

Measurement: Hardware Event Counters

Vojtěch Horký

Peter Libič

Petr Tůma

2010 – 2023

This work is licensed under a “CC BY-NC-SA 3.0” license. Created to support the Charles University Performance Evaluation lecture. See <http://d3s.mff.cuni.cz/teaching/performance-evaluation> for details.

Contents

1	Overview	1
2	Hardware	1
3	Tool: perf	4
4	Tool: PAPI	8
5	Tool: LIKWID	9
6	Analysis	11

1 Overview

Hardware Performance Event Counters

Modern processors include features to report performance relevant events.

Events

Selected performance relevant events in processor operation, typically:

- individual instruction execution steps,
- memory cache operations,
- and many more.

Counters

Processor registers that count the occurrence of selected events.

Information collected through hardware performance event counters can help explain how the executing software interacts with the processor hardware.

2 Hardware

Hardware Architecture

Detectors

A detector is the hardware circuit responsible for reporting event occurrence.

Typical detector features:

- one detector per event type PERFEVTSEL Event Select

- configurable through additional event mask PERFEVTSEL Unit Mask
- reporting event count per cycle for events that can happen multiple times simultaneously

Counters

A counter is the hardware register that counts configured event occurrences.

Typical counter features:

- count in user mode or kernel mode or both
- count on single thread or on all threads across core
- count active state cycles or cycles with inactive-to-active transitions
- active state defined as event count per cycle above or below threshold
- generate interrupt on counter overflow

Architectural Performance Monitoring

Monitoring features compatible across multiple processor models.

- Standard events (clock, instructions, cache misses, branch misses)
- Standard MSR layout for configuring counters
- Fixed function performance counters

Model Specific Performance Monitoring

Monitoring features specific to single processor model.

- Model specific events (typically hundreds)
- Precise event based sampling
- Off core monitoring

For Intel processors, the list of available performance monitoring events used to be a part of the Intel Architectures Software Developer's Manual. Recent versions of the manual no longer contain such list, instead an online list is available at <https://perfmon-events.intel.com>.

For AMD processors, the list of available performance monitoring events is included in the Processor Programming Reference.

Counting Speculations

Speculative execution is essential to performance. But how should speculative events be counted ?

Instruction Retirement

Instruction results subject to speculation are not visible immediately. At instruction retirement phase, the processor decides whether:

- speculation was valid and results are made visible, or
- speculation was invalid and results are dropped as bogus.

Counting speculation:

- exact behavior depends on event type
 - some events inherently include speculation (branch misses)
 - events counted at retirement can typically exclude speculation
- counting at retirement introduces delay between event and count

Processor Event Based Sampling

What if more information about specific event is needed ?

Record Sampling

Processor can save detailed information on event into memory buffer. Obviously this can only be done rarely or too much data is collected.

Event record contains:

- general purpose register content
- event related data (relevant address, load latency, load source)

Stored on counter overflow or on random memory load (latency sampling).

Debug Store

Event records are placed in debug store memory buffer. Interrupt is generated on buffer (near) full. Store also used for branch trace sampling.

Accessing Counters

Counter configuration done through MSR registers.

Writing

Writing requires kernel mode (WRMSR). Restricted due to security implications.

Reading

Reading typically possible from user mode (RDPMC). Features such as multiplexing or overflow handling require kernel mode anyway.

Sharing

Counters in use can be recognized in PERF_GLOBAL_INUSE MSR. Kernel context switches counters that it configured.

Example: Sandy Bridge

Numbers

3 fixed counters per thread (instructions retired and two clock types) 8 general purpose counters per core (4 per thread)
7 architectural performance events Around 200 model specific performance events 2 off core event counters (separate event types)

Features

PEBS for 4 general purpose counters and 7 event types Load latency sampling with PEBS Store destination sampling with PEBS Skid compensation for INST_RETIRED

Common Issues

Certain common issues are to be expected whenever using counters.

Inherent Properties

- Some measurement configurations not supported by hardware
- Some measurement configurations require multiplexing
 - Multiplexing can happen quite invisibly
 - Multiplexing can introduce bias
- Some sampling approaches inaccurate in principle

Wrong Usage

Relying on counter name to guess what it does is naive.

Bugs

Some processors are known to misreport values of some counters.

Example: Estimating Memory Boundedness

The goal is to determine whether workload waits for memory.

L1-DCACHE-LOADS and L1-DCACHE-LOAD-MISSES

Looks like perfect pair of counters for estimating L1 traffic ?

- L1-DCACHE-LOADS counts instructions

- L1-DCACHE-LOAD-MISSES counts cache lines
- L1-DCACHE-LOAD-MISSES counts writes with RFO

MEM-LOAD-UOPS-RETIRED with various masks

Used to determine source of data in case of L1 miss ?

- Prefetch counted as hit in LFB or L1
- False positives on hits in L2
- False positives on hits in L3
- Reads due to writes with RFO not counted

1

The example is based on the paper by Molka et al.: Detecting Memory Boundedness ... doi:10.1145/3030207.3030223. Examine Figure 2 for details on the processor architecture. Examine Figure 3 for behavior of MEM-LOAD-UOPS-RETIRED. The relevant workload uses 256b (32B) loads. For introduction into compute-bound vs memory-bound workloads see the paper by Williams et al.: Roofline ... doi:10.1145/1498765.1498785. For examples of roofline models with dynamic application behavior traces see the paper by Lorenzo et al.: 3DyRM ... doi:10.1007/s11227-014-1163-4.

3 Tool: perf

Linux perf system

A framework for collecting performance related information in Linux.

Kernel Interface

- Configuration syscall
 - Selected hardware events
 - Standardized cache events
 - Low level device PMU events
 - Also software generated performance events
- Using file descriptors to operate

Userspace Tool

- Configuration
- Counting
- Sampling

Userspace perf Command

Higher-level tool to measure low-level performance in Linux. Basic features:

- Event counting
- Event-based sampling
- Supports many events:
 - Kernel provided hardware events
 - Platform specific hardware events
 - Raw event setting interface (fun to use)
 - Software events
 - perf list
- Automatic multiplexing

Gotchas

File descriptor use equals number of events times threads. Some events wrong or misleading (need to read kernel source).

¹Based on Molka et al.: Detecting Memory Boundedness ... doi:10.1145/3030207.3030223

perf stat command

Command to count the events of specified program.

```
perf stat [-e <EVENT> | --event=EVENT] [-a] <command>
```

Common events to specify:

- cpu-cycles/cycles
- instructions
- cache-references
- cache-misses
- branch-instrucions/branches
- branch-misses
- page-faults
- context-switches/cs
- cpu-migrations/migrations

Example Output

```
> perf stat <command>
...
Performance counter stats for '<command>':

 837.477207 task-clock (msec)    # 0.998 CPUs utilized
      89 context-switches      # 0.106 K/sec
       3 cpu-migrations        # 0.004 K/sec
    2,155 page-faults          # 0.003 M/sec
1,974,600,340 cycles            # 2.358 GHz                    (83.27%)
1,103,816,503 stalled-cycles-frontend # 55.90% frontend cycles idle (83.34%)
 682,412,732 stalled-cycles-backend # 34.56% backend cycles idle (66.74%)
2,124,238,728 instructions     # 1.08 insns per cycle
                                   # 0.52 stalled cycles per insn (83.45%)
 200,423,362 branches          # 239.318 M/sec                (83.40%)
   29,812 branch-misses       # 0.01% of all branches       (83.33%)

0.839128135 seconds time elapsed
```

Kernel Interface

First, opening counting mechanism, getting the file descriptor:

- perf_event_open() syscall

Then manipulation using:

- ioctl(), prctl() – control mechanism
- read(), mmap() – getting the data

And closing the descriptor:

- close()

perf_event_open() Syscall

```
#include <linux/perf_event.h>
```

```
#include <linux/hw_breakpoint.h>
```

```
int perf_event_open(struct perf_event_attr *attr,
                   pid_t pid, int cpu, int group_fd,
                   unsigned long flags);
```

- No library helper – call using syscall() function.
- For details, see manpage. Excerpts shown here.

Interesting perf_event_attr fields

```

struct perf_event_attr {
    __u32 type;          /* Type of event */
    __u32 size;         /* Size of structure, use sizeof */
    __u64 config;       /* Type-specific configuration */

    union {
        __u64 sample_period; /* Period of sampling */
        __u64 sample_freq;   /* Frequency of sampling */
    };

    __u64 sample_type; /* Specifies values included in sample */
    __u64 read_format; /* Specifies values returned in read */
    __u64 disabled : 1, /* off by default */
        inherit : 1, /* children inherit it */
        pinned : 1, /* must always be on PMU */
        exclusive : 1, /* only group on PMU */
        exclude_user : 1, /* don't count user */
        exclude_kernel : 1, /* don't count kernel */
        exclude_hv : 1, /* don't count hypervisor */
        exclude_idle : 1, /* don't count when idle */

    __u64 branch_sample_type; /* enum perf_branch_sample_type */
    __u64 sample_regs_user; /* user regs to dump on samples */
    __u32 sample_stack_user; /* size of stack to dump on samples */

    ...
}

```

Interesting perf_event_attr fields

Event type field

- PERF_TYPE_HARDWARE - several predefined events
- PERF_TYPE_HW_CACHE - standardized cache events
- PERF_TYPE_RAW - raw config written into event select MSR
- /sys/devices/*/type - raw config written through PMU driver

Event config field

- Specific format for each type
- Currently up to 24 bytes (config, config1, config2)
- Check /sys/devices/*/format/* for PMU driver fields
- Need to know specific system hardware

Cache Event Settings

The cache event settings are hardcoded in the kernel sources. For example, see the definition of static `__initconst const u64 skl_hw_cache_event_ids` in the Linux kernel sources at `arch/x86/events/intel/core.c` (online https://elixir.bootlin.com/linux/latest/C/ident/skl_hw_cache_event_ids):

- L1D READ OP COUNT = 0x81D0
Event Select D0, UMask 81 = MEM_UOPS_RETIRED.ALL_LOADS
- L1D READ MISS COUNT = 0x151
Event Select 51, UMask 01 = L1D.REPLACEMENT
- so the first one counts instructions, the second one counts lines.

Reading counters

Counters can be read by `read()` syscall, returns this data:

```

struct read_format {
    u64 value;          /* The value of the event */
    u64 time_enabled; /* if PERF_FORMAT_TOTAL_TIME_ENABLED */
    u64 time_running; /* if PERF_FORMAT_TOTAL_TIME_RUNNING */
    u64 id;            /* if PERF_FORMAT_ID */
};

```

Reading samples

Samples can be read from a ring buffer mapped using `mmap()`, uses this header, data is event type specific:

```

struct perf_event_mmap_page {
    __u32 version; /* version number of this structure */
    __u32 compat_version; /* lowest version this is compat with */
};

```

```

__u32 lock;          /* seqlock for synchronization */
__u32 index;        /* hardware counter identifier */
__s64 offset;       /* add to hardware counter value */
__u64 time_enabled; /* time event active */
__u64 time_running; /* time event on CPU */
__u64 capabilities; /* capabilities union */

...

__u64 data_head;    /* head in the data section */
__u64 data_tail;    /* user-space written tail */
}

```

Counter control – ioctl()

To change the status of counter, we can call `ioctl()`. Some interesting commands:

- `PERF_EVENT_IOC_ENABLE`
- `PERF_EVENT_IOC_DISABLE`
- `PERF_EVENT_IOC_RESET`

More perf Examples

The examples use the prime sieve example from <https://github.com/d-iii-s/teaching-performance-evaluation/tree/master/src/experiment-prime-sieve>.

Provide variability indication for instruction count.

```
perf stat -e instructions -r 5 ./checked-smart
```

Count in user mode only.

```
perf stat -e instructions:u -r 5 ./checked-smart
```

Count in kernel mode only.

```
perf stat -e instructions:k -r 5 ./checked-smart
```

See how counter multiplexing interpolates results.

Increase the count of instruction events requested

until interpolated results are printed.

```
perf stat -e instructions -r 5 ./checked-smart
```

```
perf stat -e instructions,instructions -r 5 ./checked-smart
```

```
perf stat -e instructions,instructions,instructions -r 5 ./checked-smart
```

...

Counter groups are measured together even with counter multiplexing.

```
perf stat -e '{instructions,instructions}' -r 5 ./checked-smart
```

Predefined counter groups are available.

```
perf stat -M CacheMisses ./checked-smart
```

Count across all processes while a command is running.

May require administrative privileges.

```
perf stat -a -e instructions -r 5 ./checked-smart
```

Report results for each processor separately.

May require administrative privileges.

```
perf stat -A -a -e instructions -r 5 ./checked-smart
```

Record and report memory access information.

```
perf mem record ./basic
```

```
perf mem report
```

Attach to a running process.

```
perf stat -p <PID> ...
```

Display interactive results.

4 Tool: PAPI

PAPI

PAPI is a standard library that provides application level interface to performance counters on multiple hardware and operating system platforms.

PAPI does not contain code to access performance counters, support from operating system kernel is required.

Main features of PAPI are:

- per thread virtual performance counters,
- support for platform independent events,
- support for counter multiplexing.

<http://icl.cs.utk.edu/papi>

Programming with PAPI

Concepts

Native event is a performance event supported on a specific platform.

Preset event is a standardized performance event that the library supports on multiple platforms. Some preset events are derived from multiple native events.

Event set is a set of events that are monitored together.

Example simplified from PAPI distribution. Error checking omitted for brevity.

```
// Initialize the library.
PAPI_library_init (PAPI_VER_CURRENT);

// Define an event set to be used.
int events = PAPI_NULL;
PAPI_create_eventset (&events);
PAPI_add_event (events, PAPI_TOT_INS);
PAPI_add_event (events, PAPI_TOT_CYC);

// Start counting using the event set.
long long values [2];
PAPI_start (events);
...
PAPI_read (events, values);
...
PAPI_stop (events, values);

...

// Define an event set with a native event.
int events = PAPI_NULL;
PAPI_create_eventset (&events);
int native;
PAPI_event_name_to_code ("RESOURCE_STALLS:ANY", &native);
PAPI_add_event (events, native);

...
```



```

// Initialize the library.
PAPI_library_init (PAPI_VER_CURRENT);
PAPI_multiplex_init ();

// Create event set with multiplex flag.
PAPI_create_eventset (&events);
PAPI_assign_eventset_component (events, 0);
PAPI_set_multiplex (events);

...

// Close the library.
PAPI_cleanup_eventset (events);
PAPI_destroy_eventset (&events);
PAPI_shutdown ();

```

5 Tool: LIKWID

Hardware Topology Information

```

> likwid-topology -g
Socket 0:
+-----+
| +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ |
| | 0 16 | | 2 18 | | 4 20 | | 6 22 | | 8 24 | | 10 26 | | 12 28 | | 14 30 | |
| +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ |
| | 32kB | | 32kB | | 32kB | | 32kB | | 32kB | | 32kB | | 32kB | | 32kB | |
| +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ |
| | 256kB | | 256kB | | 256kB | | 256kB | | 256kB | | 256kB | | 256kB | |
| +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ |
| |                                     20MB                                     |
| +-----+
Socket 1:
+-----+
| +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ |
| | 1 17 | | 3 19 | | 5 21 | | 7 23 | | 9 25 | | 11 27 | | 13 29 | | 15 31 | |
| +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ |
| | 32kB | | 32kB | | 32kB | | 32kB | | 32kB | | 32kB | | 32kB | | 32kB | |
| +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ |
| | 256kB | | 256kB | | 256kB | | 256kB | | 256kB | | 256kB | | 256kB | |
| +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ |
| |                                     20MB                                     |
| +-----+

```

Performance Counter Groups

```

> likwid-perfctr -a
BRANCH      Branch prediction miss rate/ratio
CACHES      Some data from the CBOXes
CLOCK       Power and Energy consumption
ENERGY      Power and Energy consumption
MEM         Main memory bandwidth in MBytes/s
MEM         Local and remote memory accesses
NUMA
QPI         QPI traffic between sockets
...
> likwid-perfctr -H -g CACHES
Group CACHES:
Formulas:
L1 to L2 Load [MBytes/s] = 1.0E-06*L1D_REPLACEMENT*64/time
L1 to L2 Evict [MBytes/s] = 1.0E-06*L1D_M_EVICT*64/time
L1 to L2 bandwidth [MBytes/s] = 1.0E-06*(L1D_REPLACEMENT+L1D_M_EVICT)*64/time
L1 to L2 data volume [GBytes] = 1.0E-09*(L1D_REPLACEMENT+L1D_M_EVICT)*64
L2 to L3 Load [MBytes/s] = 1.0E-06*L2_LINES_IN_ALL*64/time
L2 to L3 Evict [MBytes/s] = 1.0E-06*L2_LINES_OUT_DIRTY_ALL*64/time

```

$L2 \text{ to } L3 \text{ bandwidth [MBytes/s]} = 1.0E-06 * (L2_LINES_IN_ALL + L2_LINES_OUT_DIRTY_ALL) * 64 / \text{time}$
 $L2 \text{ to } L3 \text{ data volume [GBytes]} = 1.0E-09 * (L2_LINES_IN_ALL + L2_LINES_OUT_DIRTY_ALL) * 64$
 $L3 \text{ avg clock [MHz]} = 1.E-06 * (\text{SUM}(CBOX_C3)) / 8$
 $L3 \text{ to Memory data volume [MBytes/s]} = 1.0E-06 * (\text{SUM}(CBOX_C1)) * 64 / \text{time}$
 $L3 \text{ to Memory data volume [MBytes]} = 1.0E-06 * (\text{SUM}(CBOX_C1)) * 64$
 $\text{Memory Read BW [MBytes/s]} = 1.0E-06 * (\text{CAS_COUNT_RD} + \text{CAS_COUNT_RD} + \text{CAS_COUNT_RD} + \text{CAS_COUNT_RD}) * 64.0 / \text{time}$
 $\text{Memory Write BW [MBytes/s]} = 1.0E-06 * (\text{CAS_COUNT_WR} + \text{CAS_COUNT_WR} + \text{CAS_COUNT_WR} + \text{CAS_COUNT_WR}) * 64.0 / \text{time}$
 $\text{Memory BW [MBytes/s]} = 1.0E-06 * (\text{CAS_COUNT_RD} + \text{CAS_COUNT_RD} + \text{CAS_COUNT_RD} + \text{CAS_COUNT_RD} + \text{CAS_COUNT_WR} + \text{CAS_COUNT_WR} + \text{CAS_COUNT_WR} + \text{CAS_COUNT_WR}) * 64.0 / \text{time}$
 $\text{Memory data volume [GBytes]} = 1.0E-09 * (\text{CAS_COUNT_RD} + \text{CAS_COUNT_RD} + \text{CAS_COUNT_RD} + \text{CAS_COUNT_RD} + \text{CAS_COUNT_WR} + \text{CAS_COUNT_WR} + \text{CAS_COUNT_WR} + \text{CAS_COUNT_WR}) * 64.0$

-

Group to measure cache transfers between L1 and Memory

Power Consumption Measurement

```
> likwid-powermeter -p <command>
```

```
...
```

Event	Counter	Sum	Min	Max	Avg
INSTR_RETIRED_ANY STAT	FIXC0	41896186054	6229866	25190405634	5237023256
CPU_CLK_UNHALTED_CORE STAT	FIXC1	117822959664	54450033	86042981021	14727869958
CPU_CLK_UNHALTED_REF STAT	FIXC2	107195405790	49991172	77859622797	13399425723
PWR_PKG_ENERGY STAT	PWR0	633.8951	0	633.8951	79.2368875
PWR_DRAM_ENERGY STAT	PWR3	0	0	0	0

```
...
```

Metric	Sum	Min	Max	Avg
Runtime (RDTS) [s] STAT	193.45664	24.18208	24.18208	24.18208
Runtime unhaltd [s] STAT	35.78692096	0.01653836	26.13424	4.47336512
Clock [MHz] STAT	28438.283	3488.899	3638.386	3554.785375
CPI STAT	28.455989	1.033677	9.002153	3.556998625
Energy [J] STAT	633.8951	0	633.8951	79.2368875
Power [W] STAT	26.21342	0	26.21342	3.2766775
Energy DRAM [J] STAT	0	0	0	0
Power DRAM [W] STAT	0	0	0	0

Intel processors export power consumption estimates through their RAPL (Running Average Power Limiting) interfaces. AMD processors export power consumption estimates through their APM (Application Power Management) interfaces. Both interfaces estimate power consumption from other indicators of processor activity, but studies show them to be often reasonably accurate.

See the paper by Hackenberg et al.: Power Measurement Techniques ... doi:10.1109/ISPASS.2013.6557170 for a measurement accuracy study. The authors indicate that the power consumption measurement interfaces deliver samples at about millisecond resolution, often accurate to within units of percent, but with multiple issues that indicate careful evaluation of the measurement results is required.

More likwid Examples

The listing uses the prime sieve example from <https://github.com/d-iii-s/teaching-performance-evaluation/tree/master/src/experiment-prime-sieve>.

```
# List performance groups supported on current architecture.
```

```
likwid-perfctr -a
```

```
# See how predictable the branches are in the example.
```

```
likwid-perfctr -g BRANCH -C 1 ./smart
```

```
likwid-perfctr -g BRANCH -C 1 ./checked-smart
```

```
# See what energy savings are delivered by optimizations in the example.
```

```
# Try this with and without frequency scaling.
```

```
likwid-powermeter -p ./smart
```

```
likwid-powermeter -p ./checked-smart
```

```
# Display power consumption continuously.
```

```
# Try this with and without frequency scaling.
```

```
while true ; do likwid-powermeter -p ; done
```

Try the examples on different machines too.

6 Analysis

Example: Top Down Analysis

The goal is to identify major performance bottlenecks.

Analysis Hierarchy

Defined to reflect modern processor architecture. Classifies individual pipeline slots.

- Frontend bound slots Slots where no μ ops are issued despite backend not stalled
- Backend bound slots Slots where no μ ops are issued because backend is stalled
- Bad speculation slots Slots with μ ops that were eventually discarded as bogus
- Retirement slots Slots with μ ops that retire normally (this is good)

Further hierarchy steps for each branch follow.

2

The example is based on the paper by on Yasin: A Top Down Method ... doi:10.1109/ISPASS.2014.6844459. Examine Figure 1 for details on the general processor architecture. Examine Figure 2 for the hierarchy of slot classifications. Figure 4 provides an example flow chart to classify memory related stalls in backend bound slots. Appendix I gives the specific hardware performance counters used to apply the slot classification.

Looking at Figures 5 and 6 gives an impression of what kind of information the analysis delivers. The figures include mostly backend bound workloads, some information can also be gleaned from comparing different parallelism levels (a shift from cache bound to memory bound slots indicates potential cache capacity issue).

Figure 8 demonstrates how differences in processor architecture are reflected on the classification.

Also fun to note is the fact that some figures contain negative values, due to counter errata.

The top down analysis is (partially) supported by several existing software tools, including the 'likwid' tool and the 'perf' tool:

```
# System wide top down analysis mode.
taskset -c 0 perf stat --topdown --all-cpus ./basic
```

```
Performance counter stats for 'system wide':
```

	retiring	bad speculation	frontend bound	backend bound
S0-D0-C0 2	6.0%	1.1%	6.8%	86.0%

```
3.001612386 seconds time elapsed
```

```
# Group based top down analysis mode on level 1.
perf stat -M TopdownL1 ./basic
```

```
Performance counter stats for './basic':
```

73,344,162	CPU_CLK_UNHALTED.REF_XCLK	#	6.1 %	tma_retiring	
		#	0.9 %	tma_bad_speculation	(41.60%)
56,318,031	INT_MISC.RECOVERY_CYCLES_ANY				(41.63%)
73,097,178	CPU_CLK_UNHALTED.ONE_THREAD_ACTIVE				(41.66%)
9,452,489,999	CPU_CLK_UNHALTED.THREAD				(41.70%)
2,285,808,627	UOPS_RETIRED.RETIRE_SLOTS				(41.73%)
2,495,316,639	UOPS_ISSUED.ANY				(41.74%)
73,352,017	CPU_CLK_UNHALTED.REF_XCLK	#	7.3 %	tma_frontend_bound	
		#	85.8 %	tma_backend_bound	(41.74%)
2,770,484,463	IDQ_UOPS_NOT_DELIVERED.CORE				(41.71%)
56,543,382	INT_MISC.RECOVERY_CYCLES_ANY				(33.33%)
73,114,649	CPU_CLK_UNHALTED.ONE_THREAD_ACTIVE				(33.29%)
9,450,320,747	CPU_CLK_UNHALTED.THREAD				(41.61%)
2,460,247,529	UOPS_ISSUED.ANY				(41.57%)

²Based on Yasin: A Top Down Method ... doi:10.1109/ISPASS.2014.6844459

```
3.079335575 seconds time elapsed
```

```
# Level 1 analysis revealed significant tma_backend_bound percentage,  
# continue group based top down analysis mode on backend bound level 2.  
perf stat -M tma_backend_bound_group ./basic
```

```
# Level 2 analysis revealed significant tma_core_bound and tma_memory_bound percentages,  
# continue group based top down analysis mode on core and memory bound levels 3.  
perf stat -M tma_core_bound_group ./basic  
perf stat -M tma_memory_bound_group ./basic
```

```
# Level 3 analysis revealed significant tma_store_bound percentage,  
# continue group based top down analysis mode on store bound level 4.  
perf stat -M tma_store_bound_group ./basic
```

```
likwid-perfctr -g TMA -C 0 ./basic
```

```
...  
+-----+-----+  
|      Metric      | Core 0 |  
+-----+-----+  
| Runtime (RDTSC) [s] | 3.0060 |  
| Runtime unhaltd [s] | 3.3564 |  
|      Clock [MHz]   | 3090.3635 |  
|      CPI           | 3.6198 |  
|      IPC           | 0.2763 |  
|      Total Slots   | 36381924196 |  
|      Slots Retired | 1980635404 |  
|      Fetch Bubbles | 2301988821 |  
|      Recovery Bubbles | 200105344 |  
|      Front End [%]  | 6.3273 |  
|      Speculation [%] | 1.0787 |  
|      Retiring [%]   | 5.4440 |  
|      Back End [%]   | 87.1500 |  
+-----+-----+
```

One example application of the top down analysis can be found in the comparative performance visualization tool available at <https://lpentecost.github.io/uarchperfviz>, described in detail in the paper by Pentecost et al.: CHAMPVis: Comparative Hierarchical Analysis ... doi:10.1109/ProTools49597.2019.00013. The live tool demo helps investigate the performance of benchmarks from the PARSEC suite using the top down analysis on an (unfortunately unspecified) CPU.