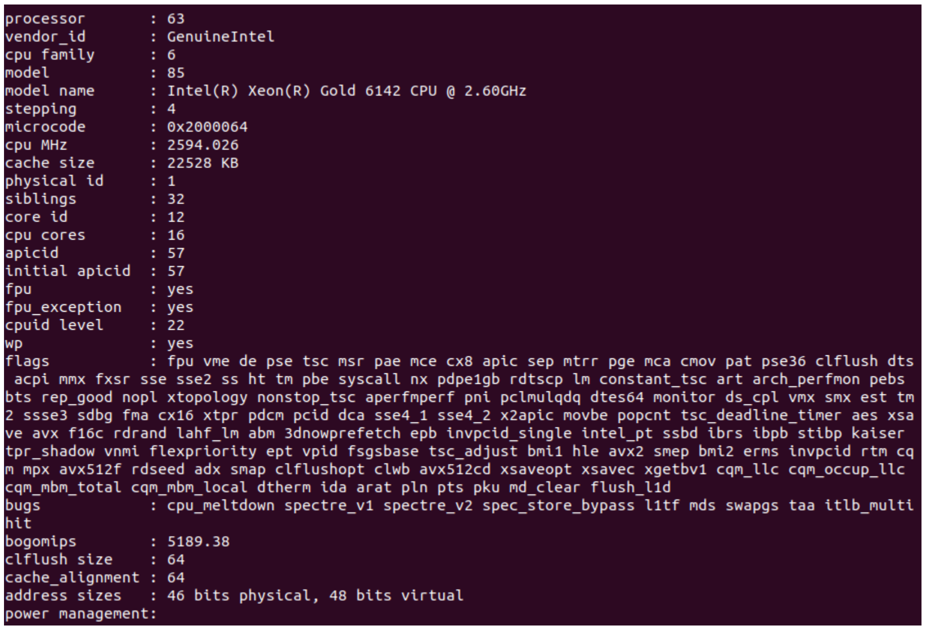
Optional project for CS250A

Semen Pyankov

March 2020

1 System

In this project I've decided to choose the source code from the analysis of transactional memory (Github). This code runs on 10 threads and consists of a loop that run transaction with several memory modifications. Loop size is 1000000 iterations. During the analysis was used Intel Xeon Gold 6142 processor with 32 threads (2 per each of 16 cores). Here is the data from /proc/cpuinfo:

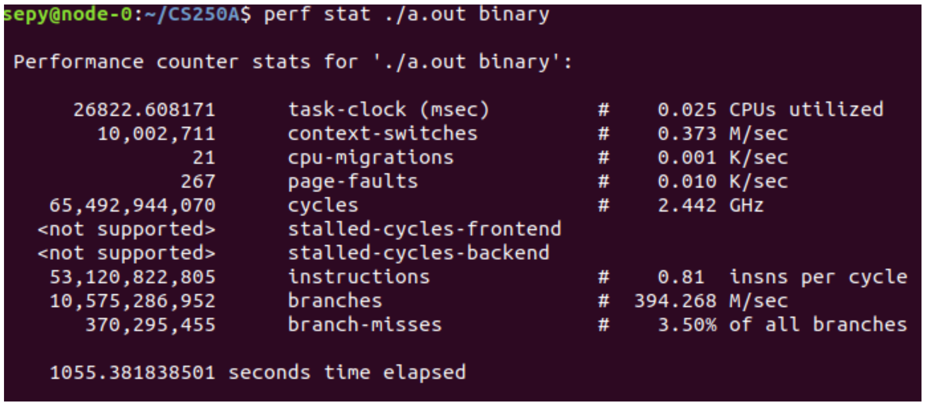


```
processor       : 63
vendor_id       : GenuineIntel
cpu family      : 6
model           : 85
model name      : Intel(R) Xeon(R) Gold 6142 CPU @ 2.60GHz
stepping        : 4
microcode       : 0x2000064
cpu MHz         : 2594.026
cache size      : 22528 KB
physical id     : 1
siblings        : 32
core id         : 12
cpu cores       : 16
apicid          : 57
initial apicid  : 57
fpu             : yes
fpu_exception   : yes
cpuid level     : 22
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts
 acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs
bts rep_good nopl xtopology nonstop_tsc aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm
2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsa
ve avx f16c rdrand lahf_lm abm 3dnowprefetch epb invpcid_single intel_pt ssbd ibrs ibpb stibp kaiser
tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm cq
m mpx avx512f rdseed adx smap clflushopt clwb avx512cd xsaveopt xsavec xgetbv1 cqm_llc cqm_occup_llc
cqm_mbm_total cqm_mbm_local dtherm ida arat pln pts pku md_clear flush_l1d
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds swapgs taa itlb_multi
hit
bogomips        : 5189.38
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:
```

Figure 1: Processor information (one core out of 16)

2 IPC

To analyse the code, was used perf utility (version @@@). With command ”perf stat ./a.out binary” IPC, number of instructions, branch misses, and some other data can be measured:



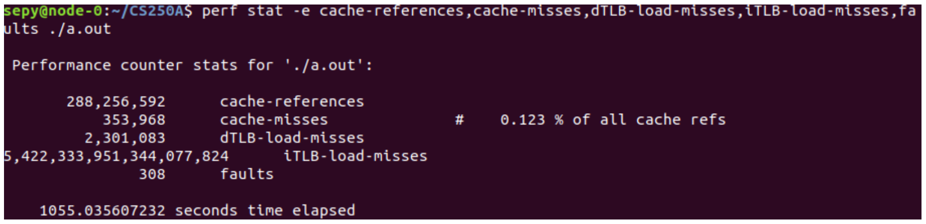
```
sepy@node-0:~/CS250A$ perf stat ./a.out binary

 Performance counter stats for './a.out binary':

     26822.608171      task-clock (msec)         #    0.025 CPUs utilized
       10,002,711      context-switches          #    0.373 M/sec
               21      cpu-migrations            #    0.001 K/sec
              267      page-faults               #    0.010 K/sec
   65,492,944,070      cycles                    #    2.442 GHz
  <not supported>      stalled-cycles-frontend
  <not supported>      stalled-cycles-backend
   53,120,822,805      instructions              #    0.81  insns per cycle
   10,575,286,952      branches                  #  394.268 M/sec
      370,295,455      branch-misses             #    3.50% of all branches

   1055.381838501 seconds time elapsed
```

Figure 2: perf data (IPC)

According to received data on 50B instructions, IPC is 0.81; branch misses rate is 3.5% (370M out of 10575M branches). About half of branches are from the loop conditional jumps – these branches can be easily predicted. Other branches appears when we check the beginning of a transaction. The abort ratio in this program is relatively low, therefore branch can be predicted as ”taken” with a low miss rate.

3 Cache misses, TLB misses, page faults

With command ”perf stat -e cache-references, cache-misses, dTLB-load-misses, iTLB-load-misses, faults ./a.out” we can measure cache miss rate, TLB misses and page faults:



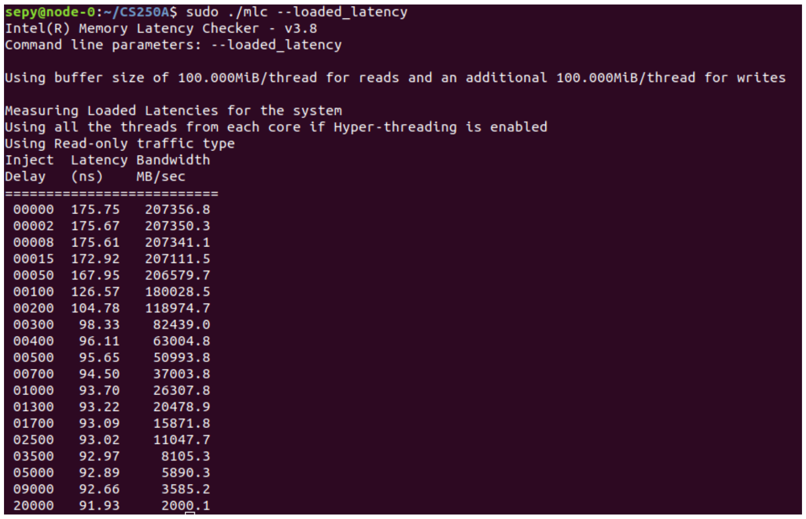
```
sepy@node-0:~/CS250A$ perf stat -e cache-references,cache-misses,dTLB-load-misses,iTLB-load-misses,fa
ults ./a.out

 Performance counter stats for './a.out':

       288,256,592      cache-references
           353,968      cache-misses              #    0.123 % of all cache refs
         2,301,083      dTLB-load-misses
5,422,333,951,344,077,824      iTLB-load-misses
               308      faults

    1055.035607232 seconds time elapsed
```

Figure 3: perf data (cache, TLB, page faults)

Cache misses rate is relatively low (0.123%) due to the spatial locality (we write to tmp[i], tmp[i+1], tmp[i+2]).

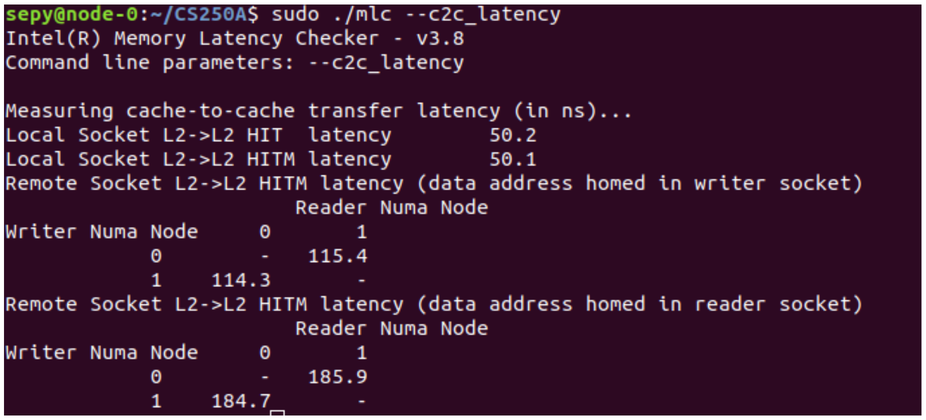
4 Average memory latency

Using Intel MLC (Memory Latency Checker, v3.8) we can measure the average memory and cache latency:



```
sepy@node-0:~/CS250A$ sudo ./mlc --loaded_latency
Intel(R) Memory Latency Checker - v3.8
Command line parameters: --loaded_latency

Using buffer size of 100.000MiB/thread for reads and an additional 100.000MiB/thread for writes

Measuring Loaded Latencies for the system
Using all the threads from each core if Hyper-threading is enabled
Using Read-only traffic type
Inject  Latency Bandwidth
Delay   (ns)    MB/sec
==========================
 00000  175.75   207356.8
 00002  175.67   207350.3
 00008  175.61   207341.1
 00015  172.92   207111.5
 00050  167.95   206579.7
 00100  126.57   180028.5
 00200  104.78   118974.7
 00300   98.33    82439.0
 00400   96.11    63004.8
 00500   95.65    50993.8
 00700   94.50    37003.8
 01000   93.70    26307.8
 01300   93.22    20478.9
 01700   93.09    15871.8
 02500   93.02    11047.7
 03500   92.97     8105.3
 05000   92.89     5890.3
 09000   92.66     3585.2
 20000   91.93     2000.1
```

Figure 4: MLC analysis (memory latency)



```
sepy@node-0:~/CS250A$ sudo ./mlc --c2c_latency
Intel(R) Memory Latency Checker - v3.8
Command line parameters: --c2c_latency

Measuring cache-to-cache transfer latency (in ns)...
Local Socket L2->L2 HIT  latency        50.2
Local Socket L2->L2 HITM latency        50.1
Remote Socket L2->L2 HITM latency (data address homed in writer socket)
                        Reader Numa Node
Writer Numa Node     0       1
            0        -   115.4
            1    114.3       -
Remote Socket L2->L2 HITM latency (data address homed in reader socket)
                        Reader Numa Node
Writer Numa Node     0       1
            0        -   185.9
            1    184.7       -
```

Figure 5: MLC analysis (cache to cache latency)