Performance limits and profiling

Potential performance limits

Why does my software not go faster? Below some of the possible hardware performance limits:

- Flops (floating-point operations)
- Ops (operations) that include all types of computer instructions
- Memory *bandwidth*
- Memory *latency*
- Instruction queue (instruction cache)
- Networks
- Disk

Potential performance limits

Why does my softwa
performance limits:

- Flops (floatin
- Ops (operations) th
- Memory *bandwidth*
- Memory *latency*
- Instruction queue (instruction cache)
- Networks
- Disk

Computational scientists still consider flops as the primary performance limit. While this might have been true years ago, the reality is that flops seldom limit performance in modern architectures.

Potential performance limits

Why does my software not go faster? Below some of the possible hardware performance limits:

- Flops (floating-point operati
- Ops (operations) that includ
- Memory *bandwidth*
- Memory *latency*
- Instruction queue (instruction cache)
- Networks
- Disk

Integer operations are also a more frequent limi than commonly assumed, especially with high dimensional arrays where the index calculatio become more complex.

Potential performance limits

Why does my software not g
performance limits:

- Flops (floating-point oper
- Ops (operations) that in
- Memory *bandwidth*
- Memory *latency*
- Instruction queue (instruction cache)
- Networks
- Disk

Bandwidth is the best rate at which data can be moved. For bandwidth to be the limit, the code should use a **streaming** approach, where data are contiguous, and all the values used.

Potential performance limits

Why does my software not go faster? Below some of the possible hardware performance limits:

- Flops (floating-point operations)
- Ops (operations) that include all types of computer instructions
- Memory *bandwidth*
- Memory *latency*
- Instruction queue (i
- Networks
- Disk

When a streaming approach is not possible, latency is the more appropriate limit. Latency is the time required for the first byte or word of data to be transferred.

Speeds and feeds

We can break down all of these limitations into two major categories:

- *Speeds:* how fast operations can be done:

Depends on the available hardware resources (parallelism) and on the operations (e.g. addition, divisions, FMA)

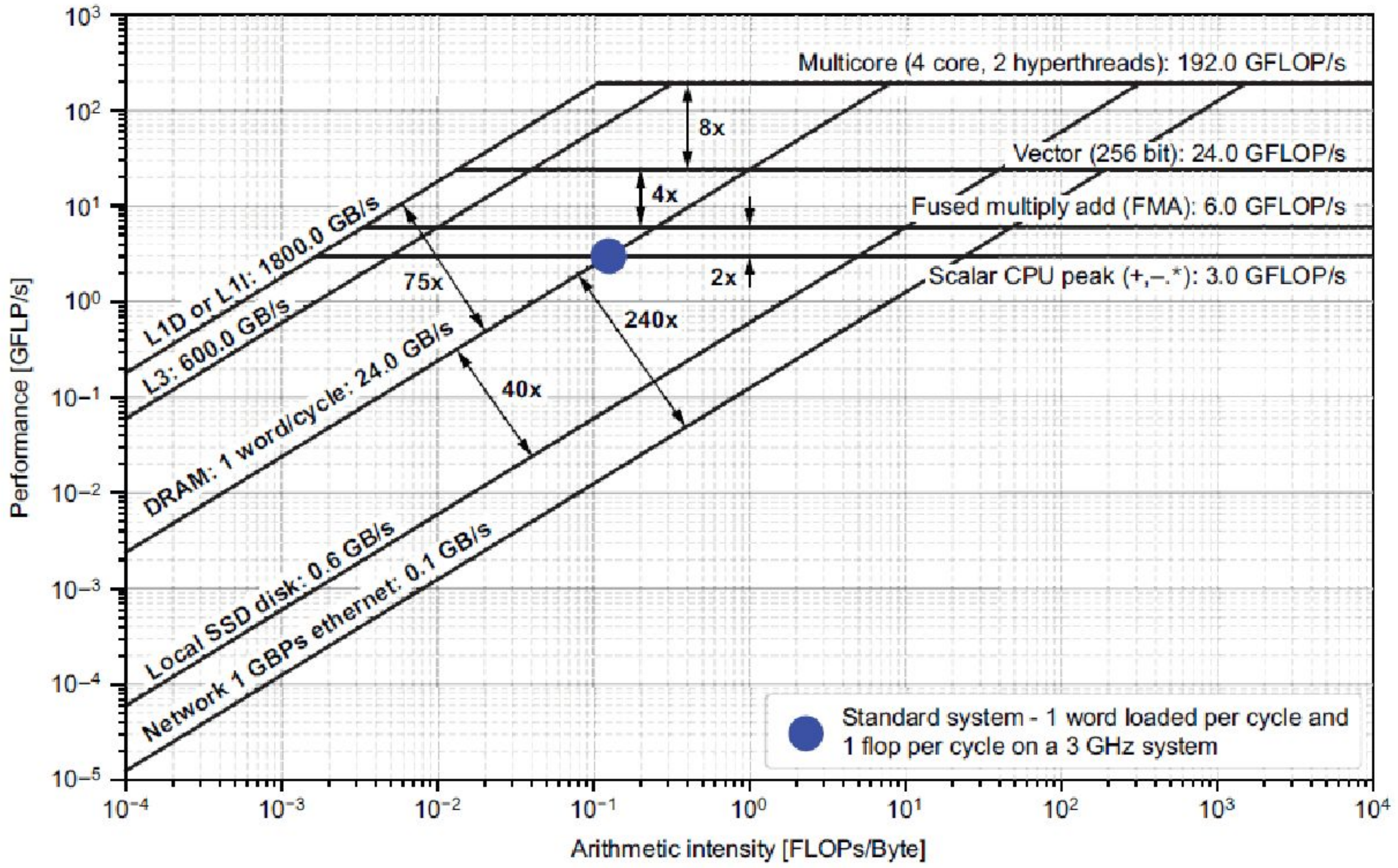
- *Feeds*

Feeds include the memory bandwidth through the cache hierarchy, as well as network and disk **bandwidth**. For applications that cannot get *streaming* behavior, the **latency** of memory, network and disk feed are more important.

Roofline Model

Feeds and speeds shown on a **roofline** plot.

- The conventional scalar CPU is close to the 1 word loaded per cycle and 1 flop per cycle.
- The multipliers for the increase in flops are due to the fused multiply-add instruction, vectorization, multiple cores, and hyperthreads.
- The relative speeds of memory movement are also shown.



Roofline Model

- ***Performance*** *[Gflops/sec]:* in an application, measures the number of flops executed in a second

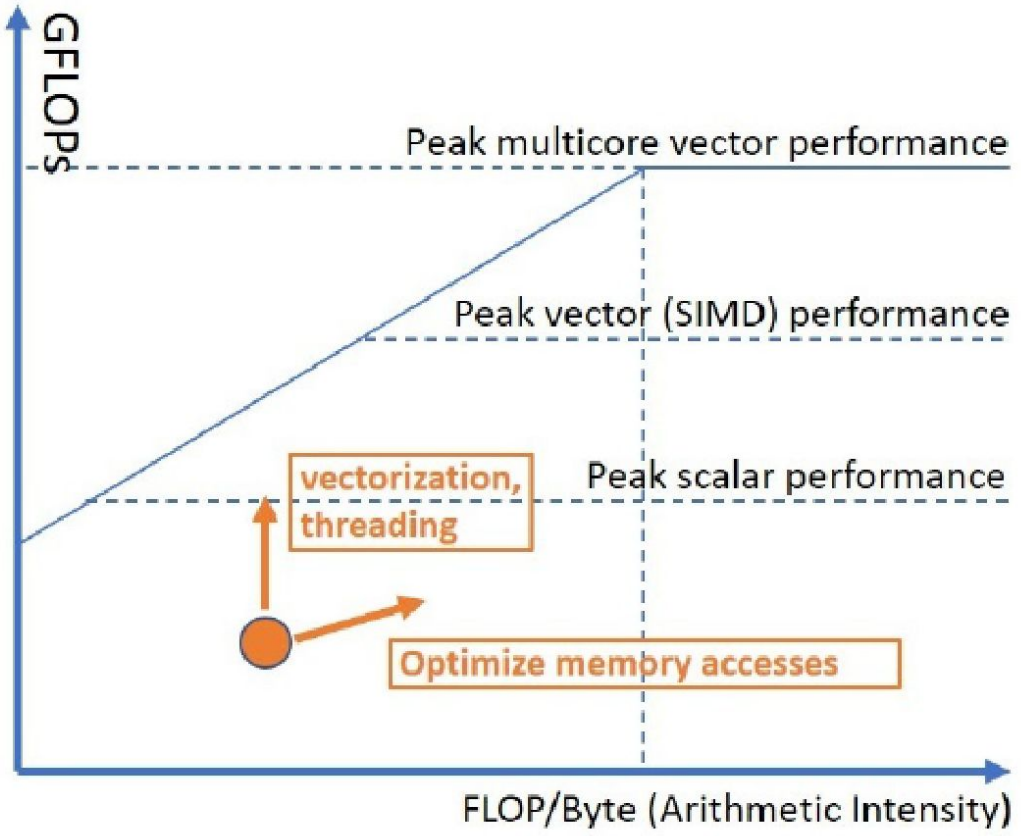
- ***Arithmetic intensity*** *[flops/B]:* in an application, measures the number of flops executed per memory operation (transferred bytes)

- ***Memory bandwidth**= Performance @ Arithmetic Intensity=1 flop/B*

- ***Machine balance**:* the number of flops that can be executed divided by the memory bandwidth for a computing hardware

Roofline analysis of a program

- The goal of optimizing performance using the Roofline model is to move the points up on the chart.
- **Vectorization** and **threading** move the dot up
- **Optimizing memory accesses** (increasing arithmetic intensity) will move the dot to the right, and also likely improve performance.


GFLOPs
Peak multicore vector performance
Peak vector (SIMD) performance
Peak scalar performance
vectorization, threading
Optimize memory accesses
FLOP/Byte (Arithmetic Intensity)

Determine your hardware capabilities: Benchmarking

In determining hardware performance, we use a mixture of theoretical and empirical measurements.

- the theoretical value provides an upper bound to performance,
- the empirical measurement confirms what can be achieved with a simplified kernel

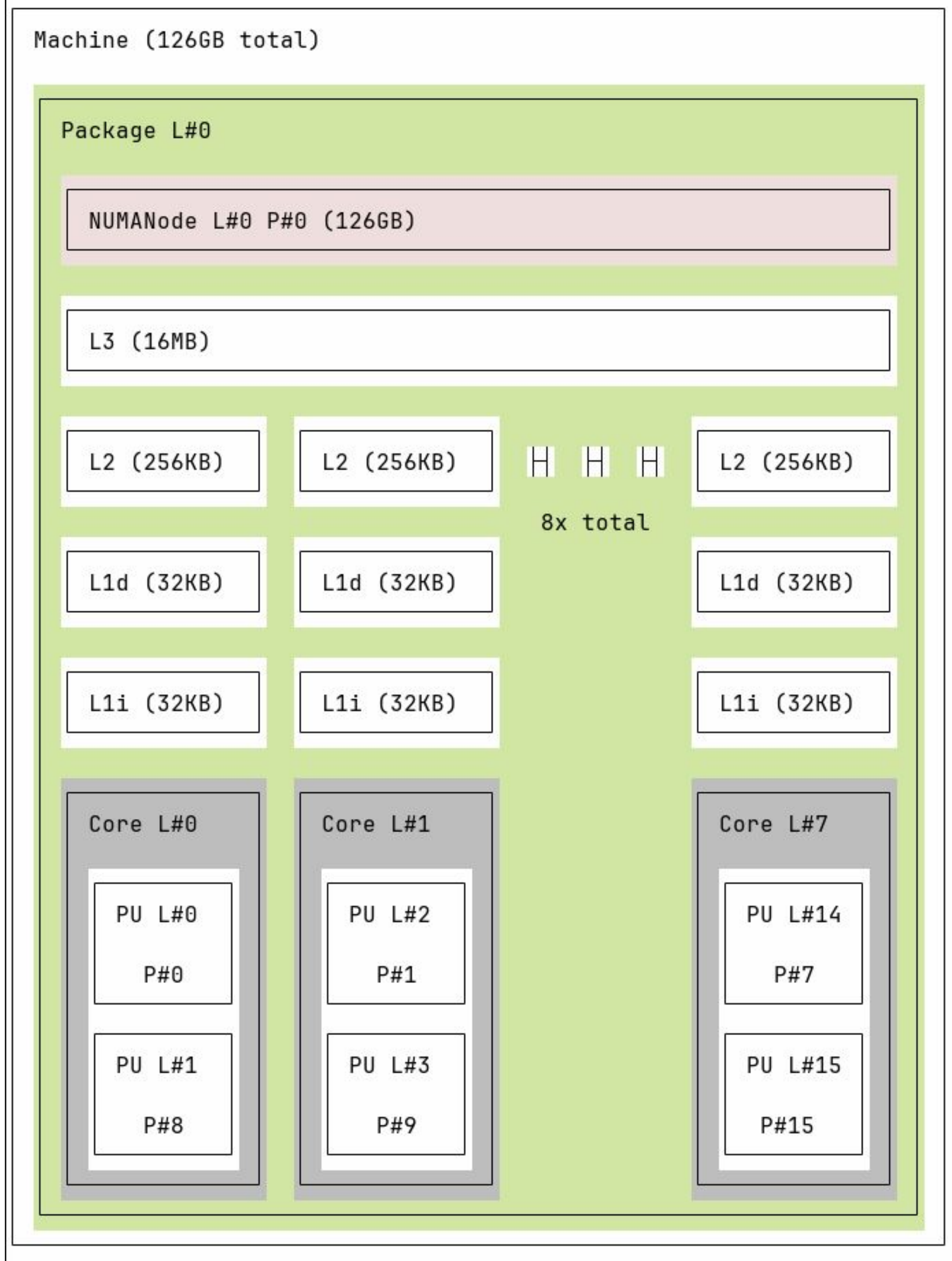
A suite to hardware capabilities measurement can be found here:

https://github.com/EssentialsofParallelComputing/Chapter3

hwloc software package

The Portable Hardware Locality (hwloc) software package provides a portable abstraction (across OS, versions, architectures, ...) of the hierarchical topology of modern architectures.

```
[sal@optiplex] lstopo --no-io --of ascii
```



Theoretical maximum flops

The theoretical maximum flops (FT) can be calculated as:

$$F_T = C_v \times f_c \times I_c = \text{Virtual Cores} \times \text{Clock Rate} \times \text{Flops/Cycle}$$

where:

$\mathbf{C_v}$ is the number of virtual cores

$\boldsymbol{f_c}$ is the maximum clock rate when all the processors are used.

$\boldsymbol{I_c}$ are instructions per cycle, including the number of simultaneous operations that can be executed by the vector unit. We can include the fused multiply-add (FMA) instruction as 2x factor

Theoretical memory bandwidth

For most large computational problems, we can assume that there are large arrays that need to be loaded from main memory through the cache hierarchy

$$B_T = MTR \times M_c \times T_w \times N_s = \text{Data Transfer Rate} \times \text{Memory Channels} \times \text{Bytes Per Access} \times \text{Sockets}$$

where:

MTR is the memory transfer rate given in millions of transfers per sec (MT/s).

T_w is the memory transfer width (e.g. 64 bits)

M_c are the memory channels (usually 2 per CPU)

N_S is the number of sockets

Theoretical vs empirical measurement

Theoretical numbers are almost never possible in practice. They assume perfect conditions:

- All memory accesses are aligned and contiguous.
- Every CPU instruction is doing a floating-point operation.
- No stalls, no cache misses, no synchronization overhead.
- Full vectorization, perfect parallelism, no thermal throttling.

Why empirical measurements fall short

Memory-bound operations:

- Even a simple load/store might be limited by latency, NUMA, or memory conflicts.
- Prefetchers and caches help but rarely eliminate all delays.

Compute-bound operations

- Compilers may fail to fully **vectorize** your loops.
- Pipeline stalls, dependency chains, and instruction mix reduce effective FLOPs/sec.

Software overhead: Function calls, branches, synchronization (threads, OpenMP), OS interrupts.

Hardware quirks: Thermal throttling, memory controllers rtc.

Empirical measurement of bandwidth and flops

To determine the memory bandwidth, we can measure the time for reading and writing a large array of data.

```
 [sal@optiplex] likwid-bench -t copy -w N:100MB

 […]
MByte/s:                        25037.41
 […]
```

It can be significantly lower than theoretical BW for several reasons:

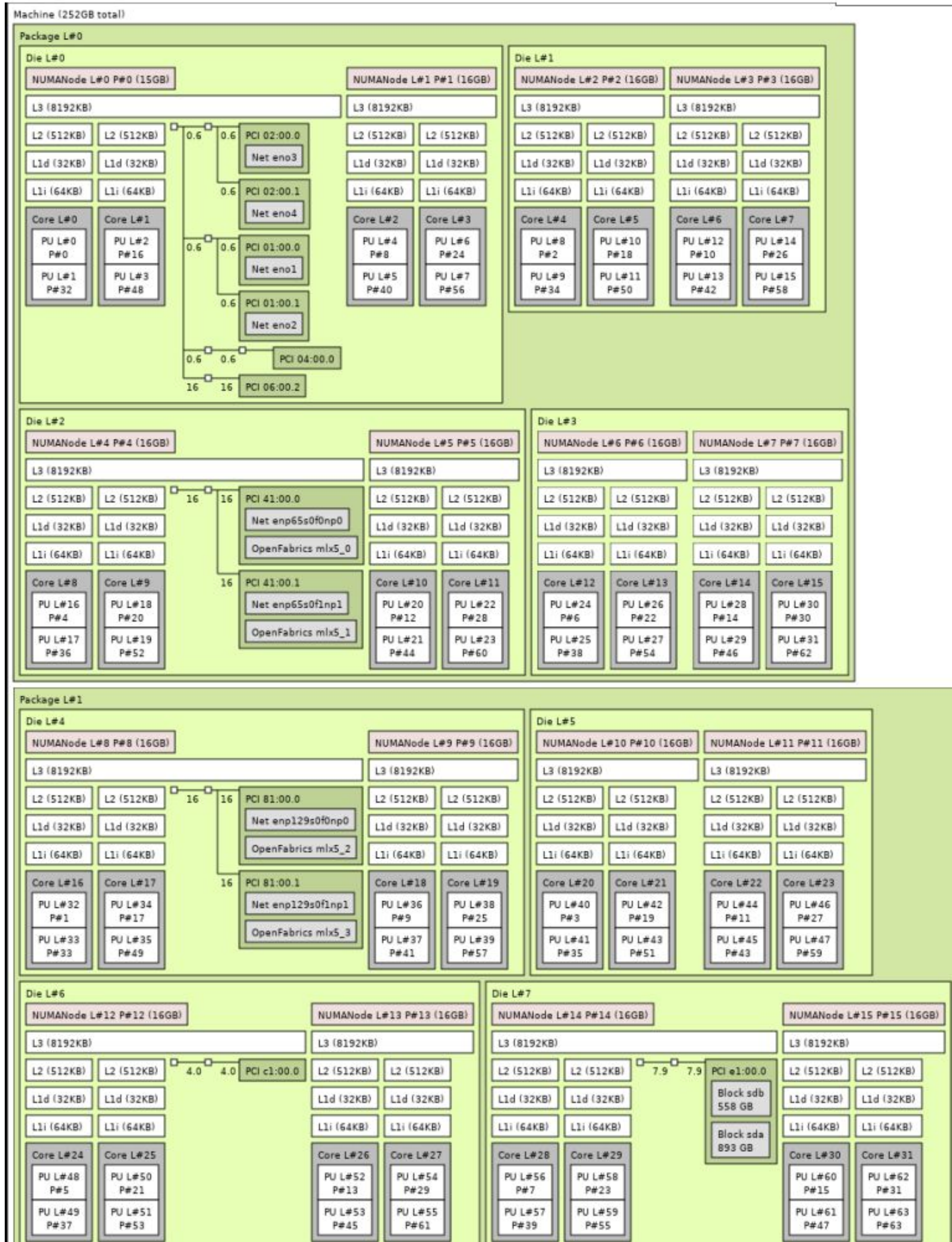
- Suboptimal memory BIOS settings (e.g., incorrect memory interleaving)
- Memory contention (multiple CPU cores share memory channels)
- NUMA awareness

Cluster node memory bandwidth

```
[sal@cluster] lstopo
```

- Dual socket configuration
- 4 dice for socket
- 2 numa nodes per die
- 2 physical cores per numa node
- Up to 8 channels per socket

$$B_T = 2400\ (\mathrm{MT/s}) * 8\mathrm{B} * 8 * 2 = 307\mathrm{GB/s}$$

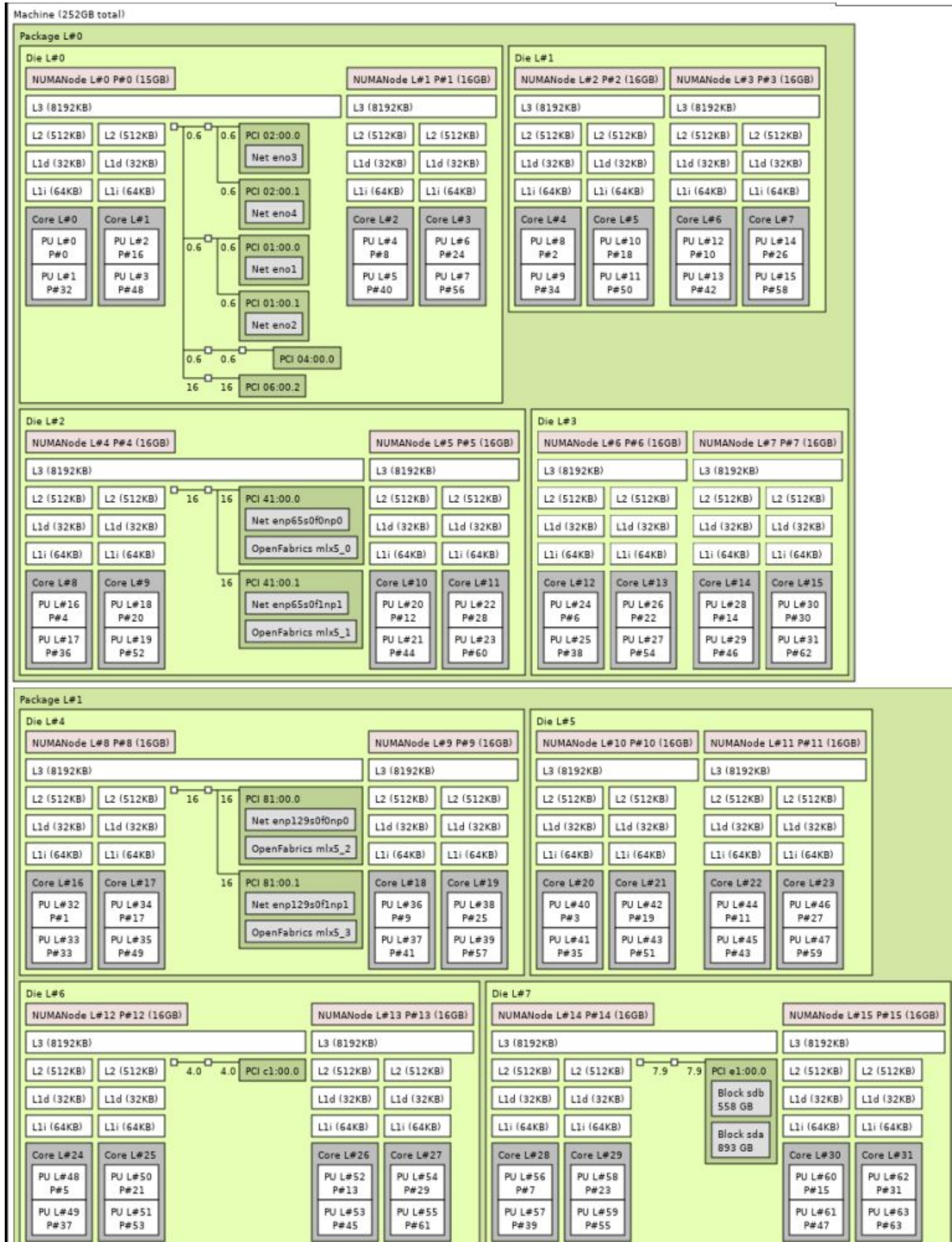


Cluster node memory bandwidth

```
[sal@cluster] lstopo
```

```
[sal@cluster] likwid-bench -t copy -w
N:1000MB

[…]
MByte/s:                    66802.19
[…]
```



Cluster node memory bandwidth

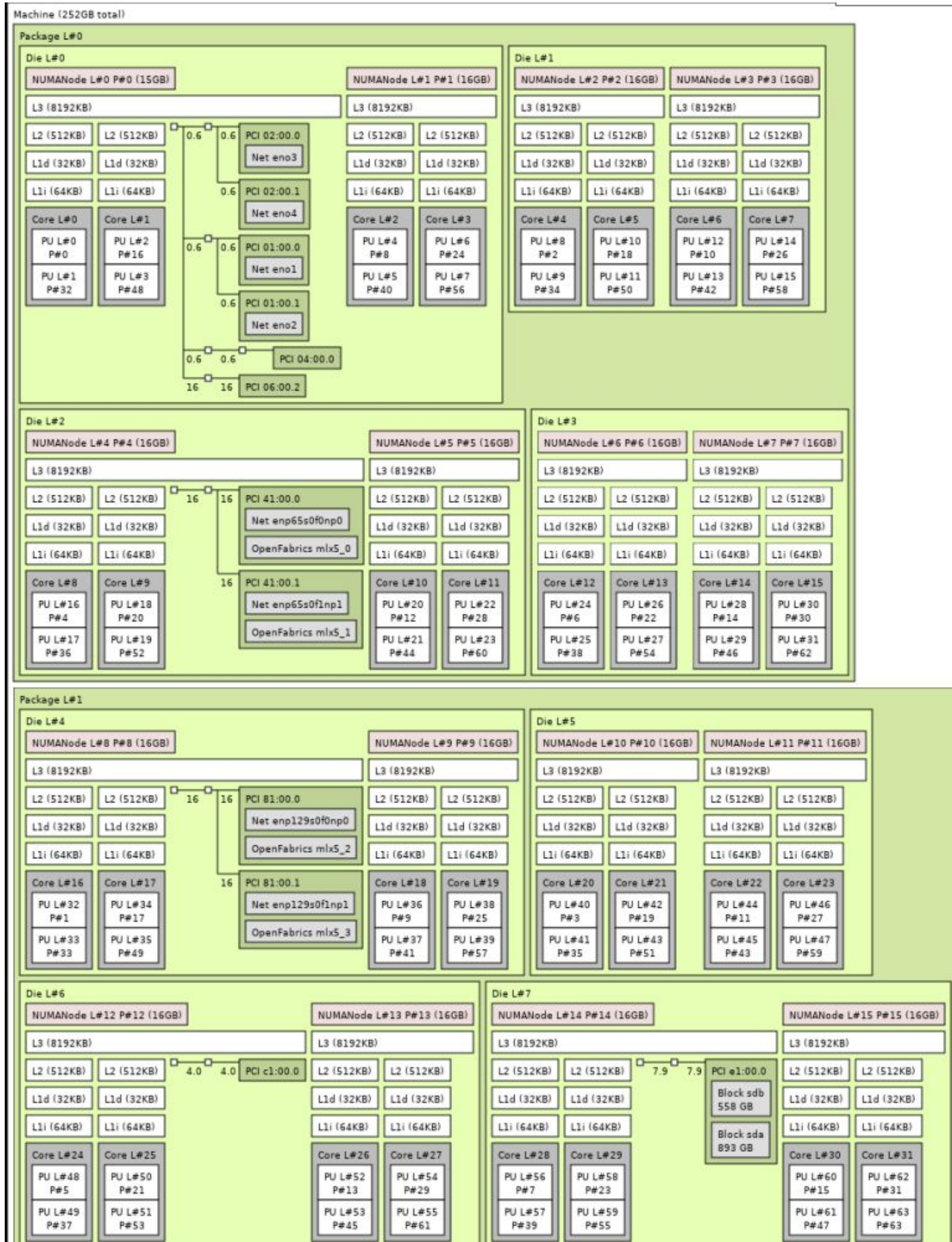
```
[sal@cluster] lstopo
```

```
[sal@cluster] likwid-bench -t copy -w
N:1000MB

[…]
MByte/s:                 66802.19
[…]
```

Cluster node are configured with 2 channels per socket

B_T = 2400 (MT/s) *8B*2*2 = 76GB/s



Empirical measurement of bandwidth and flops

To determine the maximum flop, we can use suitable Linux tools

```
 [sal@optiplex] likwid-bench -t peakflops_avx_fma -w N:1MB

 […]
MFlops/s:                    596722.60
 […]
```

Theoretical maximum flops:

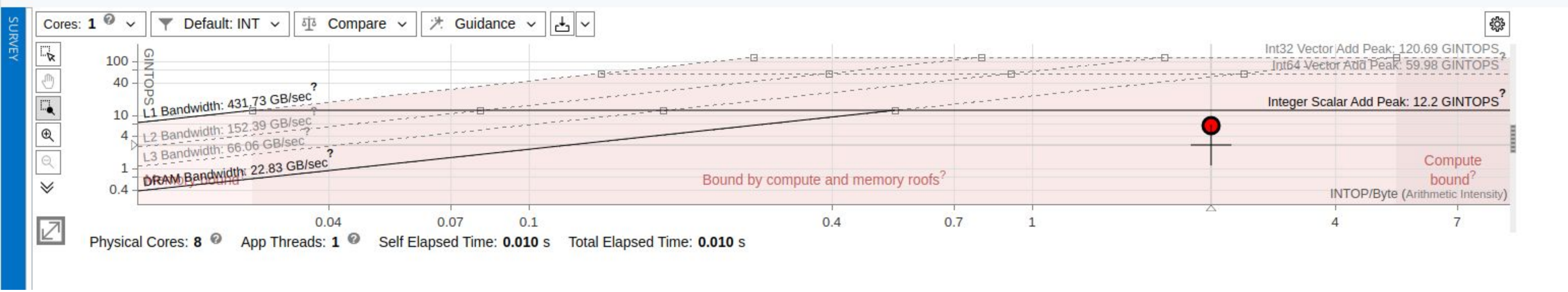
- Per FMA instruction: 4 doubles × 2 operations = **8 FLOPs**.

- 2 FMA per core (eight cores): 2 × 8 × 8 FLOPs = **128 FLOPs**

- Max freq @4.7GHz = 128 × 4.7= 601 **GFLOPs/sec**

Empirical measurement of bandwidth and flops

```
[sal@optiplex] advisor  --collect=roofline --project-dir=./roofline_results -- ./main
```


SURVEY
Cores: 1
Default: INT
Compare
Guidance
GINTOPS
100
40
10
4
1
0.4
L1 Bandwidth: 431.73 GB/sec
L2 Bandwidth: 152.39 GB/sec
L3 Bandwidth: 66.06 GB/sec
DRAM Bandwidth: 22.83 GB/sec
Memory Bound
Int32 Vector Add Peak: 120.69 GINTOPS
Int64 Vector Add Peak: 59.98 GINTOPS
Integer Scalar Add Peak: 12.2 GINTOPS
Bound by compute and memory roofs
Compute bound
INTOP/Byte (Arithmetic Intensity)
0.04
0.07
0.1
0.4
0.7
1
4
7

Physical Cores: 8 App Threads: 1 Self Elapsed Time: **0.010** s Total Elapsed Time: **0.010** s

Performance metrics

CPU Utilization

The CPU utilization can be calculated as:

$$CPU\ Utilization = \frac{CPU\_CLK\_UNHALTED.REF\_TSC}{TSC},$$

where:

CPU_CLK_UNHALTED.REF_TSC counts the number of reference cycles when the core is not in a halt state.

TSC is for timestamp counter

CPU Utilization

The CPU utilization can be calculated as:

$CPU\ Util$

where:

CPU_CLK_U

when the core is

is a performance monitoring events accessible using the Performance Monitoring Unit (PMU).

$ sudo perf stat -e cpu_clk_unhalted.ref_tsc sleep 1

TSC is for timestamp counter

CPU Utilization

The CPU utilization can be calculated as:

$CPU\ Util$

where:

CPU_CLK_U

when the core is

You can read this performance counter directly from Model-Specific Registers (MSRs).

```
$ sudo rdmsr -p 0 0x309
```

0x309 is the MSR for *IA32_FIXED_CTR0*

TSC is for timestamp counter

CPU Utilization

The CPU utilization can be calculated as:

$$CPU\ Utilization = \frac{CPU\_CLK\_UNHALTED.REF\_TSC}{TSC},$$

where:

CPU_CLK_UNH

when the core i

TSC is for

the TSC (Time Stamp Counter) is a 64-bit counter that increments at a constant rate. The canonical way to read the TSC is the RDTSC asm instruction.

CPI and IPC

IPC can be calculated as:

$$IPC = \frac{INST\_RETIRED.ANY}{CPU\_CLK\_UNHALTED.THREAD}$$

Where *INST_RETIRED.ANY* counts the number of retired instructions, and CPU_CLK_UNHALTED.THREAD counts the number of core cycles while the thread is not in a halt state.

CPI and IPC

IPC can be calculated as:

$$IPC = \frac{INST}{CPU\_CLK\_}$$

Where $INST\_RETIRED.ANY$ co
CPU_CLK_UNHALTED.THRE
thread is not in a halt state.

In Intel PMU terminology, the suffix "THREAD" refers to a hardware thread, i.e., one logical CPU (like CPU0, CPU1, etc.), not a software thread!

CPI and IPC

IPC can be calculated as:

$$IPC = \frac{INST\_RETIRED.ANY}{CPU\_CLK\_UNHALTED.THREAD}$$

Where *INST_RETIRED.ANY* counts the number of retired instructions, and `CPU_CLK_UNHALTED.THREAD` counts the number of core cycles while the thread is not in a halt state.

Is the **core clock** (actual running frequency) not the **reference clock** always running at the **TSC frequency**

CPI and IPC

- CPI is the inverse of IPC

$$CPI = \frac{1}{IPC}$$

- Memory intensive applications usually have a low IPC (0–1)

- computationally intensive workloads tend to have a high IPC (4–6).

Performance

The performance depends on two factors:

$$Performance = \frac{instructions \times frequency}{cycles} = IPC \times frequency$$

From the perspective of benchmarking, IPC and frequency are two independent metrics.

Cache Miss

Performance greatly suffers when a memory request misses the cache:

Memory Hierarchy Component	Latency (cycle/time)
L1 Cache	4 cycles (~1 ns)
L2 Cache	10-25 cycles (5-10 ns)
L3 Cache	~40 cycles (20 ns)
Main Memory	200+ cycles (100 ns)

Linux *perf* users can collect the number of L1 cache misses by running:

```
$ perf stat -e mem_load_retired.fb_hit,mem_load_retired.l1_miss,
mem_load_retired.l1_hit,mem_inst_retired.all_loads -- a.exe
 29580 mem_load_retired.fb_hit
 19036 mem_load_retired.l1_miss
497204 mem_load_retired.l1_hit
546230 mem_inst_retired.all_loads
```

misses are around 3.5% of loads

Cache Miss

We can further break down L1 data misses and analyze L2 cache behavior

```
$ perf stat -e mem_load_retired.l1_miss,
mem_load_retired.l2_hit,mem_load_
19521 mem_load_retired.l1_miss
12360 mem_load_retired.l2_hit
7188 mem_load_retired.l2_miss
```

37% of loads that missed in the L1 D-cache also missed in the L2 cache

Mispredicted Branch

Linux *`perf`* users can check the number of branch mispredictions by running

```
$ perf stat -e branches,branch-misses -- a.exe
358209 branches
14026 branch-misses # 3,92% of all branches
# or simply do:
$ perf stat -- a.exe
```

Performance Metrics

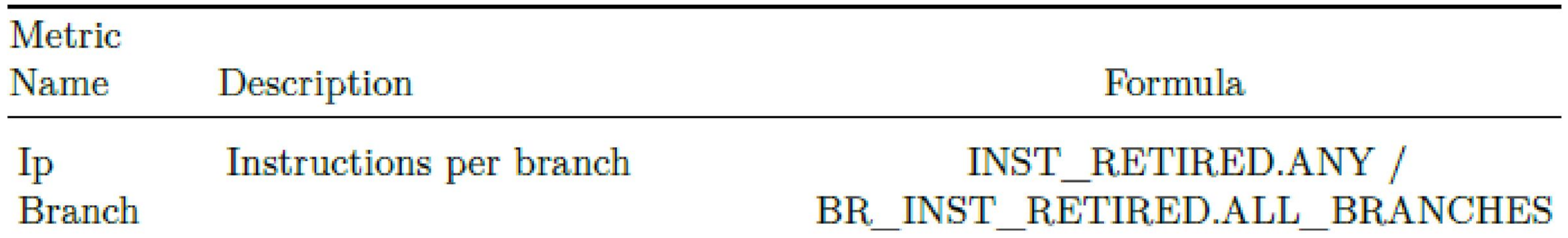
Performance engineers frequently use metrics, which are built on top of raw events. Metrics combine and process raw events into **interpretable quantities** that reflect performance characteristics.

They allow **identifying limiting factors** (e.g., memory bandwidth, compute efficiency).

A full list of metrics can be found here:

https://github.com/intel/perfmon/blob/main/TMA_Metrics-full.xlsx

Performance Metrics: examples (1)

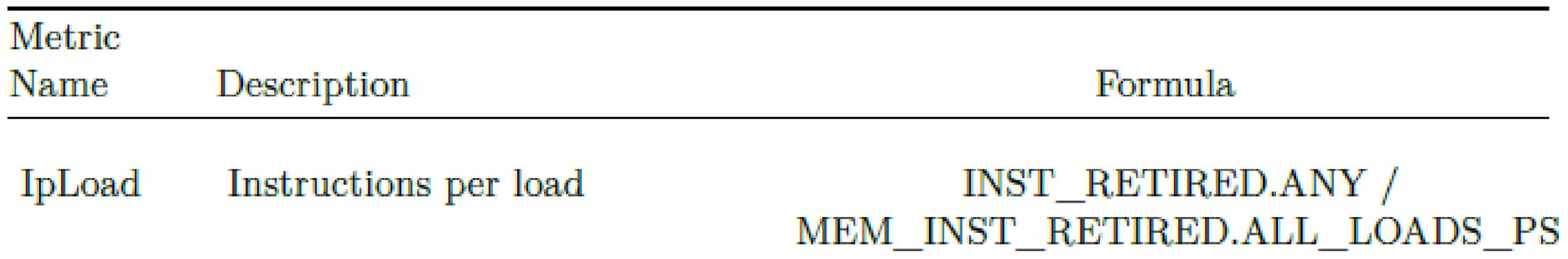


Metric Name	Description	Formula
Ip Branch	Instructions per branch	INST_RETIRED.ANY / BR_INST_RETIRED.ALL_BRANCHES

Instructions per branch indicates the branch density of your code.

- **High IPB:** Few branches per many instructions. Code is mostly straight-line (**compute-heavy**). Good for pipelines and vectorization.
- **Low IPB:** Frequent branching. Code has lots of control flow (if/else, loops, function calls). Can hurt performance if branch prediction fails.

Performance Metrics: examples (2)

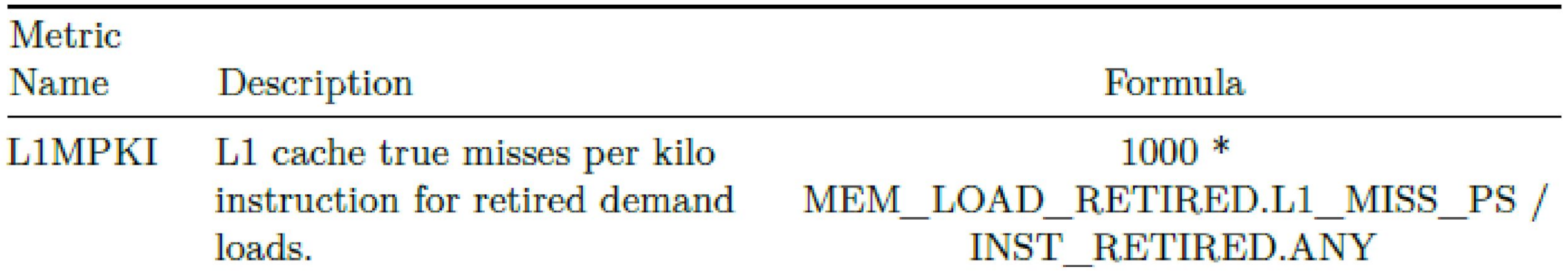


Metric Name	Description	Formula
IpLoad	Instructions per load	INST_RETIRED.ANY / MEM_INST_RETIRED.ALL_LOADS_PS

IpLoad measures the average number of instructions executed per memory load

- High IPL → few loads per many instructions (compute-heavy, good locality)
- Low IPL → many loads (memory-bound, frequent data fetching).

Performance Metrics: examples (3)

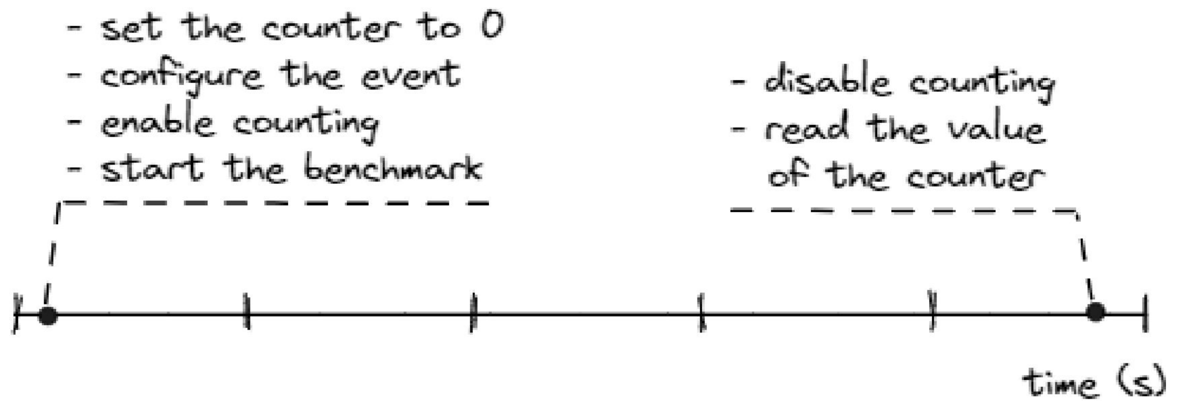


Metric Name	Description	Formula
L1MPKI	L1 cache true misses per kilo instruction for retired demand loads.	1000 * MEM_LOAD_RETIRED.L1_MISS_PS / INST_RETIRED.ANY

L1MPKI stands for L1 Misses Per Kilo Instructions. It is used to quantify how often your program misses in the L1 cache relative to how many instructions it executes.

- **Low L1MPKI** → most loads/stores are satisfied from L1 → excellent cache locality.
- **High L1MPKI** → many accesses miss L1 → CPU must fetch from L2/L3/memory → performance penalty.

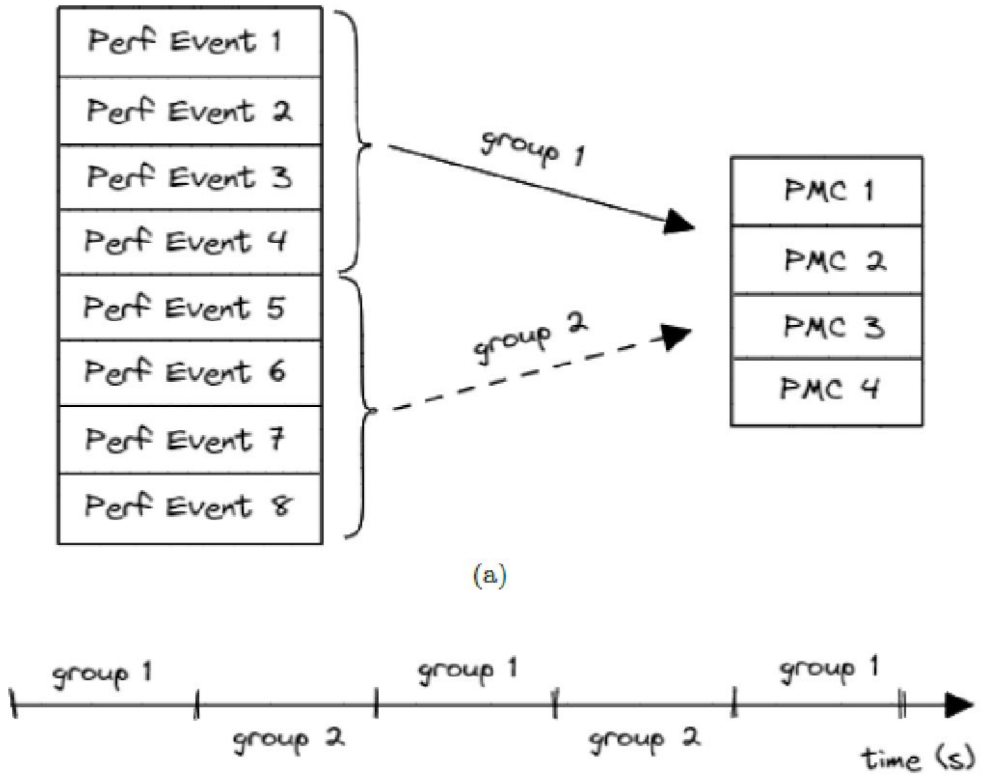
Collecting Performance Monitoring Events



The steps outlined in Figure roughly represent what a typical analysis tool will do to count performance events.

```
$ perf stat -- ./my_program.exe
10580290629 cycles      # 3,677 GHz
8067576938 instructions  # 0,76 insn per cycle
3005772086 branches     # 1044,472 M/sec
239298395 branch-misses  # 7,96% of all branches
```

Multiplexing and Scaling Events



If you need to collect more events than the number of available PMCs, the analysis tool uses time multiplexing

Code Instrumentation + PMC

Marker APIs analyzes the performance of a specific code region, not an entire application.

Most performance analysis tools provide specific *marker APIs* that let you do that. Examples are:

- Intel VTune has `__itt_task_begin /__itt_task_end` functions.
- AMD uProf has `amdProfileResume / amdProfilePause` functions.
- Likwid has `likwid_markerStartRegion / likwid_markerStopRegion` functions.

Marker APIs

Marker APIs hybrid approach combines the benefits of instrumentation and performance event counting

- They use **hardware performance counters** (via perf_event or MSRs)
- They abstract from low level details and provides discovery tools for identifying available events on your CPU
- the start function reads current counter values. The stop function calculates the delta and stores it. At the end, it aggregates results and prints or exports them.

Example with likwid

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

void foo() {
    LIKWID_MARKER_START("foo");
    double sum = 0.0;
    for (int i = 0; i < 1000000; i++)
      sum += i * 0.5;
    LIKWID_MARKER_STOP("foo");
    printf("sum = %f\n", sum);
}

int main() {
    LIKWID_MARKER_INIT;
    foo();
    LIKWID_MARKER_CLOSE;
    return 0;
}
```

- Marker APIs identifies the code regions to be profiled.
- Multiple regions can be selected.

Example with likwid

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

void foo() {
    LIKWID_MARKER_START("foo");
    double sum = 0.0;
    for (int i = 0; i < 1000000; i++)
      sum += i * 0.5;
    LIKWID_MARKER_STOP("foo");
    printf("sum = %f\n", sum);
}

int main() {
    LIKWID_MARKER_INIT;
    foo();
    LIKWID_MARKER_CLOS
    return 0;
}
```

```
$ gcc -DLIKWID_PERFMON example.c -o example -llikwid
[...]
$ likwid-perfctr -g MEM_DP -m ./example
```

Example with likwid

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

void foo() {
    LIKWID_MARKER_START("foo");
    double sum = 0.0;
    for (int i = 0; i < 1000000; i++)
      sum += i * 0.5;
    LIKWID_MARKER_STOP("foo");
    printf("sum = %f\n", sum);
}

int main() {
    LIKWID_MARKER_INIT;
    foo();
    LIKWID_MARKER_CLOSE
    return 0;
}
```

Use specific flags when compile

```
$ gcc -DLIKWID_PERFMON example.c -o example -llikwid
[...]
$ likwid-perfctr -g MEM_DP -m ./example
```

Example with likwid

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

void foo() {
    LIKWID_MARKER_START("foo");
    double sum = 0.0;
    for (int i = 0; i < 1000000; i++)
      sum += i * 0.5;
    LIKWID_MARKER_STOP("foo");
    printf("sum = %f\n", sum);
}

int main() {
    LIKWID_MARKER_INIT;
    foo();
    LIKWID_MARKER_CLOSE
    return 0;
}
```

Run `likwid-perfctr` with `-m`

```
$ gcc -DLIKWID_PERFMON exa    e.c -o example -llikwid
[...]
$ likwid-perfctr -g MEM_DP -m ./example
```

Code Instrumentation overhead

The additional instrumentation code causes additional overhead.

- Code instrumentation should always have a fixed cost, otherwise will interfere with the measurements.

To lower the overhead we can:

- decrease the sampling rate
- Reduce the amount of data stored by using “online” algorithms for calculating mean, variance, min, max, and other metrics.

Sampling

Sampling can be performed in 2 different modes:

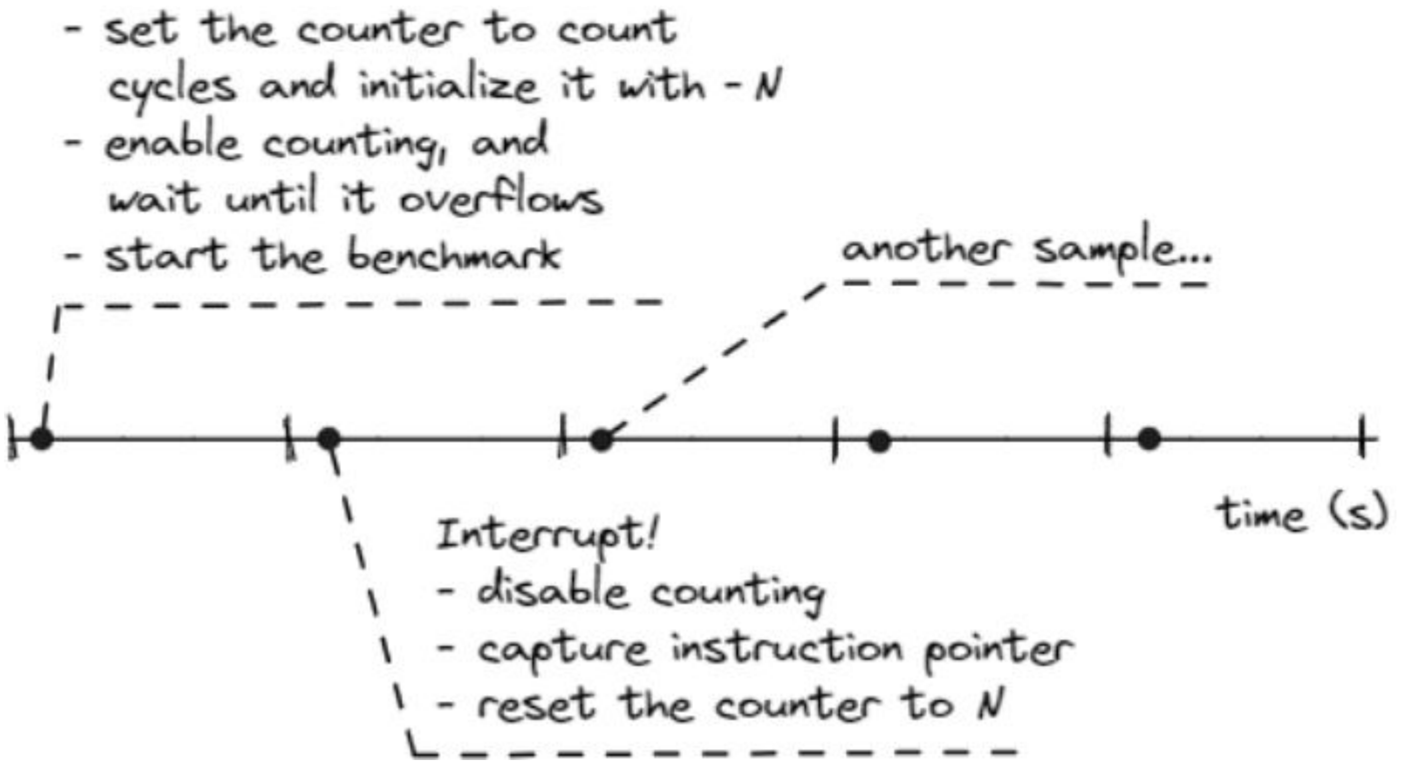
- user-mode:
  - it is a pure software approach (we set up an OS timer. Upon expiration, the application receives a SIGPROF)
  - can be used to identify hotspots
  - incurs higher runtime overhead than EBS.

- hardware event based sampling (EBS):
  - uses hardware PMCs to trigger interrupts
  - can be used for additional analysis types that involve PMCs, e.g., sampling on cache-misses, Top-down Microarchitecture Analysis, etc.

Finding Hotspots

When the counter overflows it triggers a **Performance Monitoring Interrupt (PMI)**, also known as SIGPROF.

At the start of a benchmark, we configure the event that we want to sample. **Sampling on cycles** is a default since we want to know where the program spends most of the time.



- set the counter to count cycles and initialize it with -N
- enable counting, and wait until it overflows
- start the benchmark
another sample...
time (s)
Interrupt!
- disable counting
- capture instruction pointer
- reset the counter to N

Finding Hotspots

When the counter overflows it

```
$ time -p perf record -F 1000 -- ./x264 -o /dev/null --slow --threads 1
../Bosphorus_1920x1080_120fps_420_8bit_YUV.y4m
[ perf record: Captured and wrote 1.625 MB perf.data (35035 samples) ]
real 36.20 sec
$ perf report -n --stdio
# Samples: 35K of event 'cpu_core/cycles/'
# Event count (approx.): 156756064947
# Overhead Samples Shared Object Symbol
# ........ ....... ............. ..............
  7.50%    2620    x264          [.] x264_8_me_search_ref
  7.38%    2577    x264          [.] refine_subpel.lto_priv.0
  6.51%    2281    x264          [.] x264_8_pixel_satd_8x8_internal_avx2
  6.29%    2212    x264          [.] get_ref_avx2.lto_priv.0
  5.07%    1787    x264          [.] x264_8_pixel_avg2_w16_sse2
  3.26%    1145    x264          [.] x264_8_mc_chroma_avx2
  2.88%    1013    x264          [.]
x264_8_pixel_satd_16x8_internal_avx2
  2.87%    1006    x264          [.] x264_8_pixel_avg2_w8_mmx2
  2.58%     904    x264          [.] x264_8_pixel_satd_8x8_avx2
  2.51%     882    x264          [.] x264_8_pixel_sad_16x16_sse2
....
```

Finding Hotspots

recording phase with sample rate F=*1000 per sec*

When the counter overflows it

```
$ time -p perf record -F 1000 -- ./x264 -o /dev/null --slow --threads 1
../Bosphorus_1920x1080_120fps_420_8bit_YUV.y4m
[ perf record: Captured and wrote 1.625 MB perf.data (35035 samples) ]
real 36.20 sec
$ perf report -n --stdio
# Samples: 35K of event 'cpu_core/cycles/'
# Event count (approx.): 156756064947
# Overhead Samples Shared Object Symbol
# ........ ....... ........... .............
   7.50%    2620    x264          [.] x264_8_me_search_ref
   7.38%    2577    x264          [.] refine_subpel.lto_priv.0
   6.51%    2281    x264          [.] x264_8_pixel_satd_8x8_internal_avx2
   6.29%    2212    x264          [.] get_ref_avx2.lto_priv.0
   5.07%    1787    x264          [.] x264_8_pixel_avg2_w16_sse2
   3.26%    1145    x264          [.] x264_8_mc_chroma_avx2
   2.88%    1013    x264          [.]
x264_8_pixel_satd_16x8_internal_avx2
   2.87%    1006    x264          [.] x264_8_pixel_avg2_w8_mmx2
   2.58%     904    x264          [.] x264_8_pixel_satd_8x8_avx2
   2.51%     882    x264          [.] x264_8_pixel_sad_16x16_sse2
....
```

Finding Hotspots

When the counter overflows it

```
$ time -p perf record -F 1000 -- ./x264 -o /dev/null --slow --threads 1
../Bosphorus_1920x1080_120fps
[ perf record: Captured and w
real 36.20 sec
$ perf report -n --stdio
# Samples: 35K of event 'cpu_core/cycles/'
# Event count (approx.): 156756064947
# Overhead Samples Shared Object Symbol
# ........ ....... ........... .............
  7.50%     2620    x264          [.] x264_8_me_search_ref
  7.38%     2577    x264          [.] refine_subpel.lto_priv.0
  6.51%     2281    x264          [.] x264_8_pixel_satd_8x8_internal_avx2
  6.29%     2212    x264          [.] get_ref_avx2.lto_priv.0
  5.07%     1787    x264          [.] x264_8_pixel_avg2_w16_sse2
  3.26%     1145    x264          [.] x264_8_mc_chroma_avx2
  2.88%     1013    x264          [.]
x264_8_pixel_satd_16x8_internal_avx2
  2.87%     1006    x264          [.] x264_8_pixel_avg2_w8_mmx2
  2.58%      904    x264          [.] x264_8_pixel_satd_8x8_avx2
  2.51%      882    x264          [.] x264_8_pixel_sad_16x16_sse2
....
```

report phase (-n show number of samples)

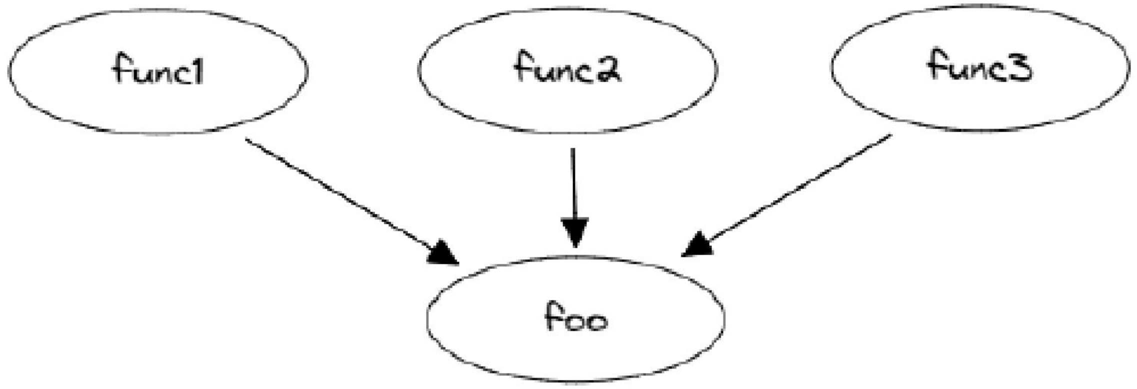
Finding Hotspots

```
$ time -p perf record -F 1000 -- ./x264 -o /dev/null --slow --threads 1
../Bosphorus_1920x1080_120fps_420_8bit_YUV.y4m
[ perf record: Captured and wrote 1.625 MB perf.data (35035 samples) ]
real 36.20 sec
$ perf report -n --stdio
# Samples: 35K of event 'cpu_core/cy
# Event count (approx.): 156756064
# Overhead Samples Shared Object Sy
# ........ ....... ........... .............
  7.50%    2620    x264          [.] x264_8_me_search_ref
  7.38%    2577    x264          [.] refine_subpel.lto_priv.0
  6.51%    2281    x264          [.] x264_8_pixel_satd_8x8_internal_avx2
  6.29%    2212    x264          [.] get_ref_avx2.lto_priv.0
  5.07%    1787    x264          [.] x264_8_pixel_avg2_w16_sse2
  3.26%    1145    x264          [.] x264_8_mc_chroma_avx2
  2.88%    1013    x264          [.]
x264_8_pixel_satd_16x8_internal_avx2
  2.87%    1006    x264          [.] x264_8_pixel_avg2_w8_mmx2
  2.58%     904    x264          [.] x264_8_pixel_satd_8x8_avx2
  2.51%     882    x264          [.] x264_8_pixel_sad_16x16_sse2
....
```

Total number of events (35000*4500000) □ 35K samples, 1000 samples/sec @4.5 GHz

Collecting Call Stacks

- When the hottest function is called from multiple functions, we need to know who call it.



- Profiling tools achieve this by capturing the call stack of the process

- There are several methods to collect the call stack.
  - Using the the frame pointer: `perf record --call-graph fp`
  - Using debug info: `perf record --call-graph dwarf`
  - Using hardware features (Intel Last Branch Record (LBR)) `perf record --call-graph lbr`

Collecting Call Stacks

```
$ perf record --call-graph lbr -- ./a.out
$ perf report -n --stdio --no-children
# Samples: 65K of event 'cycles:ppp'
# Event count (approx.): 61363317007
# Overhead Samples  Command Shared Object    Symbol
# ........ ........ ....... ................ ......................
  99.96%      65217 a.out   a.out            [.] foo
  |
  --99.96%--foo
  |
  |--55.52%--func1
  |          main
  |          __libc_start_main
  |          _start
  |
  |--33.32%--func2
  |          main
  |          __libc_start_main
  |          _start
  |
  --11.12%--func3
            main
            __libc_start_main
            _start
```

Collecting Call Stacks

```
$ perf record --call-graph lbr -- ./a.out
$ perf report -n --stdio --no-children
# Samples: 65K of event 'cycles:ppp'
# Event count (approx.): 61363317007
# Overhead Samples  Command Shared Object    Symbol
# ........ ....... ....... ................ ......................
  99.96%      652            .out            [.] foo
  |
  --99.96%--foo
  |
  |--55.52%--func1
  |          main
  |          __libc_start_main
  |          _start
  |
  |--33.32%--func2
  |          main
  |          __libc_start_main
  |          _start
  |
  --11.12%--func3
            main
            __libc_start_main
            _start
```

hotspot

perf report

`perf record` interprets and displays data collected by perf record.

- You can use interactive **TUI** (text-based user interface) or **STDIO** output

- if code is compiled with –g you can see both assembly and source code: e.g. `perf report --stdio -s srcline` shows the code line that triggered the sample

- You can also use `perf annotate` to see the corresponding assembly lines

Exercise

In the google shared folder there is a *sieve.c* file that find prime numbers using the so-called "Eratostene sieve“.

You can try to profile and optimize it. Here are some hints:

1. Try to use `perf record` to identify the hottest lines of code
2. First: improve the algorithm!
3. Try to use vectorization, parallelization, and unrolling
4. Finally, check if there is an additional SIMD instruction that can help you.

Bibliography

Yuliana Zamora, Robert Robey, Parallel and High Performance Computing, manning publications, (cap.3)

Denis, Bakhvalov, Performance Analysis And Tuning On Modern CPUs: Second Edition