CPU Hardware Architecture and Performance Optimization

G. Amadio (CERN)

Performance Analysis on Modern CPUs

Performance is challenging

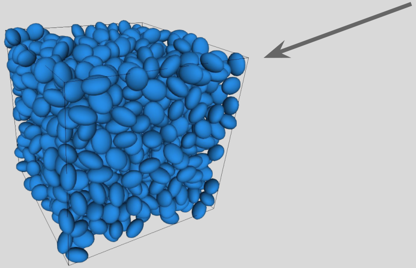
- Measuring Performance
  - Instrumentation and measurement has some overhead
  - Sophisticated hardware architecture (out of order, superscalar)
  - Variable CPU frequency scaling (turbo boost, thermal throttling)
  - Often missing symbols (JIT, interpreted languages, stripped binaries)
  - Unreliable stack unwinding (deep call stacks, inlining, missing frame pointers)
- Optimization and Tuning
  - Floating-point arithmetics is complicated (denormals)
  - Memory access patterns, fragmentation, (mis-)alignment
  - Concurrency issues (shared resources with hyperthreading, contention)
  - Reliance on compiler optimizations (exceptions vs vectorization, dead code)

Instrumentation-Based Profiling

- Use a timer and print out how long a section of code takes to run
  - Simplest form of instrumentation
  - Make changes and measure again
- Use an instrumentation-based profiler
  - May need to compile application with profiling information (`-g -pg`)
  - Run the application and analyze the output file
  - Examples: gprof, valgrind, uftrace
  - Yields number of calls for each function, unlike sampling
  - Usually suffers from high overhead
  - Cannot use in production systems

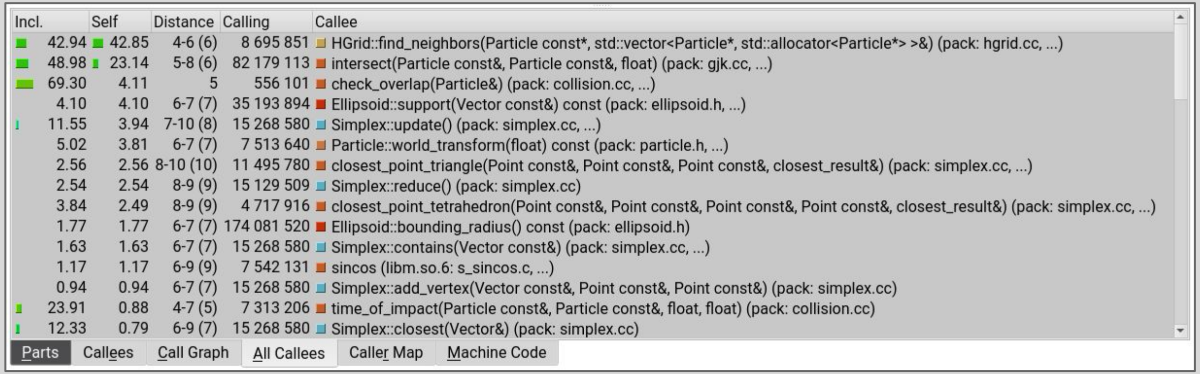
Flat profile example using gprof

```
$ pack -f 0.5 examples/ellipsoids # compiled with -O2 -g -pg, simulates a packing of ellipsoids, as shown below
100.00% 0.5000 0.0000/min 2.1e-01 ev/s 4.9 s
$ file gmon.out
gmon.out: GNU prof performance data - version 1
$ gprof --no-graph pack | head -n 20
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self               self     total
 time   seconds   seconds     calls   s/call   s/call  name
 34.05      0.95     0.95   7677145     0.00     0.00  HGrid::find_neighbors(Particle const*, std::vector<Particle*>&)
 24.73      1.64     0.69  66007037     0.00     0.00  intersect(Particle const&, Particle const&, float)
  7.89      1.86     0.22  31828514     0.00     0.00  Ellipsoid::support(Vector const&) const
  5.38      2.01     0.15   6685781     0.00     0.00  Particle::world_transform(float) const
  4.30      2.13     0.12 140620355     0.00     0.00  Ellipsoid::bounding_radius() const
  3.94      2.24     0.11  10459271     0.00     0.00  closest_point_triangle(Point&, Point&, Point&, result&)
  3.23      2.33     0.09  13858812     0.00     0.00  Simplex::add_vertex(Vector const&, Point const&, Point const&)
  2.15      2.39     0.06  13858812     0.00     0.00  Simplex::update()
  2.15      2.45     0.06   4288132     0.00     0.00  closest_point_tetrahedron(Point&, Point&, Point&, Point& result&)
  1.79      2.50     0.05  13732871     0.00     0.00  Simplex::reduce()
  1.79      2.55     0.05    481841     0.00     0.00  check_overlap(Particle&)
  1.79      2.60     0.05      1000     0.00     0.00  Ellipsoid::name() const
  1.43      2.64     0.04   6784500     0.00     0.00  time_of_impact(Particle const&, Particle const&, float, float)
  1.43      2.68     0.04   3342851     0.00     0.00  Simplex::reset()
  1.43      2.72     0.04                             _init
```



Flat profile example using valgrind

```
$ valgrind --tool=callgrind -- pack -f 0.5 examples/ellipsoids # no need for -pg
==2140677== Callgrind, a call-graph generating cache profiler
==2140677== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==2140677== Using Valgrind-3.23.0 and LibVEX; rerun with -h for copyright info
==2140677== Command: pack -f 0.5 examples/ellipsoids
==2140677==
==2140677== For interactive control, run 'callgrind_control -h'.
100.00% 0.5000 0.0000/min 6.7e-03 ev/s 150.0 s
==2140677==
==2140677== Events    : Ir
==2140677== Collected : 29183525425
==2140677==
==2140677== I   refs:      29,183,525,425
$ kcachegrind callgrind.out.2140677
```

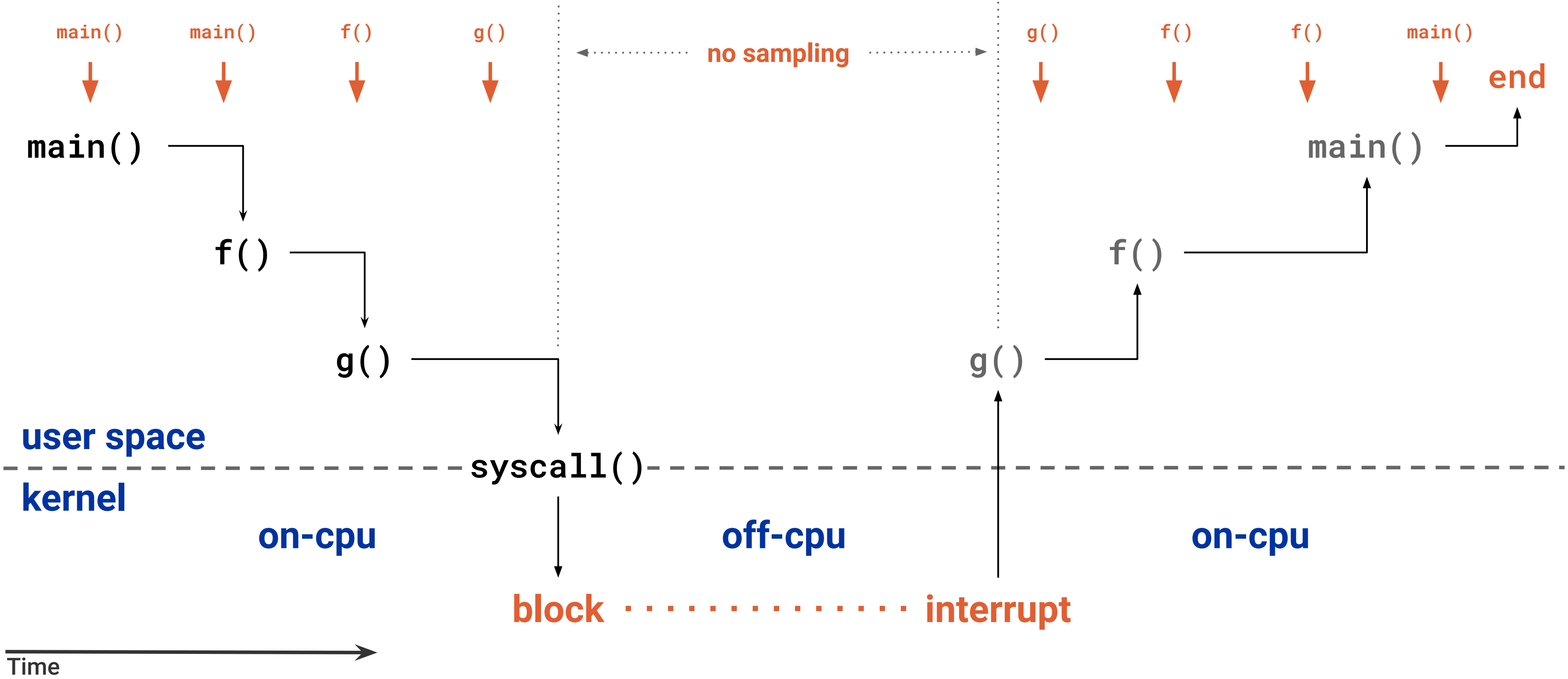


Incl.	Self	Distance	Calling	Callee
42.94	42.85	4-6 (6)	8 695 851	HGrid::find_neighbors(Particle const*, std::vector<Particle*, std::allocator<Particle*> >&) (pack: hgrid.cc, ...)
48.98	23.14	5-8 (6)	82 179 113	intersect(Particle const&, Particle const&, float) (pack: gjk.cc, ...)
69.30	4.11	5	556 101	check_overlap(Particle&) (pack: collision.cc, ...)
4.10	4.10	6-7 (7)	35 193 894	Ellipsoid::support(Vector const&) const (pack: ellipsoid.h, ...)
11.55	3.94	7-10 (8)	15 268 580	Simplex::update() (pack: simplex.cc, ...)
5.02	3.81	6-7 (7)	7 513 640	Particle::world_transform(float) const (pack: particle.h, ...)
2.56	2.56	8-10 (10)	11 495 780	closest_point_triangle(Point const&, Point const&, Point const&, closest_result&) (pack: simplex.cc, ...)
2.54	2.54	8-9 (9)	15 129 509	Simplex::reduce() (pack: simplex.cc)
3.84	2.49	8-9 (9)	4 717 916	closest_point_tetrahedron(Point const&, Point const&, Point const&, Point const&, closest_result&) (pack: simplex.cc, ...)
1.77	1.77	6-7 (7)	174 081 520	Ellipsoid::bounding_radius() const (pack: ellipsoid.h)
1.63	1.63	6-7 (7)	15 268 580	Simplex::contains(Vector const&) (pack: simplex.cc, ...)
1.17	1.17	6-9 (9)	7 542 131	sincos (libm.so.6: s_sincos.c, ...)
0.94	0.94	6-7 (7)	15 268 580	Simplex::add_vertex(Vector const&, Point const&, Point const&) (pack: simplex.cc)
23.91	0.88	4-7 (5)	7 313 206	time_of_impact(Particle const&, Particle const&, float, float) (pack: collision.cc)
12.33	0.79	6-9 (7)	15 268 580	Simplex::closest(Vector&) (pack: simplex.cc)

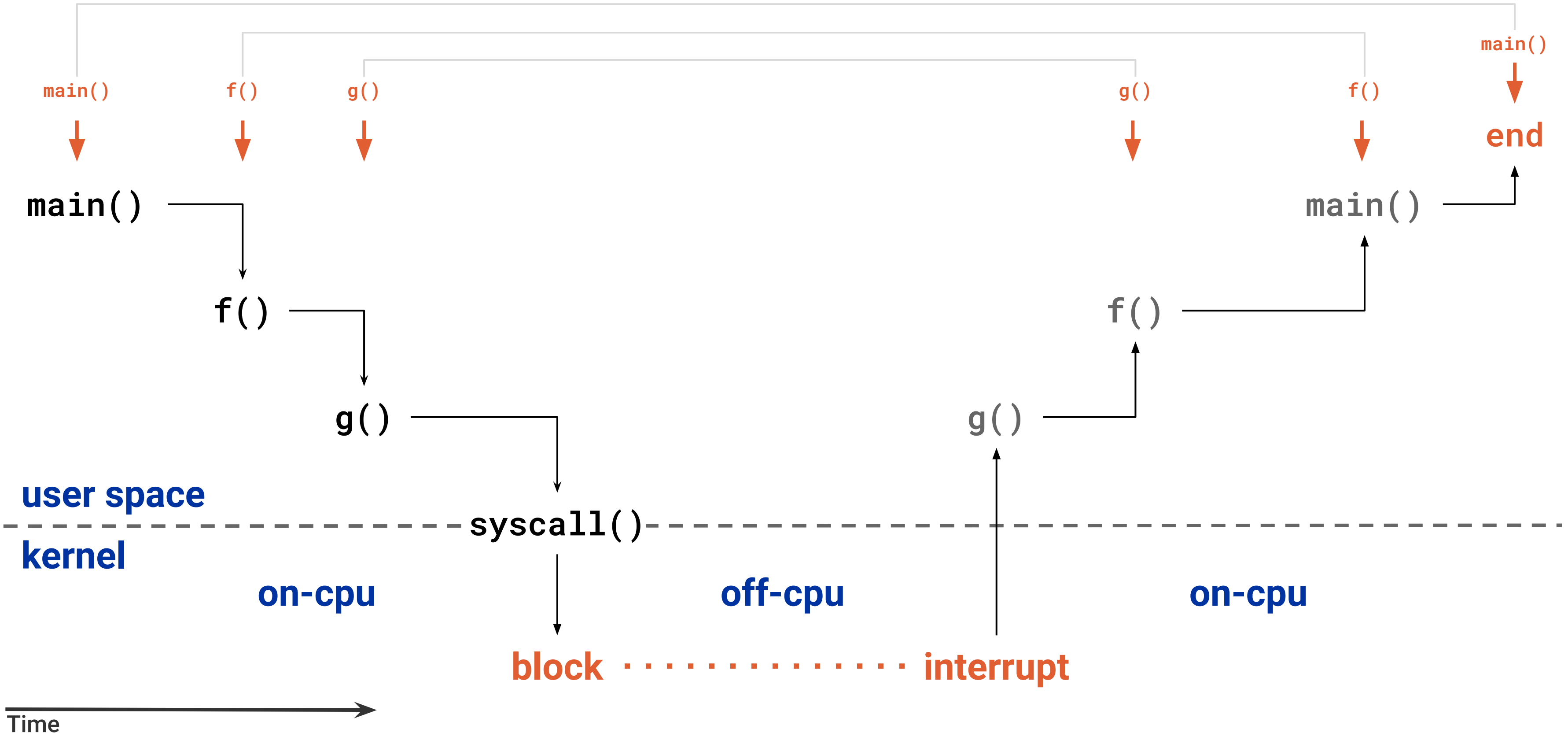
perf – Performance analysis tools for Linux

- Official Linux profiler (source code is part of the kernel itself)
- Both hardware and software based performance monitoring
- Much lower overhead compared with instrumentation-based profiling
- Kernel and user space
- Counting and Sampling
    - Counting — count occurrences of a given event (e.g. cache misses)
    - Event-based Sampling — a sample is recorded when a threshold of events has occurred
    - Time-based Sampling — samples are recorded at a given fixed frequency
    - Instruction-based Sampling — processor follows instructions and samples events they create
- Static and Dynamic Tracing
    - Static — pre-defined tracepoints in software
    - Dynamic — tracepoints created using uprobes (user) or kprobes (kernel)

Sampling


main()
main()
f()
g()
no sampling
g()
f()
f()
main()
end
main()
f()
g()
main()
f()
g()
user space
syscall()
kernel
on-cpu
off-cpu
on-cpu
block
interrupt
Time

Tracing


main()
f()
g()
g()
f()
main()
end
main()
f()
g()
g()
f()
main()
user space
syscall()
kernel
on-cpu
off-cpu
on-cpu
block
interrupt
Time

perf – subcommands

```
bash ~ $ perf

 usage: perf [--version] [--help] [OPTIONS] COMMAND [ARGS]

 The most commonly used perf commands are:
   annotate        Read perf.data (created by perf record) and display annotated code
   archive         Create archive with object files with build-ids found in perf.data file
   c2c             Shared Data C2C/HITM Analyzer.
   config          Get and set variables in a configuration file.
   data            Data file related processing
   diff            Read perf.data files and display the differential profile
   evlist          List the event names in a perf.data file
   list            List all symbolic event types
   mem             Profile memory accesses
   record          Run a command and record its profile into perf.data
   report          Read perf.data (created by perf record) and display the profile
   sched           Tool to trace/measure scheduler properties (latencies)
   script          Read perf.data (created by perf record) and display trace output
   stat            Run a command and gather performance counter statistics
   timechart       Tool to visualize total system behavior during a workload
   top             System profiling tool.
   version         display the version of perf binary
   probe           Define new dynamic tracepoints
   trace           strace inspired tool

 See 'perf help COMMAND' for more information on a specific command.
```

Flat profile example using perf

```
$ pack -f 0.5 examples/ellipsoids # compiled with -O2 -g
100.00% 0.5000 0.0001/min 3.2e-01 ev/s 3.1 s
$ perf record -F 1000 -e cycles -- pack -f 0.5 examples/ellipsoids
perf record -F 1000 -e cycles -- pack -f 0.5 examples/ellipsoids
100.00% 0.5000 0.0002/min 2.9e-01 ev/s 3.5 s
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.138 MB perf.data (3431 samples) ]
$ perf report --stdio | sed -ne /Overhead/,25p
# Overhead  Command  Shared Object      Symbol
# ........  .......  .................  .......................................
#
    34.07%  pack     pack               [.] HGrid::find_neighbors
    29.13%  pack     pack               [.] intersect
     8.23%  pack     pack               [.] Ellipsoid::support
     5.74%  pack     pack               [.] Particle::world_transform
     3.72%  pack     pack               [.] closest_point_tetrahedron
     3.65%  pack     pack               [.] closest_point_triangle
     2.42%  pack     pack               [.] Simplex::update
     2.27%  pack     pack               [.] Ellipsoid::bounding_radius
     2.25%  pack     pack               [.] Simplex::contains
     2.19%  pack     pack               [.] check_overlap
     0.95%  pack     libm.so.6          [.] __sincos
     0.61%  pack     pack               [.] HGrid::insert
     0.51%  pack     pack               [.] Simplex::reduce
     0.37%  pack     pack               [.] Simplex::closest
```

CPU Features for Performance Analysis

- Performance Monitoring Unit (PMU)
  - Performance monitoring counters (PMC)
    - Hardware: cycles, instructions, branches, stalled cycles in frontend/backend, etc
    - PMUs have several slots (usually 4–6) for counting hardware events together
    - Core PMU (CPU related events) and Uncore PMUs (I/O, caches, memory, interconnect)
    - If more events need to be measured than fit in a PMU, this needs to be done via multiplexing
- Varies depending on hardware vendor/model
  - Basic events have equivalents in most hardware
  - More specific events may only be available on certain hardware models
  - Some events have the same name, but count different things (e.g. cache misses)
- Profilers make use of hardware/software events
  - Software events: page faults, context switches, migrations, etc
- Intel VTune, AMD μprof, macOS Instruments, Linux perf, etc

perf – hardware and software events

```
bash ~ $ perf list hw cache

List of pre-defined events (to be used in -e):

  branch-instructions OR branches                    [Hardware event]
  branch-misses                                      [Hardware event]
  cache-misses                                       [Hardware event]
  cache-references                                   [Hardware event]
  cpu-cycles OR cycles                               [Hardware event]
  instructions                                       [Hardware event]
  stalled-cycles-backend OR idle-cycles-backend      [Hardware event]
  stalled-cycles-frontend OR idle-cycles-frontend    [Hardware event]

  L1-dcache-load-misses                              [Hardware cache event]
  L1-dcache-loads                                    [Hardware cache event]
  L1-dcache-prefetches                               [Hardware cache event]
  L1-icache-load-misses                              [Hardware cache event]
  L1-icache-loads                                    [Hardware cache event]
  branch-load-misses                                 [Hardware cache event]
  branch-loads                                       [Hardware cache event]
  dTLB-load-misses                                   [Hardware cache event]
  dTLB-loads                                         [Hardware cache event]
  iTLB-load-misses                                   [Hardware cache event]
  iTLB-loads                                         [Hardware cache event]
```

```
bash ~ $ perf list sw

List of pre-defined events (to be used in -e):

  alignment-faults                                   [Software event]
  bpf-output                                         [Software event]
  context-switches OR cs                             [Software event]
  cpu-clock                                          [Software event]
  cpu-migrations OR migrations                       [Software event]
  dummy                                              [Software event]
  emulation-faults                                   [Software event]
  major-faults                                       [Software event]
  minor-faults                                       [Software event]
  page-faults OR faults                              [Software event]
  task-clock                                         [Software event]

  duration_time                                      [Tool event]
```

perf – Intel Skylake events

```
bash ~ $ perf list pipeline

List of pre-defined events (to be used in -e):

pipeline:
  arith.divider_active
       [Cycles when divide unit is busy executing divide or square root operations. Accounts for integer and floating-point operations]
  baclears.any
       [Counts the total number when the front end is resteered, mainly when the BPU cannot provide a correct prediction]
  br_inst_retired.all_branches
       [All (macro) branch instructions retired Spec update: SKL091]
  br_inst_retired.all_branches_pebs
       [All (macro) branch instructions retired Spec update: SKL091 (Must be precise)]
  br_inst_retired.conditional
       [Conditional branch instructions retired Spec update: SKL091 (Precise event)]
  br_inst_retired.far_branch
       [Counts the number of far branch instructions retired Spec update: SKL091 (Precise event)]
  br_inst_retired.near_call
       [Direct and indirect near call instructions retired Spec update: SKL091 (Precise event)]
  br_inst_retired.near_return
       [Return instructions retired Spec update: SKL091 (Precise event)]
  br_inst_retired.near_taken
       [Taken branch instructions retired Spec update: SKL091 (Precise event)]
  br_inst_retired.not_taken
       [Counts all not taken macro branch instructions retired Spec update: SKL091 (Precise event)]
  br_misp_retired.all_branches
       [All mispredicted macro branch instructions retired]
  ...
```

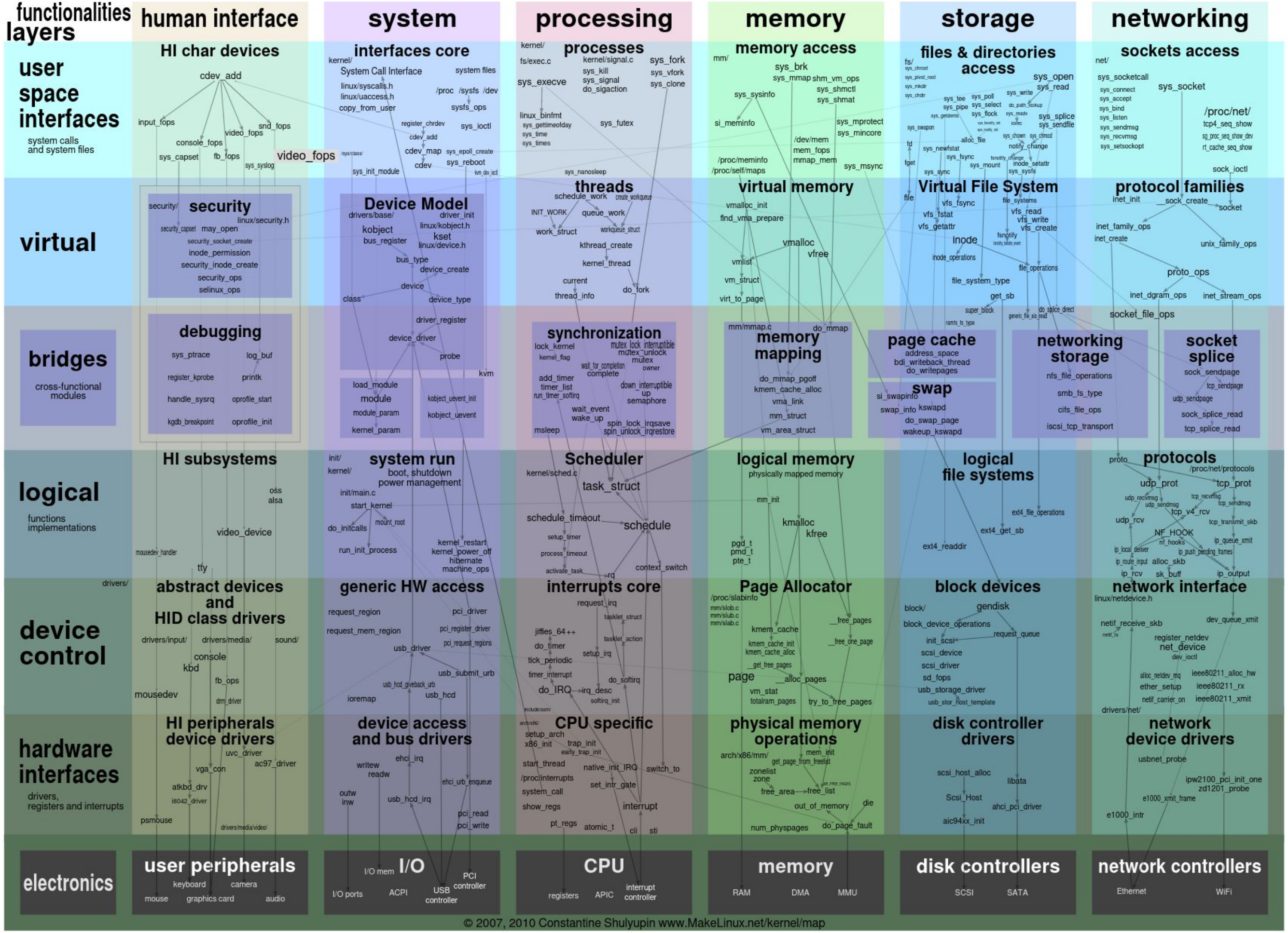
perf – AMD Ryzen events

```
bash ~ $ perf list core

List of pre-defined events (to be used in -e):

core:
  ex_div_busy
       [Div Cycles Busy count]
  ex_div_count
       [Div Op Count]
  ex_ret_brn
       [Retired Branch Instructions]
  ex_ret_brn_far
       [Retired Far Control Transfers]
  ex_ret_brn_ind_misp
       [Retired Indirect Branch Instructions Mispredicted]
  ex_ret_brn_misp
       [Retired Branch Instructions Mispredicted]
  ex_ret_brn_resync
       [Retired Branch Resyncs]
  ex_ret_brn_tkn
       [Retired Taken Branch Instructions]
  ex_ret_brn_tkn_misp
       [Retired Taken Branch Instructions Mispredicted]
  ex_ret_cond
       [Retired Conditional Branch Instructions]
  ex_ret_cond_misp
       [Retired Conditional Branch Instructions Mispredicted]
  ...
```

perf – static tracepoint events

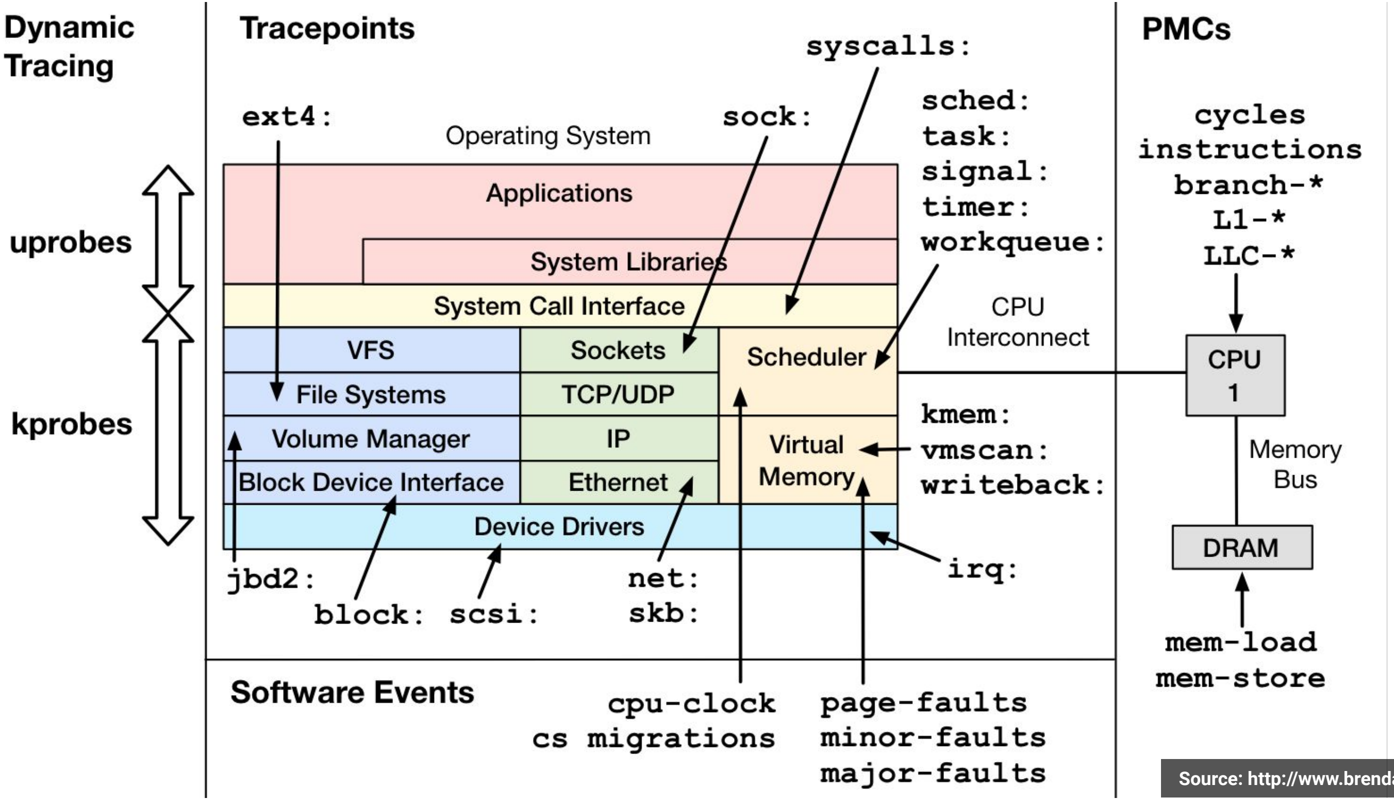
```
bash ~ $ sudo perf list 'sched:*'

List of pre-defined events (to be used in -e):

  sched:sched_kthread_stop                           [Tracepoint event]
  sched:sched_kthread_stop_ret                       [Tracepoint event]
  sched:sched_migrate_task                           [Tracepoint event]
  sched:sched_move_numa                              [Tracepoint event]
  sched:sched_pi_setprio                             [Tracepoint event]
  sched:sched_process_exec                           [Tracepoint event]
  sched:sched_process_exit                           [Tracepoint event]
  sched:sched_process_fork                           [Tracepoint event]
  sched:sched_process_free                           [Tracepoint event]
  sched:sched_process_wait                           [Tracepoint event]
  sched:sched_stat_runtime                           [Tracepoint event]
  sched:sched_stick_numa                             [Tracepoint event]
  sched:sched_swap_numa                              [Tracepoint event]
  sched:sched_switch                                 [Tracepoint event]
  sched:sched_wait_task                              [Tracepoint event]
  sched:sched_wake_idle_without_ipi                  [Tracepoint event]
  sched:sched_wakeup                                 [Tracepoint event]
  sched:sched_wakeup_new                             [Tracepoint event]
  sched:sched_waking                                 [Tracepoint event]
```

Map of the Linux Kernel



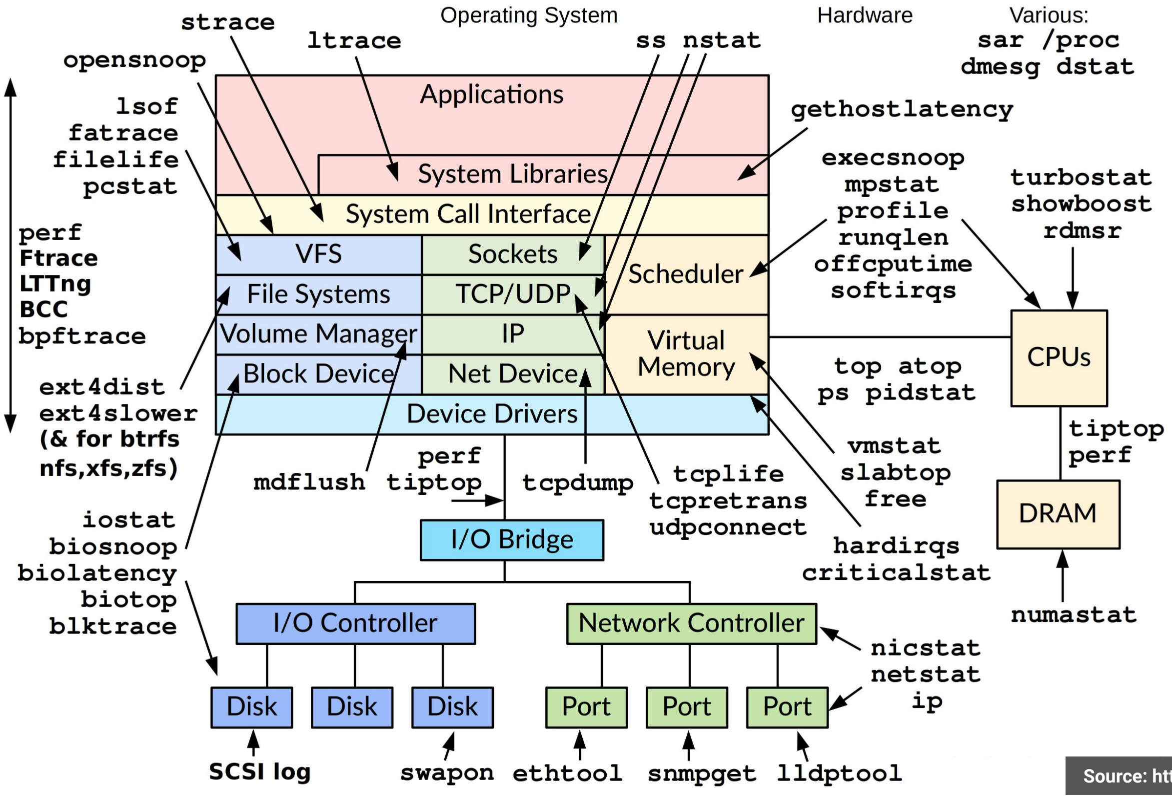
© 2007, 2010 Constantine Shulyupin www.MakeLinux.net/kernel/map

perf – event sources



Source: http://www.brendangregg.com/perf.html

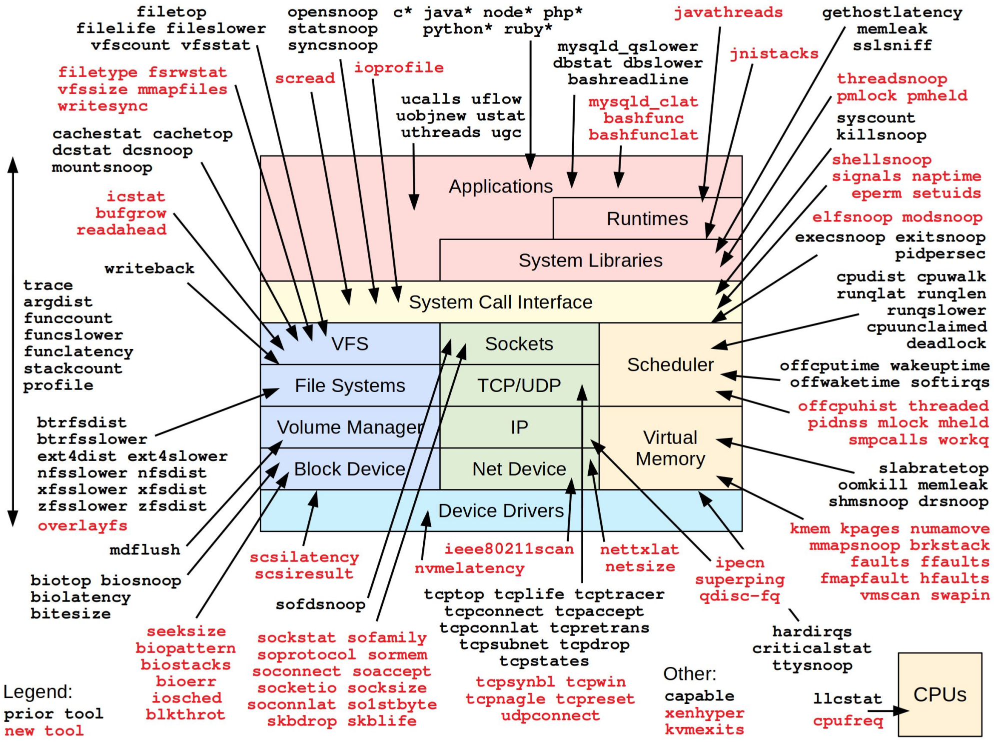
Linux Observability Tools



Source: http://www.brendangregg.com/perf.html

Linux eBPF-based Observability Tools

New tools developed for the book **BPF Performance Tools: Linux System and Application Observability** by Brendan Gregg (Addison Wesley, 2019), which also covers **prior BPF tools**



filetop
filelife fileslower
vfscount vfsstat

opensnoop
statsnoop
syncsnoop

c* java* node* php*
python* ruby*

javathreads

gethostlatency
memleak
sslsniff

filetype fsrwstat
vfssize mmapfiles
writesync

scread

ioprofile

ucalls uflow
uobjnew ustat
uthreads ugc

mysqld_qslower
dbstat dbslower
bashreadline

mysqld_clat
bashfunc
bashfunclat

jnistacks

threadsnoop
pmlock pmheld

syscount
killsnoop

cachestat cachetop
dcstat dcsnoop
mountsnoop

shellsnoop
signals naptime
eperm setuids

icstat
bufgrow
readahead

elfsnoop modsnoop

execsnoop exitsnoop
pidpersec

writeback

trace
argdist
funccount
funcslower
funclatency
stackcount
profile

cpudist cpuwalk
runqlat runqlen
runqslower
cpuunclaimed
deadlock

offcputime wakeuptime
offwaketime softirqs

offcpuhist threaded
pidnss mlock mheld
smpcalls workq

Applications

Runtimes

System Libraries

System Call Interface

VFS

Sockets

Scheduler

File Systems

TCP/UDP

Volume Manager

IP

Virtual Memory

Block Device

Net Device

Device Drivers

btrfsdist
btrfsslower
ext4dist ext4slower
nfsslower nfsdist
xfsslower xfsdist
zfsslower zfsdist
overlayfs

slabratetop
oomkill memleak
shmsnoop drsnoop

kmem kpages numamove
mmapsnoop brkstack
faults ffaults
fmapfault hfaults
vmscan swapin

mdflush

scsilatency
scsiresult

ieee80211scan
nvmelatency

nettxlat
netsize

ipecn
superping
qdisc-fq

biotop biosnoop
biolatency
bitesize

sofdsnoop

tcptop tcplife tcptracer
tcpconnect tcpaccept
tcpconnlat tcpretrans
tcpsubnet tcpdrop
tcpstates

hardirqs
criticalstat
ttysnoop

seeksize
biopattern
biostacks
bioerr
iosched
blkthrot

sockstat sofamily
soprotocol sormem
soconnect soaccept
socketio socksize
soconnlat so1stbyte
skbdrop skblife

tcpsynbl tcpwin
tcpnagle tcpreset
udpconnect

Other:
capable
xenhyper
kvmexits

llcstat
cpufreq

CPUs

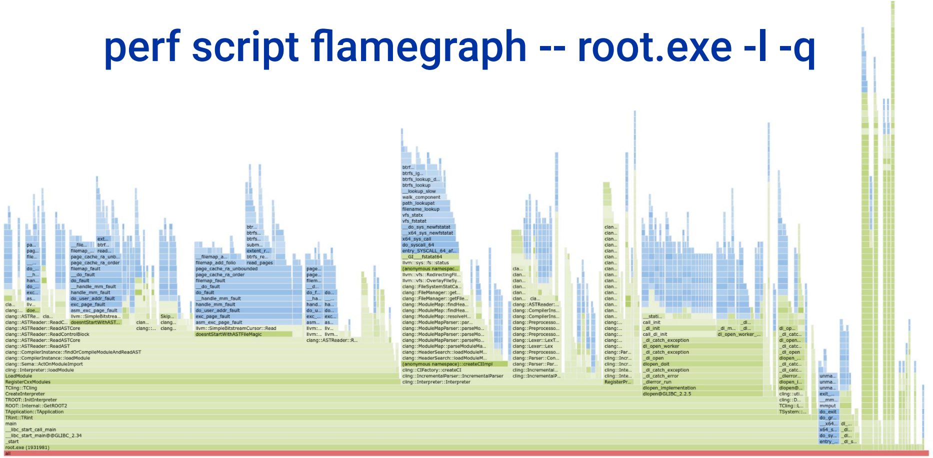
Legend:
prior tool
new tool

Source: http://www.brendangregg.com/perf.html

Flamegraphs

- Visualization tool by Brendan Gregg
    - https://www.brendangregg.com/flamegraphs.html
- Call stacks on the vertical axis
- Number of samples as width
- Easy to identify where time is spent
- Not very good for in-depth analysis
- Built-in support now exists in perf
- Creates browseable HTML file

perf script flamegraph -- root.exe -l -q



Avoid broken stack traces and missing symbols

- Compile code with debugging information (`-g`)
- Add `-fno-omit-frame-pointer` to compile options to keep frame pointer
- Install system packages with debugging info for the kernel and system libs

When recording data:

- Use `--call-graph=fp/dwarf` + DWARF debugging information
- Use precise events to avoid skidding (`cycles:pp` instead of just `cycles`)
- Adjust sampling rate to avoid large amounts of data and high overhead
- Sample events in a group if computing derived metrics (e.g. instr. per cycle)
- See `man perf-list` for more information on events and their modifiers

Frame Pointer

- Saved/restored on each function call
- Lightweight and accurate backtraces
- DWARF backtraces not as accurate
- High overhead for very short functions

Simple square and cube functions

```
float square(float x)
{
    return x * x;
}

float cube(float x)
{
    return x * square(x);
}
```

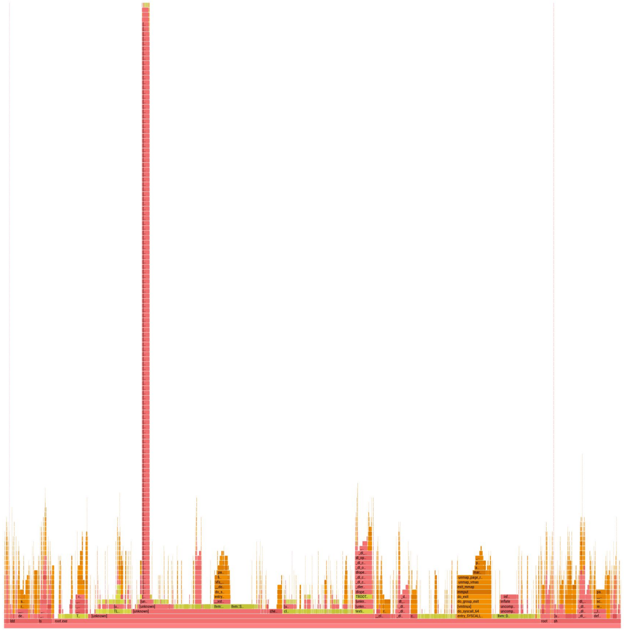
Without frame pointer

```
0000000000000000 <square>:
   0:	c5 fa 59 c0          	vmulss %xmm0,%xmm0,%xmm0
   4:	c3                   	ret
   5:	66 66 2e 0f 1f 84 00 	data16 cs nopw 0x0(%rax,%rax,1)
   c:	00 00 00 00

0000000000000010 <cube>:
  10:	c5 f8 28 c8          	vmovaps %xmm0,%xmm1
  14:	e8 00 00 00 00       	call   19 <cube+0x9>
  19:	c5 fa 59 c1          	vmulss %xmm1,%xmm0,%xmm0
  1d:	c3                   	ret
```

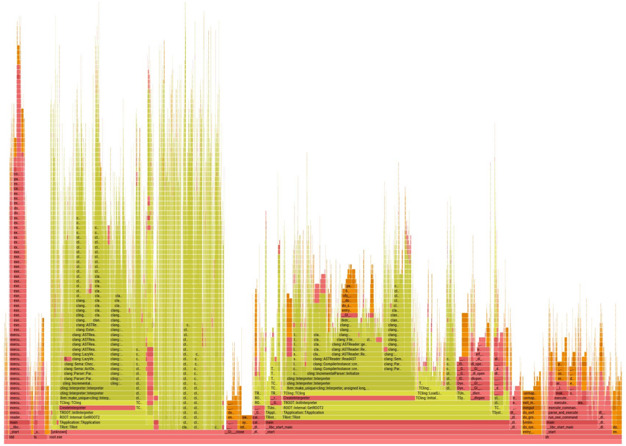
With frame pointer

```
0000000000000000 <square>:
   0:	c5 fa 59 c0          	vmulss %xmm0,%xmm0,%xmm0
   4:	c3                   	ret
   5:	66 66 2e 0f 1f 84 00 	data16 cs nopw 0x0(%rax,%rax,1)

0000000000000010 <cube>:
  10:	55                   	push   %rbp
  11:	c5 f8 28 c8          	vmovaps %xmm0,%xmm1
  15:	48 89 e5             	mov    %rsp,%rbp
  18:	e8 00 00 00 00       	call   1d <cube+0xd>
  1d:	5d                   	pop    %rbp
  1e:	c5 fa 59 c1          	vmulss %xmm1,%xmm0,%xmm0
  22:	c3                   	ret
```

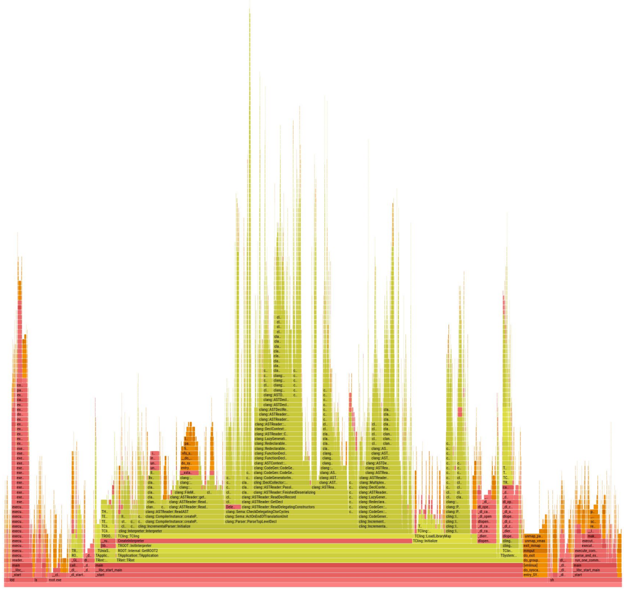
ROOT startup flamegraph for various configurations



perf record --call-graph=fp
(debugging info not available)

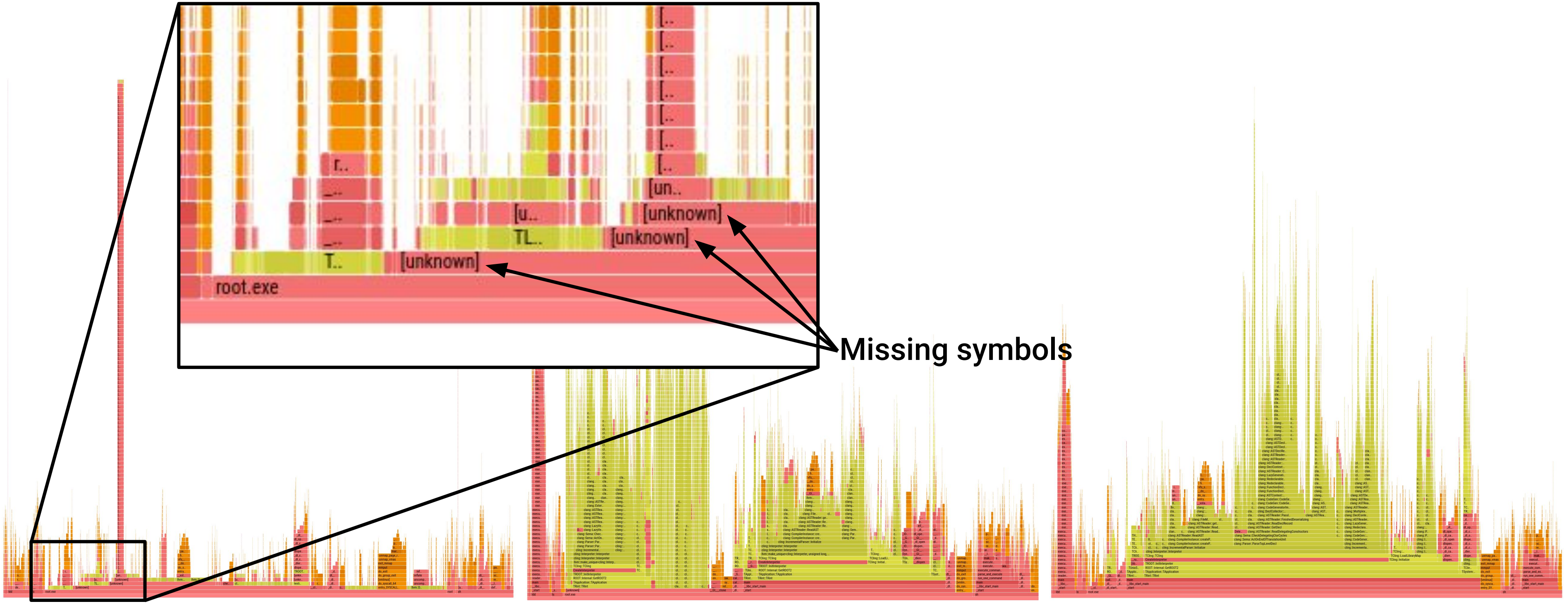


perf record --call-graph=dwarf
(frame pointer not available)



perf record --call-graph=fp
(frame pointer and debugging info)

ROOT startup flamegraph for various configurations



Missing symbols

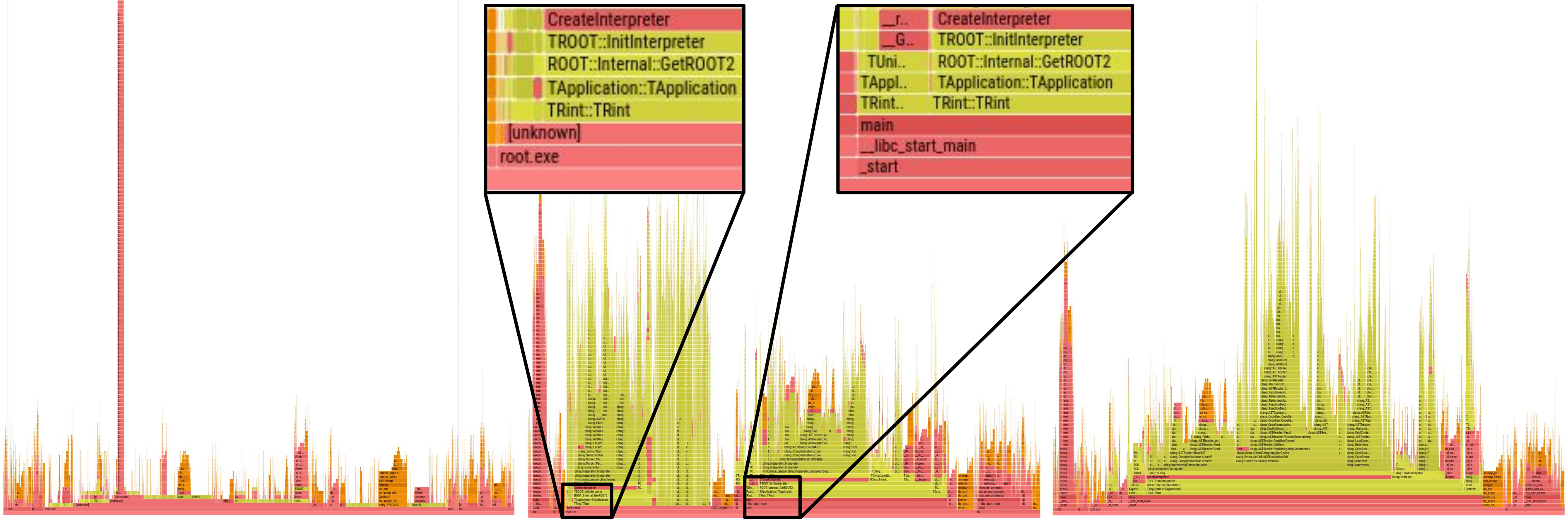
perf record --call-graph=fp
(debugging info not available)

perf record --call-graph=dwarf
(frame pointer not available)

perf record --call-graph=fp
(frame pointer and debugging info)

ROOT startup flamegraph for various configurations

Broken stack unwinding

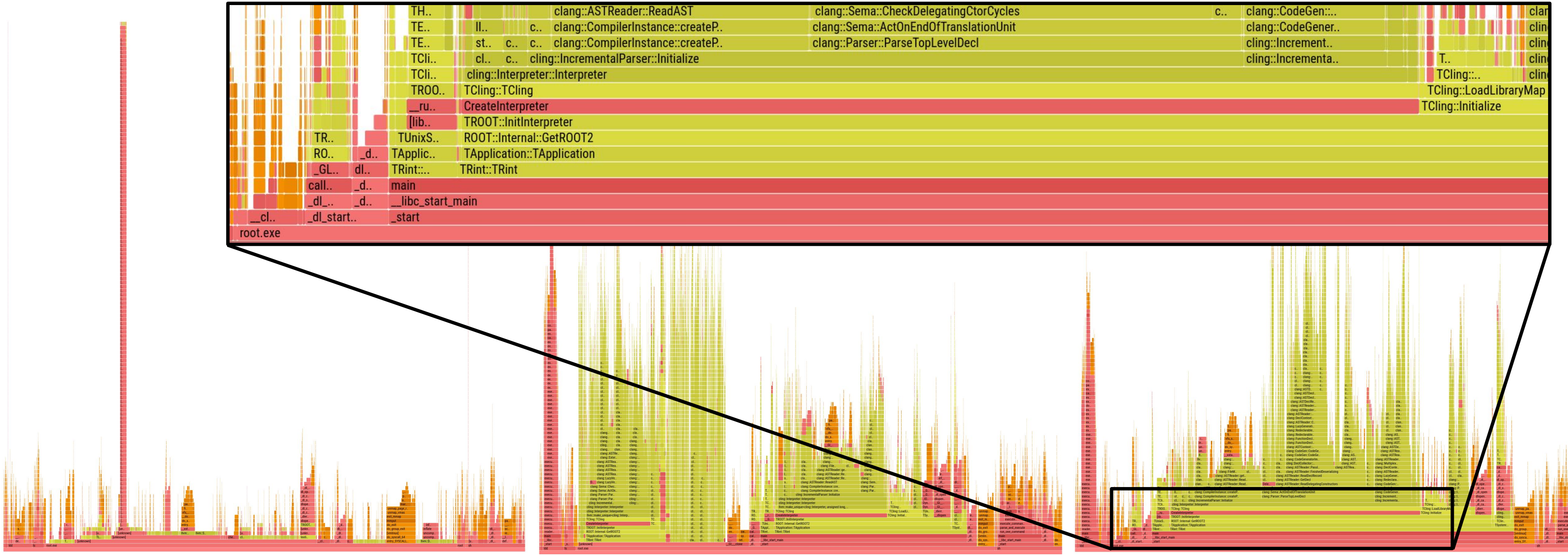


perf record --call-graph=fp
(debugging info not available)

perf record --call-graph=dwarf
(frame pointer not available)

perf record --call-graph=fp
(frame pointer and debugging info)

ROOT startup flamegraph for various configurations



Correctly merged stacks

perf record --call-graph=fp
(debugging info not available)

perf record --call-graph=dwarf
(frame pointer not available)

perf record --call-graph=fp
(frame pointer and debugging info)

perf stat – counting cycles vs instructions vs wall time

```
# measure ROOT startup 20 times and print stats with averages and deviations

 $ perf stat -d -r 20 -- root.exe -l -q >/dev/null

 Performance counter stats for 'root.exe -l -q' (20 runs):

         119.72 msec task-clock                #    0.442 CPUs utilized          ( +-  0.76% )
            579      context-switches          #    0.005 M/sec                  ( +-  4.34% )
             13      cpu-migrations            #    0.109 K/sec                  ( +-  6.94% )
          11260      page-faults               #    0.094 M/sec                  ( +-  0.49% )
      493768274      cycles                    #    4.125 GHz                    ( +-  0.75% )  (66.72%)
       33420383      stalled-cycles-frontend   #    6.77% frontend cycles idle   ( +-  1.56% )  (75.75%)
      177325752      stalled-cycles-backend    #   35.91% backend cycles idle    ( +-  1.87% )  (79.76%)
      532310517      instructions              #    1.08  insn per cycle
                                               #    0.33  stalled cycles per insn ( +-  0.35% )  (82.16%)
      107905661      branches                  #  901.351 M/sec                  ( +-  0.26% )  (82.38%)
        2282743      branch-misses             #    2.12% of all branches        ( +-  0.77% )  (78.52%)
      246528817      L1-dcache-loads           # 2059.290 M/sec                  ( +-  1.12% )  (71.14%)
        5628008      L1-dcache-load-misses     #    2.28% of all L1-dcache hits  ( +-  1.30% )  (63.57%)
 <not supported>      LLC-loads
 <not supported>      LLC-load-misses

         0.2709 +- 0.0205 seconds time elapsed  ( +-  7.58% )
```

```
# same measurements again, to show difference in noise for wall time, cycles, instructions

 $ perf stat -d -r 20 -- root.exe -l -q >/dev/null

 Performance counter stats for 'root.exe -l -q' (20 runs):

         118.38 msec task-clock                #    0.565 CPUs utilized          ( +-  0.73% )
            433      context-switches          #    0.004 M/sec                  ( +- 12.62% )
             12      cpu-migrations            #    0.103 K/sec                  ( +-  5.57% )
          11267      page-faults               #    0.095 M/sec                  ( +-  0.50% )
      488189557      cycles                    #    4.124 GHz                    ( +-  0.73% )  (60.32%)
       32509432      stalled-cycles-frontend   #    6.66% frontend cycles idle   ( +-  1.70% )  (78.43%)
      175081210      stalled-cycles-backend    #   35.86% backend cycles idle    ( +-  1.45% )  (83.54%)
      533538019      instructions              #    1.09  insn per cycle
                                               #    0.33  stalled cycles per insn ( +-  0.35% )  (84.97%)
      108436560      branches                  #  915.999 M/sec                  ( +-  0.29% )  (84.34%)
        2279445      branch-misses             #    2.10% of all branches        ( +-  1.05% )  (81.41%)
      244414949      L1-dcache-loads           # 2064.653 M/sec                  ( +-  0.94% )  (71.80%)
        5720566      L1-dcache-load-misses     #    2.34% of all L1-dcache hits  ( +-  1.35% )  (55.19%)
 <not supported>      LLC-loads
 <not supported>      LLC-load-misses

         0.2093 +- 0.0220 seconds time elapsed  ( +- 10.53% )


# (ratio of wall clock durations)
$ bc -l <<< "0.2709 / 0.2093"
1.29431438127090301003

# (ratio of cycles measurements)
$ bc -l <<< "493768274 / 488189557"
1.01142735832835522944

# (ratio of instructions measurements)
$ bc -l <<< "532310517 / 533538019"
0.99769931671917086006
```

Intel’s Last Branch Record

- Useful when frame pointers are not available
- Use with `perf record -b` or `perf record --call-graph=lbr`
- Hardware registers on Intel CPUs that allow sampling branches
- Registers hold a ring buffer of the most recent branch decisions
- Useful to analyze branching behavior (branching probabilities, mispredictions)
- Available on AMD Zen4 or later CPUs
  - On older CPUs, some events provide similar functionality
- Articles describing LBR on LWN.net
  - An introduction to last branch records [LWN.net]
  - Advanced usage of last branch records [LWN.net]

Precise CPU Events for Sampling

- PMU counts events on a per-core basis
    - Sample is taken when counter reaches threshold
    - Fixed frequency sampling achieved by predicting/adjusting the threshold
    - Instruction-level parallelism and speculative execution introduce noise and skidding
        - Only one base pointer per thread
        - Many instructions in flight on the core at the same time
        - Shared resources mean mixed counting when using hyperthreading
- Intel Processor Event-Based Sampling (PEBS)
    - Instruction pointer (and auxiliary information) stored in a designated area
    - No interrupts during sampling, reduced or no skidding
- AMD Instruction-Based Sampling (IBS)
    - Tracks instructions rather than events, marks every Nth instruction to be tracked
    - Two forms: IBS Fetch sampling (front-end) and IBS Op sampling (back-end)

“Precise Event Sampling on AMD Versus Intel: Quantitative and Qualitative Comparison” doi: 10.1109/TPDS.2023.3257105

Instructions vs Micro-operations (µops)

Instructions from a CISC instruction set are usually broken into one or more RISC-like operations in hardware. For example, an addition of two values from memory may be broken into memory loads into registers, the addition itself, then memory stores.
These operations are usually called **micro-ops** and abbreviated as **µops**. Some PMUs have hardware events that allow counting separately µops issued, executed, and retired.

While instructions are usually split into simpler µops, the µops can instead be fused together when instructions are decoded in the front-end of the processor. **Microfusion** is when µops from the same machine instruction are fused together, and **macrofusion** is when µops from distinct instructions are fused.

Instructions Retired vs Executed

Instructions executed refers to any instructions that have been processed by the CPU. For example, a multiplication of two numbers that has loaded the inputs, calculated the results and stored it somewhere. This metric includes speculatively executed instructions on branches that may have been discarded later on.

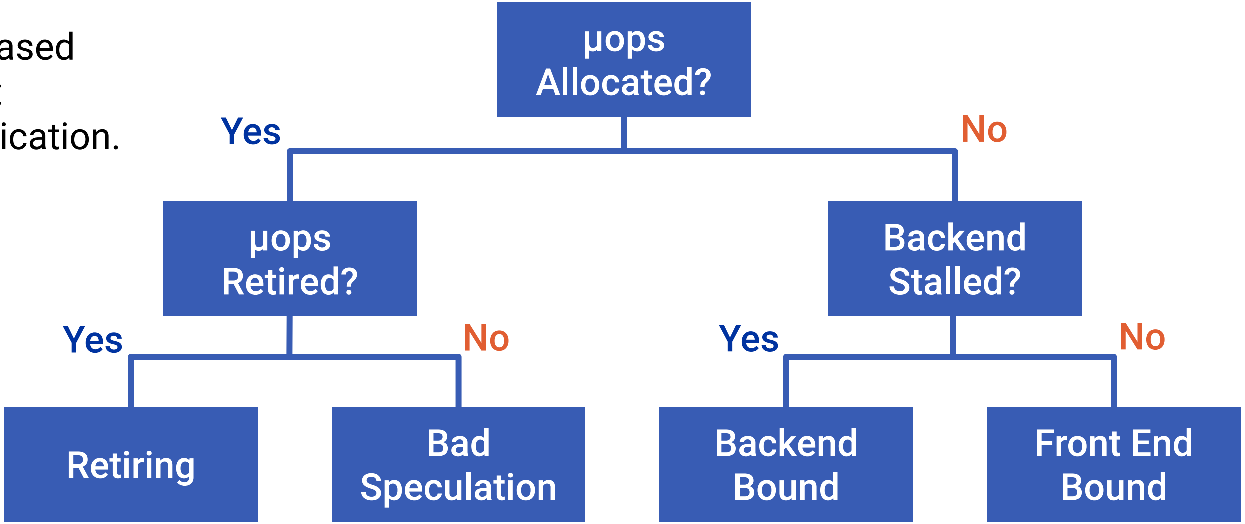
Instructions retired refers to executed instructions that have actually contributed to the main line of execution of a program, that is, that has not been discarded as speculatively executed.

Instructions per cycle (IPC) is a measure of the instruction-level parallelism, or how many instructions were retired on average in each CPU cycle. CPI (cycles per instruction) is also common. Typically up to 4 instructions per cycle can be executed on AMD/Intel CPUs.

Top-Down Microarchitecture Analysis

The Top-Down Characterization is a hierarchical organization of event-based metrics that identifies the dominant performance bottlenecks in an application.

Its aim is to show, on average, how well the CPU’s pipelines are being utilized while running an application.

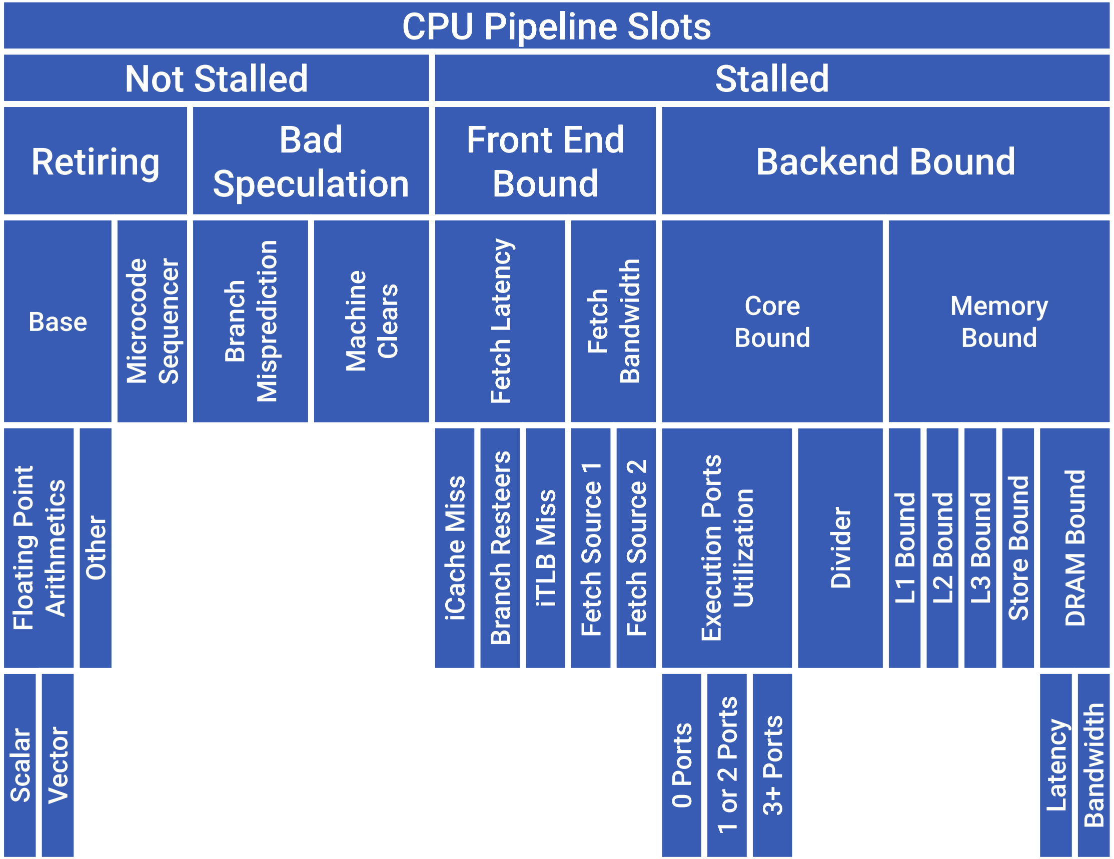


Ahmad Yasin, "A Top-Down method for performance analysis and counters architecture," *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Monterey, CA, 2014, pp. 35-44, doi: 10.1109/ISPASS.2014.6844459.

`https://www.intel.com/content/www/us/en/develop/documentation/vtune-cookbook/top/methodologies/top-down-microarchitecture-analysis-method.html`

Top-Down Microarchitecture Analysis

- Retiring
    - Useful Work
- Bad Speculation
    - Branching Issues
- Front End Bound
    - Instruction Fetch Issues
- Back End Bound
    - Core Bound
        - Port Utilization
        - Execution Latency
    - Memory Bound
        - Cache misses
        - Memory Bandwidth



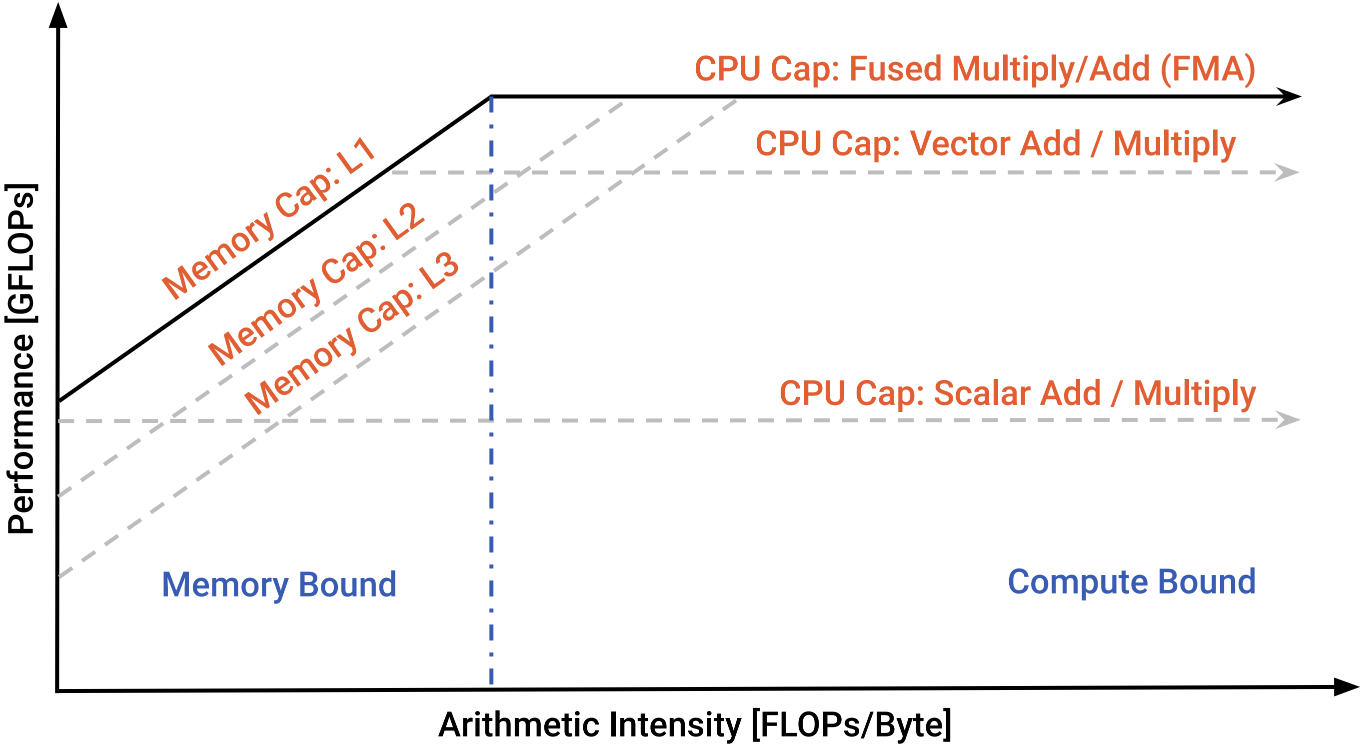
Ahmad Yasin, "A Top-Down method for performance analysis and counters architecture," *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Monterey, CA, 2014, pp. 35-44, doi: 10.1109/ISPASS.2014.6844459

Expected Ranges of Pipeline Slots for Each Category

Category	Client/Desktop Application	Server/Database Distributed Application	High Performance Computing (HPC) Application
Retiring	20 – 50%	10 – 30%	30 – 70%
Back-End Bound	20 – 40%	20 – 60%	20 – 40%
Front-End Bound	5 – 10%	10 – 25%	5 – 10%
Bad Speculation	5 – 10%	5 – 10%	1 – 5%

https://www.intel.com/content/www/us/en/develop/documentation/vtune-cookbook/top/methodologies/top-down-microarchitecture-analysis-method.html

Roofline Performance Model



https://www.intel.com/content/www/us/en/developer/articles/guide/intel-advisor-roofline.html

perf – recording and reporting data

```
bash ~ $ perf record -g -F max -- root.exe -l -q
info: Using a maximum frequency rate of 32500 Hz

[ perf record: Woken up 6 times to write data ]
[ perf record: Captured and wrote 2.003 MB perf.data (7035 samples) ]
bash ~ $ perf report -q --stdio -c root.exe | head -n 20
# comm: root.exe
    82.13%     0.00%  root.exe                 [.] _start
            |
            ---_start
               __libc_start_main@@GLIBC_2.34
               __libc_start_call_main
               main
               |
               |--79.94%--TRint::TRint
               |          |
               |          |--76.22%--TApplication::TApplication
               |          |          |
               |          |           --76.14%--ROOT::Internal::GetROOT2
               |          |                     TROOT::InitInterpreter
               |          |                     |
               |          |                     |--69.43%--CreateInterpreter
               |          |                     |          |
               |          |                     |           --69.41%--TCling::TCling
               |          |                     |                     |
               |          |                     |                     |--32.52%--RegisterCxxModules
```

perf – flat profile report

```
bash ~ $ perf report -q --stdio --call-graph=none -c root.exe | head -n 25
# comm: root.exe
    82.13%     0.01%  root.exe                [.] main
    82.13%     0.00%  libc.so.6               [.] __libc_start_call_main
    82.13%     0.00%  libc.so.6               [.] __libc_start_main@@GLIBC_2.34
    82.13%     0.00%  root.exe                [.] _start
    79.94%     0.00%  libRint.so.6.30.06      [.] TRint::TRint
    76.22%     0.00%  libCore.so.6.30.06      [.] TApplication::TApplication
    76.14%     0.00%  libCore.so.6.30.06      [.] ROOT::Internal::GetROOT2
    76.14%     0.00%  libCore.so.6.30.06      [.] TROOT::InitInterpreter
    69.43%     0.00%  libCling.so.6.30.06     [.] CreateInterpreter
    69.41%     0.00%  libCling.so.6.30.06     [.] TCling::TCling
    38.76%     0.00%  libCling.so.6.30.06     [.] clang::CompilerInstance::loadModule
    38.52%     0.00%  libCling.so.6.30.06     [.] clang::CompilerInstance::findOrCompileModuleAndReadAST
    38.32%     0.21%  libCling.so.6.30.06     [.] clang::ASTReader::ReadAST
    32.52%     0.00%  libCling.so.6.30.06     [.] RegisterCxxModules
    32.26%     0.00%  libCling.so.6.30.06     [.] LoadModule
    31.87%     0.00%  libCling.so.6.30.06     [.] cling::Interpreter::loadModule
    31.75%     0.01%  libCling.so.6.30.06     [.] clang::Sema::ActOnModuleImport
    26.63%     0.00%  libCling.so.6.30.06     [.] cling::Interpreter::Interpreter
    22.80%     0.28%  [kernel.kallsyms]       [k] entry_SYSCALL_64
    22.51%     0.29%  [kernel.kallsyms]       [k] asm_exc_page_fault
    22.14%     0.36%  [kernel.kallsyms]       [k] do_syscall_64
    21.68%     0.31%  [kernel.kallsyms]       [k] exc_page_fault
    19.17%     0.38%  [kernel.kallsyms]       [k] do_user_addr_fault
    19.08%     0.00%  libCling.so.6.30.06     [.] cling::IncrementalParser::ParseInternal
```

perf – flat profile report by self-time

```
bash ~ $ perf report -q --stdio --call-graph=none --no-children --percent-limit 0.75 -c root.exe
# comm: root.exe
     5.85%  libz.so.1.3.1           [.] inflate_fast
     4.11%  libCling.so.6.30.06     [.] llvm::SimpleBitstreamCursor::Read
     2.63%  [kernel.kallsyms]       [k] unmap_page_range
     2.27%  libCling.so.6.30.06     [.] llvm::BitstreamCursor::readRecord
     1.89%  [kernel.kallsyms]       [k] __mod_lruvec_state
     1.78%  [kernel.kallsyms]       [k] srso_untrain_ret
     1.77%  [kernel.kallsyms]       [k] srso_return_thunk
     1.53%  [kernel.kallsyms]       [k] trace_hardirqs_off
     1.43%  libz.so.1.3.1           [.] adler32_z
     1.32%  [kernel.kallsyms]       [k] __lruvec_stat_mod_folio
     1.31%  [kernel.kallsyms]       [k] clear_page_rep
     1.31%  ld-linux-x86-64.so.2    [.] _dl_lookup_symbol_x
     1.26%  libCling.so.6.30.06     [.] llvm::StringMapImpl::LookupBucketFor
     1.10%  [kernel.kallsyms]       [k] preempt_count_add
     1.04%  [kernel.kallsyms]       [k] __mod_memcg_lruvec_state
     0.97%  [kernel.kallsyms]       [k] link_path_walk
     0.94%  [kernel.kallsyms]       [k] preempt_count_sub
     0.92%  libz.so.1.3.1           [.] inflate_table
     0.85%  [kernel.kallsyms]       [k] percpu_counter_add_batch
     0.81%  libc.so.6               [.] _int_malloc
     0.79%  libz.so.1.3.1           [.] inflate
     0.76%  ld-linux-x86-64.so.2    [.] do_lookup_x
```

perf – hierarchical profile report

```
bash ~ $ perf report -q --stdio --call-graph=none --hierarchy --percent-limit 1 --comm root.exe
# comm: root.exe
    92.93%        root.exe
       47.60%        [kernel.kallsyms]
           2.63%        [k] unmap_page_range
           1.89%        [k] __mod_lruvec_state
           1.78%        [k] srso_untrain_ret
           1.77%        [k] srso_return_thunk
           1.53%        [k] trace_hardirqs_off
           1.32%        [k] __lruvec_stat_mod_folio
           1.31%        [k] clear_page_rep
           1.10%        [k] preempt_count_add
           1.04%        [k] __mod_memcg_lruvec_state
       26.75%        libCling.so.6.30.06
           4.11%        [.] llvm::SimpleBitstreamCursor::Read
           2.27%        [.] llvm::BitstreamCursor::readRecord
           1.26%        [.] llvm::StringMapImpl::LookupBucketFor
        9.10%        libz.so.1.3.1
           5.85%        [.] inflate_fast
           1.43%        [.] adler32_z
        4.56%        libc.so.6
                        no entry >= 1.00%

        2.96%        ld-linux-x86-64.so.2
           1.31%        [.] _dl_lookup_symbol_x
        1.12%        libCore.so.6.30.06
                        no entry >= 1.00%
```

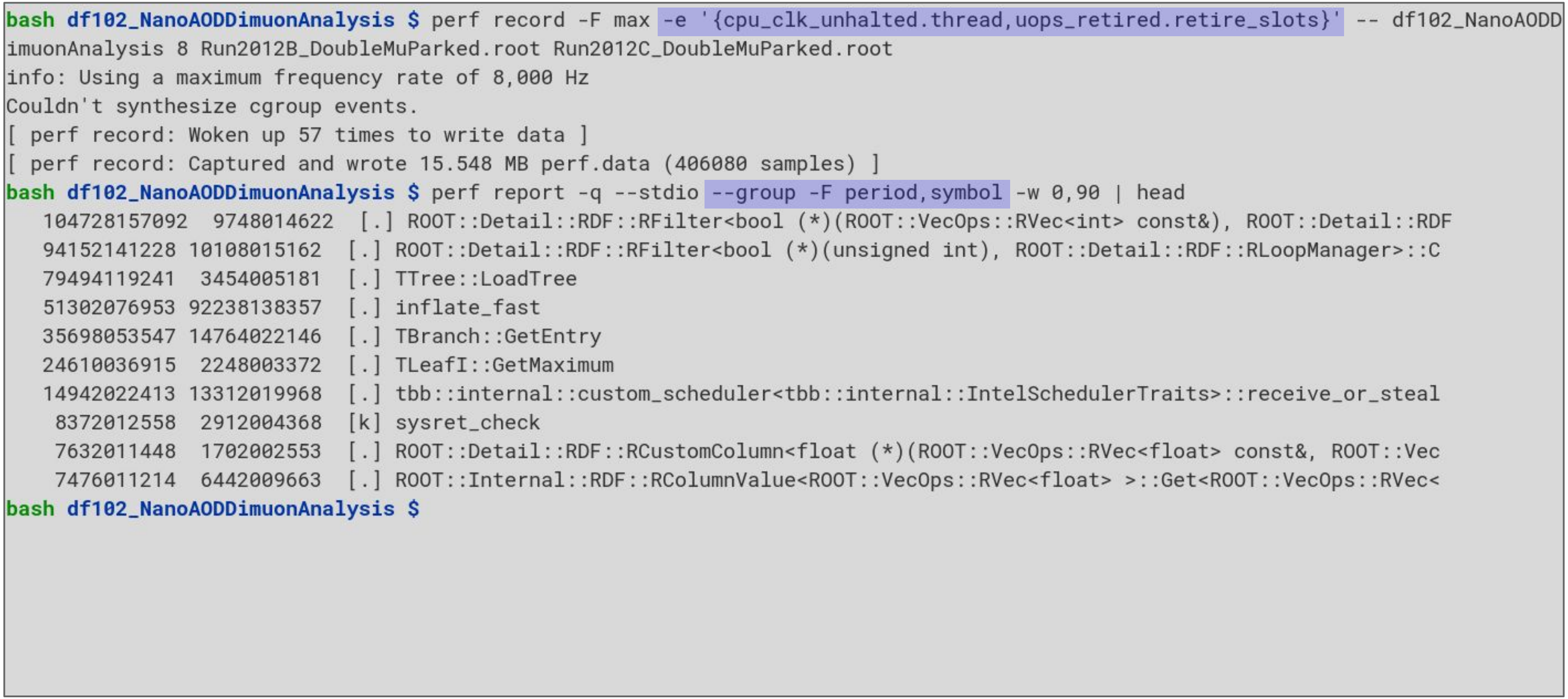
perf – pre-packaged metrics (Intel CPU)

```
bash ~ $ perf list metrics

Metrics:

  Backend_Bound
       [This category represents fraction of slots where no uops are delivered due to a lack of required resources for accepting new uops in the Backend]
  Bad_Speculation
       [This category represents fraction of slots wasted due to incorrect speculations]
  BpTB
       [Branch instructions per taken branch]
  CLKS
       [Per-Logical Processor actual clocks when the Logical Processor is active]
  CPI
       [Cycles Per Instruction (per Logical Processor)]
  CPU_Utilization
       [Average CPU Utilization]
  CoreIPC
       [Instructions Per Cycle (per physical core)]
  Frontend_Bound
       [This category represents fraction of slots where the processor's Frontend undersupplies its Backend]
  ILP
       [Instruction-Level-Parallelism (average number of uops executed when there is at least 1 uop executed)]
  IPC
       [Instructions Per Cycle (per Logical Processor)]
  Instructions
       [Total number of retired Instructions]
  IpB
       [Instructions per Branch (lower number means higher occurance rate)]
  IpCall
       [Instruction per (near) call (lower number means higher occurance rate)]
  IpL
       [Instructions per Load (lower number means higher occurance rate)]
```

perf – pre-packaged metrics (Intel CPU)

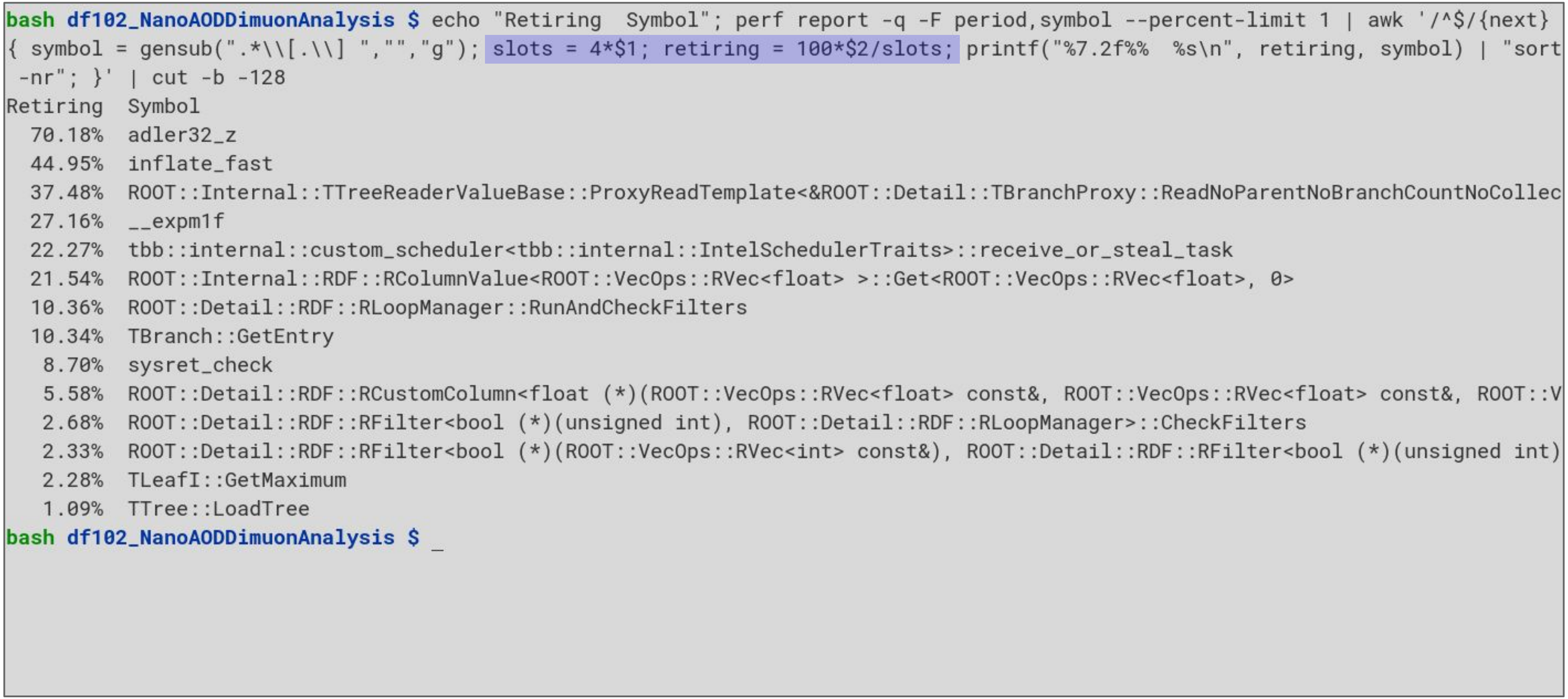
```
bash ~ $ perf stat -M Frontend_Bound,Backend_Bound,Bad_Speculation,Retiring -- root -l -q


 Performance counter stats for 'root -l -q':

         535853293      cycles
                                                  #     0.32  Frontend_Bound              (50.07%)
         676507752      idq_uops_not_delivered.core                                       (50.07%)
         803157447      uops_issued.any           #     0.10  Bad_Speculation
                                                  #     0.28  Backend_Bound               (49.93%)
         540449552      cycles
                                                  #     0.31  Retiring                    (49.93%)
         676523326      idq_uops_not_delivered.core                                       (49.93%)
          19393734      int_misc.recovery_cycles                                          (49.93%)
         667220596      uops_retired.retire_slots                                         (49.93%)

       0.243072802 seconds time elapsed

       0.158384000 seconds user
       0.088028000 seconds sys

bash ~ $
```

Example – using perf + awk to get percent retiring



```
bash df102_NanoAODDimuonAnalysis $ perf record -F max -e '{cpu_clk_unhalted.thread,uops_retired.retire_slots}' -- df102_NanoAODD
imuonAnalysis 8 Run2012B_DoubleMuParked.root Run2012C_DoubleMuParked.root
info: Using a maximum frequency rate of 8,000 Hz
Couldn't synthesize cgroup events.
[ perf record: Woken up 57 times to write data ]
[ perf record: Captured and wrote 15.548 MB perf.data (406080 samples) ]
bash df102_NanoAODDimuonAnalysis $ perf report -q --stdio --group -F period,symbol -w 0,90 | head
   104728157092  9748014622  [.] ROOT::Detail::RDF::RFilter<bool (*)(ROOT::VecOps::RVec<int> const&), ROOT::Detail::RDF
   94152141228 10108015162  [.] ROOT::Detail::RDF::RFilter<bool (*)(unsigned int), ROOT::Detail::RDF::RLoopManager>::C
   79494119241  3454005181  [.] TTree::LoadTree
   51302076953 92238138357  [.] inflate_fast
   35698053547 14764022146  [.] TBranch::GetEntry
   24610036915  2248003372  [.] TLeafI::GetMaximum
   14942022413 13312019968  [.] tbb::internal::custom_scheduler<tbb::internal::IntelSchedulerTraits>::receive_or_steal
    8372012558  2912004368  [k] sysret_check
    7632011448  1702002553  [.] ROOT::Detail::RDF::RCustomColumn<float (*)(ROOT::VecOps::RVec<float> const&, ROOT::Vec
    7476011214  6442009663  [.] ROOT::Internal::RDF::RColumnValue<ROOT::VecOps::RVec<float> >::Get<ROOT::VecOps::RVec<
bash df102_NanoAODDimuonAnalysis $
```

Example – using perf + awk to get percent retiring



```
bash df102_NanoAODDimuonAnalysis $ echo "Retiring  Symbol"; perf report -q -F period,symbol --percent-limit 1 | awk '/^$/{next}
{ symbol = gensub(".*\\[.\\] ","","g"); slots = 4*$1; retiring = 100*$2/slots; printf("%7.2f%%  %s\n", retiring, symbol) | "sort
 -nr"; }' | cut -b -128
Retiring  Symbol
  70.18%  adler32_z
  44.95%  inflate_fast
  37.48%  ROOT::Internal::TTreeReaderValueBase::ProxyReadTemplate<&ROOT::Detail::TBranchProxy::ReadNoParentNoBranchCountNoCollec
  27.16%  __expm1f
  22.27%  tbb::internal::custom_scheduler<tbb::internal::IntelSchedulerTraits>::receive_or_steal_task
  21.54%  ROOT::Internal::RDF::RColumnValue<ROOT::VecOps::RVec<float> >::Get<ROOT::VecOps::RVec<float>, 0>
  10.36%  ROOT::Detail::RDF::RLoopManager::RunAndCheckFilters
  10.34%  TBranch::GetEntry
   8.70%  sysret_check
   5.58%  ROOT::Detail::RDF::RCustomColumn<float (*)(ROOT::VecOps::RVec<float> const&, ROOT::VecOps::RVec<float> const&, ROOT::V
   2.68%  ROOT::Detail::RDF::RFilter<bool (*)(unsigned int), ROOT::Detail::RDF::RLoopManager>::CheckFilters
   2.33%  ROOT::Detail::RDF::RFilter<bool (*)(ROOT::VecOps::RVec<int> const&), ROOT::Detail::RDF::RFilter<bool (*)(unsigned int)
   2.28%  TLeafI::GetMaximum
   1.09%  TTree::LoadTree
bash df102_NanoAODDimuonAnalysis $ _
```

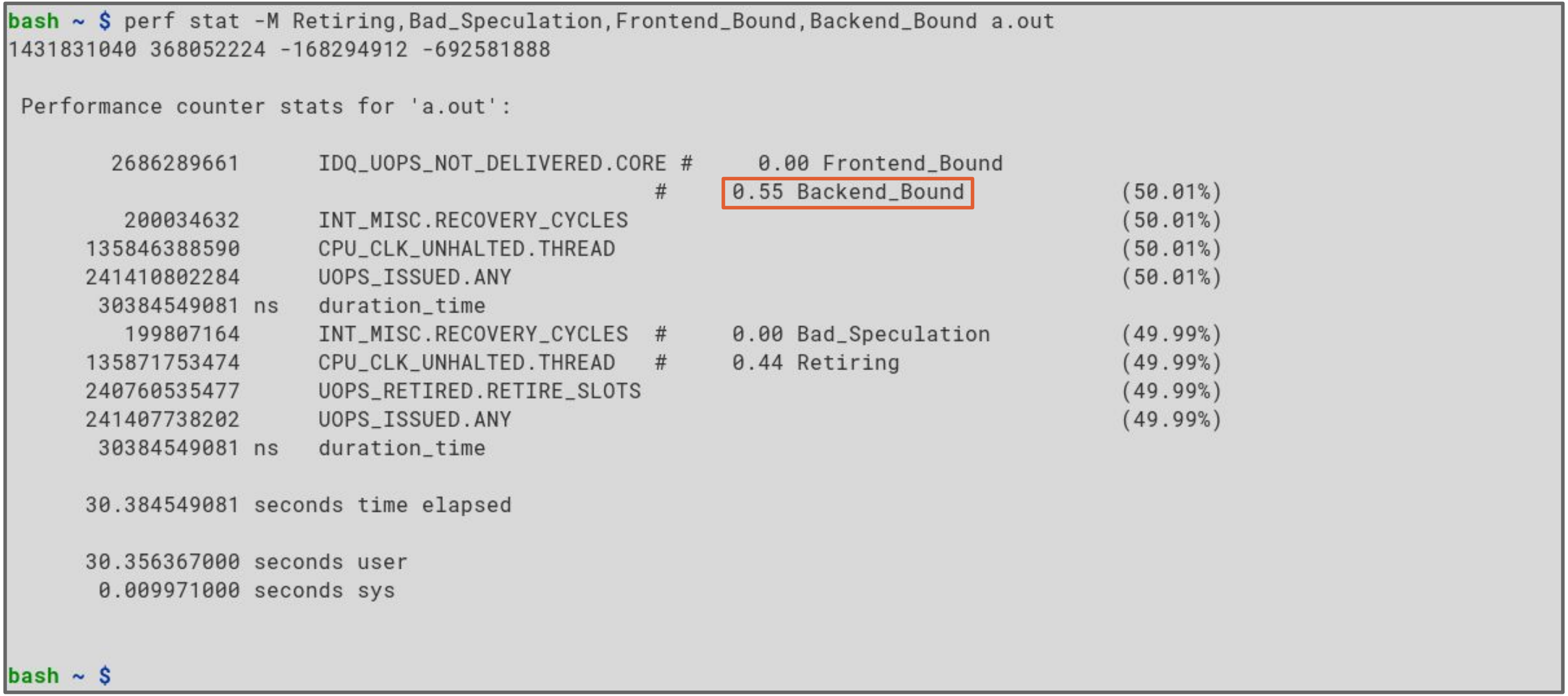
Matrix Multiplication

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

// This version has minor modifications applied, the
// original version is linked at the bottom of the slide

#define SIZE 1024
#define LENGTH 32

int **mkmatrix(int rows, int cols);
void zeromatrix(int rows, int cols, int **m);
void freematrix(int rows, int **m);

int **mmult(int rows, int cols,
            int **m1, int **m2, int **m3) {
    int i, j, k;

    for (i=0; i<rows; i++) {
        for (j=0; j<cols; j++) {
            m3[i][j] = 0;
            for (k=0; k<cols; k++)
                m3[i][j] += m1[i][k] * m2[k][j];
        }
    }
    return(m3);
}
```

https://github.com/llvm-mirror/test-suite/blob/master/SingleSource/Benchmarks/Shootout/matrix.c

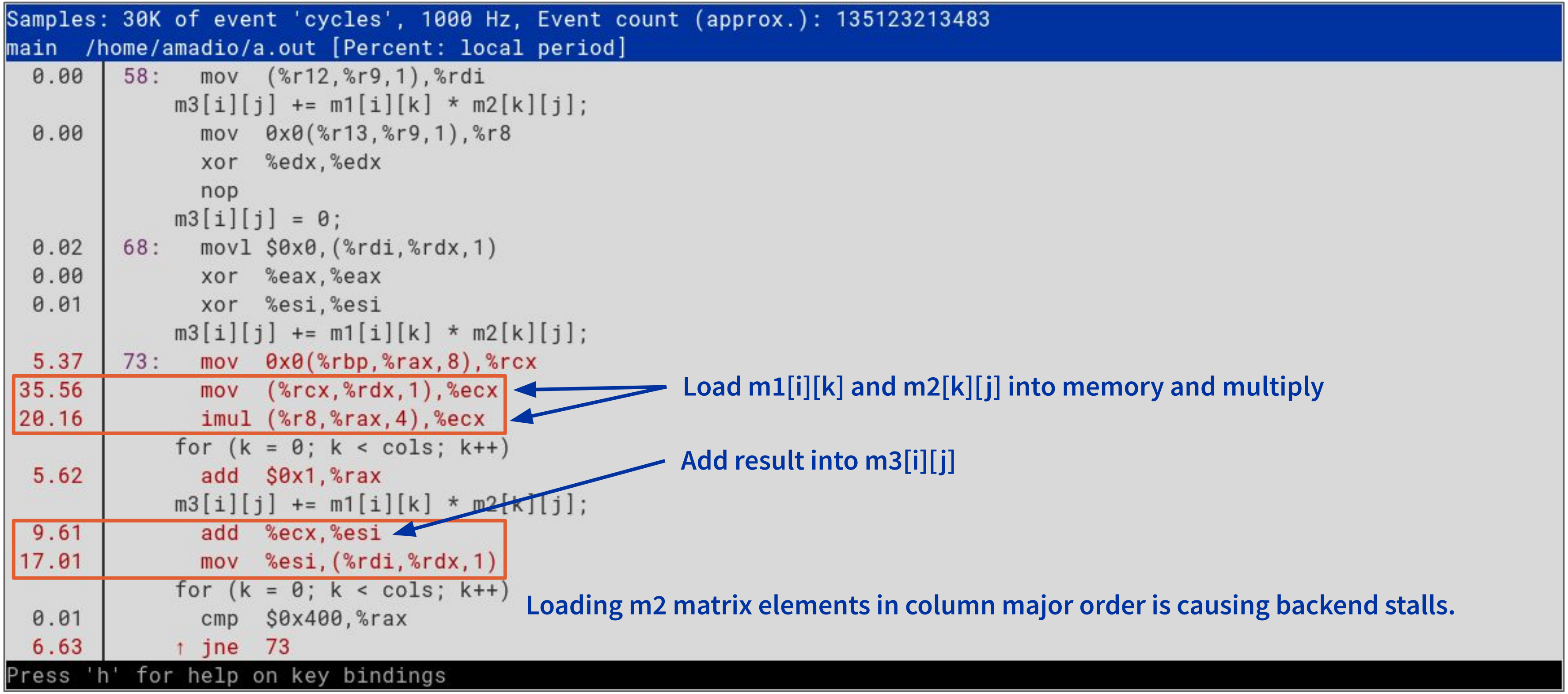
```c
int main(int argc, char *argv[]) {

    int i, n = ((argc == 2) ? atoi(argv[1]) : LENGTH);

    int **m1 = mkmatrix(SIZE, SIZE);
    int **m2 = mkmatrix(SIZE, SIZE);
    int **mm = mkmatrix(SIZE, SIZE);

    zeromatrix(SIZE, SIZE, mm);

    for (i=0; i<n; i++)
      mm = mmult(SIZE, SIZE, m1, m2, mm);

    printf("%d %d %d %d\n",
           mm[0][0], mm[2][3], mm[3][2], mm[4][4]);

    freematrix(SIZE, m1);
    freematrix(SIZE, m2);
    freematrix(SIZE, mm);
    return(0);
}
```

Simple Top-Down Analysis with perf



```
bash ~ $ perf stat -M Retiring,Bad_Speculation,Frontend_Bound,Backend_Bound a.out
1431831040 368052224 -168294912 -692581888

 Performance counter stats for 'a.out':

        2686289661      IDQ_UOPS_NOT_DELIVERED.CORE #     0.00 Frontend_Bound
                                                  #     0.55 Backend_Bound            (50.01%)
         200034632      INT_MISC.RECOVERY_CYCLES                                      (50.01%)
      135846388590      CPU_CLK_UNHALTED.THREAD                                       (50.01%)
      241410802284      UOPS_ISSUED.ANY                                               (50.01%)
       30384549081 ns   duration_time
         199807164      INT_MISC.RECOVERY_CYCLES  #     0.00 Bad_Speculation          (49.99%)
      135871753474      CPU_CLK_UNHALTED.THREAD   #     0.44 Retiring                 (49.99%)
      240760535477      UOPS_RETIRED.RETIRE_SLOTS                                     (49.99%)
      241407738202      UOPS_ISSUED.ANY                                               (49.99%)
       30384549081 ns   duration_time

      30.384549081 seconds time elapsed

      30.356367000 seconds user
       0.009971000 seconds sys


bash ~ $
```

Annotated Source



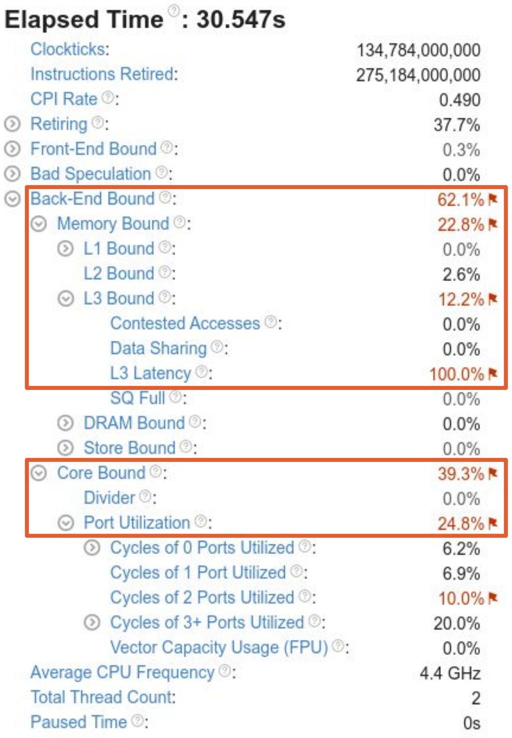
```
Samples: 30K of event 'cycles', 1000 Hz, Event count (approx.): 135123213483
main  /home/amadio/a.out [Percent: local period]
  0.00 │ 58:   mov   (%r12,%r9,1),%rdi
       │     m3[i][j] += m1[i][k] * m2[k][j];
  0.00 │       mov   0x0(%r13,%r9,1),%r8
       │       xor   %edx,%edx
       │       nop
       │     m3[i][j] = 0;
  0.02 │ 68:   movl  $0x0,(%rdi,%rdx,1)
  0.00 │       xor   %eax,%eax
  0.01 │       xor   %esi,%esi
       │     m3[i][j] += m1[i][k] * m2[k][j];
  5.37 │ 73:   mov   0x0(%rbp,%rax,8),%rcx
 35.56 │       mov   (%rcx,%rdx,1),%ecx
 20.16 │       imul  (%r8,%rax,4),%ecx
       │     for (k = 0; k < cols; k++)
  5.62 │       add   $0x1,%rax
       │     m3[i][j] += m1[i][k] * m2[k][j];
  9.61 │       add   %ecx,%esi
 17.01 │       mov   %esi,(%rdi,%rdx,1)
       │     for (k = 0; k < cols; k++)
  0.01 │       cmp   $0x400,%rax
  6.63 │     ↑ jne   73
Press 'h' for help on key bindings
```

Load m1[i][k] and m2[k][j] into memory and multiply

Add result into m3[i][j]

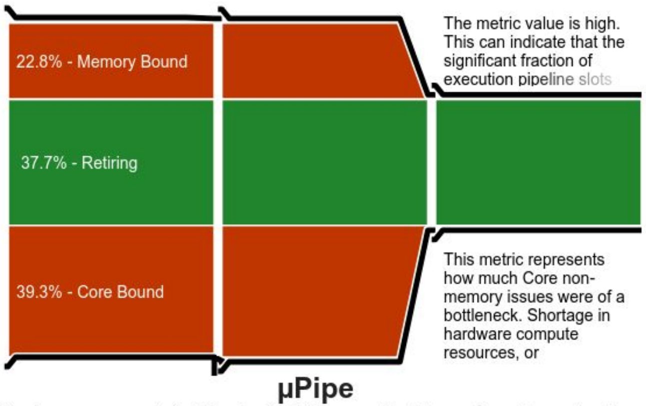
Loading m2 matrix elements in column major order is causing backend stalls.

Top-Down Analysis with Intel VTune Profiler

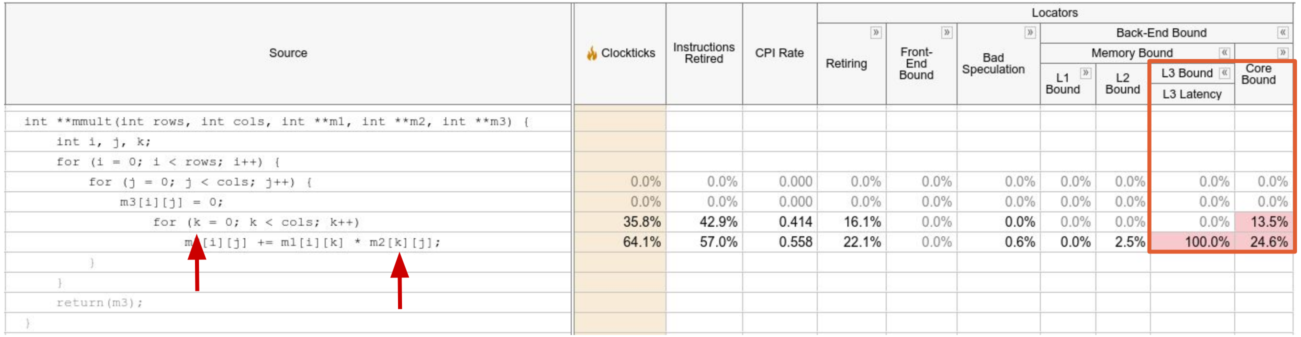


Elapsed Time: 30.547s

Clockticks:	134,784,000,000
Instructions Retired:	275,184,000,000
CPI Rate:	0.490
Retiring:	37.7%
Front-End Bound:	0.3%
Bad Speculation:	0.0%
Back-End Bound:	62.1%
Memory Bound:	22.8%
L1 Bound:	0.0%
L2 Bound:	2.6%
L3 Bound:	12.2%
Contested Accesses:	0.0%
Data Sharing:	0.0%
L3 Latency:	100.0%
SQ Full:	0.0%
DRAM Bound:	0.0%
Store Bound:	0.0%
Core Bound:	39.3%
Divider:	0.0%
Port Utilization:	24.8%
Cycles of 0 Ports Utilized:	6.2%
Cycles of 1 Port Utilized:	6.9%
Cycles of 2 Ports Utilized:	10.0%
Cycles of 3+ Ports Utilized:	20.0%
Vector Capacity Usage (FPU):	0.0%
Average CPU Frequency:	4.4 GHz
Total Thread Count:	2
Paused Time:	0s

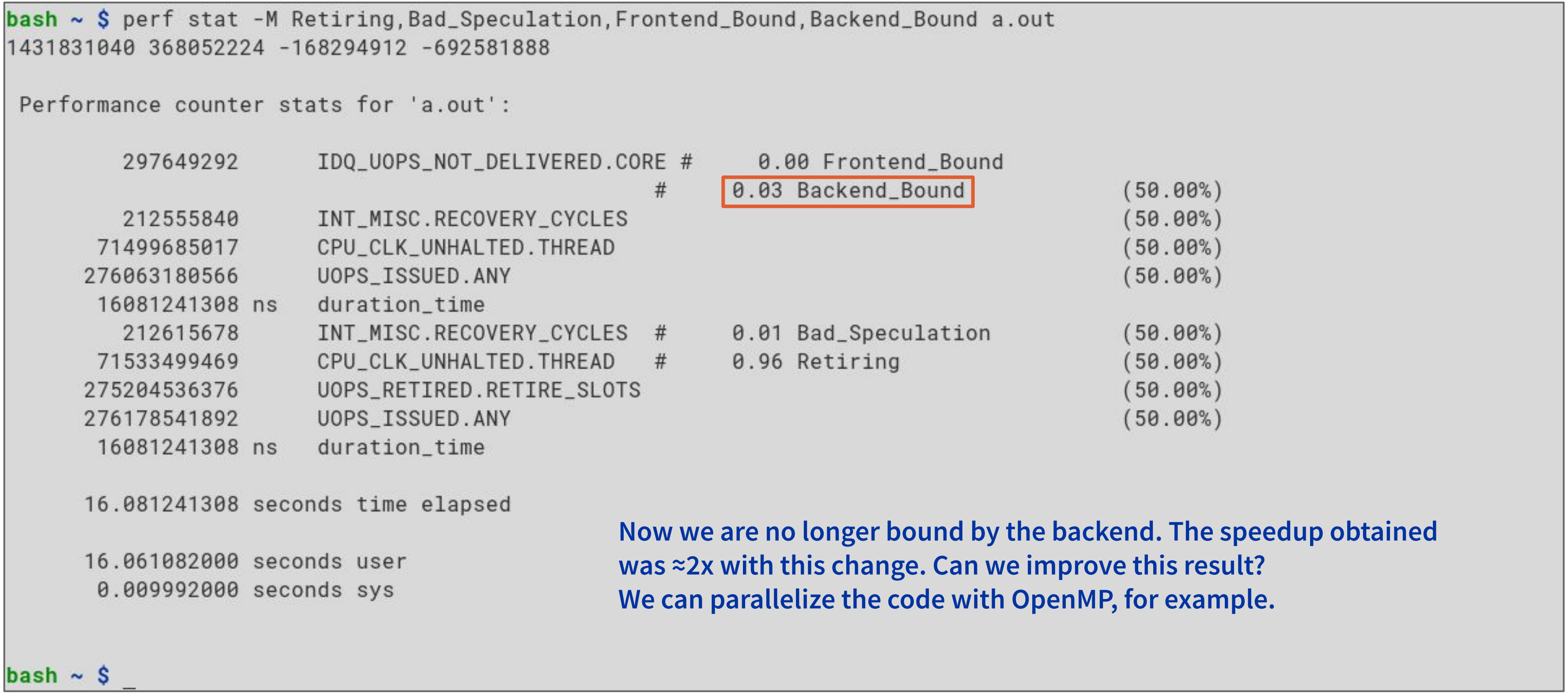


As shown by the red arrows, the loop is being performed in column major order, which in C/C++ is not optimal, because the memory layout is row major. Therefore, we need to perform a loop inversion for the indices `j` and `k` to improve performance.



Source	Clockticks	Instructions Retired	CPI Rate	Locators						
				Retiring	Front-End Bound	Bad Speculation	Back-End Bound			
							Memory Bound			Core Bound
							L1 Bound	L2 Bound	L3 Bound	
									L3 Latency	
`int **mmult(int rows, int cols, int **m1, int **m2, int **m3) {`										
`int i, j, k;`										
`for (i = 0; i < rows; i++) {`										
`for (j = 0; j < cols; j++) {`	0.0%	0.0%	0.000	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
`m3[i][j] = 0;`	0.0%	0.0%	0.000	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
`for (k = 0; k < cols; k++)`	35.8%	42.9%	0.414	16.1%	0.0%	0.0%	0.0%	0.0%	0.0%	13.5%
`m [i][j] += m1[i][k] * m2[k][j];`	64.1%	57.0%	0.558	22.1%	0.0%	0.6%	0.0%	2.5%	100.0%	24.6%
`}`										
`}`										
`return(m3);`										
`}`										

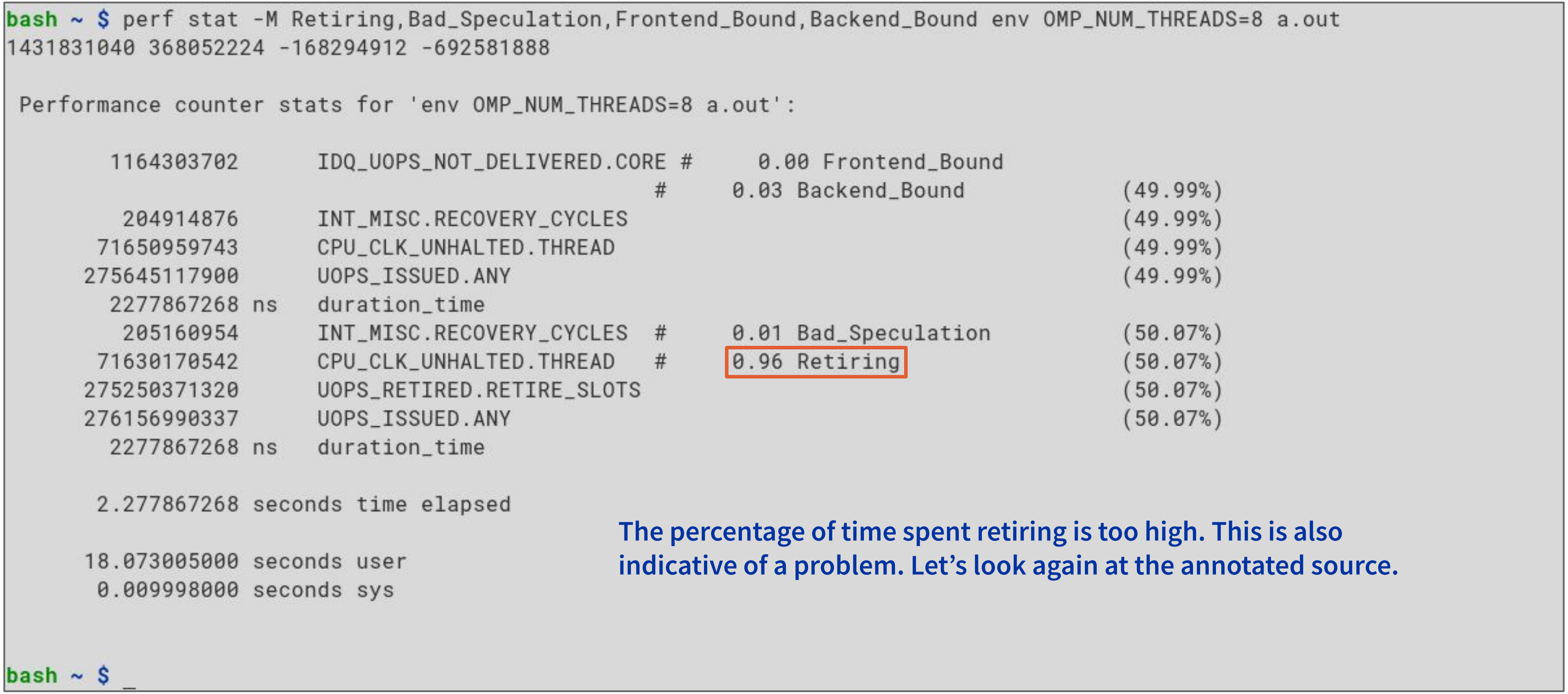
Loop inversion solves the problem



```
bash ~ $ perf stat -M Retiring,Bad_Speculation,Frontend_Bound,Backend_Bound a.out
1431831040 368052224 -168294912 -692581888

 Performance counter stats for 'a.out':

        297649292      IDQ_UOPS_NOT_DELIVERED.CORE #     0.00 Frontend_Bound
                                                  #     0.03 Backend_Bound            (50.00%)
        212555840      INT_MISC.RECOVERY_CYCLES                                       (50.00%)
      71499685017      CPU_CLK_UNHALTED.THREAD                                        (50.00%)
     276063180566      UOPS_ISSUED.ANY                                                (50.00%)
      16081241308 ns   duration_time
        212615678      INT_MISC.RECOVERY_CYCLES   #     0.01 Bad_Speculation          (50.00%)
      71533499469      CPU_CLK_UNHALTED.THREAD    #     0.96 Retiring                 (50.00%)
     275204536376      UOPS_RETIRED.RETIRE_SLOTS                                      (50.00%)
     276178541892      UOPS_ISSUED.ANY                                                (50.00%)
      16081241308 ns   duration_time

     16.081241308 seconds time elapsed

     16.061082000 seconds user
      0.009992000 seconds sys


bash ~ $ _
```

Now we are no longer bound by the backend. The speedup obtained was ≈2x with this change. Can we improve this result?
We can parallelize the code with OpenMP, for example.

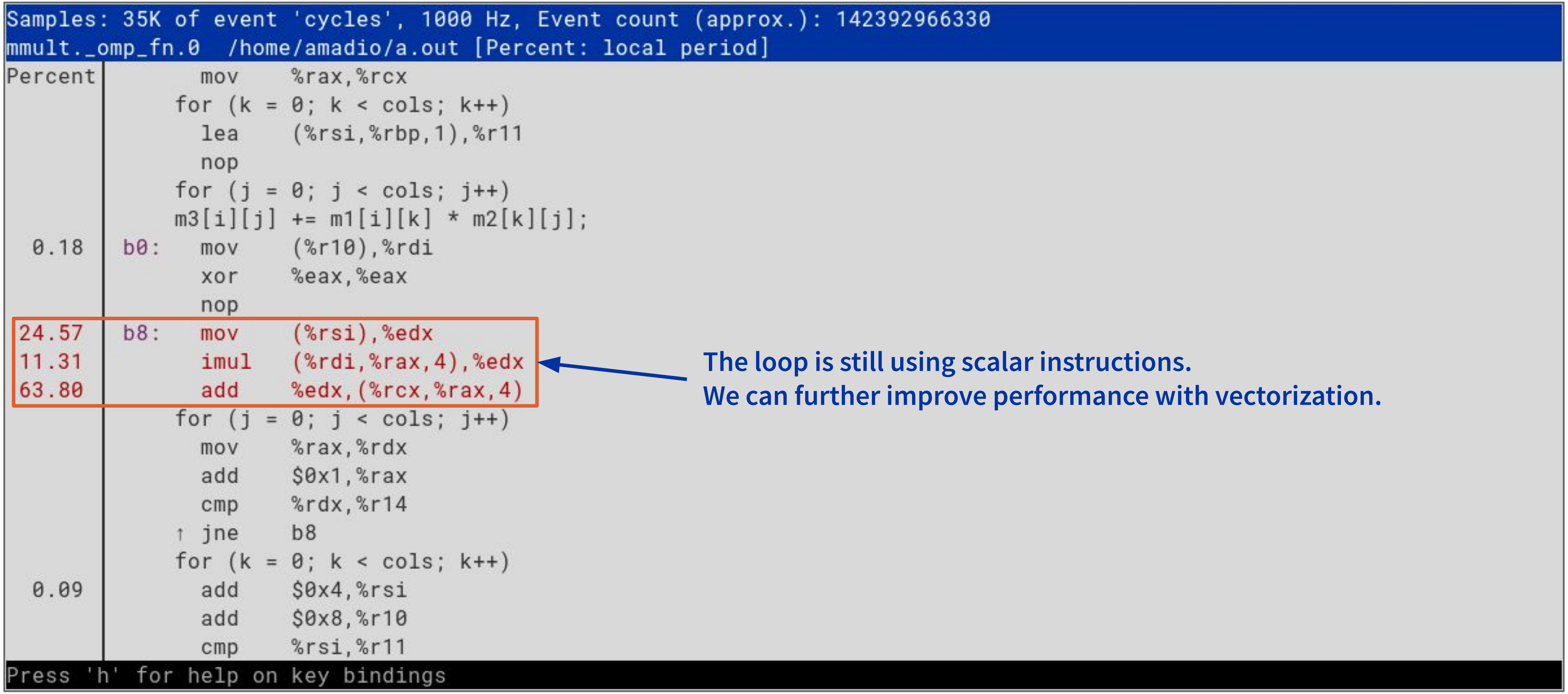
Parallel code with OpenMP gains more performance



```
bash ~ $ perf stat -M Retiring,Bad_Speculation,Frontend_Bound,Backend_Bound env OMP_NUM_THREADS=8 a.out
1431831040 368052224 -168294912 -692581888

 Performance counter stats for 'env OMP_NUM_THREADS=8 a.out':

        1164303702      IDQ_UOPS_NOT_DELIVERED.CORE #     0.00 Frontend_Bound
                                                  #     0.03 Backend_Bound            (49.99%)
         204914876      INT_MISC.RECOVERY_CYCLES                                     (49.99%)
       71650959743      CPU_CLK_UNHALTED.THREAD                                      (49.99%)
      275645117900      UOPS_ISSUED.ANY                                              (49.99%)
        2277867268 ns   duration_time
         205160954      INT_MISC.RECOVERY_CYCLES   #     0.01 Bad_Speculation          (50.07%)
       71630170542      CPU_CLK_UNHALTED.THREAD    #     0.96 Retiring                 (50.07%)
      275250371320      UOPS_RETIRED.RETIRE_SLOTS                                    (50.07%)
      276156990337      UOPS_ISSUED.ANY                                              (50.07%)
        2277867268 ns   duration_time

       2.277867268 seconds time elapsed

      18.073005000 seconds user
       0.009998000 seconds sys


bash ~ $ _
```

The percentage of time spent retiring is too high. This is also indicative of a problem. Let's look again at the annotated source.

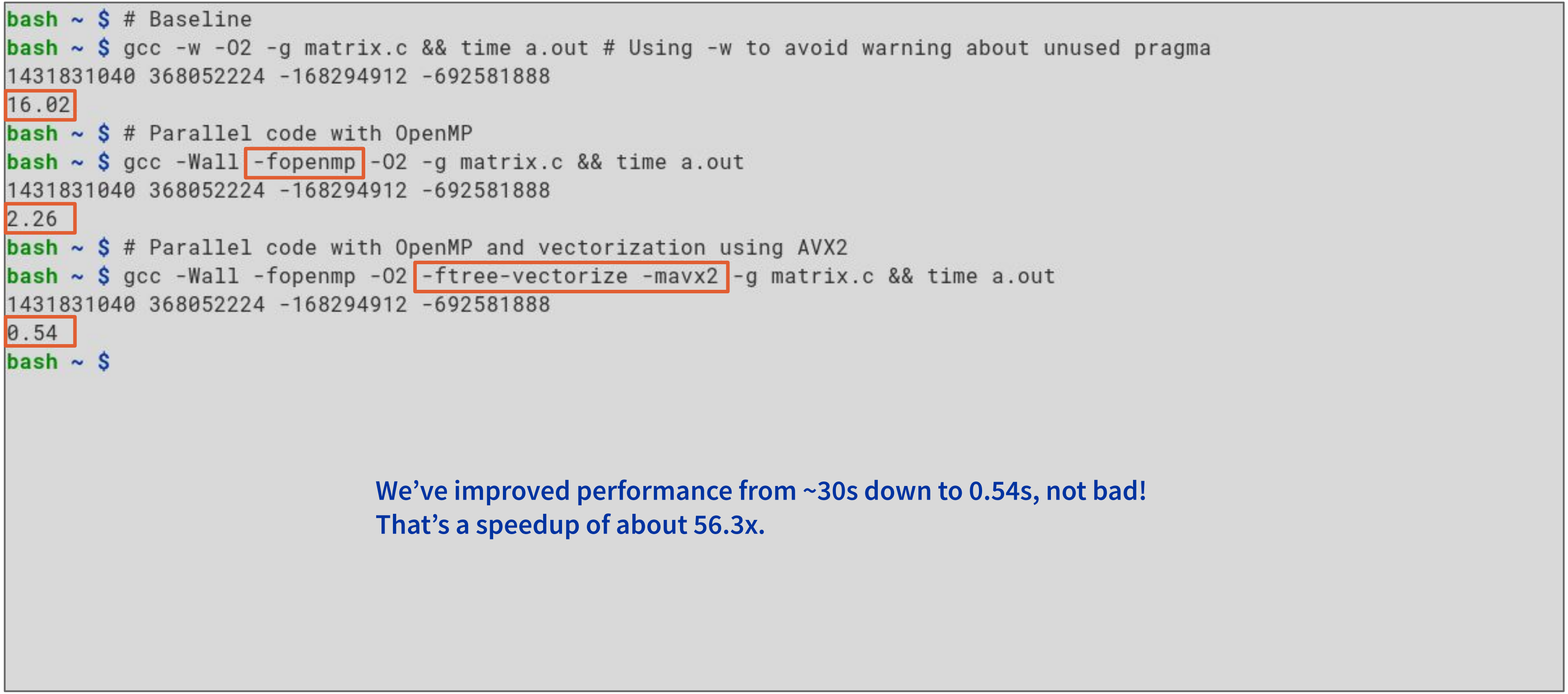
Annotated Source with perf annotate



```
Samples: 35K of event 'cycles', 1000 Hz, Event count (approx.): 142392966330
mmult._omp_fn.0  /home/amadio/a.out [Percent: local period]
Percent│       mov    %rax,%rcx
       │     for (k = 0; k < cols; k++)
       │       lea    (%rsi,%rbp,1),%r11
       │       nop
       │     for (j = 0; j < cols; j++)
       │     m3[i][j] += m1[i][k] * m2[k][j];
  0.18 │b0:    mov    (%r10),%rdi
       │       xor    %eax,%eax
       │       nop
 24.57 │b8:    mov    (%rsi),%edx
 11.31 │       imul   (%rdi,%rax,4),%edx
 63.80 │       add    %edx,(%rcx,%rax,4)
       │     for (j = 0; j < cols; j++)
       │       mov    %rax,%rdx
       │       add    $0x1,%rax
       │       cmp    %rdx,%r14
       │     ↑ jne    b8
       │     for (k = 0; k < cols; k++)
  0.09 │       add    $0x4,%rsi
       │       add    $0x8,%r10
       │       cmp    %rsi,%r11
Press 'h' for help on key bindings
```

The loop is still using scalar instructions.
We can further improve performance with vectorization.

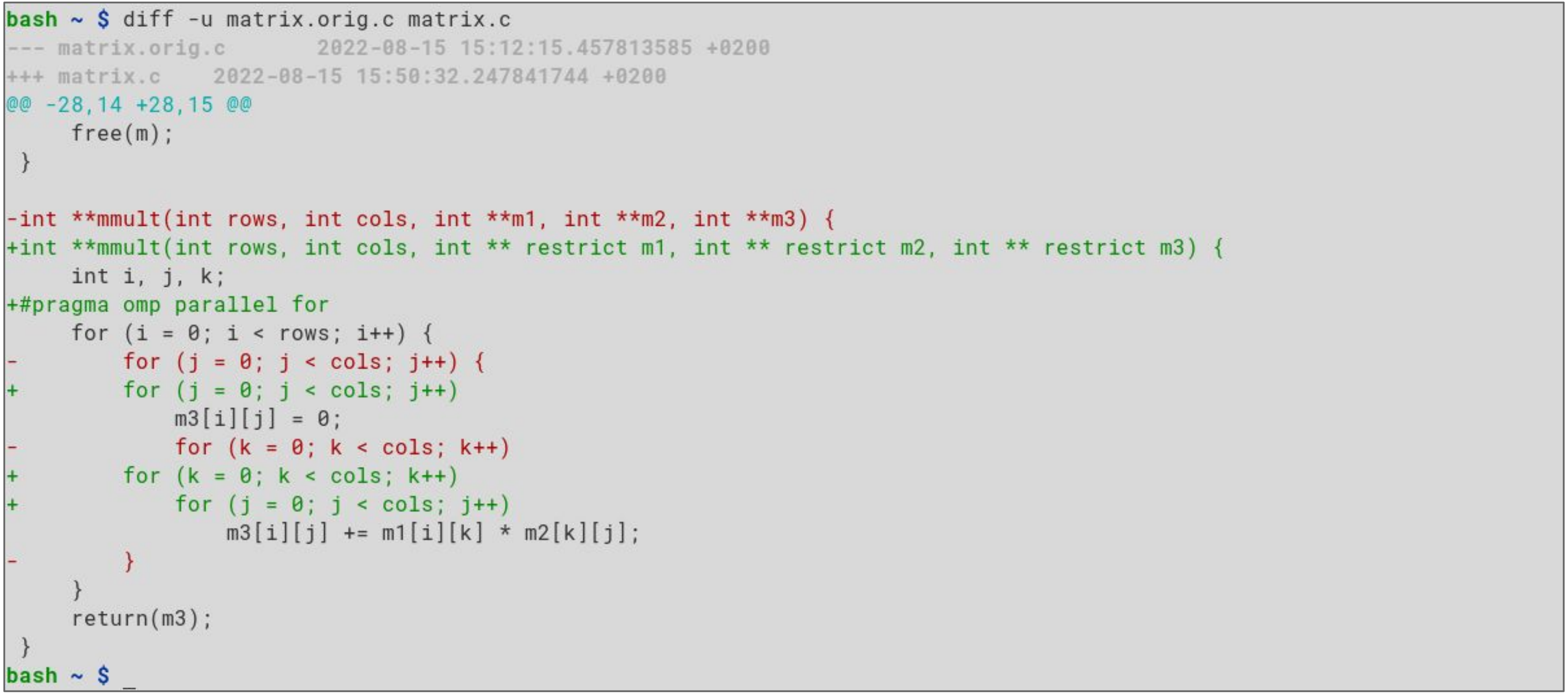
Vectorization significantly improves performance



```
bash ~ $ # Baseline
bash ~ $ gcc -w -O2 -g matrix.c && time a.out # Using -w to avoid warning about unused pragma
1431831040 368052224 -168294912 -692581888
16.02
bash ~ $ # Parallel code with OpenMP
bash ~ $ gcc -Wall -fopenmp -O2 -g matrix.c && time a.out
1431831040 368052224 -168294912 -692581888
2.26
bash ~ $ # Parallel code with OpenMP and vectorization using AVX2
bash ~ $ gcc -Wall -fopenmp -O2 -ftree-vectorize -mavx2 -g matrix.c && time a.out
1431831040 368052224 -168294912 -692581888
0.54
bash ~ $
```

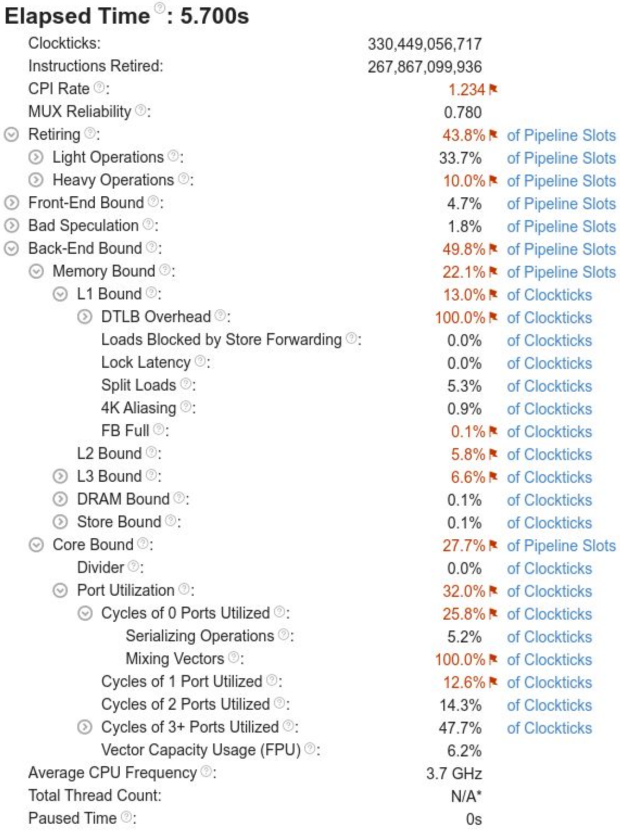
We’ve improved performance from ~30s down to 0.54s, not bad!
That’s a speedup of about 56.3x.

Comparison between initial and final versions

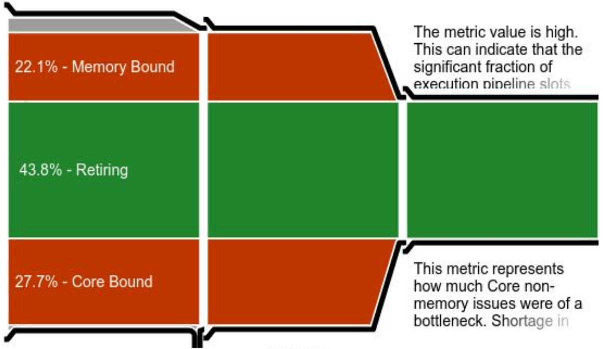


```
bash ~ $ diff -u matrix.orig.c matrix.c
--- matrix.orig.c       2022-08-15 15:12:15.457813585 +0200
+++ matrix.c    2022-08-15 15:50:32.247841744 +0200
@@ -28,14 +28,15 @@
     free(m);
 }

-int **mmult(int rows, int cols, int **m1, int **m2, int **m3) {
+int **mmult(int rows, int cols, int ** restrict m1, int ** restrict m2, int ** restrict m3) {
     int i, j, k;
+#pragma omp parallel for
     for (i = 0; i < rows; i++) {
-        for (j = 0; j < cols; j++) {
+        for (j = 0; j < cols; j++)
             m3[i][j] = 0;
-            for (k = 0; k < cols; k++)
+        for (k = 0; k < cols; k++)
+            for (j = 0; j < cols; j++)
                 m3[i][j] += m1[i][k] * m2[k][j];
-        }
     }
     return(m3);
 }
bash ~ $ _
```

Final performance summary in VTune (10x runtime)



```
Elapsed Time: 5.700s
  Clockticks:                                    330,449,056,717
  Instructions Retired:                          267,867,099,936
  CPI Rate:                                                1.234 ⚑
  MUX Reliability:                                         0.780
  Retiring:                                               43.8% ⚑  of Pipeline Slots
    Light Operations:                                     33.7%    of Pipeline Slots
    Heavy Operations:                                     10.0% ⚑  of Pipeline Slots
  Front-End Bound:                                         4.7%    of Pipeline Slots
  Bad Speculation:                                         1.8%    of Pipeline Slots
  Back-End Bound:                                         49.8% ⚑  of Pipeline Slots
    Memory Bound:                                         22.1% ⚑  of Pipeline Slots
      L1 Bound:                                           13.0% ⚑  of Clockticks
        DTLB Overhead:                                   100.0% ⚑  of Clockticks
        Loads Blocked by Store Forwarding:                 0.0%    of Clockticks
        Lock Latency:                                      0.0%    of Clockticks
        Split Loads:                                       5.3%    of Clockticks
        4K Aliasing:                                       0.9%    of Clockticks
        FB Full:                                           0.1% ⚑  of Clockticks
      L2 Bound:                                            5.8% ⚑  of Clockticks
      L3 Bound:                                            6.6% ⚑  of Clockticks
      DRAM Bound:                                          0.1%    of Clockticks
      Store Bound:                                         0.1%    of Clockticks
    Core Bound:                                           27.7% ⚑  of Pipeline Slots
      Divider:                                             0.0%    of Clockticks
      Port Utilization:                                   32.0% ⚑  of Clockticks
        Cycles of 0 Ports Utilized:                       25.8% ⚑  of Clockticks
          Serializing Operations:                          5.2%    of Clockticks
          Mixing Vectors:                                100.0% ⚑  of Clockticks
        Cycles of 1 Port Utilized:                        12.6% ⚑  of Clockticks
        Cycles of 2 Ports Utilized:                       14.3%    of Clockticks
        Cycles of 3+ Ports Utilized:                      47.7%    of Clockticks
        Vector Capacity Usage (FPU):                       6.2%
  Average CPU Frequency:                                 3.7 GHz
  Total Thread Count:                                       N/A*
  Paused Time:                                                0s
```



The metric value is high. This can indicate that the significant fraction of execution pipeline slots

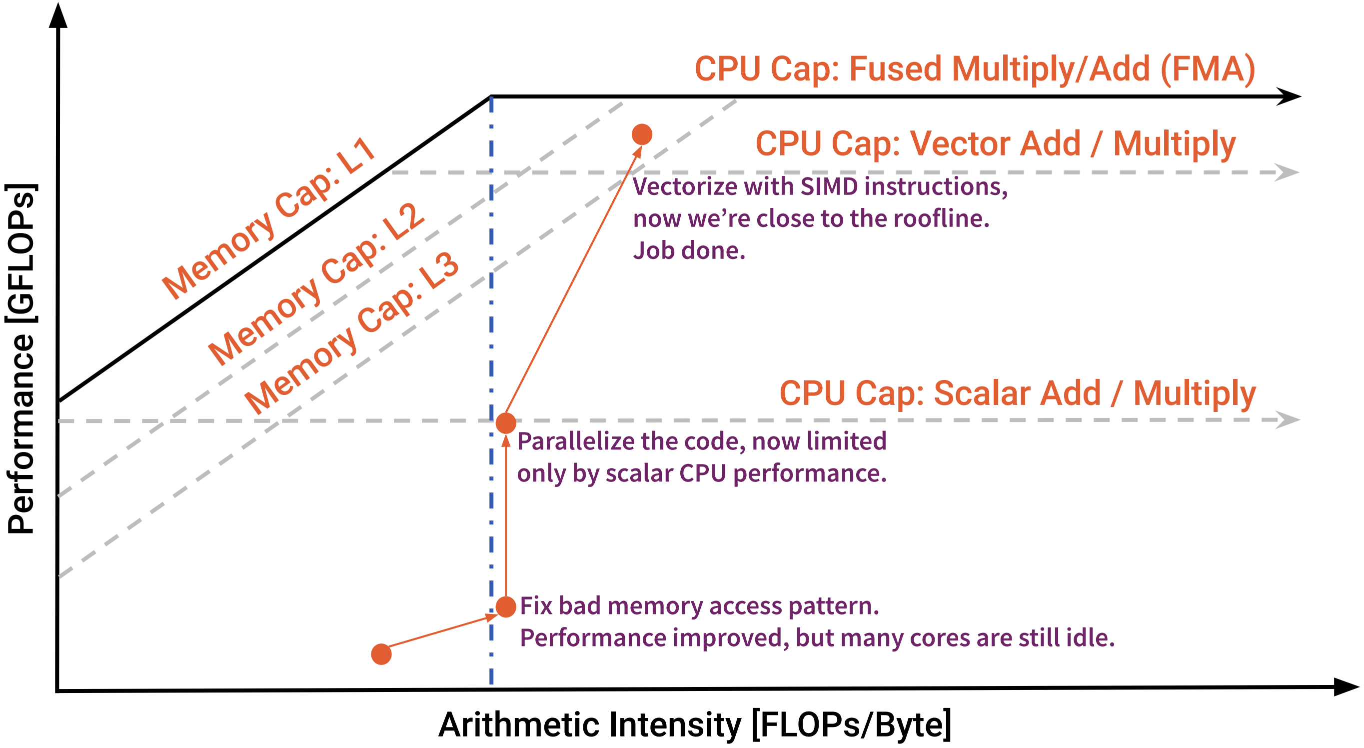
This metric represents how much Core non-memory issues were of a bottleneck. Shortage in

µPipe

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

Category	Client/Desktop Application	Server/Database Distributed Application	High Performance Computing (HPC) Application
Retiring	20 – 50%	10 – 30%	30 – 70%
Back-End Bound	20 – 40%	20 – 60%	20 – 40%
Front-End Bound	5 – 10%	10 – 25%	5 – 10%
Bad Speculation	5 – 10%	5 – 10%	1 – 5%

Matrix Multiplication Roofline Performance



Vectorize with SIMD instructions,
now we're close to the roofline.
Job done.

Parallelize the code, now limited
only by scalar CPU performance.

Fix bad memory access pattern.
Performance improved, but many cores are still idle.

https://www.intel.com/content/www/us/en/developer/articles/guide/intel-advisor-roofline.html