

## Performance tools part 2: Performance analysis with hardware metrics

likwid-perfctr




# Probing performance behavior

- How do we find out about the performance properties and requirements of a parallel code?  
Profiling via advanced tools is often overkill
- A coarse overview is often sufficient: **likwid-perfctr**

Simple end-to-end measurement of hardware performance metrics

Operating modes:

- Wrapper
- Stethoscope
- Timeline
- Marker API

Preconfigured and extensible  
metric groups, list with  
**likwid-perfctr -a** 

BRANCH: Branch prediction miss rate/ratio  
CACHE: Data cache miss rate/ratio  
CLOCK: Clock frequency of cores  
DATA: Load to store ratio  
FLOPS\_DP: Double Precision MFlops/s  
FLOPS\_SP: Single Precision MFlops/s  
L2: L2 cache bandwidth in MBytes/s  
L2CACHE: L2 cache miss rate/ratio  
L3: L3 cache bandwidth in MBytes/s  
L3CACHE: L3 cache miss rate/ratio  
MEM: Main memory bandwidth in MBytes/s  
TLB: TLB miss rate/ratio  
ENERGY: Power and energy consumption

# likwid-perfctr wrapper mode

```
$ likwid-perfctr -g L2 -C S1:0-3 ./a.out
```

```
-----  
CPU name:      Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz [...]  
-----
```

```
<<<< PROGRAM OUTPUT >>>>
```

```
Group 1: L2
```

Event	Counter	Core 14	Core 15	Core 16	Core 17
INSTR_RETIRED_ANY	FIXC0	1298031144	1965945005	1854182290	1862521357
CPU_CLK_UNHALTED_CORE	FIXC1	2353698512	2894134935	2894645261	2895023739
CPU_CLK_UNHALTED_REF	FIXC2	2057044629	2534405765	2535218217	2535560434
L1D_REPLACEMENT	PMC0	212900444	200544877	200389272	200387671
L2_TRANS_L1D_WB	PMC1	112464863	99931184	99982371	99976697
ICACHE_MISSES	PMC2	21265	26233	12646	12363

Always measured for Intel CPUs

Configured metrics (this group)

```
[... statistics output omitted ...]
```

Metric	Core 14	Core 15	Core 16	Core 17
Runtime (RDTSC) [s]	1.1314	1.1314	1.1314	1.1314
Runtime unhalted [s]	1.0234	1.2583	1.2586	1.2587
Clock [MHz]	2631.6699	2626.4367	2626.0579	2626.0468
CPI	1.8133	1.4721	1.5611	1.5544
L2D load bandwidth [MBytes/s]	12042.7388	11343.8446	11335.0428	11334.9523
L2D load data volume [GBytes]	13.6256	12.8349	12.8249	12.8248
L2D evict bandwidth [MBytes/s]	6361.5883	5652.6192	5655.5146	5655.1937
L2D evict data volume [GBytes]	7.1978	6.3956	6.3989	6.3985
L2 bandwidth [MBytes/s]	18405.5299	16997.9477	16991.2728	16990.8453
L2 data volume [GBytes]	20.8247	19.2321	19.2246	19.2241

Derived metrics

# likwid-perfctr stethoscope mode

---

- likwid-perfctr counts events on cores; it has no notion of what kind of code is running (if any)

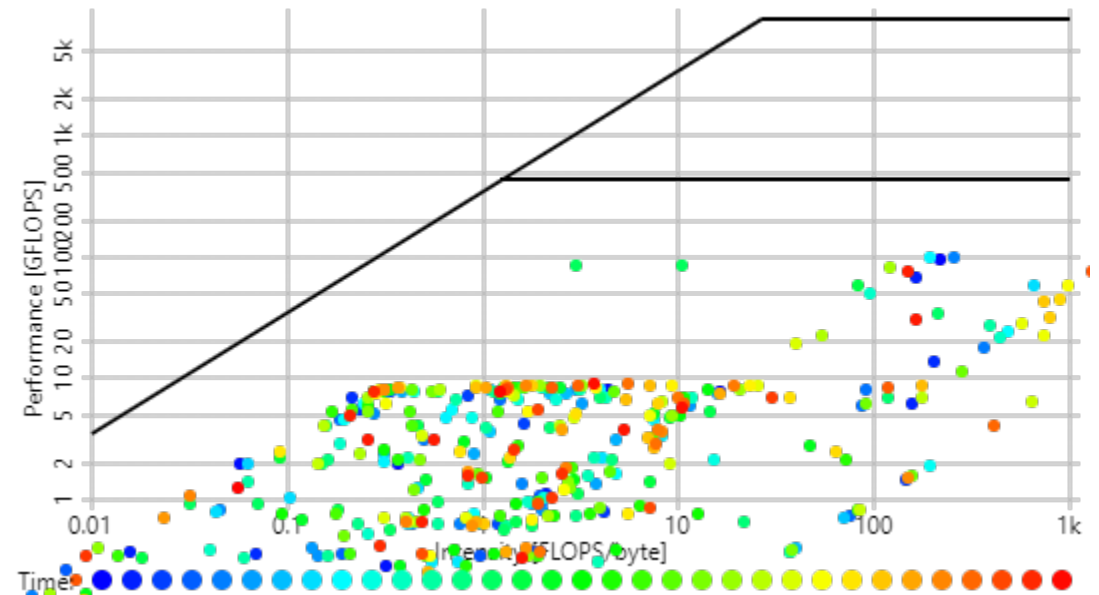
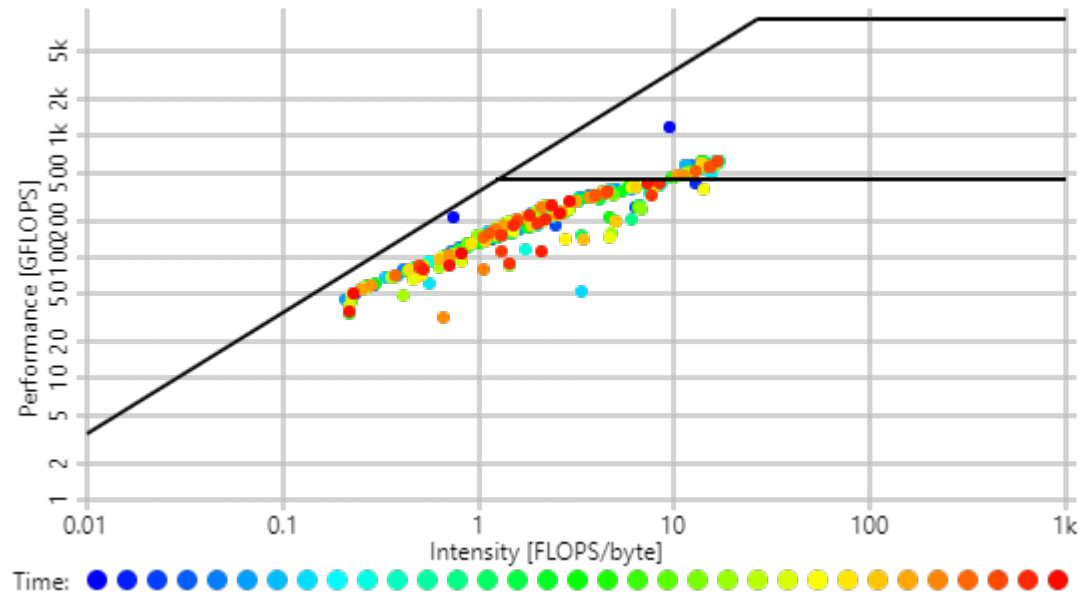
This allows you to “listen” to what is currently happening, **without any overhead:**

```
$ likwid-perfctr -c N:0-11 -g FLOPS_DP -S 10s
```

- Can be used as cluster/server monitoring tool
- Frequent use: monitor a long-running parallel application from outside

# Cluster monitoring with likwid-perfctr

Two jobs on the NHR@FAU “Fritz” cluster



<https://github.com/ClusterCockpit>

# likwid-perfctr with MarkerAPI

- The MarkerAPI can restrict measurements to **code regions**
- The API only reads counters.  
The configuration of the counters is still done by **likwid-perfctr**
- Multiple named regions allowed, accumulation over multiple calls
- Inclusive and overlapping regions allowed

- **Caveat:** Marker API can cause overhead; do not call too frequently!

```
#include <likwid-marker.h>

LIKWID_MARKER_INIT; // must be called from serial region
. . .
LIKWID_MARKER_START("Compute");
. . .
LIKWID_MARKER_STOP("Compute");
. . .
LIKWID_MARKER_START("Postprocess");
. . .
LIKWID_MARKER_STOP("Postprocess");
. . .
LIKWID_MARKER_CLOSE; // must be called from serial region
```

# likwid-perfctr with MarkerAPI: OpenMP code (C)

```
#include <likwid-marker.h>

int main(...) {
    LIKWID_MARKER_INIT;
    #pragma omp parallel
    {
        LIKWID_MARKER_REGISTER("MatrixAssembly");
    }
    ...
    #pragma omp parallel
    {
        LIKWID_MARKER_START("MatrixAssembly");
        #pragma omp for
        for(int i=0; i<N; ++i) { /* Loop */ }
        LIKWID_MARKER_STOP("MatrixAssembly");
    }
    ...
    LIKWID_MARKER_CLOSE;
}
```

Optional: Prepare data structures (reduced overhead on 1<sup>st</sup> marker call)

Call markers in parallel region if data should be taken on all threads

<https://github.com/RRZE-HPC/likwid/wiki/TutorialMarkerC>

# likwid-perfctr with MarkerAPI: OpenMP code (Fortran)

```
program p
  use likwid
  call likwid_markerInit
  !$omp parallel
    call likwid_markerRegisterRegion ("MatrixAssembly")
  !$omp end parallel
  ...
  !$omp parallel
    call likwid_markerStartRegion ("MatrixAssembly")
    !$omp do
    do i=1,N
      ! Loop
    enddo
    !$omp end do
    call likwid_markerStopRegion ("MatrixAssembly")
  !$omp end parallel
  ...
  call likwid_markerClose
end program p
```

<https://github.com/RRZE-HPC/likwid/wiki/TutorialMarkerF90>



# Compiling, linking, and running with marker API

## Compile:

```
cc -I /path/to/likwid.h -DLIKWID_PERFMON -c program.c
```

## Link:

```
cc -L /path/to/liblikwid program.o -o program -llikwid
```

## Run:

```
likwid-perfctr -C <CPULIST> -g <GROUP> -m ./program
```

## MPI:

```
likwid-mpirun -np 4 -pin <PINEXPR> -g <GROUP> -m ./program
```

→ One separate block of output for every marked region

Activate LIKWID  
macros (C only)

Activate  
markers

# So... what should I look at first?

Focus on **resource utilization** and **instruction decomposition**!

Metrics to measure:

- Operation throughput (Flops/s)
- Overall instruction throughput (IPC,CPI)
- **Instruction breakdown**:
  - FP instructions
  - loads and stores
  - branch instructions
  - other instructions
- Instruction breakdown to **SIMD width** (scalar, SSE, AVX, AVX512 for x86)
- **Data volumes** and **bandwidths** to main memory (GB and GB/s)
- Data volumes and bandwidth to different cache levels (GB and GB/s)

Useful diagnostic metrics are:

- Clock frequency (GHz)
- Power (W)

All the above metrics can be acquired using performance groups:

**MEM\_DP, MEM\_SP, BRANCH, DATA, L2, L3**

# Example: triangular matrix-vector multiplication

```
#define N 10000 // matrix in memory
#define ROUNDS 10
// Initialization
fillMatrix(mat, N*N, M_PI);
fillMatrix(bvec, N, M_PI);

// Calculation loop
#pragma omp parallel
{
    for (int k = 0; k < ROUNDS; k++) {
        #pragma omp for private(current,j)
        for (int i = 0; i < N; i++) {
            current = 0;
            for (int j = i; j < N; j++)
                current += mat[(i*N)+j] * bvec[j];
            cvec[i] = current;
        }
        while (cvec[N>>1] < 0) {dummy();break;}
    }
}
```



Guard with impossible condition and `dummy()` function call required if `cvec` not used afterwards

# Example: triangular matrix-vector multiplication

```
#include <likwid-marker.h>
[...] // defines, fillMatrix, init data
LIKWID_MARKER_INIT;
#pragma omp parallel
{
    for (int k = 0; k < ROUNDS; k++) {
        LIKWID_MARKER_START("Compute");
        #pragma omp for private(current,j)
        for (int i = 0; i < N; i++) {
            current = 0;
            for (int j = i; j < N; j++)
                current += mat[(i*N)+j] * bvec[j];
            cvec[i] = current;
        }
        LIKWID_MARKER_STOP("Compute");
        while (cvec[N>>1] < 0) {dummy();break;}
    }
}
LIKWID_MARKER_CLOSE;
```



# Example: triangular matrix-vector multiplication

```
$ likwid-perfctr -C 0,1,2 -g L2 -m ./a.out
```

```
-----  
CPU type:      Intel Core SandyBridge EN/EP processor  
CPU clock:     3.09 GHz  
-----
```

```
=====  
Group 1: Region Compute  
=====
```

```
+-----+-----+-----+-----+  
| Region Info | Core 0 | Core 1 | Core 2 |  
+-----+-----+-----+-----+  
| RDTSC Runtime [s] | 0.161382 | 0.161365 | 0.161365 |  
| call count | 10 | 10 | 10 |  
+-----+-----+-----+-----+
```

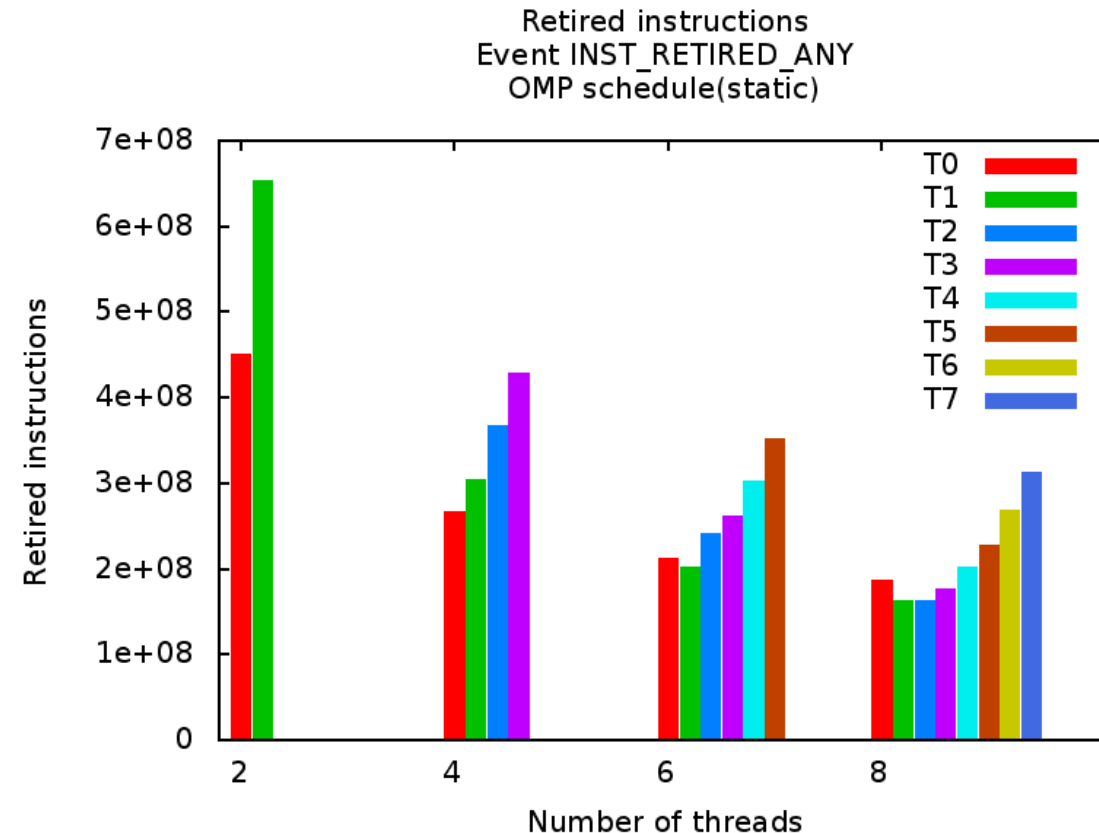
```
+-----+-----+-----+-----+  
| Event | Counter | Core 0 | Core 1 | Core 2 |  
+-----+-----+-----+-----+  
| INSTR_RETIRED_ANY | FIXC0 | 2.626800e+08 | 3.187585e+08 | 3.780255e+08 |  
| CPU_CLK_UNHALTED_CORE | FIXC1 | 4.972802e+08 | 4.961411e+08 | 4.933711e+08 |  
| CPU_CLK_UNHALTED_REF | FIXC2 | 4.972801e+08 | 4.961404e+08 | 4.933714e+08 |  
| L1D_REPLACEMENT | PMC0 | 5.490278e+07 | 3.927353e+07 | 2.364295e+07 |  
| L1D_M_EVICT | PMC1 | 2.920200e+04 | 2.876600e+04 | 2.861000e+04 |  
| ICACHE_MISSES | PMC2 | 4.649000e+03 | 4.984000e+03 | 5.321000e+03 |  
+-----+-----+-----+-----+
```

???

# Example: triangular matrix-vector multiplication

Retired instructions are misleading!

Waiting in implicit OpenMP barrier executes many instructions



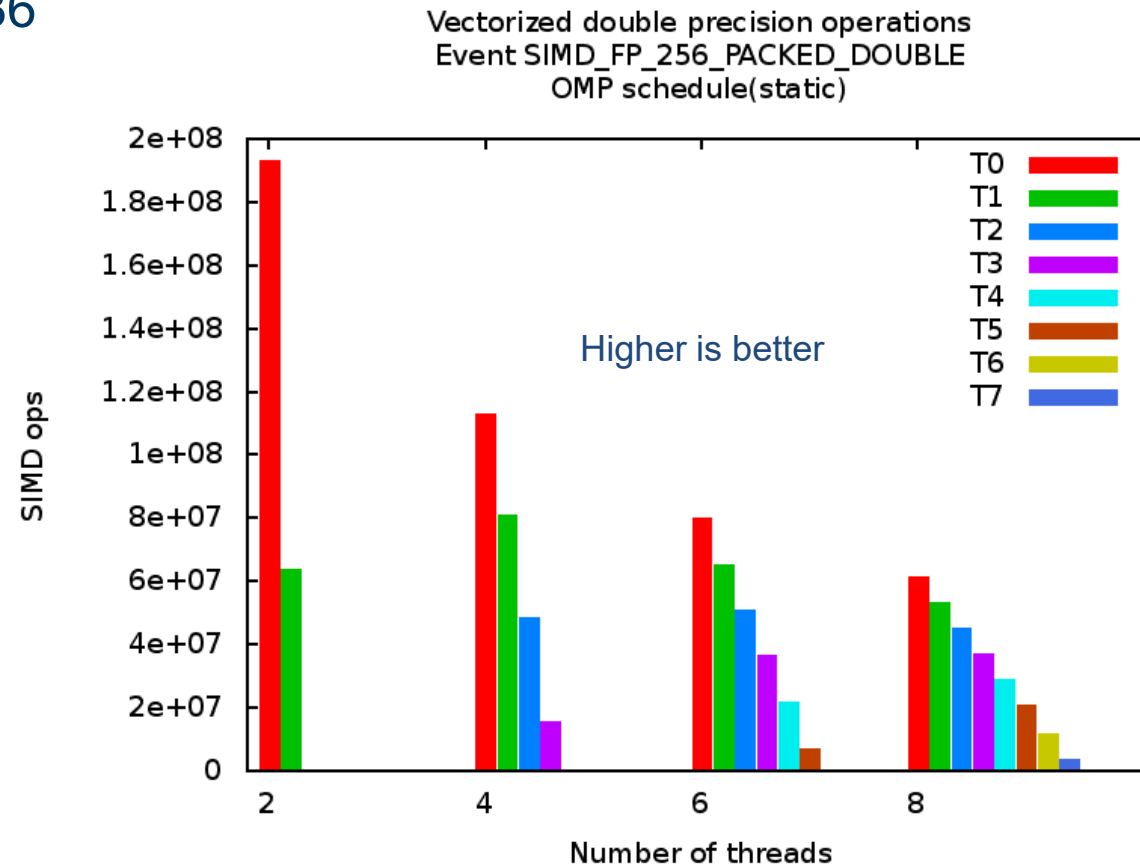
We need to measure actual work (or use a tool that can separate user from runtime lib instructions)

# Example: triangular matrix-vector multiplication

Floating-point instructions reliable ↔ useful work metric

## Caveats

- FP instr. counters from SandyBridge to Haswell are only qualitatively correct
- Masked SIMD lanes cannot be counted directly on x86

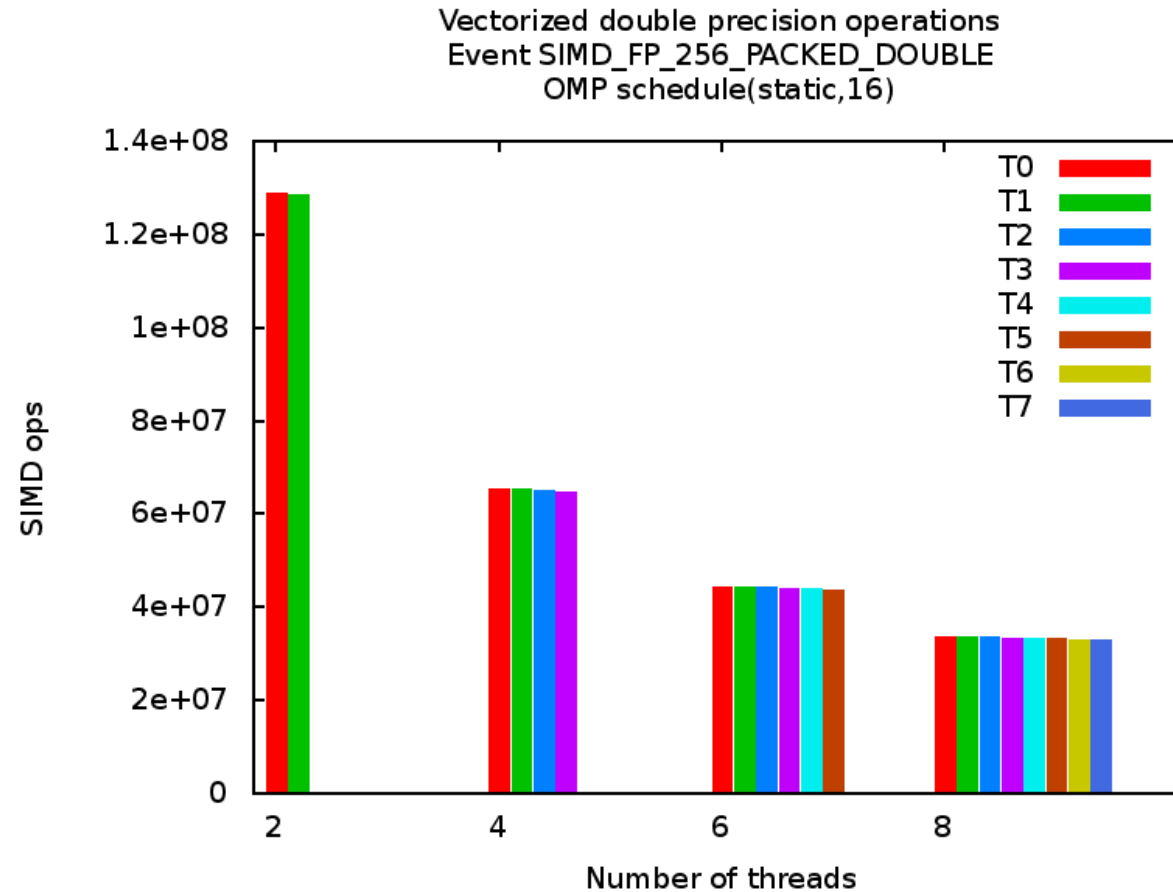


# Example: triangular matrix-vector multiplication

Changing OMP schedule to **static** with **chunk size 16** ↔ smaller work packages per thread

No imbalance anymore!

Is it also faster?





# Summary of hardware performance monitoring

- Useful **only if you know what you are looking for**
  - Hardware event counting bears the potential of acquiring massive amounts of data for nothing!
- **Resource-based metrics** are most useful
  - Cache lines transferred, work executed, loads/stores, cycles
  - Instructions, CPI, cache misses may be misleading
- **Caveat: Processor work != user work**
  - Waiting time in libraries (OpenMP, MPI) may cause lots of instructions
  - → distorted application characteristic
- Another very useful application of PM: **validating performance models!**
  - Roofline is data centric → measure data volume through memory hierarchy