{4}{3\pi} \frac{Z(Z+\zeta)}{A} N_A (\alpha r_0)^2 \frac{1-v}{\epsilon} \int_0^{\rho_{\max}} G(Z, E, v, \rho)\, d\rho, \tag{154}$$

where

$$G(Z, E, v, \rho) = \Phi_e + (m/\mu)^2 \Phi_\mu,$$
$$\Phi_{e,\mu} = B_{e,\mu} L'_{e,\mu}$$

and

$$\Phi_{e,\mu} = 0 \quad \text{whenever} \quad \Phi_{e,\mu} < 0.$$

B_e and B_μ do not depend on Z, A, and are given by

$$B_e = [(2+\rho^2)(1+\beta) + \xi(3+\rho^2)] \ln\left(1 + \frac{1}{\xi}\right) + \frac{1-\rho^2-\beta}{1+\xi} - (3+\rho^2);$$
$$\approx \frac{1}{2\xi}[(3-\rho^2) + 2\beta(1+\rho^2)] \quad \text{for} \quad \xi \geq 10^3;$$

$$B_\mu = \left[(1+\rho^2)\left(1 + \frac{3\beta}{2}\right) - \frac{1}{\xi}(1+2\beta)(1-\rho^2)\right] \ln(1+\xi) + \frac{\xi(1-\rho^2-\beta)}{1+\xi} + (1+2\beta)(1-\rho^2);$$
$$B_\mu \approx \frac{\xi}{2}[(5-\rho^2) + \beta(3+\rho^2)] \quad \text{for} \quad \xi \leq 10^{-3};$$

Also,

$$\xi = \frac{\mu^2 v^2}{4m^2} \frac{(1-\rho^2)}{(1-v)}; \quad \beta = \frac{v^2}{2(1-v)};$$

$$L'_e = \ln \frac{A^* Z^{-1/3} \sqrt{(1+\xi)(1+Y_e)}}{1 + \frac{2m\sqrt{e} A^* Z^{-1/3}(1+\xi)(1+Y_e)}{Ev(1-\rho^2)}} - \frac{1}{2} \ln\left[1 + \left(\frac{3mZ^{1/3}}{2\mu}\right)^2 (1+\xi)(1+Y_e)\right];$$

$$L'_\mu = \ln \frac{(\mu/m) A^* Z^{-1/3} \sqrt{(1+1/\xi)(1+Y_\mu)}}{1 + \frac{2m\sqrt{e} A^* Z^{-1/3}(1+\xi)(1+Y_\mu)}{Ev(1-\rho^2)}} - \ln\left[\frac{3}{2} Z^{1/3} \sqrt{(1+1/\xi)(1+Y_\mu)}\right].$$

For faster computing, the expressions for $L'_{e,\mu}$ are further algebraically transformed. The functions $L'_{e,\mu}$ include the nuclear size correction [KP71] in comparison with parameterization [KP70] :

$$Y_e = \frac{5 - \rho^2 + 4\beta(1+\rho^2)}{2(1+3\beta)\ln(3+1/\xi) - \rho^2 - 2\beta(2-\rho^2)};$$
$$Y_\mu = \frac{4 + \rho^2 + 3\beta(1+\rho^2)}{(1+\rho^2)(\frac{3}{2}+2\beta)\ln(3+\xi) + 1 - \frac{3}{2}\rho^2};$$
$$\rho_{\max} = [1 - 6\mu^2/E^2(1-v)]\sqrt{1 - 4m/Ev}.$$

Comment on the Calculation of the Integral $\int d\rho$ in Eq.(154)

The integral $\int_0^{\rho_{\max}} G(Z, E, v, \rho)\, d\rho$ is computed with the substitutions:

$$t = \ln(1-\rho),$$
$$1 - \rho = \exp(t),$$
$$1 + \rho = 2 - \exp(t),$$
$$1 - \rho^2 = e^t(2 - e^t).$$

After that,

$$\int_0^{\rho_{\max}} G(Z, E, v, \rho)\, d\rho = \int_{t_{\min}}^0 G(Z, E, v, \rho)\, e^t\, dt, \tag{155}$$

where

$$t_{\min} = \ln \frac{\frac{4m}{\epsilon} + \frac{12\mu^2}{EE'}\left(1 - \frac{4m}{\epsilon}\right)}{1 + \left(1 - \frac{6\mu^2}{EE'}\right)\sqrt{1 - \frac{4m}{\epsilon}}}.$$

To compute the integral of Eq.(155) with an accuracy better than 0.5%, Gaussian quadrature with $N = 8$ points is sufficient.

The function $\zeta(E, Z)$ in Eq.(154) serves to take into account the process on atomic electrons (inelastic atomic form factor contribution). To treat the energy loss balance correctly, the following approximation, which is an algebraic transformation of the expression in Ref. [Kel98], is used:

$$\zeta(E, Z) = \frac{0.073 \ln \frac{E/\mu}{1+\gamma_1 Z^{2/3} E/\mu} - 0.26}{0.058 \ln \frac{E/\mu}{1+\gamma_2 Z^{1/3} E/\mu} - 0.14};$$
$$= 0 \quad \text{if the numerator is negative.}$$

For $E \leq 35\,\mu$, $\zeta(E, Z) = 0$. Also $\gamma_1 = 1.95 \cdot 10^{-5}$ and $\gamma_2 = 5.30 \cdot 10^{-5}$.

The above formulae make use of the Thomas-Fermi model which is not good enough for light elements. For hydrogen ($Z = 1$) the following parameters must be changed:

- $A^* = 183 \Rightarrow 202.4$;
- $\gamma_1 = 1.95 \cdot 10^{-5} \Rightarrow 4.4 \cdot 10^{-5}$;
- $\gamma_2 = 5.30 \cdot 10^{-5} \Rightarrow 4.8 \cdot 10^{-5}$.

How to improve performance?

- Look for pair production model in Geant4 Physics Manual
  - Rework expressions for cross sections with pencil/paper to reduce arithmetic operations
- Avoid unnecessary calls to G4Log function when calculating zeta
- Remove data dependencies
  - Break up large for loop into several smaller for loops
  - Compute together things that don't depend on each other
  - Hide latency from divisions
  - When calling G4Log, input is already available
  - Move common expressions out of for loop
- Remove code duplication from the two classes with essentially the same version of this function by inheriting the base version in the derived class

Return early to avoid unnecessary divisions

```
commit 1c0b893bef0fe6f32fd8593989882239628262a9
Author: Guilherme Amadio <amadio@cern.ch>

    G4MuPairProductionModel: make early return condition more explicit

diff --git a/source/processes/electromagnetic/muons/src/G4MuPairProductionModel.cc b/source/processes/electromagnetic/muons/src/G4MuPairProductionMode
l.cc
index f7ba48dc7d..e0b7c550f0 100644
--- a/source/processes/electromagnetic/muons/src/G4MuPairProductionModel.cc
+++ b/source/processes/electromagnetic/muons/src/G4MuPairProductionModel.cc
@@ -321,6 +321,9 @@ G4double G4MuPairProductionModel::ComputeDMicroscopicCrossSection(
   static const G4double g1h  = 4.4e-5 ;
   static const G4double g2h  = 4.8e-5 ;

+  if (pairEnergy <= 4.0 * electron_mass_c2)
+    return 0.0;
+
   G4double totalEnergy  = tkin + particleMass;
   G4double residEnergy  = totalEnergy - pairEnergy;
   G4double massratio    = particleMass/electron_mass_c2;
@@ -334,7 +337,6 @@ G4double G4MuPairProductionModel::ComputeDMicroscopicCrossSection(
   G4double c8 = 6.*particleMass*particleMass;
   G4double alf = c7/pairEnergy;
   G4double a3 = 1. - alf;
-  if (a3 <= 0.) { return cross; }

   // zeta calculation
   G4double bbb,g1,g2;

lines 685-712/3673 18%
```

Moving the early return up reduces unnecessary divisions.
Also a3 <= 0 is harder to understand than the new form.

Result of expensive call to G4Log not always used

```
// zeta calculation
G4double bbb,g1,g2;
if( Z < 1.5 ) { bbb = bbbh ; g1 = g1h ; g2 = g2h ; }
else          { bbb = bbbtf; g1 = g1tf; g2 = g2tf; }

G4double zeta = 0;
G4double zeta1 =
  0.073*G4Log(totalEnergy/(particleMass+g1*z23*totalEnergy))-0.26;
if ( zeta1 > 0.)
{
  G4double zeta2 =
    0.058*G4Log(totalEnergy/(particleMass+g2*z13*totalEnergy))-0.14;
  zeta  = zeta1/zeta2 ;
}

G4double z2 = Z*(Z+zeta);
G4double screen0 = 2.*electron_mass_c2*sqrte*bbb/(z13*pairEnergy);
G4double a0 = totalEnergy*residEnergy;
G4double a1 = pairEnergy*pairEnergy/a0;
G4double bet = 0.5*a1;
G4double xi0 = 0.25*massratio2*a1;
G4double del = c8/a0;
_
                                                        357,1          48%
```

this if statement for treating hydrogen differently can be replaced by branchless code (index based on boolean result)

this call to G4Log can be avoided when `zeta1 <= 0.0`

result of division used right away

Avoid calling G4Log by replacing condition

```
@@ -350,14 +350,15 @@ G4double G4MuPairProductionModel::ComputeDMicroscopicCrossSection(
   if( Z < 1.5 ) { bbb = bbbh ; g1 = g1h ; g2 = g2h ; }
   else          { bbb = bbbtf; g1 = g1tf; g2 = g2tf; }

-  G4double zeta = 0;
-  G4double zeta1 =
-    0.073*G4Log(totalEnergy/(particleMass+g1*z23*totalEnergy))-0.26;
-  if ( zeta1 > 0.)
+  G4double zeta = 0.0;
+  G4double z1exp = totalEnergy / (particleMass + g1*z23*totalEnergy);
+
+  // 35.221047195922 is the root of zeta1(x) = 0.073 * log(x) - 0.26, so the
+  // condition below is the same as zeta1 > 0.0, but without calling log(x)
+  if (z1exp > 35.221047195922)
   {
-    G4double zeta2 =
-      0.058*G4Log(totalEnergy/(particleMass+g2*z13*totalEnergy))-0.14;
-    zeta  = zeta1/zeta2 ;
+    G4double z2exp = totalEnergy / (particleMass + g2*z13*totalEnergy);
+    zeta = (0.073 * G4Log(z1exp) - 0.26) / (0.058 * G4Log(z2exp) - 0.14);
   }

   G4double z2 = Z*(Z+zeta);
lines 2339-2361/3238 74%
```

We can avoid calling G4Log by replacing the condition with an equivalent one for the input argument of G4Log.

Big loop with many data dependencies (cont.)

```
G4double tmn = G4Log(tmnexp);
G4double sum = 0.;

// Gaussian integration in ln(1-ro)
for (G4int i=0; i<8; ++i)
{
  G4double a4 = G4Exp(tmn*xgi[i]);
  G4double a5 = a4*(2.-a4) ;
  G4double a6 = 1.-a5 ;
  G4double a7 = 1.+a6 ;
  G4double a9 = 3.+a6 ;
  G4double xi = xi0*a5 ;
  G4double xii = 1./xi ;
  G4double xi1 = 1.+xi ;
  G4double screen = screen0*xi1/a5 ;
  G4double yeu = 5.-a6+4.*bet*a7 ;
  G4double yed = 2.*(1.+3.*bet)*G4Log(3.+xii)-a6-a1*(2.-a6) ;
  G4double ye1 = 1.+yeu/yed ;
  G4double ale = G4Log(bbb/z13*sqrt(xi1*ye1)/(1.+screen*ye1)) ;
  G4double cre = 0.5*G4Log(1.+2.25*z23*xi1*ye1/massratio2) ;
  G4double be;
"/srv/geant4/src/geant4-10.6.r9/source/processes/electromagnetic/muons/src/G4MuPairProductionModel.cc" 712 lines --50%--
Press ENTER or type command to continue
```

Observations:

- Big for loop with fixed iteration count, but no vectorization
  - Loop has common expressions that can be moved out
- Variable names make code hard to understand
- Many data dependencies reduce parallelism
  - Results of divisions and sqrt used immediately
    - Result of `tmn = G4Log(tmnexp)` used immediately
    - Results of divisions and sqrt used inside call to G4Log
  - G4Log is called (and inlined!) 4 times just here
    - G4Log **inlined 10 times** just in this function!

hottest source lines shown by perf annotate

Use perf annotate to find hottest parts of the code

```
Samples: 2M of event 'cycles', 100000 Hz, Event count (approx.): 113912086093
G4hPairProductionModel::ComputeDMicroscopicCrossSection  /srv/geant4/install/gcc-10.2.0/10.6.r9-MT/lib64/libG4processes.so [Perc
  0.46          vdivsd       %xmm0,%xmm7,%xmm7
              G4double ale=G4Log(bbb/z13*sqrt(xi1*ye1)/(1.+screen*ye1)) ;
  2.33          vmovsd       -0x88(%rbp),%xmm0
  0.00          vdivsd       0x140(%r10),%xmm0,%xmm15
              G4double ye1 = 1.+yeu/yed ;
                vaddsd       %xmm2,%xmm7,%xmm7
              G4double ale=G4Log(bbb/z13*sqrt(xi1*ye1)/(1.+screen*ye1)) ;
  0.15          vmulsd       %xmm7,%xmm11,%xmm0
  0.78          vucomisd     %xmm0,%xmm3
  0.32        → ja           649179 <G4hPairProductionModel::ComputeDMicroscopicCrossSection(double, double, double)+0x13e9>
  0.31          vsqrtsd      %xmm0,%xmm0,%xmm0
  2.48          vmulsd       %xmm0,%xmm15,%xmm0
  0.45          vmovsd       -0x50(%rbp),%xmm15
                vfmadd132sd  %xmm7,%xmm2,%xmm15
                vdivsd       %xmm15,%xmm0,%xmm15
              G4LogConsts::dp2uint64(double):
              tmp.d = x;
  1.99          vmovq        %xmm15,%rax
              G4LogConsts::getMantExponent(double, double&):
              uint64_t le = (n >> 52);
  0.15          vmovq        %xmm15,%rdx
Press 'h' for help on key bindings
```

hottest instructions

Big loop with many data dependencies

```
for (G4int i=0; i<8; ++i)
{
  G4double a4 = G4Exp(tmn*xgi[i]);      // a4 = (1.-asymmetry)
  G4double a5 = a4*(2.-a4) ;
  G4double a6 = 1.-a5 ;
  G4double a7 = 1.+a6 ;
  G4double a9 = 3.+a6 ;
  G4double xi = xi0*a5 ;
  G4double xii = 1./xi ;
  G4double xi1 = 1.+xi ;
  G4double screen = screen0*xi1/a5 ;
  G4double yeu = 5.-a6+4.*bet*a7 ;
  G4double yed = 2.*(1.+3.*bet)*G4Log(3.+xii)-a6-a1*(2.-a6) ;
  G4double ye1 = 1.+yeu/yed ;
  G4double ale = G4Log(bbb/z13*sqrt(xi1*ye1)/(1.+screen*ye1)) ;
  G4double cre = 0.5*G4Log(1.+2.25*z23*xi1*ye1/massratio2) ;
  G4double be;

  if (xi <= 1.e3) {
    be = ((2.+a6)*(1.+bet)+xi*a9)*G4Log(1.+xii)+(a5-bet)/xi1-a9;
  } else {
    be = (3.-a6+a1*a7)/(2.*xi);
  }
  G4double fe = (ale-cre)*be;
  if ( fe < 0.) fe = 0. ;

  G4double ymu = 4.+a6 +3.*bet*a7 ;
  G4double ymd = a7*(1.5+a1)*G4Log(3.+xi)+1.-1.5*a6 ;
  G4double ym1 = 1.+ymu/ymd ;
                                                                  363,26        57%
```

Data dependencies between arithmetic operations can create execution latency even without cache misses.

Split it into two loops to hide latency from divisions

```
-  for (G4int i=0; i<8; ++i)
+  G4double rho[8];
+  G4double rho2[8];
+  G4double xi[8];
+  G4double xi1[8];
+  G4double xii[8];
+
+  for (G4int i = 0; i < 8; ++i)
   {
-    G4double rho = G4Exp(tmn*xgi[i]) - 1.0; // rho = -asymmetry
-    G4double rho2 = rho * rho;
-    G4double xi = xi0*(1.0-rho2) ;
-    G4double xii = 1./xi ;
-    G4double xi1 = 1.+xi ;
-    G4double screen = screen0*xi1/(1.0-rho2) ;
-    G4double yeu = 5.-rho2+4.*bet*(1.0+rho2) ;
-    G4double yed = 2.*(1.+3.*bet)*G4Log(3.+xii)-rho2-a1*(2.-rho2) ;
+    rho[i] = G4Exp(tmn*xgi[i]) - 1.0; // rho = -asymmetry
+    rho2[i] = rho[i] * rho[i];
+    xi[i] = xi0*(1.0-rho2[i]);
+    xi1[i] = 1.0 + xi[i];
+    xii[i] = 1.0 / xi[i];
+  }
+
+  for (G4int i = 0; i < 8; ++i)
+  {
+    G4double screen = screen0*xi1[i]/(1.0-rho2[i]) ;
+    G4double yeu = 5.-rho2[i]+4.*bet*(1.0+rho2[i]) ;
+    G4double yed = 2.*(1.+3.*bet)*G4Log(3.+xii[i])-rho2[i]-a1*(2.-rho2[i]) ;
lines 1088-1116
```

Data dependencies between arithmetic operations can create execution latency even without cache misses.

Breaking up long loops into smaller parts makes it possible to hide some of the latency from divisions and math function calls with instruction level parallelism.

Reduce Code Duplication

```
-#include "G4PhysicalConstants.hh"
-#include "G4Log.hh"
-#include "G4Exp.hh"
-
-using namespace std;

 G4hPairProductionModel::G4hPairProductionModel(const G4ParticleDefinition* p,
                                                const G4String& nam)
@@ -65,114 +60,3 @@ G4hPairProductionModel::~G4hPairProductionModel()
 {}

 //....oooOO0OOooo........oooOO0OOooo........oooOO0OOooo........oooOO0OOooo......
-
-G4double G4hPairProductionModel::ComputeDMicroscopicCrossSection(
-                                          G4double tkin,
-                                          G4double Z,
-                                          G4double pairEnergy)
-//  differential cross section
-{
-  static const G4double bbbtf= 183. ;
-  static const G4double bbbh = 202.4 ;
-  static const G4double g1tf = 1.95e-5 ;
-  static const G4double g2tf = 5.3e-5 ;
-  static const G4double g1h  = 4.4e-5 ;
-  static const G4double g2h  = 4.8e-5 ;
-
-  G4double totalEnergy  = tkin + particleMass;
-  G4double residEnergy  = totalEnergy - pairEnergy;
-  G4double massratio    = particleMass/electron_mass_c2 ;
lines 511-539
```

This was a copy of G4MuPairProductionModel::ComputeDMicroscopicCrossSection. We can keep only the copy from the base class.

From ~40% of initialization to ~27%, not bad!

```
bash ~ $ perf report -q --stdio --no-children -g none --percent-limit 1 -F overhead,dso,symbol
Warning:
Processed 2720752 events and lost 7 chunks!

Check IO/CPU overload!

    26.88%  libG4processes.so          [.] G4MuPairProductionModel::ComputeDMicroscopicCrossSection
    12.60%  libG4processes.so          [.] G4ElasticHadrNucleusHE::HadrNucDifferCrSec
     4.60%  libG4processes.so          [.] G4eBremsstrahlungRelModel::ComputeLPMfunctions
     4.53%  libG4processes.so          [.] G4hBremsstrahlungModel::ComputeDMicroscopicCrossSection
     3.96%  libG4global.so             [.] G4PhysicsVector::Value
     2.33%  libG4geometry.so           [.] G4Region::BelongsTo
     2.26%  libG4processes.so          [.] G4MuBremsstrahlungModel::ComputeDMicroscopicCrossSection
     1.78%  libG4processes.so          [.] G4SeltzerBergerModel::ComputeDXSectionPerAtom
     1.67%  libG4processes.so          [.] G4ProductionCutsTable::ScanAndSetCouple
     1.63%  libpythia8.so              [.] Pythia8::NNPDF::polint
     1.39%  libG4processes.so          [.] G4eBremsstrahlungRelModel::ComputeXSectionPerAtom
     1.30%  libG4geometry.so           [.] G4LogicalVolume::GetMaterial
     1.06%  libG4processes.so          [.] G4LossTableBuilder::BuildRangeTable
     1.01%  libc-2.32.so               [.] malloc


bash ~ $
```

Revisiting overview of Geant4 initialization (before)

```
bash ~ $ perf stat -r 3 -d -- taskset -c 0 /srv/geant4/install/gcc-10.2.0/10.6.r9-MT/bin/g4run -g ~/src/g4run/CMS.gdml -p pythia
:ttbar -e 0

 Performance counter stats for 'taskset -c 0 /srv/geant4/install/gcc-10.2.0/10.6.r9-MT/bin/g4run -g /home/amadio/src/g4run/CMS.g
dml -p pythia:ttbar -e 0' (3 runs):

         21454.06 msec task-clock                #    0.953 CPUs utilized           ( +-  0.07% )
             1520      context-switches          #    0.071 K/sec                   ( +- 48.20% )
                1      cpu-migrations            #    0.000 K/sec
           110280      page-faults               #    0.005 M/sec                   ( +-  0.06% )
      93708948749      cycles                    #    4.368 GHz                     ( +-  0.02% )  (74.96%)
        428488171      stalled-cycles-frontend   #    0.46% frontend cycles idle    ( +-  2.14% )  (74.95%)
      62140664026      stalled-cycles-backend    #   66.31% backend cycles idle     ( +-  0.04% )  (74.97%)
     129389101781      instructions              #    1.38  insn per cycle
                                                 #    0.48  stalled cycles per insn ( +-  0.04% )  (75.02%)
      16731397508      branches                  #  779.871 M/sec                   ( +-  0.06% )  (75.06%)
        156166747      branch-misses             #    0.93% of all branches         ( +-  0.17% )  (75.09%)
      58140887925      L1-dcache-loads           # 2710.018 M/sec                   ( +-  0.10% )  (75.02%)
        685016614      L1-dcache-load-misses     #    1.18% of all L1-dcache accesses  ( +-  1.28% )  (74.93%)
   <not supported>      LLC-loads
   <not supported>      LLC-load-misses

           22.513 +- 0.637 seconds time elapsed  ( +-  2.83% )
```

Revisiting overview of Geant4 initialization (after)

```
bash ~ $ perf stat -r 3 -d -- taskset -c 0 /srv/geant4/install/gcc-10.2.0/10.6.r10-MT/bin/g4run -g ~/src/g4run/CMS.gdml -p pythi
a:ttbar -e 0

 Performance counter stats for 'taskset -c 0 /srv/geant4/install/gcc-10.2.0/10.6.r10-MT/bin/g4run -g /home/amadio/src/g4run/CMS.
gdml -p pythia:ttbar -e 0' (3 runs):

          17517.74 msec task-clock                #    0.982 CPUs utilized            ( +-  0.20% )
               282      context-switches          #    0.016 K/sec                    ( +-  1.13% )
                 1      cpu-migrations            #    0.000 K/sec
            110368      page-faults               #    0.006 M/sec                    ( +-  0.01% )
       76145083799      cycles                    #    4.347 GHz                      ( +-  0.11% )  (74.98%)
         428362729      stalled-cycles-frontend   #    0.56% frontend cycles idle     ( +-  0.83% )  (74.98%)
       45042101836      stalled-cycles-backend    #   59.15% backend cycles idle      ( +-  0.14% )  (74.99%)
      125423223087      instructions              #    1.65  insn per cycle
                                                  #    0.36  stalled cycles per insn  ( +-  0.04% )  (74.96%)
       16930383319      branches                  #  966.471 M/sec                    ( +-  0.23% )  (74.94%)
         157044670      branch-misses             #    0.93% of all branches          ( +-  0.02% )  (75.04%)
       57470258214      L1-dcache-loads           # 3280.690 M/sec                    ( +-  0.09% )  (75.09%)
         675844171      L1-dcache-load-misses     #    1.18% of all L1-dcache accesses  ( +-  0.35% )  (75.02%)
   <not supported>      LLC-loads
   <not supported>      LLC-load-misses

           17.8476 +- 0.0237 seconds time elapsed  ( +-  0.13% )
```

DWARF can show time spent in inlined functions

```
Samples: 2M of event 'cycles', Event count (approx.): 113912086093
  Children      Self  Shared Object              Symbol
-   38.11%     0.01%  libG4processes.so          [.] G4MuPairProductionModel::ComputeCrossSectionPerAtom
   - 38.10% G4MuPairProductionModel::ComputeCrossSectionPerAtom
      - 38.10% G4MuPairProductionModel::ComputeMicroscopicCrossSection
         - 28.15% G4hPairProductionModel::ComputeDMicroscopicCrossSection
            + 19.40% G4Log (inlined)
              0.60% G4Exp (inlined)
         - 9.77% G4MuPairProductionModel::ComputeDMicroscopicCrossSection
            + 6.72% G4Log (inlined)
+   38.10%     0.19%  libG4processes.so          [.] G4MuPairProductionModel::ComputeMicroscopicCrossSection
+   29.84%    29.82%  libG4processes.so          [.] G4hPairProductionModel::ComputeDMicroscopicCrossSection
+   20.55%     0.00%  libG4processes.so          [.] G4Log (inlined)
+   13.04%     0.00%  libG4processes.so          [.] G4HadronicInteractionRegistry::InitialiseModels
+   10.70%     0.00%  libG4processes.so          [.] G4ElasticHadrNucleusHE::InitialiseModel
+   10.70%     0.00%  libG4processes.so          [.] G4ElasticHadrNucleusHE::FillData
+   10.69%     0.00%  libG4processes.so          [.] G4ElasticHadrNucleusHE::FillFq2
+   10.69%    10.30%  libG4processes.so          [.] G4ElasticHadrNucleusHE::HadrNucDifferCrSec
+   10.50%     0.00%  libG4run.so                [.] G4RunManager::Initialize
+   10.39%    10.38%  libG4processes.so          [.] G4MuPairProductionModel::ComputeDMicroscopicCrossSection
+    9.19%     0.00%  libG4run.so                [.] G4VUserPhysicsList::PreparePhysicsTable
+    9.18%     0.00%  libG4processes.so          [.] G4EmModelManager::Initialise
+    8.93%     0.00%  g4run                      [.] DetectorConstruction::Construct
Tip: See assembly instructions with percentage: perf annotate <symbol>
```

DWARF also allows to sort by source line

```
Samples: 2M of event 'stalled-cycles-backend', Event count (approx.): 68339152914
  Overhead  Source:Line
+    5.50%  G4Log.hh:250
+    4.05%  G4Exp.hh:213
+    3.64%  G4Log.hh:195
+    2.58%  G4Log.hh:258
+    2.55%  G4Log.hh:235
+    2.39%  G4Log.hh:254
+    2.14%  G4Log.hh:253
+    2.02%  G4Log.hh:244
+    1.93%  G4hPairProductionModel.cc:142
+    1.90%  G4hPairProductionModel.cc:157
+    1.88%  G4Log.hh:251
+    1.83%  G4Exp.hh:210
+    1.79%  G4Log.hh:256
+    1.45%  G4hPairProductionModel.cc:170
+    1.37%  G4Exp.hh:218
+    1.36%  G4Log.hh:148
+    1.30%  G4Log.hh:248
+    1.29%  G4Log.hh:117
+    1.20%  G4Log.hh:115
+    1.18%  G4hPairProductionModel.cc:140
+    1.18%  G4Log.hh:242
Tip: Use parent filter to see specific call path: perf report -p <regex>
```

Physics models call G4Log and G4Exp many times during initialization, so the results on the left are expected.
However, this is also an indication that there may be room for optimization in G4Log and G4Exp as well, since we see backend stall cycles without many L1 cache misses.
G4Log at least is also called many times in the event loop to fetch cross section data with energy interpolated as log(E).

DWARF also allows to sort by source line (2)

```
Samples: 1M of event 'cycles', Event count (approx.): 445734003655
  Overhead  Source:Line
-    1.65%  ThreeVector.icc:255
   - CLHEP::Hep3Vector::dot (inlined)
      + 0.64% G4PolyhedraSide::DistanceAway
      + 0.50% G4PolyhedraSide::DistanceToOneSide
      + 0.21% G4PolyhedraSide::Distance
        0.13% G4PolyhedraSide::Intersect
-    1.62%  stl_algobase.h:235
   - 1.61% std::min<double> (inlined)
      + 1.53% G4PhysicsVector::Interpolation (inlined)
-    1.59%  stl_algobase.h:259
   - 1.59% std::max<double> (inlined)
      + 1.17% G4PhysicsVector::LogVectorValue
      + 0.23% G4SteppingManager::CalculateSafety (inlined)
+    1.37%  ThreeVector.icc:75
-    1.23%  G4CrossSectionDataStore.cc:336
     G4CrossSectionDataStore::GetCrossSection
   - G4CrossSectionDataStore::ComputeCrossSection
      + 1.23% G4HadronicProcess::GetMeanFreePath
+    1.04%  fenv_private.h:416
+    0.92%  stl_vector.h:1046
+    0.92%  G4PolyhedraSide.cc:1165
Tip: Skip collecting build-id when recording: perf record -B
```

Same as before, but sampled in the event loop.

Data dependencies seem to be the culprit again

```
  G4double px = G4LogConsts::get_log_px(x);

  // for the final formula
  const G4double x2 = x * x;
  px *= x;
  px *= x2;

  const G4double qx = G4LogConsts::get_log_qx(x);

  G4double res = px / qx;

  res -= fe * 2.12194440054690582767 9e-4;
  res -= 0.5 * x2;

  res = x + res;
  res += fe * 0.693359375;

  if(original_x > G4LogConsts::LOG_UPPER_LIMIT)
    res = std::numeric_limits<G4double>::infinity();
  if(original_x < G4LogConsts::LOG_LOWER_LIMIT)  // THIS IS NAN!
    res = -std::numeric_limits<G4double>::quiet_NaN();

  return res;
"/srv/geant4/src/geant4-10.6.r9/source/global/management/include/G4Log.hh" 350 lines --71%--                250,3          72%
```

Looking at the code for G4Log, we see that the line with most stalls (G4Log.hh:250 in previous slide) is a line immediately using the result of a division after it is computed.

G4Log inlined many times, maybe that's a problem?

```
bash geant4-10.6.r9 $ git grep 'G4Log(' source/ | head
source/global/HEPNumerics/src/G4JTPolynomialSolver.cc:      l      = (G4int)(G4Log(sc) / G4Log(base) + 0.5);
source/global/HEPNumerics/src/G4JTPolynomialSolver.cc:    x = G4Exp((G4Log(-pt[n]) - G4Log(pt[0])) / (G4double) n);
source/global/management/include/G4Log.hh:inline G4double G4Log(G4double x)
source/global/management/include/G4PhysicsVector.icc:    bin = size_t(std::max(G4Log(theEnergy) * invdBin - baseBin, 0.0));
source/global/management/include/G4Pow.hh:    res = G4Log(a);
source/global/management/include/G4Pow.hh:    res = G4Log(a);
source/global/management/src/G4PhysicsLogVector.cc:  invdBin = 1. / (G4Log(theEmax / theEmin) / (G4double) (numberOfNodes - 1));
source/global/management/src/G4PhysicsLogVector.cc:  baseBin = G4Log(theEmin) * invdBin;
source/global/management/src/G4PhysicsLogVector.cc:    invdBin = 1. / G4Log(binVector[1] / edgeMin);
source/global/management/src/G4PhysicsLogVector.cc:    baseBin = G4Log(edgeMin) * invdBin;
bash geant4-10.6.r9 $ git grep -c 'G4Log(' source/ | head
source/global/HEPNumerics/src/G4JTPolynomialSolver.cc:2
source/global/management/include/G4Log.hh:1
source/global/management/include/G4PhysicsVector.icc:1
source/global/management/include/G4Pow.hh:2
source/global/management/src/G4PhysicsLogVector.cc:6
source/global/management/src/G4Pow.cc:4
source/materials/include/G4IonisParamMat.hh:1
source/materials/src/G4DensityEffectCalculator.cc:4
source/materials/src/G4Element.cc:3
source/materials/src/G4IonisParamMat.cc:11
bash geant4-10.6.r9 $ git grep -c 'G4Log(' source/processes/ | awk -F : '{sum += $2} END { print sum }'
932
```

G4Log inlined at least 932 times in physics processes.
Makes libG4processes.so 1–2% larger because of this.
(release ~300K larger / debug 10MB larger)

G4Log inlined many times, but it’s not a problem (yet)

G4Log function inlined

```
bash ~ $ perf stat -r 5 -- taskset -c 0 /srv/geant4/install/gcc-10.2.0/master-MT/bi
n/g4run -g /srv/geant4/gdml/CMS.gdml -p pythia:ttbar -e 0 2>&1 | cut -b -83

 Performance counter stats for 'taskset -c 0 /srv/geant4/install/gcc-10.2.0/master-

          17290.23 msec task-clock                #    0.961 CPUs utilized
               291      context-switches          #    0.017 K/sec
                 1      cpu-migrations            #    0.000 K/sec
            111152      page-faults               #    0.006 M/sec
       76427613719      cycles                    #    4.420 GHz
         448704342      stalled-cycles-frontend   #    0.59% frontend cycles idle
       45431920601      stalled-cycles-backend    #   59.44% backend cycles idle
      125558491994      instructions              #    1.64  insn per cycle
                                                  #    0.36  stalled cycles per ins
       17019334780      branches                  #  984.333 M/sec
         157955893      branch-misses             #    0.93% of all branches

            17.999 +- 0.392 seconds time elapsed  ( +-  2.18% )

bash ~ $ _
```

G4Log function *not* inlined

```
bash ~ $ perf stat -r 5 -- taskset -c 0 /srv/geant4/install/gcc-10.2.0/amadio-MT/bi
n/g4run -g /srv/geant4/gdml/CMS.gdml -p pythia:ttbar -e 0 2>&1 | cut -b -83

 Performance counter stats for 'taskset -c 0 /srv/geant4/install/gcc-10.2.0/amadio-

          17867.15 msec task-clock                #    0.982 CPUs utilized
               301      context-switches          #    0.017 K/sec
                 1      cpu-migrations            #    0.000 K/sec
            111092      page-faults               #    0.006 M/sec
       79215774857      cycles                    #    4.434 GHz
         440356271      stalled-cycles-frontend   #    0.56% frontend cycles idle
       44536128217      stalled-cycles-backend    #   56.22% backend cycles idle
      134984466012      instructions              #    1.70  insn per cycle
                                                  #    0.33  stalled cycles per ins
       19095299031      branches                  # 1068.738 M/sec
         166932047      branch-misses             #    0.87% of all branches

            18.1884 +- 0.0134 seconds time elapsed  ( +-  0.07% )

bash ~ $
```

No big difference, so problem is not due to code bloat

What happens if we use std::log and std::exp?

```
bash ~ $ perf stat -r 3 -d -- taskset -c 0 /srv/geant4/install/gcc-10.2.0/amadio-MT/bin/g4run -g ~/src/g4run/CMS.gdml -p pythia:
ttbar -e 0

 Performance counter stats for 'taskset -c 0 /srv/geant4/install/gcc-10.2.0/amadio-MT/bin/g4run -g /home/amadio/src/g4run/CMS.gd
ml -p pythia:ttbar -e 0' (3 runs):

         15773.13 msec task-clock                #    0.972 CPUs utilized           ( +-  0.08% )
             1284      context-switches          #    0.081 K/sec                   ( +- 40.16% )
                1      cpu-migrations            #    0.000 K/sec
           111287      page-faults               #    0.007 M/sec                   ( +-  0.05% )
      68324629875      cycles                    #    4.332 GHz                     ( +-  0.06% )  (75.04%)
        373986440      stalled-cycles-frontend   #    0.55% frontend cycles idle    ( +-  1.07% )  (74.95%)
      36522879398      stalled-cycles-backend    #   53.45% backend cycles idle     ( +-  0.26% )  (75.00%)
     137884394122      instructions              #    2.02  insn per cycle
                                                 #    0.26  stalled cycles per insn ( +-  0.05% )  (74.99%)
      19884639889      branches                  # 1260.666 M/sec                   ( +-  0.11% )  (74.98%)
         86643525      branch-misses             #    0.44% of all branches         ( +-  0.68% )  (74.98%)
      58388354221      L1-dcache-loads           # 3701.762 M/sec                   ( +-  0.08% )  (74.98%)
        694825388      L1-dcache-load-misses     #    1.19% of all L1-dcache accesses  ( +-  0.42% )  (75.09%)
   <not supported>      LLC-loads
   <not supported>      LLC-load-misses

          16.2350 +- 0.0809 seconds time elapsed  ( +-  0.50% )
```

Extra ~10% speedup! Could make sense to use std::log at initialization only.

A pinch of UNIX wisdom – on handling complexity

Rule 1 *You can't tell where a program is going to spend its time.* Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you've proven that's where the bottleneck is.

Rule 2 *Measure.* Don't tune for speed until you've measured, and even then don't unless one part of the code overwhelms the rest.

Rule 3 *Fancy algorithms are slow when **n** is small, and **n** is usually small.* Fancy algorithms have big constants. Until you know that **n** is frequently going to be big, don't get fancy. (Even if **n** does get big, use Rule 2 first.)

Rule 4 *Fancy algorithms are buggier than simple ones, and they're much harder to implement.* Use simple algorithms as well as simple data structures.

Rule 5 *Data dominates.* If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. *Data structures, not algorithms, are central to programming.*

Rule 6 There is no Rule 6.

from "Notes on C Programming", by Rob Pike

A pinch of UNIX wisdom – on handling complexity

Rule 1 ***You can't tell where a program is going to spend its time.*** **Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you've proven that's where the bottleneck is.**

Rule 2 ***Measure.*** **Don't tune for speed until you've measured, and even then don't unless one part of the code overwhelms the rest.**

Rule 3 *Fancy algorithms are slow when* ***n*** *is small, and* ***n*** *is usually small.* Fancy algorithms have big constants. Until you know that **n** is frequently going to be big, don't get fancy. (Even if **n** does get big, use Rule 2 first.)

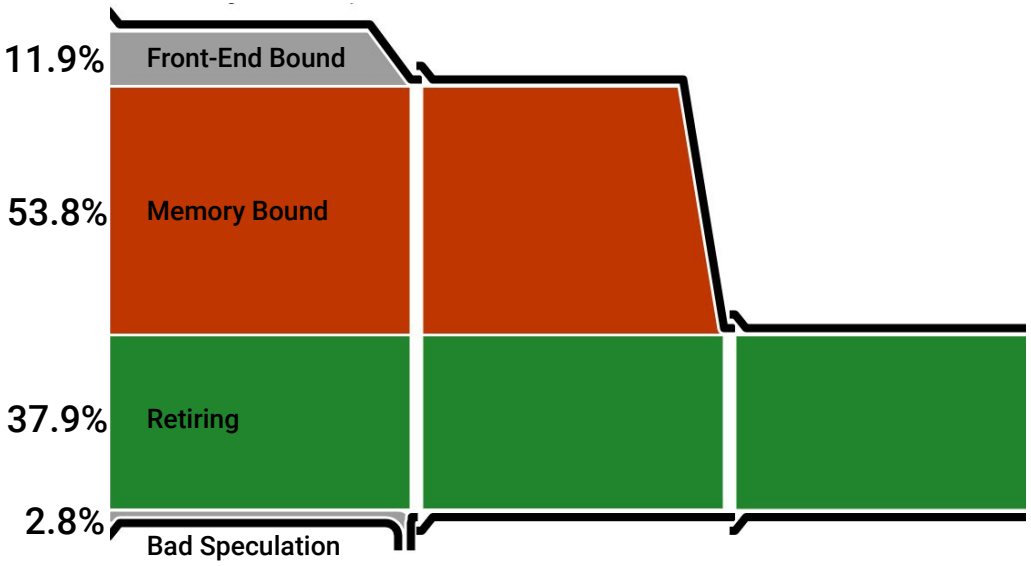
Rule 4 *Fancy algorithms are buggier than simple ones, and they're much harder to implement.* Use simple algorithms as well as simple data structures.

Rule 5 *Data dominates.* If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. *Data structures, not algorithms, are central to programming.*

Rule 6 There is no Rule 6.

from "Notes on C Programming", by Rob Pike

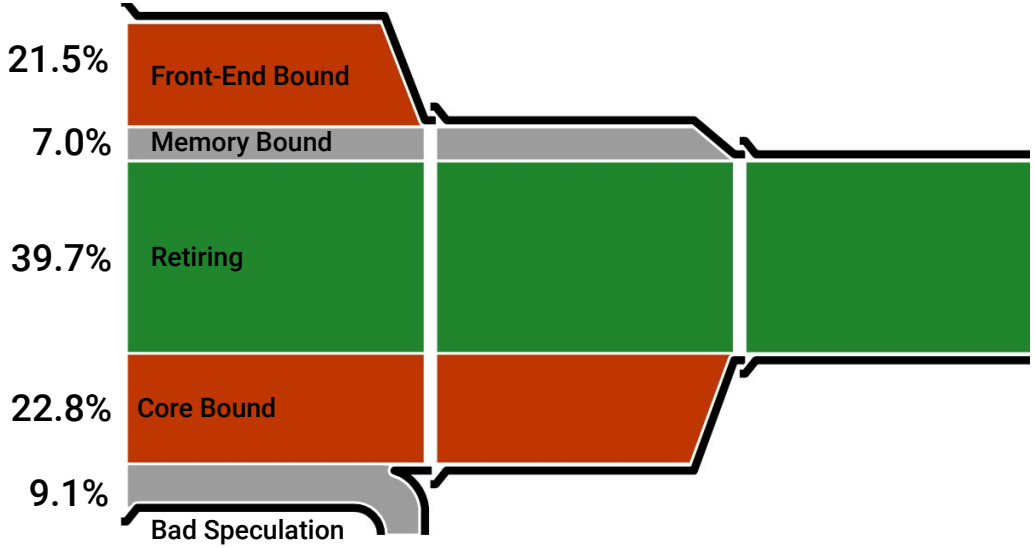
Geant4 Microarchitecture Usage on Haswell Server



Retiring:	37.9%	of Pipeline Slots
Front-End Bound:	11.9%	of Pipeline Slots
Bad Speculation:	2.8%	of Pipeline Slots
Back-End Bound:	47.3%	of Pipeline Slots
Memory Bound:	53.8%	of Pipeline Slots
L1 Bound:	48.9%	of Clockticks
DTLB Overhead:	13.3%	of Clockticks
Loads Blocked by Store Forwarding:	0.0%	of Clockticks
Lock Latency:	0.0%	of Clockticks
Split Loads:	0.0%	of Clockticks
4K Aliasing:	0.3%	of Clockticks
FB Full:	0.0%	of Clockticks
L2 Bound:	0.0%	of Clockticks
L3 Bound:	5.6%	of Clockticks
Contested Accesses:	0.0%	of Clockticks
Data Sharing:	0.0%	of Clockticks
L3 Latency:	6.9%	of Clockticks
SQ Full:	0.0%	of Clockticks
DRAM Bound:	0.0%	of Clockticks
Store Bound:	0.0%	of Clockticks
Core Bound:	0.0%	of Pipeline Slots

Mostly memory bound on Haswell

Geant4 Microarchitecture Usage on Skylake Desktop



Retiring:	39.7%	of Pipeline Slots
Front-End Bound:	21.5%	of Pipeline Slots
Bad Speculation:	9.1%	of Pipeline Slots
Back-End Bound:	29.8%	of Pipeline Slots
Memory Bound:	7.0%	of Pipeline Slots
Core Bound:	22.8%	of Pipeline Slots
Divider:	10.3%	of Clockticks
Port Utilization:	24.5%	of Clockticks
Cycles of 0 Ports Utilized:	1.9%	of Clockticks
Cycles of 1 Port Utilized:	17.1%	of Clockticks
Cycles of 2 Ports Utilized:	18.7%	of Clockticks
Cycles of 3+ Ports Utilized:	38.5%	of Clockticks
Vector Capacity Usage (FPU):	12.5%	

Mostly frontend and core bound on Skylake, quite different than Haswell

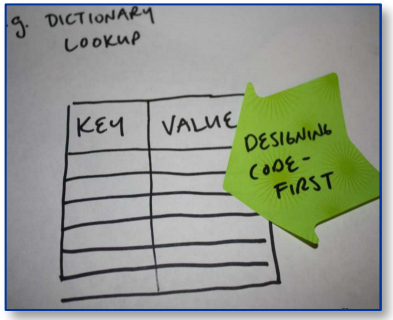
Conclusions and lessons learned

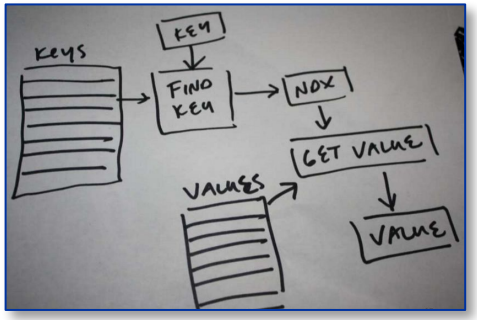
- Problems don't always happen where we expect
  - Always measure to make sure your hypothesis for the cause is correct
- The fastest thing you can do is to not do anything
  - Avoid unnecessary work in your code (e.g. checking field manager for neutral particles)
- Beware of data dependencies
  - Reoder computations to take advantage of instruction level parallelism
  - Strong dependencies can make your code slow even if L1 misses are low
- Beware of indirect accesses via pointers and calls to other shared objects
  - Patterns like `obj->GetFoo()->GetBar()->GetBaz()` are too common in C++
  - Accessing Baz becomes as expensive as traversing a list every time, bad for locality
  - Frequent calls across shared objects are expensive, it's better to merge into a single library

Data-Oriented Design

What is Data-Oriented Design?

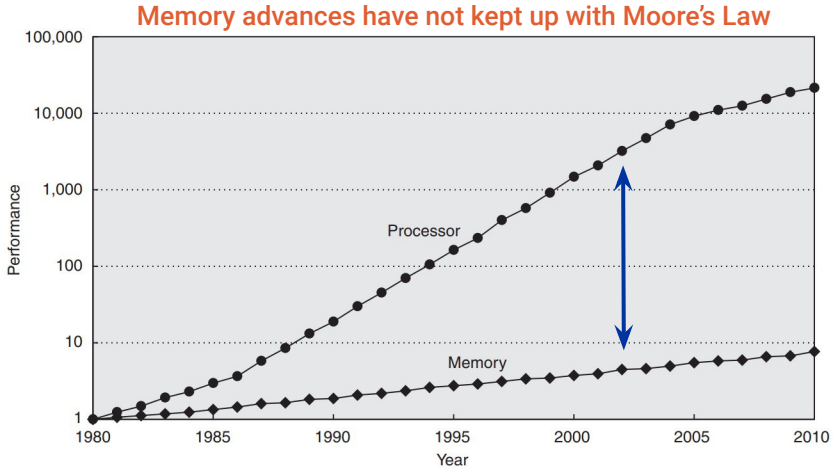
- CppCon 2014 talk by Mike Acton
- Reminder of first principles on how data structures influence final performance
- A program is something that transforms input data from one form to another
- If you don't understand the data, you don't understand the problem
- Different data ⇒ different problem
- Where there's one, there are many, optimize for the many
- You need to understand the hardware your software is running on

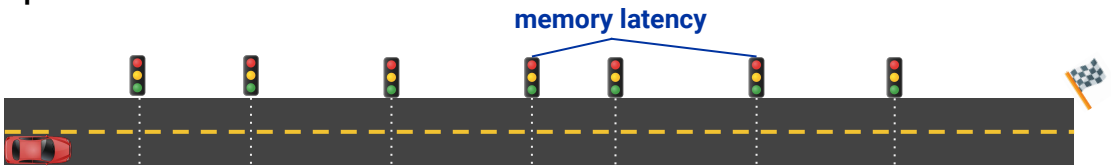




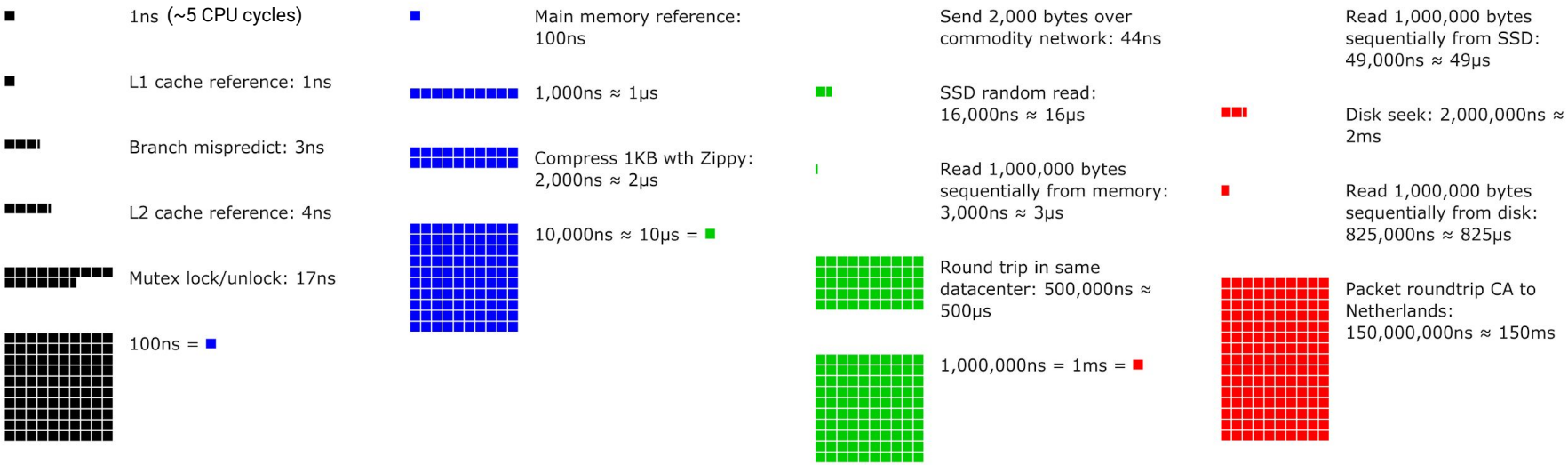
CPU vs Memory Performance Gap

- CPU performance has grown much faster than memory performance
- Memory access latency is often the main performance issue
- L3 cache has grown significantly, but L1 and L2 remain relatively small
- L2 cache misses are still expensive
- Ultimately, speed of light limits peak performance (c ≈ 30cm / ns)





Latency Numbers Every Programmer Should Know



Source: https://colin-scott.github.io/personal_website/research/interactive_latency.html

Object-Oriented vs Data-Oriented Design

Object-Oriented

- Polymorphism, abstract interfaces
  - Defer implementation to concrete types
- Classes, Inheritance, Encapsulation
  - Data available only via exposed interface
- Data and operations/behavior together
  - Extend data and behavior
  - Reuse code from parent classes
  - Less chance to optimize data layout
  - Higher demand on instruction cache
    - Many methods per object type

Data-Oriented

- Optimize memory access pattern
  - Make use of full cachelines
  - Use all available memory bandwidth
  - Spatial and temporal cache locality
- Data structures separate from code
  - Improves also front-end metrics
  - Move all movable objects together
  - Single function rather than methods
  - Commonly implemented as an Entity-Component-System in games

A pinch of UNIX wisdom – on handling complexity

Rule 1 *You can't tell where a program is going to spend its time.* Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you've proven that's where the bottleneck is.

Rule 2 *Measure.* Don't tune for speed until you've measured, and even then don't unless one part of the code overwhelms the rest.

Rule 3 *Fancy algorithms are slow when **n** is small, and **n** is usually small.* Fancy algorithms have big constants. Until you know that **n** is frequently going to be big, don't get fancy. (Even if **n** does get big, use Rule 2 first.)

Rule 4 *Fancy algorithms are buggier than simple ones, and they're much harder to implement.* Use simple algorithms as well as simple data structures.

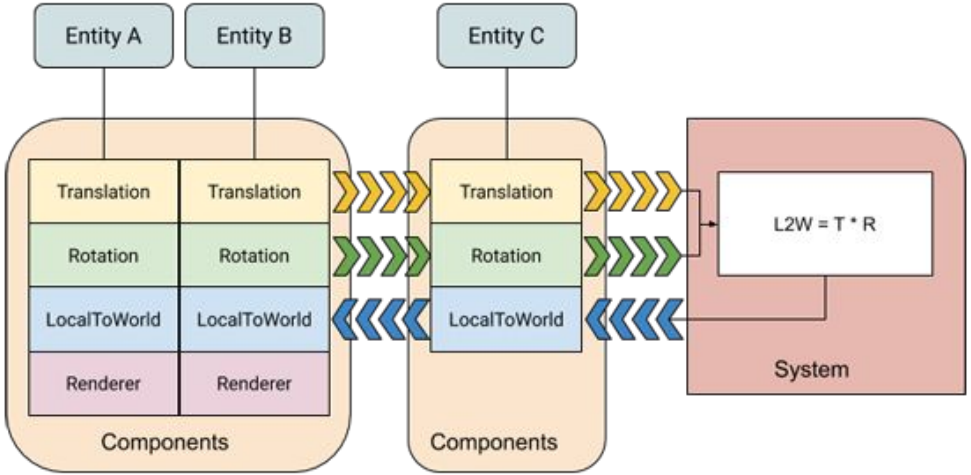
Rule 5 ***Data dominates.* If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. *Data structures, not algorithms, are central to programming.***

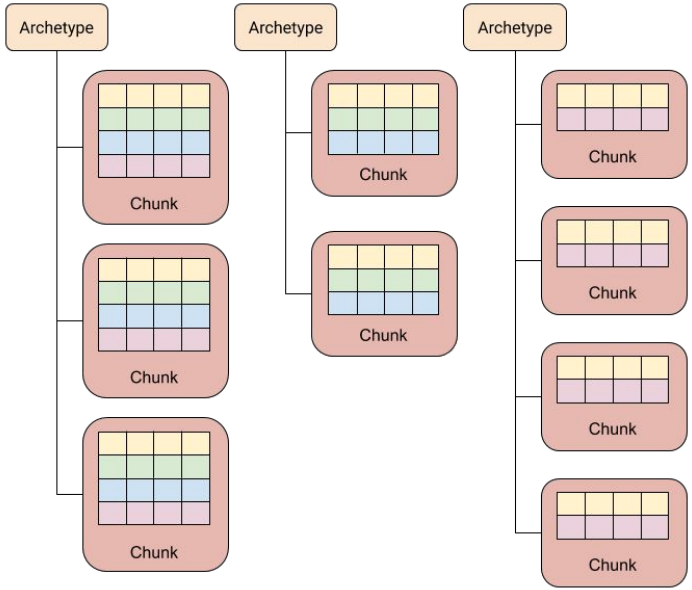
Rule 6 There is no Rule 6.

from "Notes on C Programming", by Rob Pike

Entity Component System Concepts

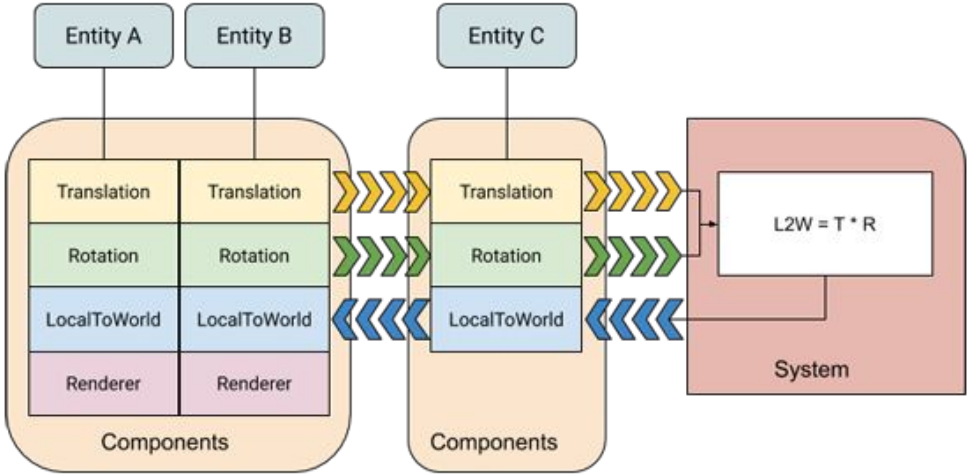
What they do it in computer games



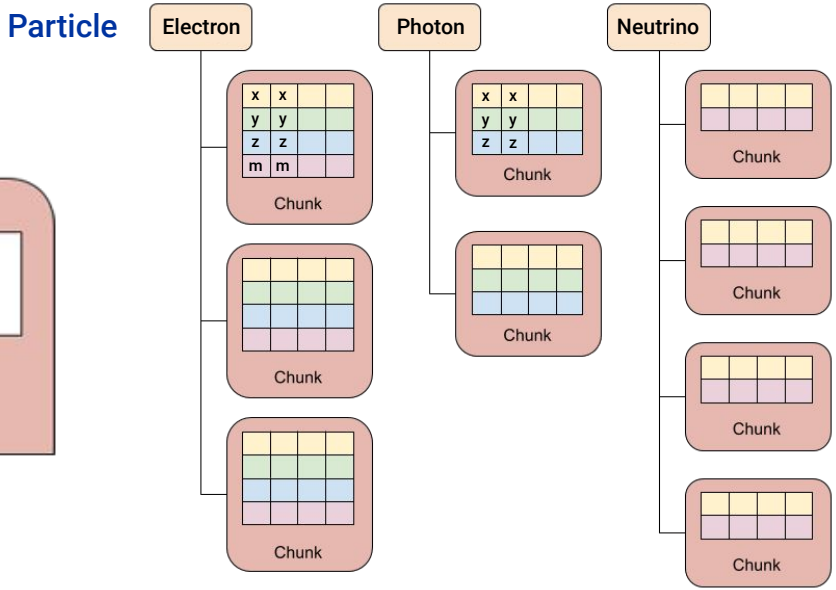


Entity Component System Concepts

What they do it in computer games



...could also be applied to detector simulation



Array of Structure vs Structure of Array Data Layouts

AoS Structure

```
struct track {
        /* event structure */
        int32_t id;
        int32_t parent;

        /* geometry data */
        float x;
        float y;
        float z;

        int32_t geometry_id;

        /* physics data */
        float vx;
        float vy;
        float vz;
        float E;
        ...
};
```

SoA Structure

```
template<unsigned int N>
struct TrackBlock {
  static constexpr unsigned int nElem = N;

  /* event structure */
  std::array<int32_t, nElem> id;
  std::array<int32_t, nElem> parent;

  /* geometry data */
  std::array<float, nElem> x;
  std::array<float, nElem> y;
  std::array<float, nElem> z;

  static constexpr mass = 0.511f;
  ...
};
```

Performance numbers for simple kinetic energy kernel

AoS Data Layout

```
Performance counter stats for 'AoS' (average of 20 runs):

          483.46 msec task-clock                #    0.999 CPUs utilized
               1      context-switches          #    0.001 K/sec
               0      cpu-migrations            #    0.000 K/sec
            1138      page-faults               #    0.002 M/sec
      1687395641      cycles                    #    3.490 GHz
       106037669      stalled-cycles-frontend   #    6.28% frontend cycles idle
      1283753418      stalled-cycles-backend    #   76.08% backend cycles idle
      1996927212      instructions              #    1.18  insn per cycle
                                                #    0.64  stalled cycles per insn
        19497066      branches                  #   40.328 M/sec
           40196      branch-misses             #    0.21% of all branches
       622021303      L1-dcache-loads           # 1286.609 M/sec
       123124358      L1-dcache-load-misses     #   19.79% of all L1-dcache accesses
        42493039      L1-icache-loads           #   87.894 M/sec
          149328      L1-icache-load-misses     #    0.35% of all L1-icache accesses
           31003      dTLB-loads                #    0.064 M/sec
            8598      dTLB-load-misses          #   27.73% of all dTLB cache accesses
              26      iTLB-loads                #    0.055 K/sec
              19      iTLB-load-misses          #   72.30% of all iTLB cache accesses

       0.484160 +- 0.000367 seconds time elapsed
```

SoA Data Layout

```
Performance counter stats for 'SoA' (average of 20 runs):

          118.78 msec task-clock                #    0.994 CPUs utilized
               1      context-switches          #    0.010 K/sec
               0      cpu-migrations            #    0.000 K/sec
            1138      page-faults               #    0.010 M/sec
       414521761      cycles                    #    3.490 GHz
       131647110      stalled-cycles-frontend   #   31.76% frontend cycles idle
       182606710      stalled-cycles-backend    #   44.05% backend cycles idle
       122945544      instructions              #    0.30  insn per cycle
                                                #    1.49  stalled cycles per insn
        17639259      branches                  #  148.508 M/sec
           38941      branch-misses             #    0.22% of all branches
        81264204      L1-dcache-loads           #  684.179 M/sec
        42555034      L1-dcache-load-misses     #   52.37% of all L1-dcache accesses
         8799361      L1-icache-loads           #   74.083 M/sec
           38984      L1-icache-load-misses     #    0.44% of all L1-icache accesses
           13098      dTLB-loads                #    0.110 M/sec
            6723      dTLB-load-misses          #   51.33% of all dTLB cache accesses
               5      iTLB-loads                #    0.040 K/sec
             112      iTLB-load-misses          # 2363.16% of all iTLB cache accesses

       0.119446 +- 0.000115 seconds time elapsed
```

About 4x faster for SoA. Much lower number of cycles, L1 loads, and TLB loads, easily auto-vectorized.

pahole – inspect layout of data structures

```
bash $ pahole -M -C G4LogicalVolume /usr/lib64/libG4geometry.so
class G4LogicalVolume {
public:

        int ()(void) * *            _vptr.G4LogicalVolume; /*     0     8 */
        static G4LVManager                  subInstanceManager; /*     0     0 */
        G4PhysicalVolumeList        fDaughters;            /*     8    24 */
        class G4String             fName;                  /*    32    32 */
        /* --- cacheline 1 boundary (64 bytes) --- */
        class G4UserLimits *        fUserLimits;           /*    64     8 */
        class G4SmartVoxelHeader * fVoxel;                 /*    72     8 */
        G4double                    fSmartless;            /*    80     8 */
        class G4Region *            fRegion;               /*    88     8 */
        G4double                    fBiasWeight;           /*    96     8 */
        class shared_ptr<const G4VisAttributes> fVisAttributes; /*   104    16 */
        class G4VSolid *            fSolid;                /*   120     8 */
        /* --- cacheline 2 boundary (128 bytes) --- */
        class G4VSensitiveDetector * fSensitiveDetector; /*   128     8 */
        class G4FieldManager *      fFieldManager;         /*   136     8 */
        class G4LVData *            lvdata;                /*   144     8 */
        G4int                       instanceID;            /*   152     4 */
        enum EVolume                fDaughtersVolumeType; /*   156     4 */
        G4bool                      fOptimise;             /*   160     1 */
        G4bool                      fRootRegion;           /*   161     1 */
        G4bool                      fLock;                 /*   162     1 */

        /* size: 168, cachelines: 3, members: 18, static members: 1 */
        /* padding: 5 */
        /* last cacheline: 40 bytes */
};
```

Members which are accessed most often are in 3 different cachelines.

Must load fName whenever checking fDaughters. Inefficient use of memory hierarchy due to mix of hot and cold data on each cacheline.

perf – memory access analysis: loads and stores

```
PERF-MEM(1)                                    perf Manual                                    PERF-MEM(1)

NAME
       perf-mem - Profile memory accesses

SYNOPSIS
       perf mem [<options>] (record [<command>] | report)

DESCRIPTION
       "perf mem record" runs a command and gathers memory operation data from it, into perf.data. Perf record options are
       accepted and are passed through.

       "perf mem report" displays the result. It invokes perf report with the right set of options to display a memory
       access profile. By default, loads and stores are sampled. Use the -t option to limit to loads or stores.

       Note that on Intel systems the memory latency reported is the use-latency, not the pure load (or store latency). Use
       latency includes any pipeline queueing delays in addition to the memory subsystem latency.

       On Arm64 this uses SPE to sample load and store operations, therefore hardware and kernel support is required. See
       perf-arm-spe(1) for a setup guide. Due to the statistical nature of SPE sampling, not every memory operation will be
       sampled.

OPTIONS
 Manual page perf-mem(1) line 1 (press h for help or q to quit)
```

perf – memory access analysis: ROOT RDataFrame

```
bash df102_NanoAODDimuonAnalysis $ perf mem record -t load -F 5000 df102_NanoAODDimuonAnalysis 8 Run2012B_DoubleMuParked.root Ru
n2012C_DoubleMuParked.root
Couldn't synthesize cgroup events.
[ perf record: Woken up 130 times to write data ]
[ perf record: Captured and wrote 33.940 MB perf.data (550719 samples) ]
bash df102_NanoAODDimuonAnalysis $ perf mem report -s mem -q --stdio 2>/dev/null
    38.64%        250297  L1 or L1 hit
    20.24%        121901  L3 or L3 hit
    18.59%        106253  LFB or LFB hit
    15.63%         50825  Remote Remote Cache (1 hop) or L3 hit
     4.22%          9863  L3 miss
     1.45%          6289  Local RAM or RAM hit
     0.99%          3978  Remote Remote RAM (1 hop) or RAM hit
     0.24%          1310  L2 or L2 hit
     0.00%             2  Uncached or N/A hit
     0.00%             1  I/O or N/A hit


bash df102_NanoAODDimuonAnalysis $ _
```

Line Fill Buffer (sits between L1 and L2)

Red flag, too many remote cache accesses

See also: https://community.intel.com/t5/Intel-Moderncode-for-Parallel/What-is-the-aim-of-the-line-fill-buffer/td-p/1180777

Using perf to measure average load latency

```
bash df102_NanoAODDimuonAnalysis $ perf stat -M Load_Miss_Real_Latency -- df102_NanoAODDimuonAnalysis 8 Run2012B_DoubleMuParked.
root Run2012C_DoubleMuParked.root

 Performance counter stats for 'df102_NanoAODDimuonAnalysis 8 Run2012B_DoubleMuParked.root Run2012C_DoubleMuParked.root':

       970,326,211      mem_load_uops_retired.hit_lfb #   188.23 Load_Miss_Real_Latency
   438,536,118,970      l1d_pend_miss.pending
     1,359,436,464      mem_load_uops_retired.l1_miss

      20.844816183 seconds time elapsed

     148.564065000 seconds user
       2.591617000 seconds sys


bash df102_NanoAODDimuonAnalysis $ perf list mem_load_uops_retired.hit_lfb | uniq

cache:
  mem_load_uops_retired.hit_lfb
       [Retired load uops which data sources were load uops missed L1 but hit
        FB due to preceding miss to the same cache line with data not ready
        Supports address when precise. Spec update: HSM30 (Precise event)]

Metric Groups:
```

in CPU cycles

Avg. load latency = L1 miss pending cycles / (L1 misses + LFB hits)

perf mem report -s mem

```
Samples: 543K of event 'cpu/mem-loads,ldlat=50/P', Event count (approx.): 172369311
  Overhead       Samples  Memory access
-   41.51%        226882  L1 or L1 hit
   + 17.49% 0xffffffffffffffff
   - 12.00% __GI___clone (inlined)
      - 12.00% start_thread
         - 12.00% tbb::internal::rml::private_worker::thread_routine
            - tbb::internal::rml::private_worker::run
               - 11.99% tbb::internal::market::process
                    tbb::internal::arena::process
                    tbb::internal::custom_scheduler<tbb::internal::IntelSchedulerTraits>::local_wait_for_all
                    tbb::internal::custom_scheduler<tbb::internal::IntelSchedulerTraits>::process_bypass_loop
                  - tbb::interface9::internal::start_for<tbb::blocked_range<unsigned int>, tbb::internal::parallel_for_body<std
                     - 10.39% 0x7f4e4344317f
                        - 10.18% 0x7f4e42dffa42
                           - ROOT::Detail::RDF::RLoopManager::RunAndCheckFilters
                              + 8.11% ROOT::Internal::RDF::RActionCRTP<ROOT::Internal::RDF::RAction<ROOT::Internal::RDF::FillPa
   + 2.10% 0x55d581065ec7
+   18.96%        113828  LFB or LFB hit
+   17.36%        114120  L3 or L3 hit
+   13.12%         59491  Remote Remote Cache (1 hop) or L3 hit
+    5.86%         14681  N/A miss

Cannot load tips.txt file, please install perf!
```

perf mem report -s dso,symbol

```
Samples: 543K of event 'cpu/mem-loads,ldlat=50/P', Event count (approx.): 172369311
  Overhead       Samples  Shared Object                Symbol
+   27.12%        131151  df102_NanoAODDimuonAnalysis  [.] ROOT::Detail::RDF::RFilter<bool (*)(unsigned int), ROOT::Detail::RDF:
+   17.33%         66980  libTree.so.6.22.02           [.] TBranch::GetBasketAndFirst
+   13.86%         62434  df102_NanoAODDimuonAnalysis  [.] ROOT::Detail::RDF::RFilter<bool (*)(ROOT::VecOps::RVec<int> const&),
+    5.51%         32071  libTreePlayer.so.6.22.02     [.] ROOT::Internal::TTreeReaderValueBase::ProxyReadTemplate<&ROOT::Detail
+    5.33%         20471  libROOTDataFrame.so.6.22.02  [.] ROOT::Detail::RDF::RLoopManager::RunAndCheckFilters
+    5.07%         21584  libTree.so.6.22.02           [.] TBranch::GetEntry
+    3.66%         24977  df102_NanoAODDimuonAnalysis  [.] ROOT::Internal::RDF::RColumnValue<ROOT::VecOps::RVec<float> >::Get<RO
     2.76%         16451  libTreePlayer.so.6.22.02     [.] ROOT::Internal::TTreeReaderValueBase::GetAddress
+    2.48%         18397  df102_NanoAODDimuonAnalysis  [.] ROOT::Internal::RDF::RColumnValue<ROOT::VecOps::RVec<int> >::Get<ROOT
+    2.28%         17109  df102_NanoAODDimuonAnalysis  [.] std::swap<ROOT::VecOps::RVec<int> >
+    2.15%         16151  df102_NanoAODDimuonAnalysis  [.] std::swap<ROOT::VecOps::RVec<float> >
+    2.02%         16377  df102_NanoAODDimuonAnalysis  [.] ROOT::Detail::RDF::RCustomColumn<float (*)(ROOT::VecOps::RVec<float>
+    1.72%         12478  df102_NanoAODDimuonAnalysis  [.] ROOT::Internal::RDF::RActionCRTP<ROOT::Internal::RDF::RAction<ROOT::I
     1.23%          4550  [kernel.kallsyms]            [k] copy_user_enhanced_fast_string
+    1.23%          5657  df102_NanoAODDimuonAnalysis  [.] ROOT::Detail::RDF::RLoopManager::CheckFilters@plt
+    0.88%          5830  libRIO.so.6.22.02            [.] TBufferFile::ReadFastArray
+    0.67%         10988  df102_NanoAODDimuonAnalysis  [.] ROOT::VecOps::InvariantMass<float>
     0.58%          8216  libHist.so.6.22.02           [.] TH1::Fill
     0.49%          3864  libTreePlayer.so.6.22.02     [.] 0x00000000000df63f
     0.36%          7884  libz.so.1.2.11               [.] inflate_table
     0.32%          4976  libRIO.so.6.22.02            [.] TBufferFile::ReadArray
```

~46% of high latency loads happen in functions related to RDF filters...

perf mem report – symbol annotation

```
Samples: 543K of event 'cpu/mem-loads,ldlat=50/P', 5000 Hz, Event count (approx.): 172369311
ROOT::Detail::RDF::RFilter<bool (*)(unsigned int), ROOT::Detail::RDF::RLoopManager>::CheckFilters  /nvme/amadio/df102_NanoAODDim
Percent │       add    $0x8,%rsp
  0.15  │       pop    %rbx
  0.68  │       pop    %rbp
  1.23  │       pop    %r12
  0.04  │       pop    %r13
  0.27  │       pop    %r14
  0.01  │       pop    %r15
        │     ← retq
        │       nop
  0.00  │       mov    %rdi,%rbx
        │     if (!fPrevData.CheckFilters(slot, entry)) {
 15.13  │       mov    0xf0(%rdi),%rdi
        │       mov    %rdx,%rbp
        │     → callq  ROOT::Detail::RDF::RLoopManager::CheckFilters@plt
        │       test   %al,%al
        │     → jne    26d60 <ROOT::Detail::RDF::RFilter<bool (*)(unsigned int), ROOT::Detail::RDF::RLoopManager>::CheckFilters(u
        │     fLastResult[slot] = false;
        │       mov    0x30(%rbx),%rdx
        │       movl   $0x0,(%rdx,%r13,4)
        │     fLastCheckedEntry[slot] = entry;
  5.40  │       mov    0x18(%rbx),%rdx
Press 'h' for help on key bindings
```

per-thread values in contiguous memory

perf mem report -s phys_daddr

```
Samples: 543K of event 'cpu/mem-loads,ldlat=50/P', Event count (approx.): 172369311
  Overhead       Samples  Data Physical Address
-    4.00%         20056  [.] 0x0000000e4fff6e08
     3.53% __GI___clone (inlined)
        start_thread
        tbb::internal::rml::private_worker::thread_routine
        tbb::internal::rml::private_worker::run
        tbb::internal::market::process
        tbb::internal::arena::process
        tbb::internal::custom_scheduler<tbb::internal::IntelSchedulerTraits>::local_wait_for_all
        tbb::internal::custom_scheduler<tbb::internal::IntelSchedulerTraits>::process_bypass_loop
      - tbb::interface9::internal::start_for<tbb::blocked_range<unsigned int>, tbb::internal::parallel_for_body<std::function<v
         - 3.05% 0x7f4e4344317f
              0x7f4e42dffa42
            - ROOT::Detail::RDF::RLoopManager::RunAndCheckFilters
               + 2.20% ROOT::Internal::RDF::RActionCRTP<ROOT::Internal::RDF::RAction<ROOT::Internal::RDF::FillParHelper<TH1D>,
                 0.86% ROOT::Detail::RDF::RFilter<bool (*)(unsigned int), ROOT::Detail::RDF::RLoopManager>::CheckFilters
+    3.75%          9553  [.] 0x0000000e4fff6df0
+    2.68%         12526  [.] 0x0000000e4fff6ee0
+    2.46%         12056  [.] 0x0000000e4fff6e20
+    2.10%         11397  [.] 0x0000000e4fff6ee8
+    2.06%         10564  [.] 0x0000000e4fff6eb0
+    1.48%          9597  [.] 0x000000058bb3c248
Press '?' for help on key bindings
```

high load latency happens in nearby addresses

perf c2c – cache to cache analysis

```
PERF-C2C(1)                                    perf Manual                                    PERF-C2C(1)

NAME
       perf-c2c - Shared Data C2C/HITM Analyzer.

SYNOPSIS
       perf c2c record [<options>] <command>
       perf c2c record [<options>] -- [<record command options>] <command>
       perf c2c report [<options>]

DESCRIPTION
       C2C stands for Cache To Cache.

       The perf c2c tool provides means for Shared Data C2C/HITM analysis. It allows you to track down the cacheline
       contentions.

       On x86, the tool is based on load latency and precise store facility events provided by Intel CPUs. On PowerPC, the
       tool uses random instruction sampling with thresholding feature.

       These events provide: - memory address of the access - type of the access (load and store details) - latency (in
       cycles) of the load access

       The c2c tool provide means to record this data and report back access details for cachelines with highest contention
       - highest number of HITM accesses.
```

See also: https://hpc-wiki.info/hpc/FalseSharing

perf c2c report

Shared Data Cache Line Table (44 entries, sorted on Total HITMs)

	Index	Cacheline Address	Node	PA cnt	Total records	Tot Hitm	LLC Load Hitm Total	Lcl	Rmt	Store Reference Total	L1Hit	L1Miss	Load Dram Lcl
+	0	0x561f510789c0	0-1	7463	13115	17.40%	5172	1368	3804	11	3	9	18
+	1	0x561f518b1740	0	5382	8536	12.44%	3698	1593	2105	1747	836	911	21
+	2	0x561f51078a00	0-1	6447	10046	10.42%	3099	892	2207	27	3	24	64
+	3	0x561f518b1e40	0	5749	8859	9.96%	2962	801	2161	181	18	163	7
+	4	0x561f518afdc0	0-1	3304	5386	7.34%	2182	738	1444	456	45	411	16
+	5	0x561f518b1a40	0	4888	7554	7.25%	2154	721	1433	2927	1244	1683	16
+	6	0x561f518b1b80	0	2923	4846	6.74%	2004	617	1387	36	1	35	19
+	7	0x561f518b1800	0	3320	5532	6.39%	1900	544	1356	1333	591	742	10
+	8	0x561f51078a40	0-1	3543	5902	4.45%	1322	352	970	21	5	16	9
+	9	0x561f518b1ac0	0	1446	2186	1.30%	387	73	314	868	327	541	16
+	10	0x561f518b2640	0-1	1364	2150	1.09%	324	10	314	290	54	236	3
+	11	0x561f518b2580	0-1	1711	2348	0.94%	279	181	98	402	105	297	4
+	12	0x561f518b1a80	0	510	1815	0.83%	247	9	238	734	148	586	3
+	13	0x561f518b24c0	0-1	1113	1535	0.82%	245	145	100	278	88	190	4
+	14	0x561f518b34c0	0-1	991	1990	0.66%	196	1	195	1108	1106	2	5
+	15	0x561f518b1840	0	347	1244	0.57%	170	7	163	346	66	280	2
+	16	0x561f518b2540	0-1	1358	2220	0.57%	170	0	170	906	904	2	3
+	17	0x561f518b2680	0-1	1723	2943	0.54%	160	1	159	1668	1470	198	6
+	18	0x561f518b3040	0-1	866	1830	0.50%	148	1	147	1050	1050	0	1
+	19	0x561f518b1ec0	0	17115	30175	0.49%	147	23	124	0	0	0	400

? - help

perf c2c report

```
Shared Data Cache Line Table     (44 entries, sorted on Total HITMs)
         ----------- Cacheline ----------      Total      Tot  ----- LLC Load Hitm -----  ---- Store Reference ----  --- Load Dram
  Index             Address  Node  PA cnt  records     Hitm    Total      Lcl      Rmt    Total    L1Hit   L1Miss       Lcl
-     0      0x561f510789c0   0-1    7463    13115   17.40%     5172     1368     3804       11        3        9        18
    + 0.00% ROOT::Detail::RDF::RFilter<bool (*)(unsigned int), ROOT::Detail::RDF::RLoopManager>::CheckFilters
    + 0.00% ROOT::Detail::RDF::RLoopManager::CheckFilters@plt
      0.00% apic_timer_interrupt
+     1      0x561f518b1740      0    5382     8536   12.44%     3698     1593     2105     1747      836      911        21
+     2      0x561f51078a00   0-1    6447    10046   10.42%     3099      892     2207       27        3       24        64
+     3      0x561f518b1e40      0    5749     8859    9.96%     2962      801     2161      181       18      163         7
+     4      0x561f518afdc0   0-1    3304     5386    7.34%     2182      738     1444      456       45      411        16
+     5      0x561f518b1a40      0    4888     7554    7.25%     2154      721     1433     2927     1244     1683        16
+     6      0x561f518b1b80      0    2923     4846    6.74%     2004      617     1387       36        1       35        19
+     7      0x561f518b1800      0    3320     5532    6.39%     1900      544     1356     1333      591      742        10
+     8      0x561f51078a40   0-1    3543     5902    4.45%     1322      352      970       21        5       16         9
+     9      0x561f518b1ac0      0    1446     2186    1.30%      387       73      314      868      327      541        16
+    10      0x561f518b2640   0-1    1364     2150    1.09%      324       10      314      290       54      236         3
+    11      0x561f518b2580   0-1    1711     2348    0.94%      279      181       98      402      105      297         4
+    12      0x561f518b1a80      0     510     1815    0.83%      247        9      238      734      148      586         3
+    13      0x561f518b24c0   0-1    1113     1535    0.82%      245      145      100      278       88      190         4
+    14      0x561f518b34c0   0-1     991     1990    0.66%      196        1      195     1108     1106        2         5
+    15      0x561f518b1840      0     347     1244    0.57%      170        7      163      346       66      280         2
+    16      0x561f518b2540   0-1    1358     2220    0.57%      170        0      170      906      904        2         3
? - help
```

perf c2c report – cacheline details (press ‘d’)

Cacheline 0x561f510789c0

	----- HITM -----		-- Store Refs --		------- CL --------				---------- cycles ----------			Total	cpu
	Rmt	Lcl	L1 Hit	L1 Miss	Off	Node	PA cnt	Code address	rmt hitm	lcl hitm	load	records	cnt
+	8.39%	9.14%	0.00%	0.00%	0x10	0-1	3	0x561f4ebc2470	408	284	339	1002	2
+	4.02%	2.63%	0.00%	0.00%	0x10	0-1	3	0x561f4ebd0d23	515	327	320	540	2
+	1.74%	1.90%	0.00%	0.00%	0x10	0-1	3	0x561f4ebd0cff	464	314	322	247	2
+	1.24%	2.27%	0.00%	0.00%	0x10	0-1	3	0x561f4ebd0d0e	360	241	309	173	2
+	1.05%	0.73%	0.00%	0.00%	0x10	0-1	3	0x561f4ebd0cfb	641	241	656	130	2
+	0.34%	0.22%	0.00%	0.00%	0x10	0-1	3	0x561f4ebd0d05	337	164	364	48	2
+	0.26%	0.29%	0.00%	0.00%	0x10	0-1	3	0x561f4ebd0d12	332	173	202	39	2
+	0.11%	0.07%	0.00%	0.00%	0x10	0-1	3	0x561f4ebd0d10	306	150	118	17	2
+	0.03%	0.00%	33.33%	11.11%	0x10	1	1	0x561f4ebd0d20	312	0	0	4	1
+	0.00%	0.00%	0.00%	33.33%	0x10	0-1	2	0x561f4ebd0d46	0	0	0	3	1
+	7.33%	8.48%	0.00%	0.00%	0x18	0-1	3	0x561f4ebc2470	389	283	343	934	5
+	3.52%	3.36%	0.00%	0.00%	0x18	0-1	3	0x561f4ebd0d23	467	372	247	529	4
+	1.42%	1.54%	0.00%	0.00%	0x18	0-1	3	0x561f4ebd0cff	404	381	330	216	4
+	1.52%	0.80%	0.00%	0.00%	0x18	0-1	3	0x561f4ebd0d0e	394	181	304	178	4
+	1.13%	0.58%	0.00%	0.00%	0x18	0-1	3	0x561f4ebd0cfb	466	293	699	129	4
+	0.21%	0.37%	0.00%	0.00%	0x18	0-1	3	0x561f4ebd0d05	349	299	435	37	4
+	0.18%	0.22%	0.00%	0.00%	0x18	0-1	2	0x561f4ebd0d12	301	176	208	26	3
+	0.08%	0.00%	0.00%	0.00%	0x18	0-1	2	0x561f4ebd0d10	296	0	220	12	2
+	0.03%	0.00%	0.00%	0.00%	0x18	0-1	2	0x561f4ebd0d03	275	0	440	3	2
+	0.00%	0.00%	33.33%	0.00%	0x18	0	1	0x561f4ebd0d20	0	0	0	1	1

? - help

perf c2c report – cacheline details, more columns

```
Cacheline 0x561f510789c0
                Total       cpu                                                  Shared
      load   records       cnt                          Symbol                    Object                    Source:Line  Node
+      339      1002         2  [.] ROOT::Detail::RDF::RLoopMa  df102_NanoAODDimuonAnalysis  ROOT::Detail::RDF::RLoopManage   1
+      320       540         2  [.] ROOT::Detail::RDF::RFilter  df102_NanoAODDimuonAnalysis  RFilter.hxx:87                   1
+      322       247         2  [.] ROOT::Detail::RDF::RFilter  df102_NanoAODDimuonAnalysis  RFilter.hxx:86                   1
+      309       173         2  [.] ROOT::Detail::RDF::RFilter  df102_NanoAODDimuonAnalysis  RFilter.hxx:99                   1
+      656       130         2  [.] ROOT::Detail::RDF::RFilter  df102_NanoAODDimuonAnalysis  RFilter.hxx:86                   1
+      364        48         2  [.] ROOT::Detail::RDF::RFilter  df102_NanoAODDimuonAnalysis  RFilter.hxx:86                   1
+      202        39         2  [.] ROOT::Detail::RDF::RFilter  df102_NanoAODDimuonAnalysis  RFilter.hxx:99                   1
+      118        17         2  [.] ROOT::Detail::RDF::RFilter  df102_NanoAODDimuonAnalysis  RFilter.hxx:99                   1
+        0         4         1  [.] ROOT::Detail::RDF::RFilter  df102_NanoAODDimuonAnalysis  RFilter.hxx:99                   1
+        0         3         1  [.] ROOT::Detail::RDF::RFilter  df102_NanoAODDimuonAnalysis  RFilter.hxx:96                   1
+      343       934         5  [.] ROOT::Detail::RDF::RLoopMa  df102_NanoAODDimuonAnalysis  ROOT::Detail::RDF::RLoopManage   0
+      247       529         4  [.] ROOT::Detail::RDF::RFilter  df102_NanoAODDimuonAnalysis  RFilter.hxx:87                   0
+      330       216         4  [.] ROOT::Detail::RDF::RFilter  df102_NanoAODDimuonAnalysis  RFilter.hxx:86                   0
+      304       178         4  [.] ROOT::Detail::RDF::RFilter  df102_NanoAODDimuonAnalysis  RFilter.hxx:99                   0
+      699       129         4  [.] ROOT::Detail::RDF::RFilter  df102_NanoAODDimuonAnalysis  RFilter.hxx:86                   0
+      435        37         4  [.] ROOT::Detail::RDF::RFilter  df102_NanoAODDimuonAnalysis  RFilter.hxx:86                   0
+      208        26         3  [.] ROOT::Detail::RDF::RFilter  df102_NanoAODDimuonAnalysis  RFilter.hxx:99                   0
+      220        12         2  [.] ROOT::Detail::RDF::RFilter  df102_NanoAODDimuonAnalysis  RFilter.hxx:99                   0
+      440         3         2  [.] ROOT::Detail::RDF::RFilter  df102_NanoAODDimuonAnalysis  RFilter.hxx:86                   0
+        0         1         1  [.] ROOT::Detail::RDF::RFilter  df102_NanoAODDimuonAnalysis  RFilter.hxx:99                   1
? - help
```

perf c2c report – cacheline details, expanded stack

```
Cacheline 0x561f510789c0
  ----- HITM -----  -- Store Refs --  ------- CL --------                    ---------- cycles ----------    Total       cpu
     Rmt      Lcl   L1 Hit  L1 Miss    Off  Node  PA cnt        Code address  rmt hitm  lcl hitm      load  records       cnt
+   0.00%    0.00%    0.00%   11.11%   0x18     0       1       0x561f4ebd0d46         0         0         0        1         1
-   7.23%    7.16%    0.00%    0.00%   0x20   0-1       3       0x561f4ebc2470       413       257       333      891         3
     ROOT::Detail::RDF::RLoopManager::CheckFilters@plt
     ROOT::Detail::RDF::RFilter<bool (*)(unsigned int), ROOT::Detail::RDF::RLoopManager>::CheckFilters
     ROOT::Detail::RDF::RFilter<bool (*)(ROOT::VecOps::RVec<int> const&), ROOT::Detail::RDF::RFilter<bool (*)(unsigned int), RO
     ROOT::Internal::RDF::RActionCRTP<ROOT::Internal::RDF::RAction<ROOT::Internal::RDF::FillParHelper<TH1D>, ROOT::Detail::RDF:◆
     ROOT::Detail::RDF::RLoopManager::RunAndCheckFilters
     0x7feaa2613a42
     0x7feaa2c5717f
     tbb::interface9::internal::start_for<tbb::blocked_range<unsigned int>, tbb::internal::parallel_for_body<std::function<void
     tbb::internal::custom_scheduler<tbb::internal::IntelSchedulerTraits>::process_bypass_loop
     tbb::internal::custom_scheduler<tbb::internal::IntelSchedulerTraits>::local_wait_for_all
     tbb::internal::arena::process
     tbb::internal::market::process
     tbb::internal::rml::private_worker::run
     tbb::internal::rml::private_worker::thread_routine
     start_thread
     __GI___clone (inlined)
+   3.63%    3.36%    0.00%    0.00%   0x20   0-1       3       0x561f4ebd0d23       469       375       334      501         3
+   1.42%    2.56%    0.00%    0.00%   0x20   0-1       3       0x561f4ebd0d0e       353       217       282      222         3
? - help
```

Data-Type Profiling

- Introduced in latest versions of perf
  - First available in perf 6.8
  - Better with perf 6.10 and later
- Data recorded with `perf mem record`
- Attributes sample addresses back to the data-type associated with that address
- Two methods to make use of it
  - `perf mem report -s type,typeoff`
  - `perf annotate --data-type=<type>`
- Still needs some help for pointer types
  - Part of LLVM's `-fdebug-info-for-profiling`

Data profiling of git fsck on Linux kernel repository

```
linux $ perf mem record --ldlat=50 -- git fsck # collect loads with latency > 50 cycles
Checking objects: 100% (10271178/10271178), done.
Checking connectivity: 10223233, done.
[ perf record: Woken up 395 times to write data ]
[ perf record: Captured and wrote 99.888 MB perf.data (1629334 samples) ]
linux $ perf mem report --stdio -s type,typeoff --percent-limit 1
# Overhead       Samples  Data Type  Data Type Offset
# ........  ............  .........  ................
    83.88%       1450665  (unknown)  (unknown) +0 (no field)
     4.30%        100109  (stack operation)  (stack operation) +0 (no field)
     3.79%          3712  struct malloc_chunk  struct malloc_chunk +8 (mchunk_size)
     1.40%          2336  struct object  struct object +0 (parsed)
     1.08%           842  struct malloc_chunk  struct malloc_chunk +32 (fd_nextsize)
linux $ perf annotate --data-type=object
Annotate type: 'struct object' in /usr/libexec/git-core/git (3282 samples):
==============================================================================
    samples     offset       size  field
        104          0         40  struct object         {
          0          0          4      unsigned int     parsed;
          0          0          4      unsigned int     type;
          0          0          4      unsigned int     flags;
        104          4         36      struct object_id oid {
        104          4         32          unsigned char*       hash;
          0         36          4          int  algo;
                                         };
                                     };
```

See also: https://lwn.net/Articles/955709/

Packing Simulation Revisited

```
$ perf record -e cycles -F 1000 -- pack -f 0.6 ellipsoids
100.00% 0.6000 0.0000/min 2.6e-02 ev/s 38.2 s
[ perf record: Woken up 5 times to write data ]
[ perf record: Captured 1.457 MB perf.data (38005 samples) ]
$ perf report --stdio --percent-limit 0.25
# Overhead  Command  Shared Object Symbol
# ........  .......  ............. .........................
#
    39.53%  pack     pack          intersect
    32.60%  pack     pack          HGrid::find_neighbors
     7.62%  pack     pack          Ellipsoid::support
     3.76%  pack     pack          closest_point_tetrahedron
     3.37%  pack     pack          closest_point_triangle
     2.77%  pack     pack          Simplex::closest
     2.46%  pack     pack          Ellipsoid::bounding_radius
     2.07%  pack     pack          check_overlap
     1.94%  pack     pack          Simplex::contains
     1.15%  pack     libm.so.6     __sincos
     0.37%  pack     pack          HGrid::make_hash
     0.36%  pack     pack          HGrid::insert
```

```
void HGrid::find_neighbors(const Particle* p, std::vector<Particle*>& neighbors) {
  hash_t hash; unsigned int mask = occupied_level_mask;
  Vector x = p->position();

  for (unsigned int level = 0; level <= MAX_LEVEL; mask >>=1, level++) {
    if (mask == 0) return; /* no more occupied levels to check */
    if ((mask & 1) == 0) continue; /* level is not occupied */

    float cell_size = MAX_CELL_SIZE / (1 << level);
    float inv_cell_size = 1.0f / cell_size;
    float delta = p->bounding_radius(t_curr) + cell_size/2.0 + EPSILON;

    hash.data.level = level;
    short int imin = (short int) floor((x[0] - delta) * inv_cell_size);
    short int jmin = (short int) floor((x[1] - delta) * inv_cell_size);
    short int kmin = (short int) floor((x[2] - delta) * inv_cell_size);
    short int imax = (short int)  ceil((x[0] + delta) * inv_cell_size);
    short int jmax = (short int)  ceil((x[1] + delta) * inv_cell_size);
    short int kmax = (short int)  ceil((x[2] + delta) * inv_cell_size);

    for (short int i = imin; i < imax; i++) {
      hash.data.x = i;
      for (short int j = jmin; j < jmax; j++) {
        hash.data.y = j;
        for (short int k = kmin; k < kmax; k++) {
          hash.data.z = k;
          if (grid.find(hash.value) != grid.end()) {
            Particle *neighbor = grid[hash.value];
            while(neighbor != NULL) {
              if (neighbor != p)
                neighbors.push_back(neighbor);
              neighbor = neighbor->next();
            }
          }
        }
      }
    }
  }
}
```

grid is an instance of std::map<hash, Particle*>, but this class stores data as an array of red-black tree nodes, with the value contiguously in memory in each node. This means that it loads all values along with the keys when traversing it.

Packing Simulation Revisited

```
$ perf record -e cycles -F 1000 -- pack -f 0.6 ellipsoids
100.00% 0.6000 0.0000/min 2.6e-02 ev/s 38.2 s
[ perf record: Woken up 5 times to write data ]
[ perf record: Captured 1.457 MB perf.data (38005 samples) ]
$ perf report --stdio --percent-limit 0.25
# Overhead  Command  Shared Object Symbol
# ........  .......  ............. .........................
#
    39.53%  pack     pack          intersect
    32.60%  pack     pack          HGrid::find_neighbors
     7.62%  pack     pack          Ellipsoid::support
     3.76%  pack     pack          closest_point_tetrahedron
     3.37%  pack     pack          closest_point_triangle
     2.77%  pack     pack          Simplex::closest
     2.46%  pack     pack          Ellipsoid::bounding_radius
     2.07%  pack     pack          check_overlap
     1.94%  pack     pack          Simplex::contains
     1.15%  pack     libm.so.6     __sincos
     0.37%  pack     pack          HGrid::make_hash
     0.36%  pack     pack          HGrid::insert
```

```
enum _Rb_tree_color { _S_red = false, _S_black = true };

_Rb_tree_node_base
{
  typedef _Rb_tree_node_base* _Base_ptr;
  typedef const _Rb_tree_node_base* _Const_Base_ptr;

  _Rb_tree_color      _M_color;
  _Base_ptr           _M_parent;
  _Base_ptr           _M_left;
  _Base_ptr           _M_right;
};

template<typename _Val>
  struct _Rb_tree_node : public _Rb_tree_node_base
  {
    typedef _Rb_tree_node<_Val>* _Link_type;

    _Val _M_value_field;

    _Val*
    _M_valptr()
    { return _M_storage._M_ptr(); }

    const _Val*
    _M_valptr() const
    { return _M_storage._M_ptr(); }
};
```

Value contiguous in memory after enum and 3 pointers.

enum = int = 4 bytes
4 pointers = 4 x 8 = 32 bytes
node = 36 bytes, misaligned with cachelines

Traversing the map to find key is expensive

```
Samples: 41K of event 'ibs_op//', 1000 Hz, Event count (approx.): 174896447829
HGrid::find_neighbors  /home/amadio/src/rocpack/pack [Percent: local period]
Samples        struct less : public binary_function<_Tp, _Tp, bool>
               {
               _GLIBCXX14_CONSTEXPR
               bool
               operator()(const _Tp& __x, const _Tp& __y) const
               { return __x < __y; }
   159           mov       -0x48(%rbp),%rdx
    88           mov       %r13,%r9
   100           mov       %rsi,%rax
   121           xchg      %ax,%ax
               if (!_M_impl._M_key_compare(_S_key(__x), __k))
  2266  240:     cmp       %rdx,0x20(%rax)
               ↓ jae       450
               { return static_cast<_Link_type>(__x->_M_right); }
   766           mov       0x18(%rax),%rax
               while (__x != 0)
   898           test      %rax,%rax
     2         ↑ jne       240
   258           mov       %r9,%rax
               find(const _Key& __k)
               {
Press 'h' for help on key bindings
```

Cache hit/miss rates with perf mem

```
bash $ perf mem record -C 0 -t load --ldlat=20 -- taskset -c 0 ./pack --seed 17 -f 0.45 bench2.in
100.00% 0.4500 0.0000/min 1.8e-02 ev/s 54.1 s
[ perf record: Woken up 64 times to write data ]
[ perf record: Captured and wrote 16.013 MB perf.data (208040 samples) ]
bash $ perf mem report --stdio -s mem
# To display the perf.data header info, please use --header/--header-only options.
#
#
# Total Lost Samples: 0
#
# Samples: 208K of event 'cpu/mem-loads,ldlat=20/P'
# Total weight : 9857968
# Sort order   : mem
#
# Overhead       Samples  Memory access
# ........  ............  ....................................
#
    90.65%        178426  L1 hit
     4.36%         12798  LFB/MAB hit
     3.95%         15602  L2 hit
     0.98%          1203  L3 hit
     0.04%             7  Remote Any cache hit
     0.01%             3  RAM hit
     0.00%             1  L3 miss
```

perf mem report -s mem,sym --hierarchy

```
Samples: 208K of event 'cpu/mem-loads,ldlat=20/P', Event count (approx.): 9857968
  Overhead       Samples     Memory access / Symbol
-   90.65%        178426     L1 hit
       47.89%         84778     [.] HGrid::find_neighbors
       22.39%         47769     [.] intersect
        8.78%         19603     [.] Simplex::closest
        3.42%          9093     [.] Simplex::add_vertex
        3.13%          6182     [.] closest_point_tetrahedron
        2.01%          3721     [.] Simplex::contains
        1.56%          3719     [.] Polyhedron::support
        0.26%           553     [.] PeriodicBoundary::reposition
-    4.36%          12798     LFB/MAB hit
        2.61%          8059     [.] intersect
        1.00%          3449     [.] HGrid::find_neighbors
-    3.95%          15602     L2 hit
        2.18%          8606     [.] HGrid::find_neighbors
        1.62%          6434     [.] intersect
-    0.98%           1203     L3 hit
        0.41%           594     [.] intersect
        0.33%           469     [.] HGrid::find_neighbors

Tip: To compute metrics for samples use perf record -e '{cycles,instructions}' ... ; perf script -F +metric
```

New data profiling can show which structures are hit

```
bash $ perf annotate --data-type=Particle

Annotate type: 'Particle' in pack (7940 samples):
==============================================================================
    samples     offset       size  field
       7940          0        160  Particle      {
          5         80          8      Shape*   m_shape;
          0         88          4      float    m_growth_rate;
          0         92          4      float    m_mass;
          0         96          4      float    m_inv_mass;
          0        100          4      unsigned int     m_tag;
          0        104          4      unsigned int     m_event_id;
          2        112          8      long unsigned int        m_hash;
          1        120          8      long unsigned int        m_collisions;
          1        128          8      Particle*        m_prev;
       7931        136          8      Particle*        m_next;
          0        144         16      union     {
          0        144         16          __m128       m_color;
          0        144         16          struct        {
          0        144          4              float    r;
          0        148          4              float    g;
          0        152          4              float    b;
          0        156          4              float    a;
                                           };
                                      };
```

When looking for adjacent particles to check for intersections, we build the list of neighbors and iterate through a list, so m_next is a hot field for slow loads (>20 cycles).

Color not used during simulation, can make it compile-time optional for better performance, and reorder hot field into the first cache line.

After optimization, less samples on hot member

```
bash $ perf annotate --data-type=Particle

Annotate type: 'Particle' in pack (7541 samples):
==============================================================================
    samples     offset       size  field
       7541          0        128  Particle      {
       7530          0          8      Particle*        m_next;
          0          8          8      Particle*        m_prev;
          2         16          8      long unsigned int        m_hash;
          2         24          8      Shape*   m_shape;
          0         32         16      Point    m_position;
          0         48         16      Quaternion       m_orientation;
          0         64         16      Vector   m_velocity;
          0         80         16      Vector   m_ang_velocity;
          0         96          4      float    m_time;
          0        100          4      float    m_growth_rate;
          0        104          4      float    m_mass;
          0        108          4      float    m_inv_mass;
          6        112          8      long unsigned int        m_collisions;
          0        120          4      unsigned int     m_tag;
          1        124          4      unsigned int     m_event_id;
                                     };
```

L1 cache hit rate also improved after optimization

```
bash $ perf mem record -C 0 -t load --ldlat=20 -- taskset -c 0 ./pack --seed 17 -f 0.45 bench2.in
100.00% 0.4500 0.0000/min 1.9e-02 ev/s 53.4 s
[ perf record: Woken up 63 times to write data ]
[ perf record: Captured and wrote 15.887 MB perf.data (205679 samples) ]
bash $ perf mem report --stdio -s mem
# To display the perf.data header info, please use --header/--header-only options.
#
#
# Total Lost Samples: 0
#
# Samples: 205K of event 'cpu/mem-loads,ldlat=20/P'
# Total weight : 9887519
# Sort order   : mem
#
# Overhead       Samples  Memory access
# ........  ............  ....................................
#
    93.01%        183502  L1 hit
     3.12%         12339  L2 hit
     3.11%          9010  LFB/MAB hit
     0.68%           813  L3 hit
     0.07%            12  Remote RAM hit
     0.01%             2  L3 miss
     0.00%             1  RAM hit
```

Summary and Conclusions

- Compiler can help with optimizing computations, not so much with memory access
- Many tools are available to inspect and optimize memory access patterns
  - pahole, perf mem, perf c2c, VTune memory access analysis
- Data-Oriented design collects key concepts to design memory efficient software
  - Separate data structures and operations on data
  - Focus on avoiding high latency and wasting memory bandwidth
- Many other indicators of bad memory access patterns
  - Backend Bound - stalled cycles at backend are a good indicator of inefficiencies
    - Core Bound - data dependencies in arithmetics, chains of high latency instructions
    - Memory Bound - not only read, but also write, some codes can be store bound
  - Cache Misses - main indicator of memory access issues
    - Need to watch for problem size: hit rate high for small workloads, inevitably higher on larger workloads
    - Cache associativity and locality can lead to complex issues, avoid loops with power of 2 trip length

“Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**. Yet we should not pass up our opportunities in that critical 3%”

— Donald Knuth

CERN