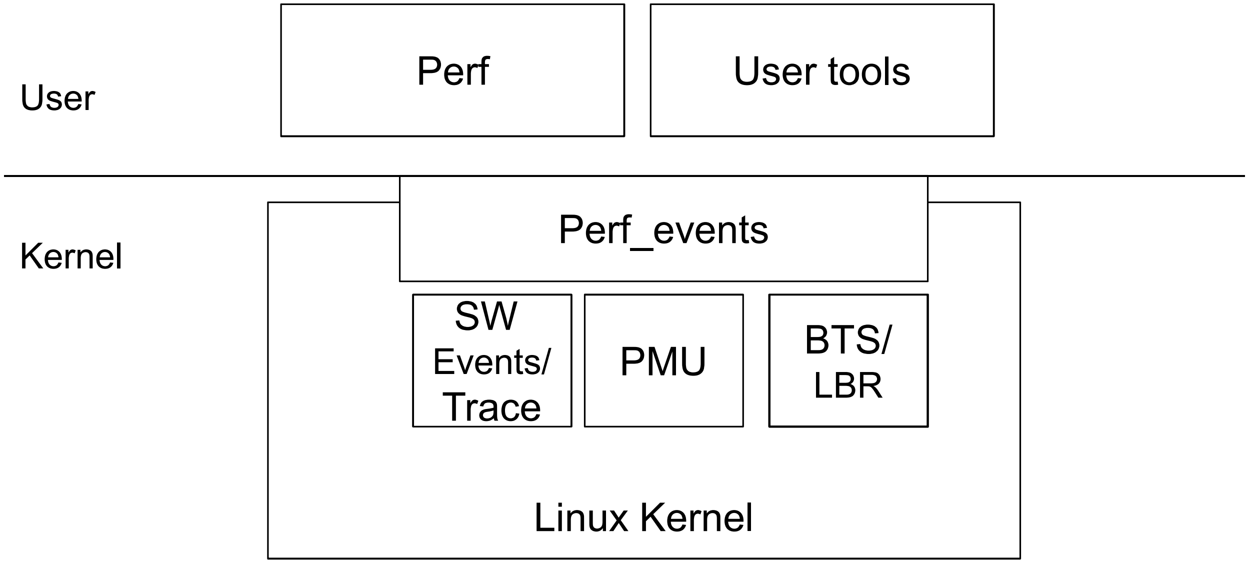
Application performance analysis

Using Perf with PMU event, PEBS, LBR and Intel PT technologies

Jin Yao (yao.jin@intel.com)

Linux “perf” overview



PMU, tracepoint, tracing framework

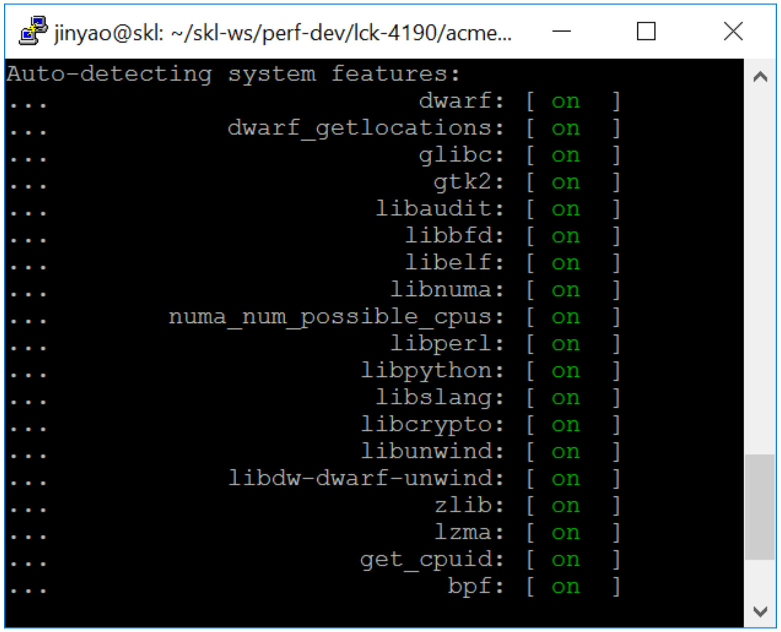
- Integrated into the Linux kernel
  - Including user tools
- Maintained by Linux community
  - With Intel contributions
- Generic: x86, other architectures
- Aims to abstract the hardware
- Supports software events
- Aims to be easy to use

Deployment

- Part of the core Linux kernel
- Fast development
- Not a separate driver
- Kernel version dependent, tightly integrated (some backports)
- Provides user interface (syscall + ring buffer)

Perf build notice

- Rebuild perf binary if use a new kernel
  - cd tools/perf; make
- Make sure the lib installed correctly

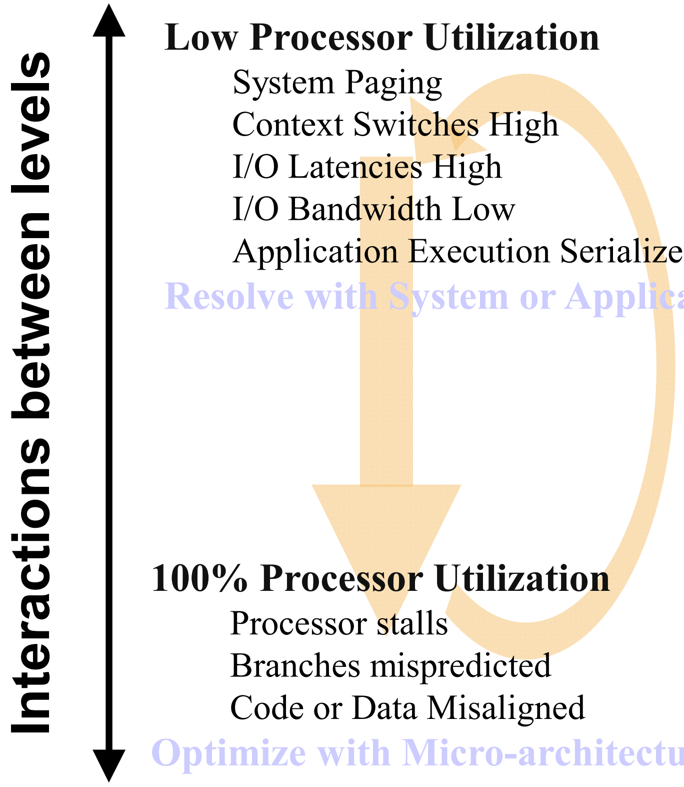


```
Auto-detecting system features:
...                         dwarf: [ on  ]
...            dwarf_getlocations: [ on  ]
...                         glibc: [ on  ]
...                          gtk2: [ on  ]
...                      libaudit: [ on  ]
...                        libbfd: [ on  ]
...                        libelf: [ on  ]
...                       libnuma: [ on  ]
...        numa_num_possible_cpus: [ on  ]
...                       libperl: [ on  ]
...                     libpython: [ on  ]
...                      libslang: [ on  ]
...                     libcrypto: [ on  ]
...                     libunwind: [ on  ]
...            libdw-dwarf-unwind: [ on  ]
...                          zlib: [ on  ]
...                          lzma: [ on  ]
...                     get_cpuid: [ on  ]
...                           bpf: [ on  ]
```

perf events I (perf list)

branch-instructions OR branches	[Hardware event]
branch-misses	[Hardware event]
bus-cycles	[Hardware event]
cache-misses	[Hardware event]
cache-references	[Hardware event]
cpu-cycles OR cycles	[Hardware event]
instructions	[Hardware event]
ref-cycles	[Hardware event]

Tuning Level Interactions



Low Processor Utilization

System Paging
Context Switches High
I/O Latencies High
I/O Bandwidth Low
Application Execution Serialized

Resolve with System or Application Tuning

100% Processor Utilization

Processor stalls
Branches mispredicted
Code or Data Misaligned

Optimize with Micro-architecture Level Tuning

Data Collection Techniques

- Sampling
  - Collection of data based on the occurrence of a particular event such as a timer or interrupt
  - Example: Perf (perf record)

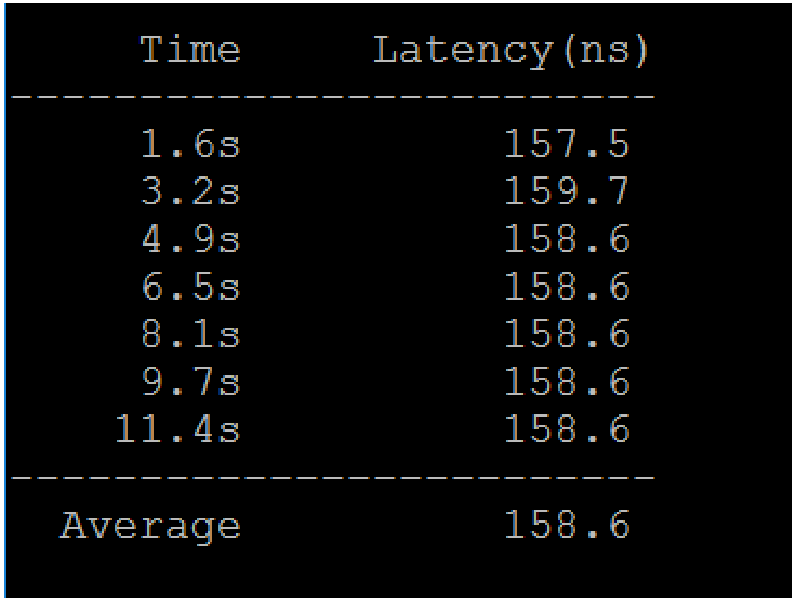
- Tracing
  - Getting log of path of application
  - Example: Perf (Intel PT)

- Instrumentation
  - Insertion of data collection instructions in the source code or object code level

- Simulation

A “mgen” workload example

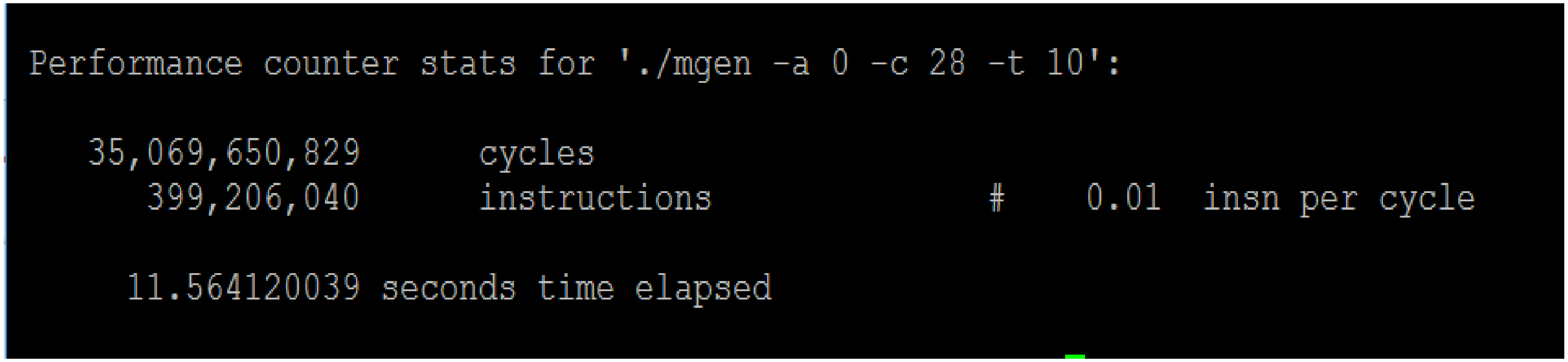
- Generate Remote Memory Access for ~10s on SKX
  - mgen -a 0 -c 28 -t 10 (memory allocated on node0, thread runs on cpu28)



```
     Time       Latency(ns)
-----------------------------
     1.6s            157.5
     3.2s            159.7
     4.9s            158.6
     6.5s            158.6
     8.1s            158.6
     9.7s            158.6
    11.4s            158.6
-----------------------------
  Average            158.6
```

Overview (by perf stat)

- perf stat -e cycles,instructions ./mgen -a 0 -c 28 -t 10

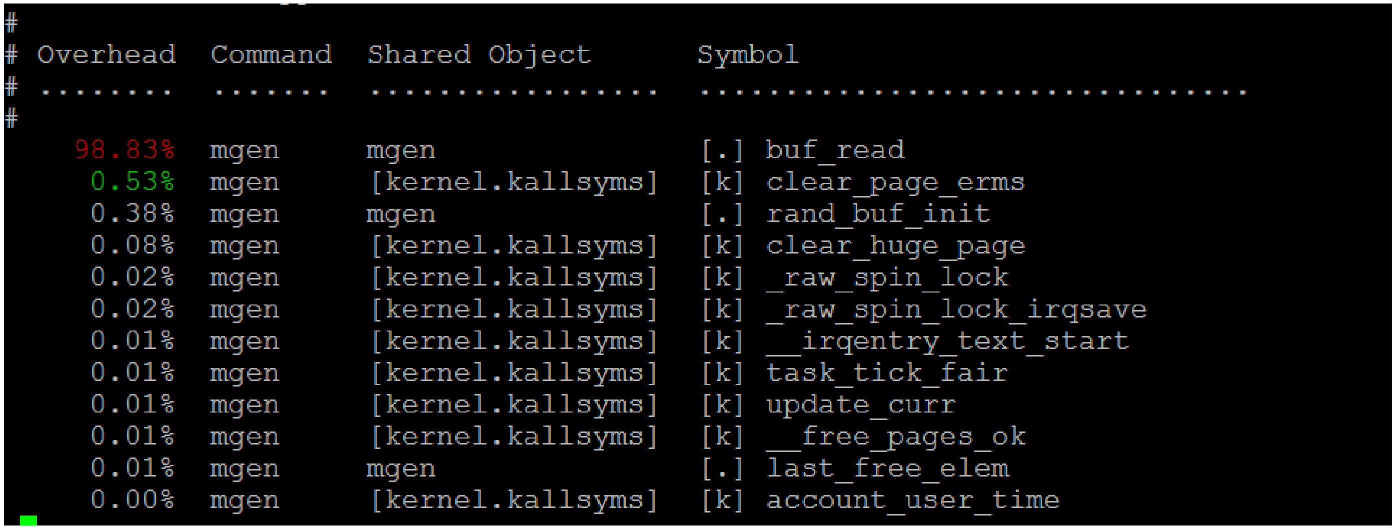


```
Performance counter stats for './mgen -a 0 -c 28 -t 10':

   35,069,650,829      cycles
      399,206,040      instructions              #    0.01  insn per cycle

     11.564120039 seconds time elapsed
```

- IPC = Instruction Per Cycle (0.01, very bad data)
- perf stat is not sampling

Who eats cycles? (by perf record/report)

- perf record -e cycles ./mgen -a 0 -c 28 -t 10
- perf report --stdio (buf_read eats 98.83% cycles)

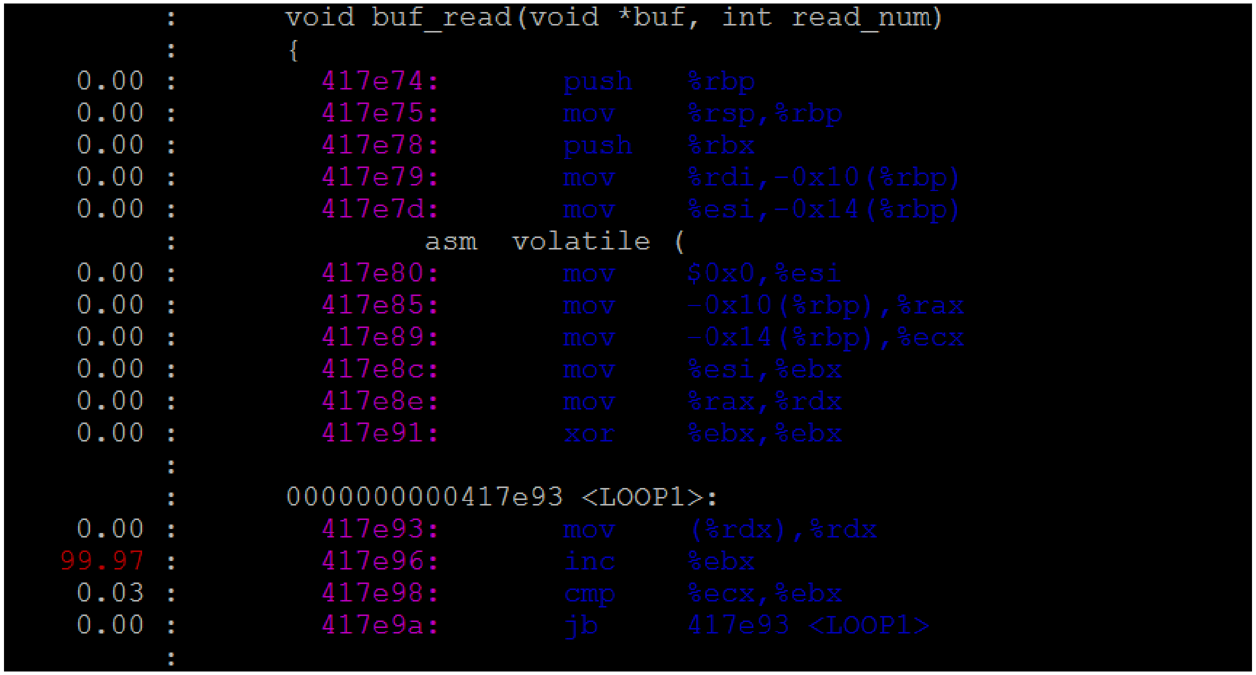


```
#
# Overhead  Command  Shared Object      Symbol
# ........  .......  .................  ................................
#
    98.83%  mgen     mgen               [.] buf_read
     0.53%  mgen     [kernel.kallsyms]  [k] clear_page_erms
     0.38%  mgen     mgen               [.] rand_buf_init
     0.08%  mgen     [kernel.kallsyms]  [k] clear_huge_page
     0.02%  mgen     [kernel.kallsyms]  [k] _raw_spin_lock
     0.02%  mgen     [kernel.kallsyms]  [k] _raw_spin_lock_irqsave
     0.01%  mgen     [kernel.kallsyms]  [k] __irqentry_text_start
     0.01%  mgen     [kernel.kallsyms]  [k] task_tick_fair
     0.01%  mgen     [kernel.kallsyms]  [k] update_curr
     0.01%  mgen     [kernel.kallsyms]  [k] __free_pages_ok
     0.01%  mgen     mgen               [.] last_free_elem
     0.00%  mgen     [kernel.kallsyms]  [k] account_user_time
```

- perf record is sampling.

Which instruction eats cycles? (by perf annotate)

- perf annotate --stdio



```
         :      void buf_read(void *buf, int read_num)
         :      {
    0.00 :        417e74:       push   %rbp
    0.00 :        417e75:       mov    %rsp,%rbp
    0.00 :        417e78:       push   %rbx
    0.00 :        417e79:       mov    %rdi,-0x10(%rbp)
    0.00 :        417e7d:       mov    %esi,-0x14(%rbp)
         :              asm  volatile (
    0.00 :        417e80:       mov    $0x0,%esi
    0.00 :        417e85:       mov    -0x10(%rbp),%rax
    0.00 :        417e89:       mov    -0x14(%rbp),%ecx
    0.00 :        417e8c:       mov    %esi,%ebx
    0.00 :        417e8e:       mov    %rax,%rdx
    0.00 :        417e91:       xor    %ebx,%ebx
         :
         :      0000000000417e93 <LOOP1>:
    0.00 :        417e93:       mov    (%rdx),%rdx
   99.97 :        417e96:       inc    %ebx
    0.03 :        417e98:       cmp    %ecx,%ebx
    0.00 :        417e9a:       jb     417e93 <LOOP1>
         :
```

- Is “inc %ebx” take 99.97% cycles in buf_read? No!

PEBS (Precise Event)

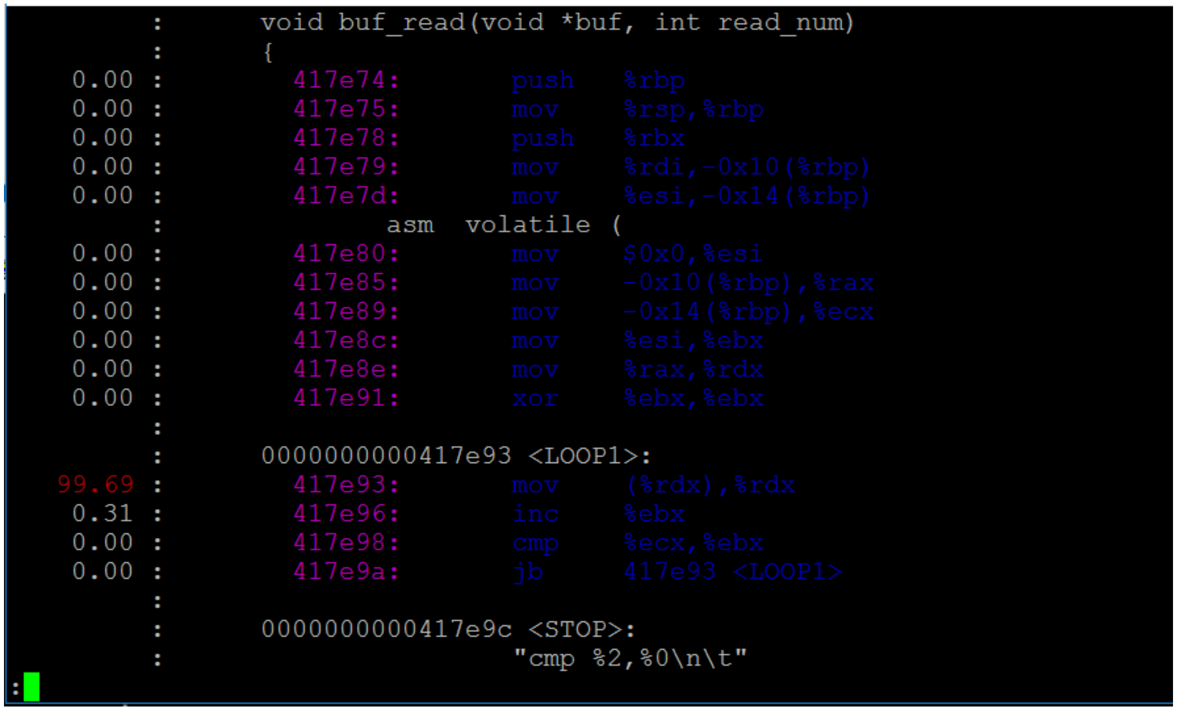
- no p - arbitrary skid
- :p - constant skid
- :pp - requested to have 0 skid (Intel PEBS events)
- :ppp - must have 0 skid (only special case)

- Run perf record with precise option again
- perf record -e cycles:pp ./mgen -a 0 -c 28 -t 10

- If only perf record <app>, default is -e cycles:ppp

PEBS (Precise Event)

- perf annotate --stdio



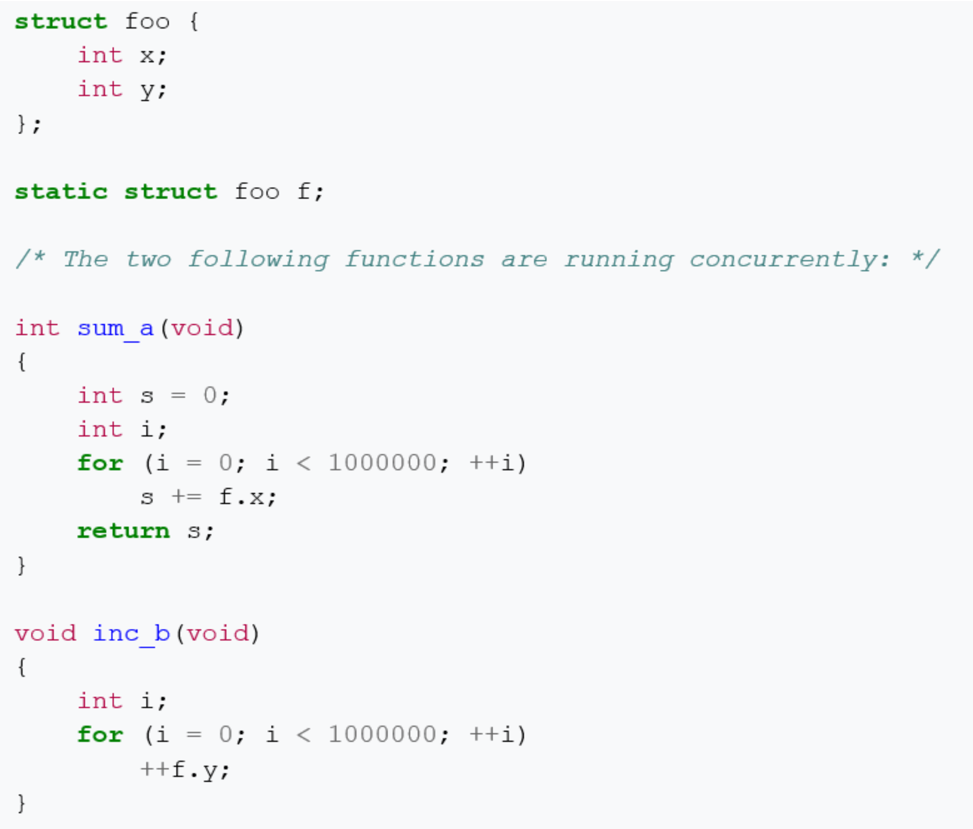
```
         :      void buf_read(void *buf, int read_num)
         :      {
    0.00 :        417e74:       push   %rbp
    0.00 :        417e75:       mov    %rsp,%rbp
    0.00 :        417e78:       push   %rbx
    0.00 :        417e79:       mov    %rdi,-0x10(%rbp)
    0.00 :        417e7d:       mov    %esi,-0x14(%rbp)
         :              asm  volatile (
    0.00 :        417e80:       mov    $0x0,%esi
    0.00 :        417e85:       mov    -0x10(%rbp),%rax
    0.00 :        417e89:       mov    -0x14(%rbp),%ecx
    0.00 :        417e8c:       mov    %esi,%ebx
    0.00 :        417e8e:       mov    %rax,%rdx
    0.00 :        417e91:       xor    %ebx,%ebx
         :
         :      0000000000417e93 <LOOP1>:
   99.69 :        417e93:       mov    (%rdx),%rdx
    0.31 :        417e96:       inc    %ebx
    0.00 :        417e98:       cmp    %ecx,%ebx
    0.00 :        417e9a:       jb     417e93 <LOOP1>
         :
         :      0000000000417e9c <STOP>:
         :                      "cmp %2,%0\n\t"
:
```

- Why instruction at 417e93 takes 99.69% cycles in buf_read?

Memory load of 417e93 (by perf c2c)

- `99.69  :       417193:          mov          (%rdx), %rdx`
- Why memory load so slow? Not hit in LLC? Not hit in local memory? Cache-line false-sharing issue?
- `perf c2c record ./mgen -a 0 -c 28 -t 10`
- `perf c2c report --stdio`
- c2c: cache to cache – Detect False-Sharing cache-lines.
- Based on Intel load latency facility.
  - Memory access of the access
  - Type of the access (e.g. remote memory hit?)
  - Latency (in cycles) of the load access

What’s False-Sharing?

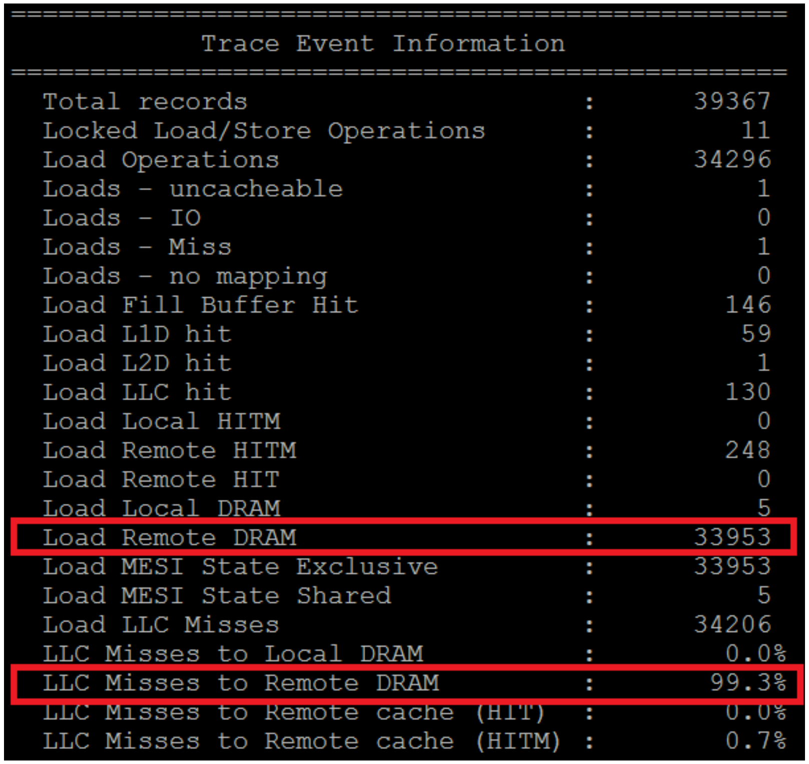


```c
struct foo {
    int x;
    int y;
};

static struct foo f;

/* The two following functions are running concurrently: */

int sum_a(void)
{
    int s = 0;
    int i;
    for (i = 0; i < 1000000; ++i)
        s += f.x;
    return s;
}

void inc_b(void)
{
    int i;
    for (i = 0; i < 1000000; ++i)
        ++f.y;
}
```

sum_a re-read x from memory even though modification of y is irrelevant.

What data address hit by 417e93 (1)

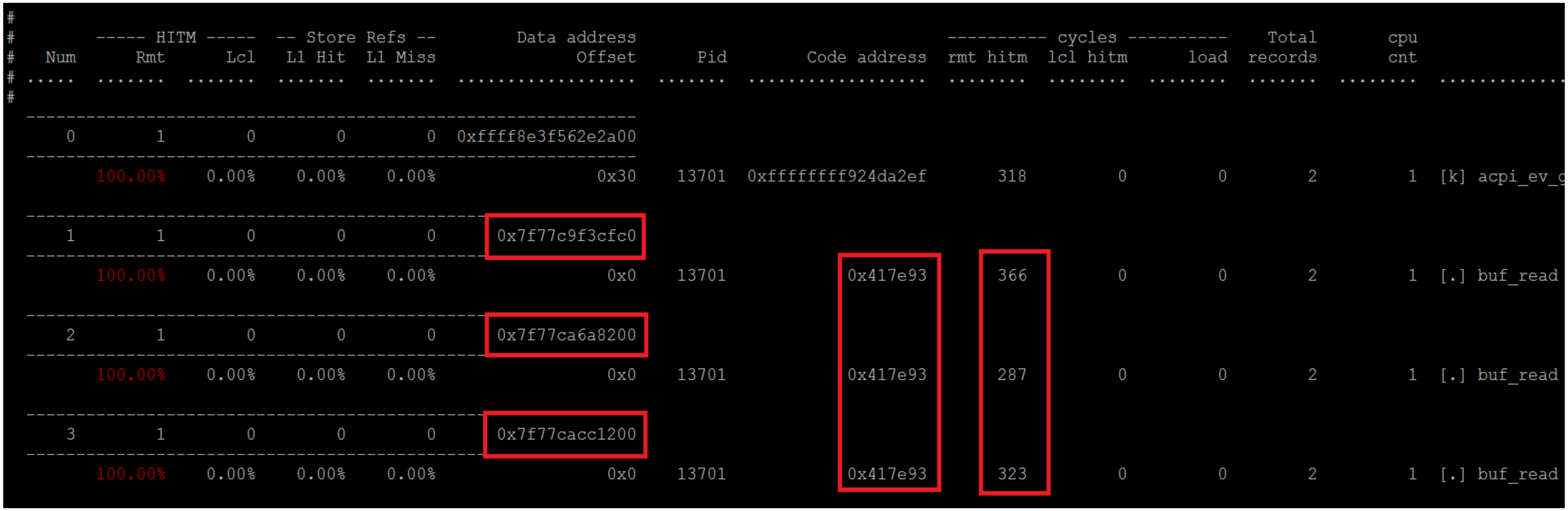
- c2c can do more than False-Sharing analysis
- perf c2c report –stdio (part of output)



```
=================================================
            Trace Event Information
=================================================
  Total records                     :      39367
  Locked Load/Store Operations      :         11
  Load Operations                   :      34296
  Loads - uncacheable               :          1
  Loads - IO                        :          0
  Loads - Miss                      :          1
  Loads - no mapping                :          0
  Load Fill Buffer Hit              :        146
  Load L1D hit                      :         59
  Load L2D hit                      :          1
  Load LLC hit                      :        130
  Load Local HITM                   :          0
  Load Remote HITM                  :        248
  Load Remote HIT                   :          0
  Load Local DRAM                   :          5
  Load Remote DRAM                  :      33953
  Load MESI State Exclusive         :      33953
  Load MESI State Shared            :          5
  Load LLC Misses                   :      34206
  LLC Misses to Local DRAM          :       0.0%
  LLC Misses to Remote DRAM         :      99.3%
  LLC Misses to Remote cache (HIT)  :       0.0%
  LLC Misses to Remote cache (HITM) :       0.7%
```

What data address hit by 417e93 (2)

- perf c2c report –stdio (part of output, actually many 417e93 entries)

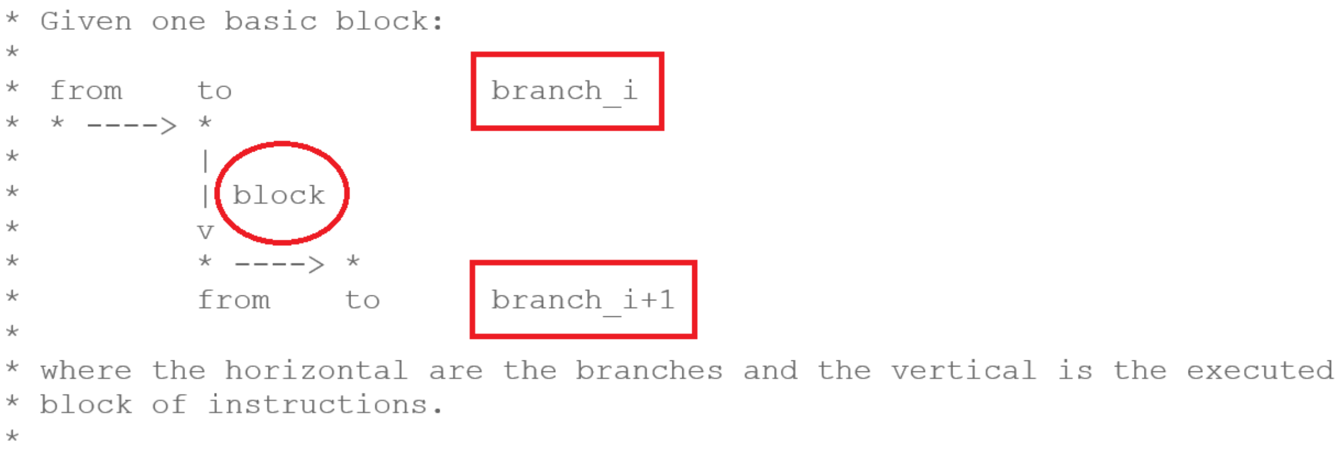


```
#
#        ----- HITM -----  -- Store Refs --        Data address                               ---------- cycles ----------    Total       cpu
#   Num      Rmt      Lcl   L1 Hit  L1 Miss              Offset      Pid        Code address  rmt hitm  lcl hitm      load  records       cnt
# .....  .......  .......  .......  .......  ..................  .......  ..................  ........  ........  ........  .......  ........  .............
#
  -------------------------------------------------------------
      0        1        0        0        0  0xffff8e3f562e2a00
  -------------------------------------------------------------
         100.00%    0.00%    0.00%    0.00%                0x30    13701  0xffffffff924da2ef       318         0         0        2         1  [k] acpi_ev_g

  -----------------------------------------------
      1        1        0        0        0      0x7f77c9f3cfc0
  -----------------------------------------------
         100.00%    0.00%    0.00%    0.00%                 0x0    13701            0x417e93       366         0         0        2         1  [.] buf_read

  -----------------------------------------------
      2        1        0        0        0      0x7f77ca6a8200
  -----------------------------------------------
         100.00%    0.00%    0.00%    0.00%                 0x0    13701            0x417e93       287         0         0        2         1  [.] buf_read

  -----------------------------------------------
      3        1        0        0        0      0x7f77cacc1200
  -----------------------------------------------
         100.00%    0.00%    0.00%    0.00%                 0x0    13701            0x417e93       323         0         0        2         1  [.] buf_read
```

- 417e93 generates a lot of remote memory access and almost no local or remote LLC hit (not false-sharing issue).

Timed LBR (Last Branch Records)

- Sampling + Tracing (h/w saves latest N branches to buffer)



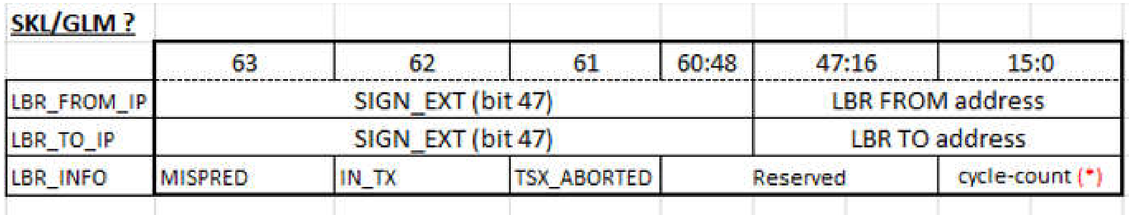
```
* Given one basic block:
*
*  from    to              branch_i
*  * ----> *
*          |
*          | block
*          v
*          * ----> *
*          from    to      branch_i+1
*
* where the horizontal are the branches and the vertical is the executed
* block of instructions.
*
```

- Tell us the cycles of code block between 2 branches.

LBR sampling

Log LBRs at sample point
Support 32 entries on SKL

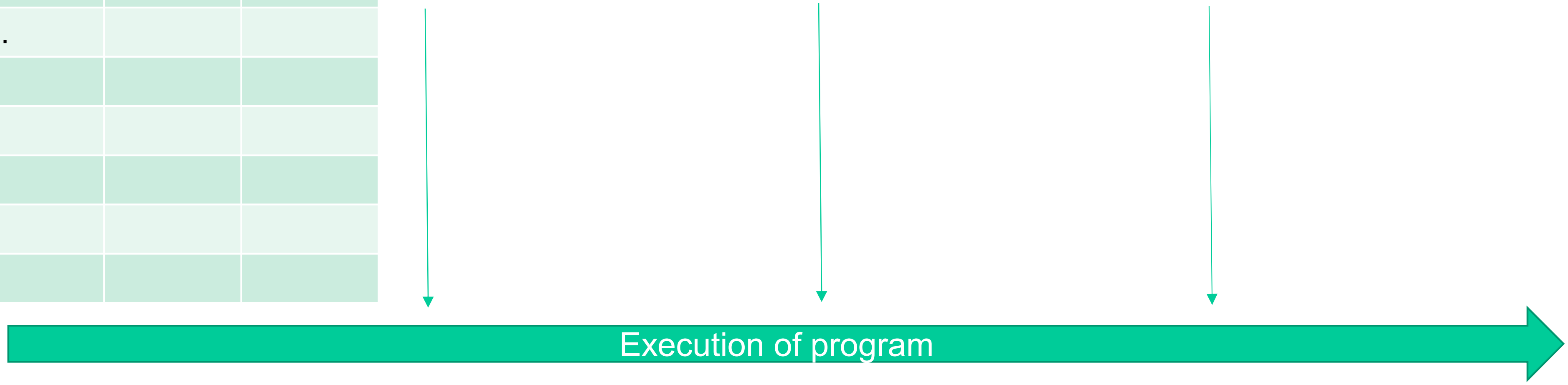
FROM	TO	CYCLES
123	456	5
…		



SKL/GLM ?

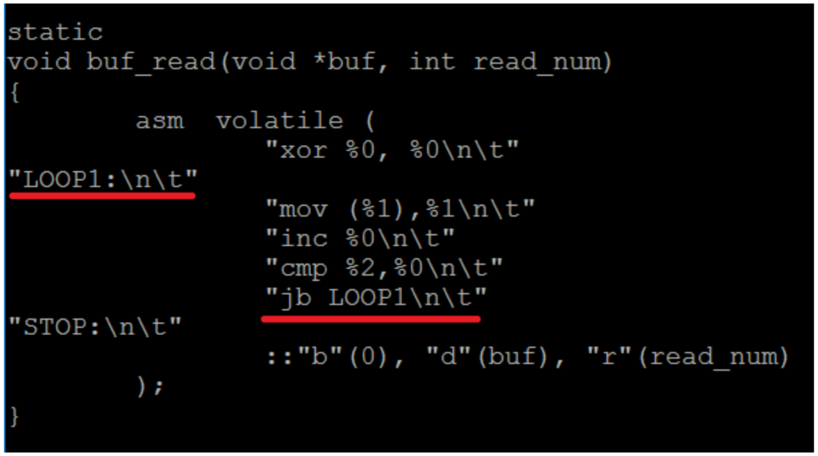
	63	62	61	60:48	47:16	15:0
LBR_FROM_IP	SIGN_EXT (bit 47)				LBR FROM address	
LBR_TO_IP	SIGN_EXT (bit 47)				LBR TO address	
LBR_INFO	MISPRED	IN_TX	TSX_ABORTED	Reserved		cycle-count (*)

Sample using
Performance Counter

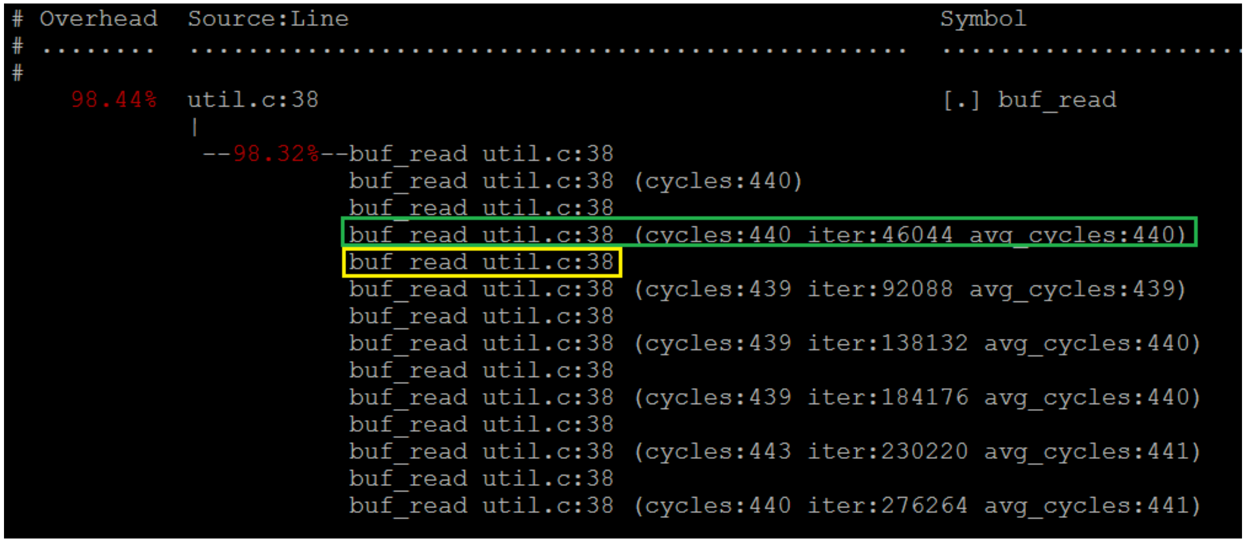


Cycles of hot code block

- perf record -b -e cycles:pp ./mgen -a 0 -c 28 -t 10
- perf report --branch-history --stdio



```
static
void buf_read(void *buf, int read_num)
{
        asm  volatile (
                "xor %0, %0\n\t"
"LOOP1:\n\t"
                "mov (%1),%1\n\t"
                "inc %0\n\t"
                "cmp %2,%0\n\t"
                "jb LOOP1\n\t"
"STOP:\n\t"
                ::"b"(0), "d"(buf), "r"(read_num)
        );
}
```



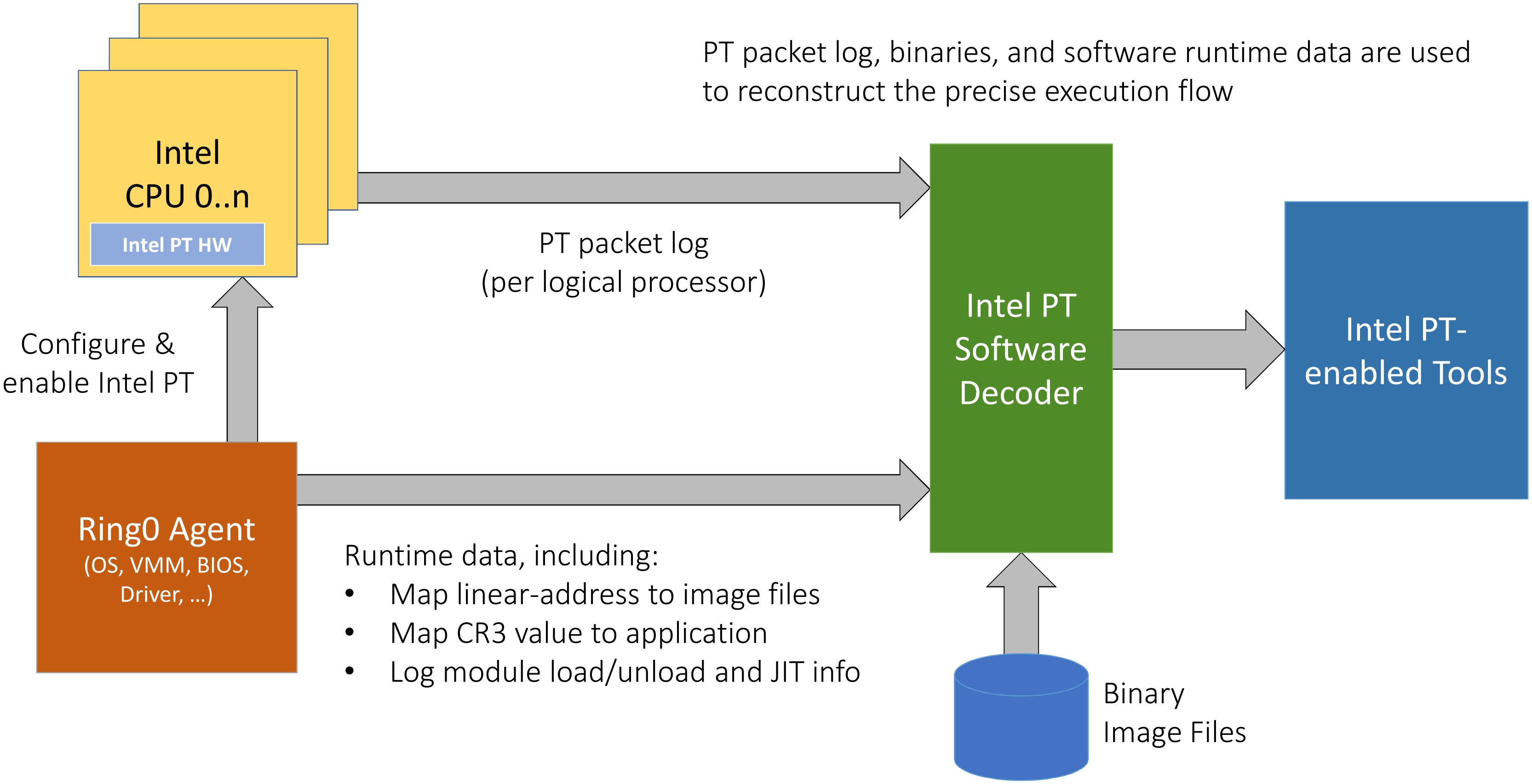
```
# Overhead  Source:Line                                                Symbol
# ........  ...............................................            ....................
#
    98.44%  util.c:38                                                  [.] buf_read
            |
             --98.32%--buf_read util.c:38
                       buf_read util.c:38 (cycles:440)
                       buf_read util.c:38
                       buf_read util.c:38 (cycles:440 iter:46044 avg_cycles:440)
                       buf_read util.c:38
                       buf_read util.c:38 (cycles:439 iter:92088 avg_cycles:439)
                       buf_read util.c:38
                       buf_read util.c:38 (cycles:439 iter:138132 avg_cycles:440)
                       buf_read util.c:38
                       buf_read util.c:38 (cycles:439 iter:184176 avg_cycles:440)
                       buf_read util.c:38
                       buf_read util.c:38 (cycles:443 iter:230220 avg_cycles:441)
                       buf_read util.c:38
                       buf_read util.c:38 (cycles:440 iter:276264 avg_cycles:441)
```

- Yellow is TO of branch X (LOOP1), green is FROM of branch X+1 (jb)
- 440 cycles is for code block from LOOP1 to jb

What is Processor Trace (PT)?

- Intel PT is a hardware feature that logs information about software execution
- Available in Skylake, Goldmont, … Broadwell also, but has many limitations and is slower
- Supports control flow tracing. Decoder can determine exact flow of software execution from trace log
    - ➢ Target <5% performance overhead. Depends on processor generation and usage model
- Can store both cycle count and timestamp information

Intel® Processor Trace Components



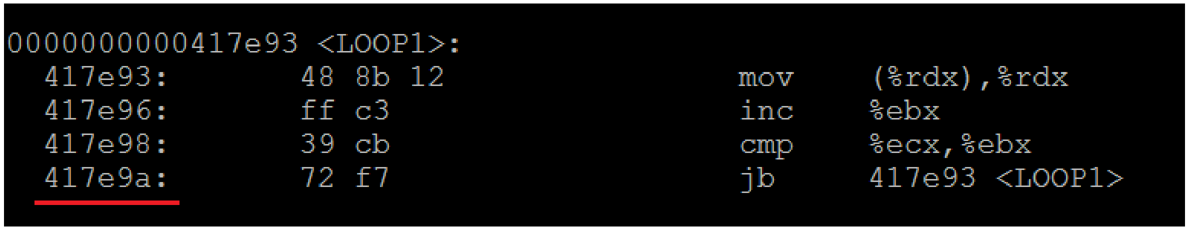
Intel
CPU 0..n
Intel PT HW
PT packet log, binaries, and software runtime data are used to reconstruct the precise execution flow
PT packet log
(per logical processor)
Configure &
enable Intel PT
Intel PT Software Decoder
Intel PT-enabled Tools
Ring0 Agent
(OS, VMM, BIOS, Driver, …)
Binary
Image Files

Runtime data, including:

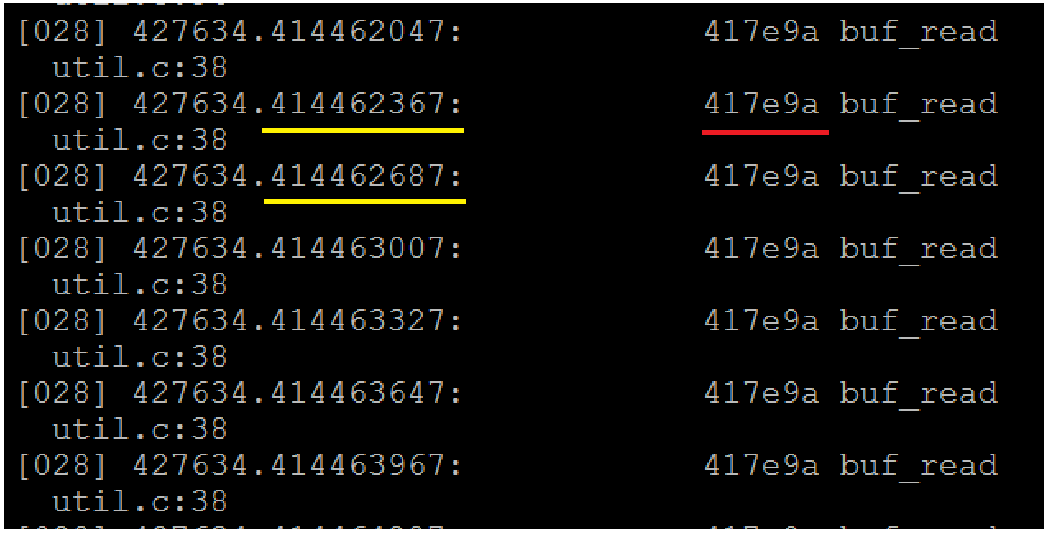
- Map linear-address to image files
- Map CR3 value to application
- Log module load/unload and JIT info

Branch timestamp (by perf PT)

- perf record -e intel_pt//u ./mgen -a 0 -c 28 -t 10
- perf script --ns -F time,cpu,sym,ip,srcline



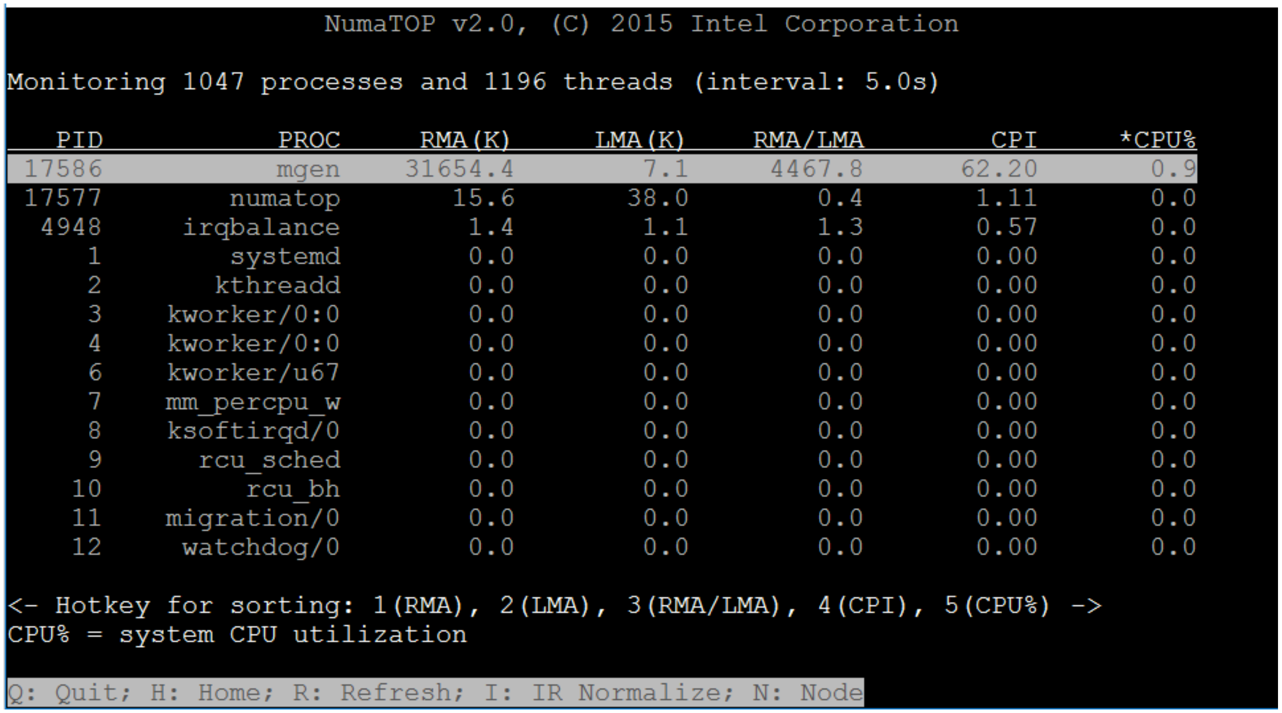
```
0000000000417e93 <LOOP1>:
  417e93:       48 8b 12                mov    (%rdx),%rdx
  417e96:       ff c3                   inc    %ebx
  417e98:       39 cb                   cmp    %ecx,%ebx
  417e9a:       72 f7                   jb     417e93 <LOOP1>
```



```
[028] 427634.414462047:             417e9a buf_read
  util.c:38
[028] 427634.414462367:             417e9a buf_read
  util.c:38
[028] 427634.414462687:             417e9a buf_read
  util.c:38
[028] 427634.414463007:             417e9a buf_read
  util.c:38
[028] 427634.414463327:             417e9a buf_read
  util.c:38
[028] 427634.414463647:             417e9a buf_read
  util.c:38
[028] 427634.414463967:             417e9a buf_read
  util.c:38
```

Other Tools - NumaTOP

- NumaTOP (runtime memory locality characterization on NUMA system)



```
                   NumaTOP v2.0, (C) 2015 Intel Corporation

Monitoring 1047 processes and 1196 threads (interval: 5.0s)

   PID            PROC     RMA(K)     LMA(K)    RMA/LMA         CPI     *CPU%
 17586            mgen    31654.4        7.1     4467.8       62.20       0.9
 17577         numatop       15.6       38.0        0.4        1.11       0.0
  4948      irqbalance        1.4        1.1        1.3        0.57       0.0
     1         systemd        0.0        0.0        0.0        0.00       0.0
     2        kthreadd        0.0        0.0        0.0        0.00       0.0
     3     kworker/0:0        0.0        0.0        0.0        0.00       0.0
     4     kworker/0:0        0.0        0.0        0.0        0.00       0.0
     6     kworker/u67        0.0        0.0        0.0        0.00       0.0
     7     mm_percpu_w        0.0        0.0        0.0        0.00       0.0
     8     ksoftirqd/0        0.0        0.0        0.0        0.00       0.0
     9       rcu_sched        0.0        0.0        0.0        0.00       0.0
    10          rcu_bh        0.0        0.0        0.0        0.00       0.0
    11     migration/0        0.0        0.0        0.0        0.00       0.0
    12      watchdog/0        0.0        0.0        0.0        0.00       0.0

<- Hotkey for sorting: 1(RMA), 2(LMA), 3(RMA/LMA), 4(CPI), 5(CPU%) ->
CPU% = system CPU utilization

Q: Quit; H: Home; R: Refresh; I: IR Normalize; N: Node
```

- http://01.org/numatop
- https://github.com/01org/numatop.git

Other Tools – LKP-tests

- LKP-tests (Linux kernel performance test tool)
- Open source tool by Intel:

    https://github.com/01org/lkp-tests.git

- Framework to run benchmarks

    Integrated ~80 benchmarks/test suites

    Flexible mechanism to configure various parameters

    Integrated ~40 monitors to monitor resource usages and statistics

- Framework for performance analysis
- Can be set up in CI environment (e.g. 0-Day CI), used for running benchmark and reproducing regression

References

Perf C2C:

https://joemario.github.io/blog/2016/09/01/c2c-blog/

LBR doc:

http://lwn.net/Articles/680985/

http://lwn.net/Articles/680996/

Perf PT doc:

https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/tools/perf/Documentation/intel-pt.txt

Adding processor trace to Linux

https://lwn.net/Articles/648154/