PERF & eBPF – TUTORIAL

PANACEA *of* PERFORMANCE PROFILING

Zvonko Kosic

September 20, 2017

CONTENTS

1 INTRODUCTION 4
1.1 Perf Tool 4
1.2 Prerequisites 4
1.2.1 Debug Symbols 4
1.2.1.1 Where are debug symbols saved? 4
1.2.2 Frame Pointer 6
1.2.2.1 Why is this important for *perf*? 7

2 SYSTEM PREREQUISITES 8
2.1 Logical Partition (LPAR) 8
2.2 Ramfs 9

3 FIRST STEPS 10
3.1 Perf List 10
3.2 Live preview 10

4 COUNTING EVENTS 12
4.1 Perf Stat 12
4.1.1 Counting Backlog 13

5 SAMPLING EVENTS 14
5.1 Perf Record 14

6 STRACE INSPIRED 18
6.1 Perf Trace 18

7 SCHEDULING 20
7.1 Perf Sched 20

8 TRACEPOINTS 24
8.1 What is a Tracepoint 24

9 PROBES 26
9.1 Userspace Probing 26
9.1.1 Backlog 28
9.1.2 Heatmap 28
9.2 Kernel Probing 29

10FLAMEGRAPH 32
10.1 Call-Stacks 32
10.2 Perf Flamegraph 33
10.3 Differential Flamegraphs 34
10.4 Java Flamegraph 34

A PERLIMINARY FEATURES 38
A.1 Build top-notch perf 38

1 INTRODUCTION

1.1 PERF TOOL

Perf is a event oriented observability tool that can help to answer specific performance questions and issues. It can be used to trace and probe the kernel and user space. It is part of the Linux kernel an available on any Linux distribution.

Perf can be used to do static tracing, dynamic tracing, record and count software, Performance Monitoring Unit (PMU) and lately also Enhanced Berkeley Packet Filter (EBPF) events.

For each of these event sources there will be examples how to count or record these events and how to interpret them. But first some fundamental topics.

1.2 PREREQUISITES

Before deep diving into performance analysis with *perf* two important questions have to be answered before proceeding. The are two compiler flags that make the life of a performance analyst difficult. Missing debug information and the omitted frame pointer.

Why these two options are critical in the performance analyst point of view is shown in the next two sections.

1.2.1 *Debug Symbols*

A debug symbols is additional information stored in the binary or in another file that expresses information about a programming language construct (variable, function, ...) that is mapped to binary code.

A symbolic debugger can gain access to names of variable or function names from the source code of the binary. Additionally the debugger can inspect variables in memory, step into and step over these language constructs in the source code with this additional information.

1.2.1.1 *Where are debug symbols saved?*

For the sake of simplicity the focus will be on Executable and Linkable Format (ELF) binaries. Every binary has several sections e.g. the `.text` section holds the executable code, `.bss` section holds statically allocated variables. Another interesting section is `.eh_frame` that in the first place held information about exception handling in `C++`. Nowadays this section is used for more then just exceptions handling. Unwinding the stack, back tracing are just a few examples how this section is used additionally.

Beware that even if C-code is compiled without exceptions the `.eh_frame` is still present and *needed* for reasons stated above.

To have a look at the `.eh_frame` section one can use the *readelf* program.

```
# readelf -debug-dump=frames /bin/ls
Contents of the .eh_frame section:

00000000 0000000000000014 00000000 CIE
  Version:               1
  Augmentation:          "zR"
  Code alignment factor: 1
  Data alignment factor: -8
  Return address column: 14
```

DWARF-2 Call Frame Information, additionally one can use `dwarfdump` or `objdump`

```
  Augmentation data:     1b

  DW_CFA_def_cfa: r15 ofs 160
  DW_CFA_nop
  DW_CFA_nop
  DW_CFA_nop

00000018 0000000000000014 0000001c FDE cie=00000000 pc=00000000000656
  DW_CFA_nop
  DW_CFA_nop
  DW_CFA_nop
  DW_CFA_nop
  DW_CFA_nop
  DW_CFA_nop
  DW_CFA_nop
```

The `.eh_frame` has a Common Information Entry (CIE) and for every frame a Frame Description Entry (FDE). With this additional info a debugger can unwind the stack even if a the frame pointer is omitted. See next section for frame pointer issues.

Each section in a binary has additional flags that describe several features of the corresponding section. Among these flags several of the flags are `ALLOC`, `LOAD`, or `READONLY`. The `ALLOC` and `LOAD` flags indicate that this section is allocated and loaded during runtime.

If source code is compiled with the `-g` compiler flag additional sections will be created. The default debugging format nowadays is the Debugging With Attributed Records Format (DWARF) debugging format (dwarfstd.org). Inspecting the binary with debug symbols one can see different DWARF sections that make up the DWARF data. The next table shows five exemplary DWARF sections of the new binary.

Section	Description
.debug_abbrev	Abbreviations used in the .debug_info section
.debug_aranges	Lookup table for mapping addresses to compilation units
.debug_frame	Call frame information
.debug_info	Core DWARF information section
.debug_line	Line number information

TABLE 1.1 Extract of some DWARF sections of a binary

Examining the flags of these section one can notice that they are `READONLY` without the `ALLOC` or `LOAD` flag.

One may ask why such a thorough analysis was done about sections and flags, and the answer is, when source code is compiled with debugging information/symbols these debugging information are not loaded at runtime nor the `.data` or `.text` sections are altered. Debugging information will not alter the performance characteristics of a workload. The downside is, the binary is bigger, hence you need extra disk space to store this information.

To remove these sections one can use the Linux command `strip`. It removes the debug sections stated above and the result is a binary that looks exactly the same as a binary where the `-g` flag was omitted.

Furthermore there is the myth that debug symbols are somehow interwoven with executable code so that the execution is slowed down. One can extract the `.text` section and compare them to see that they are the same independently if the source code was compiled with or without the `-g` flag.

First compile a source file with debug and without debug symbols. Next step is to extract the `.text` section which holds the actual code and build a hash sum.

```
# A is equal to B
# gcc -g -O2 A.c -o A && strip A
# gcc    -O2 A.c -o B
```

```
# readelf -x .text A | md5sum
# readelf -x .text B | md5sum
f53e82c6da4f3c601fbf5c22e74e5729  -
f53e82c6da4f3c601fbf5c22e74e5729  -
```

Just for educational purpose the size of the sections between `A` and `B`, where evidently no important sections are altered through debugging symbols and stripping of the binary is shown in Table 1.2.

Section	`size -A -d ./A`	`size -A -d ./B`
.interp	15	15
.note.ABI-tag	32	32
.note.gnu.build-id	36	36
.gnu.hash	52	52
.dynsym	288	288
.dynstr	167	167
.gnu.version	24	24
.gnu.version_r	32	32
.rela.dyn	216	216
.rela.plt	48	48
.init	64	64
.plt	96	96
.text	464	464
.fini	44	44
.rodata	4	4
.eh_frame_hdr	36	36
.eh_frame	132	132
.init_array	8	8
.fini_array	8	8
.jcr	8	8
.dynamic	496	496
.got	88	88
.data	16	16
.bss	8	8
.comment	52	52

TABLE 1.2 Size of sections in an ELF binary

1.2.2 *Frame Pointer*

Before going into detail what impact the frame pointer has on performance and what issues arise when using *perf*, first a little information how the frame pointer is used.

When a function is called a certain contiguous section of memory is set aside for the program called the stack. The stack saves return values, arguments to the called function and e.g. local variables (depending on the architecture). The stack works the same as the stack known from C or C++. Items are pushed or popped from the stack.

The local variables, values and arguments to the called function are grouped into a stack frame that represents a function call. With every new function call a new stack frame is allocated and the needed data is pushed onto the stack. When the function exists the data is popped off the stack. This stack of frames is called the *call-stack*. The *call-stacks* are later used in 10 FLAMEGRAPH to visualize them.

The stack pointer always points the top of the stack and is used for pushing and popping of elements to the stack. So depending on what is done on the stack the stack pointer is moving around (up and down). For simpler access of variables on the stack usually the frame pointer is used. The frame pointer points to the beginning of the stack, the return address where to jump back after completing the current function and is not manipulated during the execution of a function.

To address a variable on the stack an offset is added to the frame pointer and the variable can be accessed through this pointer. It's an auxiliary pointer to keep addressing simple, but this addressing of things on the stack works with the stack pointer too, but it is more *complicated*.

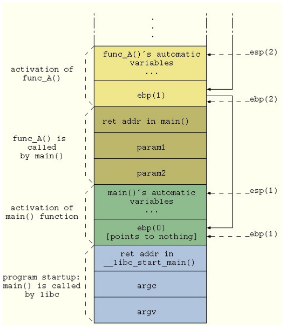


FIGURE 1.1 A sample stack that is compatible with most architectures.

Compiling a source code with optimization (`-O1` is enough) the compiler will omit the frame pointer per default and will be using the stack pointer to access local variables.

Omitting the frame pointer is performance enhancement where theoretically one can save a memory write to a cached memory, a few clock ticks in entry/exit of a function and a general purpose register is freed up. The compiler can utilize the freed register to produce code that is smaller and potentially faster.

The problem here is that a debugger will lose an easy way to generate a stack trace. If frame pointers are not omitted they can be used to walk the stack. The frame pointer is a linked list of stack frames. The debugger might still be able to generate a stack trace (when frame pointers are ommited) from a different source and that's when the `.eh_frame` section comes into play.

The debugger has to implement stack unwinding with the debug information in the *.eh_frame* section that is generated whether less `-g` is provided as a compilation flag or not.

1.2.2.1 *Why is this important for* perf?

Call-stacks are a nice way to see who is called by whom and this information can be saved by *perf* as well and the saved events/samples can be correctly correlated to the specific function calls in the call-stack. Even if the frame pointer is ommitted the call stack has to be unwinded by *perf*, it has to be implemented (architecture specific) in the same manner as for the debugger.

2 SYSTEM PREREQUISITES

2.1 LOGICAL PARTITION (LPAR)

Before using *perf* on a LPAR the LPAR has to be authorized in the Hardware Managment Console (HMC) to use the counters. See the following image an check in the activation profile under security if these checkboxes are activated when *perf* is to be used on that specific LPAR .

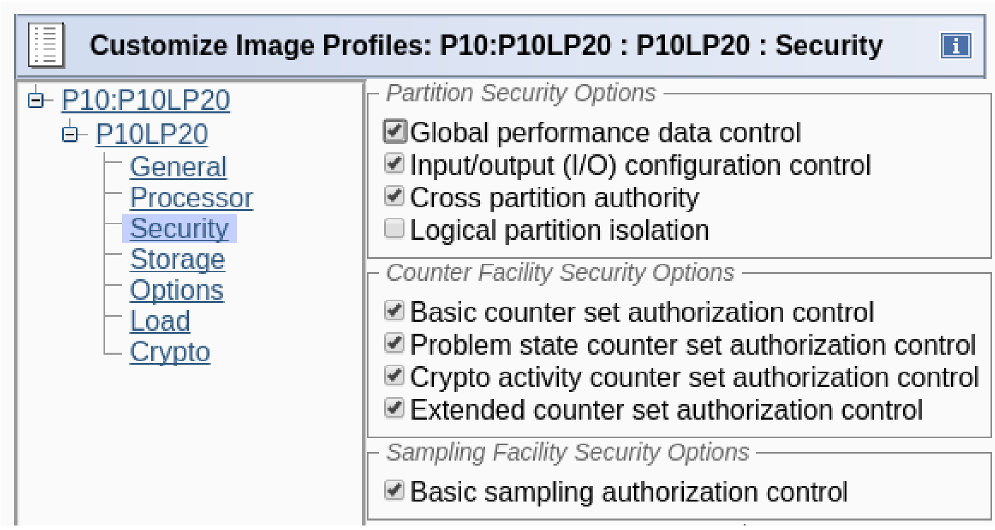


FIGURE 2.1 HMC – Activation Profile

After activating and loading a Linux it is advisable to list the counters for which the LPAR is authorized. Therefor use `lscpumf` from the *s390-tools* package.

```
# lscpumf -c
Perf event counter list for IBM z13
==============================================================================

Raw
event  Name   Description
--------------------------

r0     CPU_CYCLES

              Cycle Count.
              Counter 0 / Basic Counter Set.

r1     INSTRUCTIONS

              Instruction Count.
              Counter 1 / Basic Counter Set.

r2     L1I_DIR_WRITES

              Level-1 I-Cache Directory Write Count.
              Counter 2 / Basic Counter Set.

r3     L1I_PENALTY_CYCLES

              Level-1 I-Cache Penalty Cycle Count.
              Counter 3 / Basic Counter Set.

r4     L1D_DIR_WRITES
```

For a full list of options type `lscpumpf -h`, here only the counters for which the LPAR is authorized are listed.

```
# lscpumf -h
Usage: lscpumf -h|-v
       lscpumf [-i]
       lscpumf -c|-C
Options:
       -i    Displays detailed information.
       -c    Lists counters for which the
             LPAR is authorized.
       -C    Lists counters regardless of
             LPAR authorization.
       -s    Lists perf raw events that
             activate the sampling facility.
       -h    Displays help information,
             then exits.
       -v    Displays version information,
             then exits.
```

```
               Level-1 D-Cache Directory Write Count.
               Counter 4 / Basic Counter Set.
```

2.2 RAMFS

[2.1] A *ramdisk* is a block of random-access memory that software is treating as if the memory is a disk drive

If the interval for sampling or counting is relatively small it is advisable to use on a *ramdisk*[1]. This way one can avoid accounting of disk io or block devices events in perf.data. These events will bias the measurement.

```
# mount -t ramfs ramfs /mnt/ramfs
```

3 FIRST STEPS

3.1 PERF LIST

To get a first overview which events are known to perf use the perf command `list`. There are several groups of events that are known to perf, namely: Hardware, Software, PMU raw hardware, tracepoints and Statically Defined Tracing (SDT) events.

```
# perf list

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

  cpu-cycles OR cycles                              [Hardware event]
  instructions                                      [Hardware event]

  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]
```

To narrow the output just append the specific group identifier to `list`. Lets see which events from the PMU are known to perf.

```
# perf list pmu

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

  cpum_cf/AES_BLOCKED_CYCLES/                       [Kernel PMU event]
  cpum_cf/AES_BLOCKED_FUNCTIONS/                    [Kernel PMU event]
  cpum_cf/AES_CYCLES/                               [Kernel PMU event]
  cpum_cf/AES_FUNCTIONS/                            [Kernel PMU event]
  cpum_cf/CPU_CYCLES/                               [Kernel PMU event]
  cpum_cf/DEA_BLOCKED_CYCLES/                       [Kernel PMU event]
  cpum_cf/DEA_BLOCKED_FUNCTIONS/                    [Kernel PMU event]
  cpum_cf/DEA_CYCLES/                               [Kernel PMU event]
  cpum_cf/DEA_FUNCTIONS/                            [Kernel PMU event]
  cpum_cf/DTLB1_GPAGE_WRITES/                       [Kernel PMU event]
```

Simple regex can be also used to narrow the output for specific functions, here e.g. show only the syscalls events known to *perf*.

```
# perf list 'syscalls:*'

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

  syscalls:sys_enter_access                         [Tracepoint event]
  syscalls:sys_enter_acct                           [Tracepoint event]
  syscalls:sys_enter_add_key                        [Tracepoint event]
  syscalls:sys_enter_adjtimex                       [Tracepoint event]
  syscalls:sys_enter_alarm                          [Tracepoint event]
  syscalls:sys_enter_bdflush                        [Tracepoint event]
  syscalls:sys_enter_bind                           [Tracepoint event]
  syscalls:sys_enter_bpf                            [Tracepoint event]
  syscalls:sys_enter_brk                            [Tracepoint event]
```

3.2 LIVE PREVIEW

To get a live preview like in the Linux tool `top`, just issue `perf top` where one can similarly to `top` sort the columns.

```
# perf top # interactive
Samples: 2K of event 'cycles:ppp', Event count (approx.): 406286143513
Overhead  Shared Object                     Symbol
  37,89%  chrome                            [.] 0x0000000003e6dcb4
  37,89%  perf-6110.map                     [.] 0x00003c316eabe4f6
  14,50%  [vdso]                            [.] __vdso_gettimeofday
   9,72%  [kernel]                          [k] rb_next
   0,00%  [kernel]                          [k] _nv017513rm
   0,00%  nvidia_drv.so                     [.] 0x0000000000084f46
   0,00%  chrome                            [.] 0x00000000014eb2ff
   0,00%  nvidia_drv.so                     [.] 0x000000000005a8cd
   0,00%  [kernel]                          [k] pci_conf1_read
   0,00%  libc-2.24.so                      [.] _int_malloc
   0,00%  nvidia_drv.so                     [.] 0x000000000005a840
   0,00%  [kernel]                          [k] unix_poll
   0,00%  nvidia_drv.so                     [.] 0x000000000009ea1d
```

If only a specific event should be monitored, provide the event as an argument to `-e`, `–event`.

```
# perf top -e task-clock # Software event
# perf top -e r0         # Raw PMU event
```

COUNTING EVENTS 4

4.1 PERF STAT

The *perf* tool has two modus operandi. The (1) is counting events and the (2) is recording (sampling) events. The focus in this section will be on counting various events.

A very simple example of counting events is to call `perf stat` on ls. It shows predefined events that are associated with the `stat` command. As for every *perf* command, it can be selected which events to take under consideration.

```
# perf stat ls
A    bin      etc    lost+found  opt             proc       sbin  usr
A.c  boot     home   media     output.svg   provisioner  srv   var
app  chroot.sh       lib    mnt       perf.data    root       sys
B    dev      lib64  op        perf.data.old  run             tmp

 Performance counter stats for 'ls':

         0.425128      task-clock (msec)         #    0.753 CPUs utilized
                0      context-switches          #    0.000 K/sec
                0      cpu-migrations            #    0.000 K/sec
               94      page-faults               #    0.221 M/sec
        2,070,136      cycles                    #    4.869 GHz
        1,284,151      instructions              #    0.62  insn per cycle
   <not supported>      branches
   <not supported>      branch-misses

      0.000564777 seconds time elapsed
```

To count the number of the raw event CPU_CYCLES , one can call `perf stat` with the `-e,-event` parameter and as an argument the `r0` raw event.

```
# perf stat -e r0 ls
A    bin      etc    lost+found  opt             proc       sbin  usr
A.c  boot     home   media     output.svg   provisioner  srv   var
app  chroot.sh       lib    mnt       perf.data    root       sys
B    dev      lib64  op        perf.data.old  run             tmp

 Performance counter stats for 'ls':

        1,951,637      r0

      0.000531562 seconds time elapsed
```

To have *perf* to print the stats every second for example, than put the time in [ms] after `-I,-interval-print`.

```
# perf stat -I 1000 -e r0 lat_mem_rd 1 2048 2>&1
"stride=2048
#           time             counts unit events
     1.000112465      4,993,322,683      r0
0.00195 0.793
     2.000195215      4,997,804,011      r0
0.00293 0.793
     3.000270278      4,997,843,664      r0
     4.000333966      4,997,740,847      r0
0.00391 0.793
     4.498059029      2,487,320,780      r0
     4.498089872            158,701      r0
```

Additionally to intervall printing a nice feature is to show how the events are distributed across cores. Append `-pre-core` to the `perf stat` command.

```
# perf stat -I 1000 -a -per-core -e r0 md5sum /boot/vmlinuz 2>&1
2d177b173d7e9ff0f00dc990708bf6fc  /boot/vmlinuz
```

```
#           time core         cpus             counts unit events
     0.005954344 S3-C0           1         29,545,117      r0
     0.005954344 S3-C1           1             37,632      r0
     0.005954344 S3-C2           1             63,045      r0
     0.005954344 S3-C3           1             97,929      r0
```

4.1.1 Counting Backlog

```
perf stat -B ./lat_mem_rd 1 2048
perf stat -e task-clock,cpu-clock ./lat_mem_rd 1 2048

perf stat -e cache-misses,L1-dcache-loads,mem-loads ./lat_mem_rd 1 2048
perf stat -e cache-misses ./lat_mem_rd 1 2048

# task-clock is based only on the time spent on the profiled task,
# so that doesn’t count time spent on other tasks, it has a per thread granularity

# print out every second

perf stat -I 1000 -e mem-loads ./lat_mem_rd 1 2048
perf stat -I 1000 -e L1-dcache-loads ./lat_mem_rd 1 2048
perf stat -I 1000 -e L1-dcache-loads,cache-misses ./lat_mem_rd 1 2048

# per-core per-cpu ..

perf stat -I 1000 -e instructions –per-core lat_mem_rd 1 2048
```

SAMPLING EVENTS 5

5.1 PERF RECORD

The second modus operandi is recording (sampling) events. The default frequency of *perf* for sampling is 1000Hz. To sample at a lower frequency just append the flag `-F <freq>` to `perf record`. So e.g. to sample the cpu-cycles at 49 Hz for a simple workload one can use it as follows.

```
# perf record -F 49  -e cpu-cycles -o ./perf.data – md5sum /boot/vmlin* | cut -c 1-70
3e9fe83e5d7ecd6cea95928ae5797c4a  /boot/vmlinux-4.11.0-20170505.0.8571
5537e1d73f7e6bcfccf3340f2f0100bf  /boot/vmlinux-4.11.0-20170505.0.8571
dabc2613aaffe1ffc6eb1ae9a1a80f89  /boot/vmlinux-4.11.0-20170505.0.8571
6f6593f862c4a57a64dfb4880196bb99  /boot/vmlinux-4.11.0-20170505.0.daa6
9bd04638d9da0e44dfade5cc70af9817  /boot/vmlinuz
81cad85305e6d4f0a3c71db200edc748  /boot/vmlinuz-4.11.0-00002-g6dc0234-
81cad85305e6d4f0a3c71db200edc748  /boot/vmlinuz-4.11.0-00002-g6dc0234-
668db32e445a13f5adb9e958b02eb56a  /boot/vmlinuz-4.11.0-03422-g028fd7c-
668db32e445a13f5adb9e958b02eb56a  /boot/vmlinuz-4.11.0-03422-g028fd7c-
9bd04638d9da0e44dfade5cc70af9817  /boot/vmlinuz-4.11.0-03424-g99ce1a5-
9bd04638d9da0e44dfade5cc70af9817  /boot/vmlinuz-4.11.0-03424-g99ce1a5-
68b317e587506c383d94be767139c3fe  /boot/vmlinuz-4.11.0-20170505.0.daa6
5e11628b5ae8e092ebca56a2c3bba438  /boot/vmlinuz-4.11.0-rc6-00001-ged58
5e11628b5ae8e092ebca56a2c3bba438  /boot/vmlinuz-4.11.0-rc6-00001-ged58
5b502a065a7a7429a8e540d7759366e7  /boot/vmlinuz-4.12.0-rc4-00524-g1b35
1cb87bbf5e900fca3c61f9311ac7c066  /boot/vmlinuz-4.12.0-rc7
c4f2c214bf421ed0462c0620f808ed49  /boot/vmlinuz-4.12.0-rc7perf
c4f2c214bf421ed0462c0620f808ed49  /boot/vmlinuz-4.12.0-rc7perf.old
```

The reason for using 49 as a frequency for sampling is to avoid sampling in lockstep with other activities that can lead to misleading results.

After sampling the workload there are several ways how to examine the gathered data. The first command is `report` that shows where the events are spend in which function. Here as expected the cycles are spent in the function `md5sum` from `libcryto.so`.

```
# perf report –stdio –header
# ========
# captured on: Mon Jul 10 12:21:44 2017
# hostname : s3111p09
# os release : 4.11.0-03424-g99ce1a5-dirty
# perf version : 4.11.gead9ae
# arch : s390x
# nrcpus online : 16
# nrcpus avail : 16
# cpudesc : IBM/S390
# cpuid : IBM/S390
# total memory : 264212152 kB
# cmdline : /mnt/6ddc/kernel/latest/tools/perf record -F 49 -e cpu-cycles – md5sum
/boot/vmlinux-4.11.0-20170505.0.8571fc5.6b03710.fc25.s390xdefault
/boot/vmlinux-4.11.0-20170505.0.8571fc5.6b03710.fc25.s390xperformance
/boot/vmlinux-4.11.0-20170505.0.8571fc5.6b03710.fc25.s390xzfcpdump
/boot/vmlinux-4.11.0-20170505.0.daa6a8b.6b03710.fc25.s390xkvm
/boot/vmlinuz
/boot/vmlinuz-4.11.0-00002-g6dc0234-dirty /boot/vmlinuz-4.11.0-00002-g6dc0234-dirty.old
/boot/vmlinuz-4.11.0-03422-g028fd7c-dirty /boot/vmlinuz-4.11.0-03422-g028fd7c-dirty.old
/boot/vmlinuz-4.11.0-03424-g99ce1a5-dirty /boot/vmlinuz-4.11.0-03424-g99ce1a5-dirty.old
/boot/vmlinuz-4.11.0-20170505.0.daa6a8b.6b03710.fc25.s390xkvm /boot/vmlinuz-4.11.0-rc6-00001-ged585a7-dirty
/boot/vmlinuz-4.11.0-rc6-00001-ged585a7-dirty.old /boot/vmlinuz-4.12.0-rc4-00524-g1b35c0a-dirty
/boot/vmlinuz-4.12.0-rc7 /boot/vmlinuz-4.12.0-rc7perf /boot/vmlinuz-4.12.0-rc7perf.old
# event : name = cpu-cycles, , size = 112, { sample_period, sample_freq } = 49,
  sample_type = IP|TID|TIME|PERIOD, disabled = 1, inherit = 1, mmap = 1, comm = 1,
  freq = 1, enable_on_exec = 1, task = 1, sample_id_all = 1, exclude_guest = 1, mmap2 = 1,
  comm_exec = 1
# HEADER_CPU_TOPOLOGY info available, use -I to display
# HEADER_NUMA_TOPOLOGY info available, use -I to display
# pmu mappings: cpum_sf = 4, cpum_cf = 4, software = 1, tracepoint = 2
# HEADER_CACHE info available, use -I to display
```

```
# missing features: HEADER_TRACING_DATA HEADER_BRANCH_STACK HEADER_GROUP_DESC HEADER_AUXTRACE HEADER_STAT
# ========
#
#
# Total Lost Samples: 0
#
# Samples: 3  of event 'cpu-cycles'
# Event count (approx.): 306120000
#
# Overhead  Command  Shared Object        Symbol
# ........  .......  ...................  .......................
#
   100.00%  md5sum   libcrypto.so.1.1.0e  [.] md5_block_data_order

#
# (Tip: Show user configuration overrides: perf config -user -list)
#
```

The second way is to use `perf script` that shows a more detailed look of the samples. This mode is more of use when developing new functionality for perf and debugging perf itself. The section 9.1.2 HEATMAP will show the use of `perf script` to develop a visualization for latency examination.

```
# perf script -D

0xe8 [0x50]: event: 1
.
. ... raw event: size 80 bytes
.  0000:  00 00 00 01 00 01 00 50 ff ff ff ff 00 00 00 00  .......P........
.  0010:  00 00 00 00 00 00 02 00 00 00 03 ff 80 00 26 70  .............&p
.  0020:  00 00 00 00 00 00 02 00 5b 6b 65 72 6e 65 6c 2e  ........[kernel.
.  0030:  6b 61 6c 6c 73 79 6d 73 5d 5f 74 65 78 74 00 00  kallsyms]_text..
.  0040:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ...............

0 0xe8 [0x50]: PERF_RECORD_MMAP -1/0: [0x200(0x3ff80002670) @ 0x200]: x [kernel.kallsyms]_text

0x138 [0x80]: event: 1
.
. ... raw event: size 128 bytes
.  0000:  00 00 00 01 00 01 00 80 ff ff ff ff 00 00 00 00  ...............
.  0010:  00 00 03 ff 80 00 28 70 00 00 00 00 00 00 b8 98  ......(p........
.  0020:  00 00 00 00 00 00 00 00 2f 6c 69 62 2f 6d 6f 64  ......../lib/mod
.  0030:  75 6c 65 73 2f 34 2e 31 31 2e 30 2d 30 33 34 32  ules/4.11.0-0342
.  0040:  34 2d 67 39 39 63 65 31 61 35 2d 64 69 72 74 79  4-g99ce1a5-dirty
```

The last but not least examination method of `perf.data` is to annotate code with the samples gathered. The source code is interweaved, line-wise with the samples recorded before, here `cpu-cycles`.

```
# perf annotate -stdio
 Percent |    Source code & Disassembly of libcrypto.so.1.1.0e for cpu-cy
------------------------
         :
         :
         :
         :    Disassembly of section .text:
         :
         :    0000000000113a48 <MD5_Init@@OPENSSL_1_1_0+0x38>:
         :    md5_block_data_order():
         :    #ifndef md5_block_data_order
         :    # ifdef X
         :    #  undef X
         :    # endif
         :    void md5_block_data_order(MD5_CTX *c, const void *data_, si
         :    {
    0.00 :      113a48:       stmg    %r6,%r15,48(%r15)
    0.00 :      113a4e:       lay     %r15,-248(%r15)
    0.00 :      113a54:       stg     %r2,240(%r15)
    0.00 :      113a5a:       lgr     %r1,%r2
         :    # else
         :        MD5_LONG XX[MD5_LBLOCK];
         :    #  define X(i)   XX[i]
         :    # endif
         :
         :        A = c->A;
```

```
    0.00 :      113a5e:       l       %r2,0(%r2)
         :        B = c->B;
    0.00 :      113a62:       l       %r11,4(%r1)
         :        C = c->C;
    0.00 :      113a66:       l       %r9,8(%r1)
```

STRACE INSPIRED 6

6.1 PERF TRACE

Based on the `perf trace` command it will be shown how to use counting and recording to examine an application.

By means of a simple benchmark, here `dd` issuing some reads and writes, it will be shown that *perf* is significantly faster than its counterpart `strace`.

```
# sample command to compare strace with perf trace performance
# dd if=/dev/zero of=/dev/null bs=512 count=1000k 2>&1
1024000+0 records in
1024000+0 records out
524288000 bytes (524 MB, 500 MiB) copied, 0.290385 s, 1.8 GB/s
```

Now lets have a look how long does `strace` take to count the syscalls issued in the case of `dd`.

```
# strace counting of syscalls of dd
strace -c dd if=/dev/zero of=/dev/null bs=512 count=1000k 2>&1
1024000+0 records in
1024000+0 records out
524288000 bytes (524 MB, 500 MiB) copied, 33.4576 s, 15.7 MB/s
% time     seconds  usecs/call     calls    errors syscall
—— ——– ——– ——– ——– ——–-
 50.95    1.476851           1   1024003           read
 49.05    1.421691           1   1024003           write
  0.00    0.000048          48         1           execve
  0.00    0.000035           3        14         7 open
  0.00    0.000021           2        10           mmap
  0.00    0.000015           2        10           close
  0.00    0.000009           2         5           fstat
  0.00    0.000009           3         3           mprotect
  0.00    0.000006           2         3         3 access
  0.00    0.000006           2         3           brk
  0.00    0.000004           2         2           dup2
  0.00    0.000004           1         3           rt_sigaction
  0.00    0.000003           3         1           munmap
  0.00    0.000001           1         1           lseek
—— ——– ——– ——– ——– ——–-
100.00    2.898703               2048062        10 total
```

The running time of the native `dd` was 0.3 [s], where the counting of syscalls with `strace` took 33.5 [s] (111 times slower). Now lets see how *perf* performs.

```
# pref  counting of syscalls of dd
perf stat -e 'syscalls:sys_enter_*' dd if=/dev/zero of=/dev/null bs=512 count=1000k 2>&1 | head -n 20
1024000+0 records in
1024000+0 records out
524288000 bytes (524 MB, 500 MiB) copied, 0.39246 s, 1.3 GB/s

 Performance counter stats for 'dd if=/dev/zero of=/dev/null bs=512 count=1000k':

         1,024,003      syscalls:sys_enter_read
         1,024,003      syscalls:sys_enter_write
                 0      syscalls:sys_enter_socket
                 0      syscalls:sys_enter_socketpair
                 0      syscalls:sys_enter_bind
                 0      syscalls:sys_enter_listen
                 0      syscalls:sys_enter_accept4
                 0      syscalls:sys_enter_connect
                 0      syscalls:sys_enter_getsockname
                 0      syscalls:sys_enter_getpeername
                 0      syscalls:sys_enter_sendto
                 0      syscalls:sys_enter_recvfrom
                 0      syscalls:sys_enter_setsockopt
                 0      syscalls:sys_enter_getsockopt
```

```
                  0      syscalls:sys_enter_shutdown
                  0      syscalls:sys_enter_sendmsg
```

The count of calls for read/write are the same of course but the time of counting these events took for *perf* only 0.4 [s] compared to the 33.5 [s] of strace.

Now record the `trace` events and examine them with `perf report`. The samples are saved in a file called `perf.data`, in the directory where the *perf* command was executed. There are events that generate a lot of data so beware to have enough memory in the case of `ramfs` as decribed before or disk space.

```
# record trace
# perf trace record -a dd if=/dev/zero of=/dev/null bs=512 count=1000k 2>&1
1024000+0 records in
1024000+0 records out
524288000 bytes (524 MB, 500 MiB) copied, 0.963196 s, 544 MB/s
[ perf record: Woken up 0 times to write data ]
Warning:
Processed 4911336 events and lost 5 chunks!

Check IO/CPU overload!

Warning:
50 out of order events recorded.
[ perf record: Captured and wrote 487.895 MB perf.data (4722121 samples) ]
```

Let's have a look at the sample data with `perf report`. There are far more options to `perf report` than shown in this document so it is advisable to look at the man pages for each *perf* command.

```
# perf report  -stdio
# To display the perf.data header info, please use –header/–header-only options.
#
#
# Total Lost Samples: 0
#
# Samples: 2M of event 'raw_syscalls:sys_enter'
# Event count (approx.): 2361063
#
# Overhead  Trace output
# ........  ................................................................................
#
    38.05%  NR 4 (1, 2aa2fd6f000, 200, 2aa2fd6f000, 0, 3ff82376690)
     0.00%  NR 4 (3, 2aa26d5cbc8, 8, 0, 2aa26d5cbc8, 8)
     0.00%  NR 4 (3, 3ff9baad9f0, 870, 1ac9f0, 3ff9baad9f0, 870)
     0.00%  NR 4 (3, 3ff9b907f40, 5e8, 6f40, 3ff9b907f40, 5e8)
     0.00%  NR 4 (3, 3ff9b918500, 5e8, 17500, 3ff9b918500, 5e8)
     0.00%  NR 4 (3, 3ff9b91d348, 870, 1c348, 3ff9b91d348, 870)
     0.00%  NR 4 (3, 3ff9b94abc8, 870, 49bc8, 3ff9b94abc8, 870)
     0.00%  NR 4 (3, 3ff9b94e9e0, 870, 4d9e0, 3ff9b94e9e0, 870)
     0.00%  NR 4 (3, 3ff9b95fdb8, 870, 5edb8, 3ff9b95fdb8, 870)

                            ...
```

Each syscall has a unique id (see syscall.h) and the coresponding syscall for NR 4 is `SYS_write` with the arguments `(1, 2aa2fd6f000, 200, 2aa2fd6f000, 0, 3ff82376690)` supplied to the function call. If one would look at the complete trace, there would be other syscalls but the two most important in this trace are NR 3,4, respectively `SYS_read, SYS_write`.

7 SCHEDULING

7.1 PERF SCHED

Another very interesting *perf* command is `sched`. These are predefined events that are associated with scheduling. With `sched` one can see how and when the workload is scheduled across the CPU's, when it is woken up and led to sleep.

So e.g. to know if such a simple workload like `lat_mem_rd` is CPU affine or it is scheduled all over the CPU's, let's record a sample run.

```
perf sched record lat_mem_rd 1 2048
```

```
# show the scheduler map
perf sched map | head -n 30

                         ,*A0                                        5382.466840 secs A0 => perf:15767
                           A0                 *.                     5382.466878 secs .  => swapper:0
                           A0                  .              *B0    5382.478644 secs B0 => rcu_sched:8
                           A0                  .              *.     5382.478647 secs
                 ,*C0      A0                  .               .     5382.478680 secs C0 => ksoftirqd/4:29
                 ,*B0      A0                  .               .     5382.478682 secs
                 ,*.       A0                  .               .     5382.478685 secs
                 ,*B0      A0                  .               .     5382.498640 secs
                 ,*.       A0                  .               .     5382.498643 secs
                   .       A0             *B0  .               .     5382.508689 secs
                   .       A0             *.   .               .     5382.508691 secs
                   .       A0             *B0  .               .     5382.528655 secs
                   .       A0             *.   .               .     5382.528658 secs
                   .       A0              .  *D0              .     5382.538673 secs D0 => kworker/11:0:65
                   .       A0              .  *.               .     5382.538678 secs
                   .       A0     *E0      .   .               .     5382.658644 secs E0 => kworker/8:0:13068
                   .       A0     *.       .   .               .     5382.658646 secs
                   .       A0      .       .  *D0              .     5382.668657 secs
                   .       A0      .       .  *.               .     5382.668663 secs
                   .       A0      .       .  *D0              .     5382.798640 secs
                   .       A0      .       .  *.               .     5382.798647 secs
                   .      *F0      .       .   .               .     5382.870995 secs F0 => multipathd:261
                   .      *A0      .       .   .               .     5382.870997 secs
                   .       A0      .       .  *D0              .     5382.928688 secs
                   .       A0      .       .  *.               .     5382.928697 secs
                   .       A0      .       .   .  *G0          .     5382.992699 secs G0 => lat_mem_rd:15768
                   .      *.       .       .   .   G0          .     5382.992705 secs
                   .      *A0      .       .   .   G0          .     5382.992751 secs
                   .      *.       .       .   .   G0          .     5382.992755 secs
                   .       .       .       .  *D0  G0          .     5383.058671 secs
```

The first four columns represent the four CPUs in the system. A0, B0, C0 are ids for each executable that is called or scheduled during runtime of `lat_mem_rd`.

Every asterisk means a scheudling event. What is evident here in the first 30 rows, is that `lat_mem_rd` first runs on CPU0 and is scheduled after some time to CPU1 as indicated by the *Do where `lat_mem_rd` gets a new id.

Examining the complete trace shows that the workload is scheduled across all CPUs and not running exclusively on one or affine to one CPU.

Another nice features of `perf sched` is to report the latencies of the scheduling events per task. These latency can be correlated with the timestamp from the table below with the `perf sched map` output above.

```
# show scheduling latencies
# perf sched lat
 Task                  |   Runtime ms  | Switches | Maximum delay at

 cron:808              |      0.009 ms |        1 | max at: 3500.02 s
 perf:4737             |      2.890 ms |        1 | max at: 3507.16 s
```

```
  kworker/u64:0:4543    |      0.015 ms |        5 | max at: 3488.38 s
  khungtaskd:30         |      0.004 ms |        1 | max at: 3506.07 s
  kworker/2:0:2146      |      0.054 ms |       14 | max at: 3502.20 s
  ksoftirqd/1:14        |      0.003 ms |        1 | max at: 3488.79 s
  kworker/u64:1:4624    |      0.010 ms |        3 | max at: 3485.08 s
  kworker/0:2:414       |      0.042 ms |       13 | max at: 3487.43 s
  lat_mem_rd:(20)       |  23709.517 ms |      145 | max at: 3483.96 s
  rcu_sched:7           |      0.307 ms |      126 | max at: 3504.74 s
  pkcsslotd:825         |      0.010 ms |        2 | max at: 3487.48 s
  irqbalance:867        |      0.078 ms |        2 | max at: 3493.11 s
  jbd2/dasda1-8:317     |      0.090 ms |        9 | max at: 3501.99 s
  kworker/3:2:445       |      0.007 ms |        2 | max at: 3488.39 s
  migration/3:23        |      0.000 ms |        1 | max at: 3483.45 s
  multipathd:(3)        |      0.000 ms |       30 | max at: 3490.08 s
  gmain:837             |      0.027 ms |        6 | max at: 3484.02 s
  ksoftirqd/2:19        |      0.002 ms |        1 | max at: 3488.77 s
  ksoftirqd/0:3         |      0.007 ms |        4 | max at: 3495.80 s
  kworker/1:0:2740      |      0.058 ms |       27 | max at: 3499.83 s

  TOTAL:                |  23713.129 ms |      394 |
```

Additionally the number of context switches is reported for all task running at the time of `lat_mem_rd`.

To have even more info about the separate events call `perf sched` with the `script` argument.

```
# show detailed info about scheduling
# perf sched script | head -n 30 | cut -c 1-70

            perf  1921 [000] 12208.501300:  sched:sched_wakeup: perf:1
         swapper     0 [001] 12208.501303:  sched:sched_switch: swappe
            perf  1921 [000] 12208.501318:  sched:sched_switch: perf:1
            perf  1924 [001] 12208.501326:  sched:sched_wakeup: migrat
            perf  1924 [001] 12208.501327:  sched:sched_switch: perf:1
     migration/1    13 [001] 12208.501331:  sched:sched_switch: migrat
         swapper     0 [002] 12208.501332:  sched:sched_switch: swappe
         swapper     0 [000] 12208.502371:  sched:sched_wakeup: rcu_sc
         swapper     0 [000] 12208.502372:  sched:sched_switch: swappe
       rcu_sched     7 [000] 12208.502374:  sched:sched_switch: rcu_sc
         swapper     0 [000] 12208.510368:  sched:sched_wakeup: rcu_sc
         swapper     0 [000] 12208.510370:  sched:sched_switch: swappe
       rcu_sched     7 [000] 12208.510372:  sched:sched_switch: rcu_sc
         swapper     0 [000] 12208.518367:  sched:sched_wakeup: rcu_sc
         swapper     0 [000] 12208.518369:  sched:sched_switch: swappe
       rcu_sched     7 [000] 12208.518371:  sched:sched_switch: rcu_sc
         swapper     0 [000] 12208.526366:  sched:sched_wakeup: rcu_sc
         swapper     0 [000] 12208.526368:  sched:sched_switch: swappe
       rcu_sched     7 [000] 12208.526369:  sched:sched_switch: rcu_sc
         swapper     0 [003] 12209.027500:  sched:sched_switch: swappe
      lat_mem_rd  1924 [002] 12209.027506:  sched:sched_switch: lat_me
      lat_mem_rd  1925 [003] 12209.027544:  sched:sched_wakeup: lat_me
         swapper     0 [002] 12209.027546:  sched:sched_switch: swappe
      lat_mem_rd  1924 [002] 12209.027549:  sched:sched_switch: lat_me
         swapper     0 [000] 12209.364191:  sched:sched_wakeup: multip
         swapper     0 [000] 12209.364192:  sched:sched_switch: swappe
      multipathd   418 [000] 12209.364194:  sched:sched_switch: multip
         swapper     0 [002] 12209.580448:  sched:sched_wakeup: multip
         swapper     0 [002] 12209.580449:  sched:sched_switch: swappe
      multipathd   417 [002] 12209.580451:  sched:sched_switch: multip
```

In Linux 4.10 an additional command to `perf sched` was added, namely `timehist`. It shows the scheduler latency by event. It includes the time the task was waiting to be woken up and the scheduler latency after wakeup to running.

```
# -M migration events
# -V CPU
# -w wakup events
# perf sched timehist -MVw | head -n 20
           time    cpu  012345678  task name           wait time  sch delay   run time
                                   [tid/pid]              (msec)     (msec)     (msec)
_____ __  ___  ______ ___  ___  ___
     291.481867 [0001]   i         <idle>                  0.000      0.000      0.000
     291.481925 [0005]       i     <idle>                  0.000      0.000      0.000
     291.481963 [0005]       s     Xorg[1884]              0.000      0.000      0.037 schedule_hrtimeout_range_clock <- schedu
     291.482019 [0001]   s         irq/31-nvidia[1980]     0.000      0.000      0.152 irq_thread <- kthread <- ret_from_fork
     291.482136 [0006]             perf[6319]                                          awakened: perf[6320]
```

```
291.482140 [0001]   i         <idle>                    0.152      0.000      0.121
291.482175 [0006]       s     perf[6319]                0.000      0.000      0.000 schedule_hrtimeout_range_clock <- schedu
291.482562 [0001]             lat_mem_rd[6320]                                      awakened: kworker/u16:3[227]
291.482564 [0006]       i     <idle>                    0.000      0.000      0.389
291.482569 [0006]        m      kworker/u16:3[227]                                  migrated: terminator[6030] cpu 1 => 2
291.482572 [0006]             kworker/u16:3[227]                                    awakened: terminator[6030]
291.482574 [0006]       s     kworker/u16:3[227]        0.000      0.002      0.010 worker_thread <- kthread <- ret_from_for
291.482576 [0002]    i        <idle>                    0.000      0.000      0.000
291.482614 [0002]    s        terminator[6030]          0.000      0.004      0.037 schedule_hrtimeout_range_clock <- schedu
```

For a graphical overview of scheduling events of a workload `perf timechart` can be used. First record the needed samples.

```
# generate diagram of scheduling events + cycles
# perf timechart record  lat_mem_rd 1 2048
```

Now create a SVG out of the data.

```
perf timechart
```

A very fast SVG viewer on a Linux system is /rsvg-view-3/.

```
# rsvg-view-3 output.svg
```

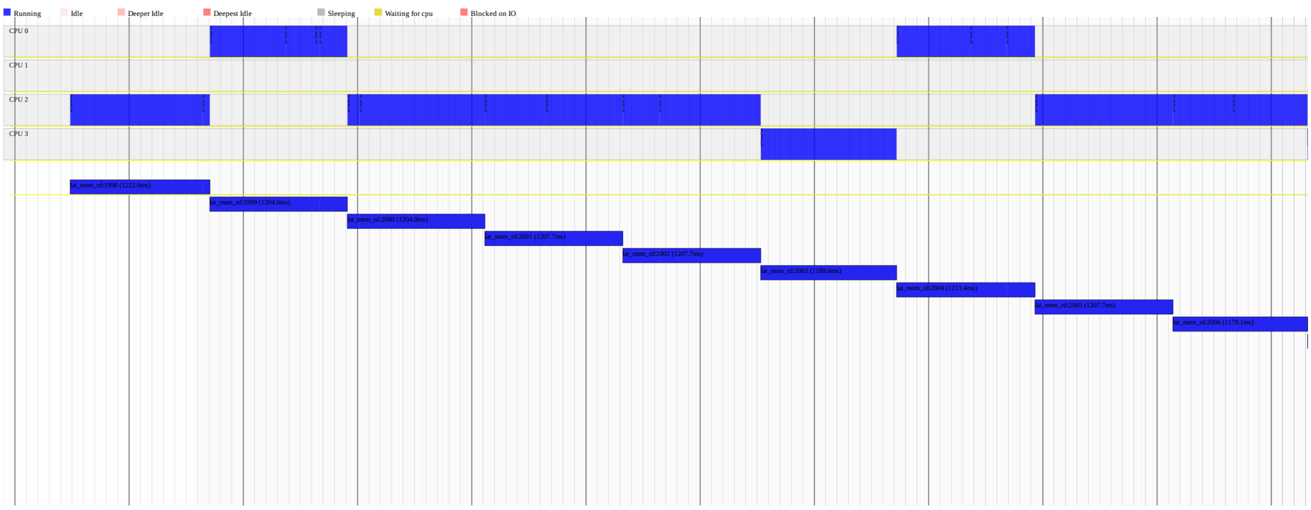


FIGURE 7.1 Timechart of `lat_mem_rd`, which is scheduled on all available CPUS

8 TRACEPOINTS

8.1 WHAT IS A TRACEPOINT

A tracepoint provides a hook to call a function (probe) everytime a tracepoint is encountered. This probes can be supplied by e.g. *perf* counting or recording variables associated to a specific tracepoint or can be added programmatically at run time.

`Tracepoints` can be `on` (probe attached) or `off` (probe not attached). When `on` beware that probes are run everytime a tracepoint is encountered and executed in the context of the caller. If the probe contains heavy computation the execution of the traced binary will be slowed down.

When `off` there is a tiny time penalty (check a condition for a branch)

```
if (tracepoint_on) { tracepoint_probe(); }
```

and a minor space penalty for a auxiliary data structure. For `Tracepoints` to work properly with *perf* one needs to install the debug symbols for the to examined binary.

PROBES 9

9.1 USERSPACE PROBING

As mentioned in the 1.2.1 DEBUG SYMBOLS section, *perf* can create `Tracepoints` at functions and source code lines. So as a first step lets have a look which functions reside in the used binary.

```
# perf probe -x lat_mem_rd -F | head -n 20
Delta
Now
alarm@plt
bandwidth
base_initialize
benchmark_loads
benchmp
benchmp_child
benchmp_child_sigchld
benchmp_child_sigterm
benchmp_childid
benchmp_getstate
benchmp_interval
benchmp_parent
benchmp_sigalrm
benchmp_sigchld
benchmp_sigterm
bread
bytes
close@plt
```

Show all functions in `lat_mem_rd` for probing, debug symbols needed.

From earlier examinations one knows that the `benchmark_loads` has the most samples. Now show or list the source code of the specified function.

```
# perf probe -x lat_mem_rd -L benchmark_loads | head -n 20
<benchmark_loads@/mnt/6ddc/benchmarks/lmbench-3.0-a9/src/lat_mem_rd.c:0>
      0  benchmark_loads(iter_t iterations, void *cookie)
      1  {
                struct mem_state* state = (struct mem_state*)cookie;
      3         register char **p = (char**)state->p[0];
                register size_t i;
      5         register size_t count = state->len / (state->line * 100) + 1;

      7         while (iterations- > 0) {
      8                 for (i = 0; i < count; ++i) {
      9                         HUNDRED;
                        }
                }

     13         use_pointer((void *)p);
     14         state->p[0] = (char*)p;
     15  }

         void
```

List source code of `benchmark_loads` function, debug symbols needed.

That's really nice, we see the source code with associated line numbers and can have a glimpse on the function and its intent. One can further dissect the overview with another feature, namely inspecting the variables available at a specific source code line. Lets have a look at line 9, where the macro `HUNDRED` is located.

```
# list all variables at line=9 in benchmark_loads function
# perf probe -x lat_mem_rd -V benchmark_loads:9
Available variables at benchmark_loads:9
        @<benchmark_loads+86>
                (unknown_type       cookie
                char** p
                iter_t iterations
                size_t count
```

```
        size_t i
        struct mem_state*    state
```

There are some control variables `i, count, iterations` but the interesting variable here is `p` . The `lat_mem_rd` benchmarks uses pointer-chasing to step with a specific stride through the caches and memory and `p` is the beginning of a circular linked list. Lets create a *Tracepoint* and gather the address of `p`.

```
# create probe for variable p
# perf probe -x lat_mem_rd 'bl9=benchmark_loads:9 addr=p' 2>&1
Added new event:
  probe_lat:bl9        (on benchmark_loads:9 in lat_mem_rd with addr=p)

You can now use it in all perf tools, such as:

       perf record -e probe_lat:bl9 -aR sleep 1
```

This new probe can now be used in every perf tool as any other event. One can count, record, ... this new event. Just for sanity checking list the new event.

```
# check for event
# perf list bl9
probe_lat:bl9                                      [Tracepoint event]
```

Lets try to count how many times we reach this *Tracepoint*.

```
# use the event either in stat or record
# perf stat -e probe_lat:bl9 lat_mem_rd 1 2048 2>&1

"stride=2048
0.00195 7.208
0.00293 7.179

    ...

 Performance counter stats for 'lat_mem_rd 1 2048':

       28,802,191      probe_lat:bl9

      23.484057250 seconds time elapsed
```

To display the value of `p`, which was gathered in the *Tracepoint*, the samples have to be recorded.

```
# record the events so we can examine the value of p variable
# perf record -e probe_lat:bl9 lat_mem_rd 1 2048 2>&1
"stride=2048
0.00195 8.615
0.00293 8.550
0.00391 8.365
0.00586 8.355
     ...

[ perf record: Woken up 2768 times to write data ]
Warning:
Processed 35165656 events and lost 109 chunks!

Check IO/CPU overload!

[ perf record: Captured and wrote 1912.943 MB perf.data (23951777 samples) ]
```

Now just call `perf script` to have an console output of the gathered samples. What is evident here is that `lat_mem_rd` starts always from the same address the pointer-chasing, the variability lies in the iteration count and stride and buffer size.

```
# show the value of p
# perf script
      lat_mem_rd  5846 [000]  5210.256441: probe_lat:bl9: (2aa01f81d7e) addr=0x3ff9affe000
      lat_mem_rd  5846 [000]  5210.256447: probe_lat:bl9: (2aa01f81d7e) addr=0x3ff9affe000
      lat_mem_rd  5846 [000]  5210.256449: probe_lat:bl9: (2aa01f81d7e) addr=0x3ff9affe000
```

```
      lat_mem_rd  5846 [000]  5210.256450: probe_lat:bl9: (2aa01f81d7e) addr=0x3ff9affe000
      lat_mem_rd  5846 [000]  5210.256451: probe_lat:bl9: (2aa01f81d7e) addr=0x3ff9affe000
      lat_mem_rd  5846 [000]  5210.256452: probe_lat:bl9: (2aa01f81d7e) addr=0x3ff9affe000
      lat_mem_rd  5846 [000]  5210.256452: probe_lat:bl9: (2aa01f81d7e) addr=0x3ff9affe000
      lat_mem_rd  5846 [000]  5210.256453: probe_lat:bl9: (2aa01f81d7e) addr=0x3ff9affe000
      lat_mem_rd  5846 [000]  5210.256454: probe_lat:bl9: (2aa01f81d7e) addr=0x3ff9affe000
      lat_mem_rd  5846 [000]  5210.256455: probe_lat:bl9: (2aa01f81d7e) addr=0x3ff9affe000
      lat_mem_rd  5846 [000]  5210.256457: probe_lat:bl9: (2aa01f81d7e) addr=0x3ff9affe000
      lat_mem_rd  5846 [000]  5210.256457: probe_lat:bl9: (2aa01f81d7e) addr=0x3ff9affe000
      lat_mem_rd  5846 [000]  5210.256458: probe_lat:bl9: (2aa01f81d7e) addr=0x3ff9affe000
      lat_mem_rd  5846 [000]  5210.256459: probe_lat:bl9: (2aa01f81d7e) addr=0x3ff9affe000
      lat_mem_rd  5846 [000]  5210.256460: probe_lat:bl9: (2aa01f81d7e) addr=0x3ff9affe000
      lat_mem_rd  5846 [000]  5210.256461: probe_lat:bl9: (2aa01f81d7e) addr=0x3ff9affe000
      lat_mem_rd  5846 [000]  5210.256462: probe_lat:bl9: (2aa01f81d7e) addr=0x3ff9affe000
      lat_mem_rd  5846 [000]  5210.256463: probe_lat:bl9: (2aa01f81d7e) addr=0x3ff9affe000
      lat_mem_rd  5846 [000]  5210.256463: probe_lat:bl9: (2aa01f81d7e) addr=0x3ff9affe000
      lat_mem_rd  5846 [000]  5210.256464: probe_lat:bl9: (2aa01f81d7e) addr=0x3ff9affe000
```

9.1.1 *Backlog*

```
perf probe -x ./lat_mem_rd -L benchmark_loads
perf probe -x ./lat_mem_rd 'bm_load=benchmark_loads'

perf probe -x ./lat_mem_rd -V benchmark_loads:9
perf probe -x ./lat_mem_rd  'bl5=benchmark_loads:9 addr=p'

perf stat -I 1000 -e probe_lat:bm_load ./lat_mem_rd 1 2048
perf record -e probe_lat:mem_load ./lat_mem_rd 1 1024

perf script
```

9.1.2 *Heatmap*

In this section *probes*[1] will be used as timestamp anchor points for latency measurement of the *fio* benchmark. For latency measurements one needs two timestamps where the difference between them is the latency.

Since *fio* exercises the disk, the two *probes* used here will be `block:block_rq_issue` and `block:block_rq_complete`. But first create a *fio* testcase.

9.1 The probes used here are static tracepoints that are already available no need to create new one

```
cat > random-read-test.fio << EOF
[random-read]
rw=randread
size=4g
directory=/tmp
EOF
```

The *fio* benchmark will be doing a random read on a 4 Gb file.

Next, use *perf* and sample the benchmark with the two *probes* selected above.

```
# perf record -e block:block_rq_issue -e block:block_rq_complete -a – fio random-read-test.fio
```

Lets have a look at the captured data. Look for the timestamps and the events just captured. This will be used in the next step to calculate the latencies and feed them to the charting software.

```
# perf script -ns
            perf 12505 [003]  3108.108454:    block:block_rq_issue: 94,0 RM 4096 () 58789608 + 8 [perf]
         swapper     0 [003]  3108.108585: block:block_rq_complete: 94,0 RM () 58789608 + 8 [0]
            perf 12505 [003]  3108.108595:    block:block_rq_issue: 94,0 RM 4096 () 58789568 + 8 [perf]
         swapper     0 [003]  3108.108723: block:block_rq_complete: 94,0 RM () 58789568 + 8 [0]
            perf 12505 [003]  3108.108740:    block:block_rq_issue: 94,0 RA 4096 () 33569728 + 8 [perf]
            perf 12505 [003]  3108.108742:    block:block_rq_issue: 94,0 RA 4096 () 33569736 + 8 [perf]
            perf 12505 [003]  3108.108744:    block:block_rq_issue: 94,0 RA 4096 () 33569744 + 8 [perf]
            perf 12505 [003]  3108.108746:    block:block_rq_issue: 94,0 RA 4096 () 33569752 + 8 [perf]
            perf 12505 [003]  3108.108748:    block:block_rq_issue: 94,0 RA 4096 () 33569760 + 8 [perf]
         swapper     0 [003]  3108.108890: block:block_rq_complete: 94,0 RA () 33569728 + 8 [0]
```

The `-ns` switch is used to display the time using 9 decimal places. The 4th column shows the timestamps and the 5th the corresponding event.

With a bit of `awk` magic the events and the corresponding timestamps are extracted and the latency is calculated.

```
perf script -ns | awk '{ gsub(/:/, "") } $5 ~ /issue/ { ts[$6, $10] = $4 }
    $5 ~ /complete/ { if (l = ts[$6, $9]) { printf "%.f %.f\n", $4 * 1000000,
    ($4 - l) * 1000000; ts[$6, $10] = 0 } }' > out.lat_us
```

Now to visualize the latencies one needs additional software that is available on *github*.

```
git clone https://github.com/brendangregg/HeatMap.git
```

The last step is to use `trace2heatmap.pl` to generate an interactive Scalable Vector Graphics (SVG) of the latencies.

```
trace2heatmap.pl -unitstime=us  -maxlat=2000 -unitslabel=us \
-grid -title "fio Latency (DASD)" out.lat_us > heatmap.svg
```

Use your preferred SVG renderer and have a look at the created figure. The more red a sqaure is the more latencies were recorded. The is a tri modal distribution of latencies that corespond to (1) the page cache (2) to the disk cache and last the request that went to disk.

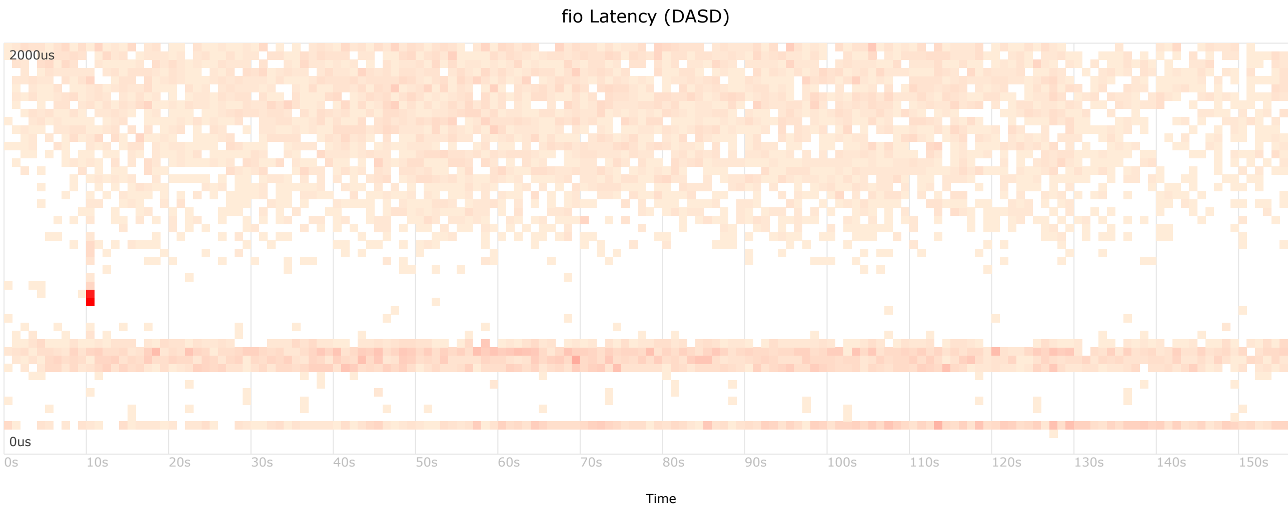


FIGURE 9.1 Latencies visualized with a heatmap

9.2 KERNEL PROBING

The nice thing about perf is that it makes no differentiation between userspace or kernelspace. What one was doing in the section before in userspace can be done similarly for the *Kernel*. The focus in this section will be on the `ping` command. Thats why the examination will be filtered from beginning on `icmp_rcv`.

List the functions of the current *Kernel* and grep for `icmp_rcv`.

```
#+BEGIN_SRC bash  :dir /sshx:root@s3111p09:/ :results value code :exports both :cache yes
# list function for probing here icmp_rcv
perf probe -k /usr/lib/debug/boot/vmlinux-4.8.0-34-generic  -F  | grep icmp_rcv 2>&1
icmp_rcv
```

Create a probe on `icmp_rcv`, this time the purpose of the probe is to count how many times `icmp_rcv` was called.

```
# create a probe on icmp_rcv
perf probe -k /usr/lib/debug/boot/vmlinux-4.8.0-34-generic   icmp_rcv 2>&1
```

```
TODO: perf stat ping
```

Record the samples and ping `pserver1` six times.

```
# recored new event probe:icmp_rcv
# perf record -e probe:icmp_rcv -aR ping -c6 pserver1.boeblingen.de.ibm.com
PING pserver1.boeblingen.de.ibm.com (9.152.140.6) 56(84) bytes of data.
64 bytes from pserver1.boeblingen.de.ibm.com (9.152.140.6): icmp_seq=1 ttl=64 time=0.547 ms
64 bytes from pserver1.boeblingen.de.ibm.com (9.152.140.6): icmp_seq=2 ttl=64 time=0.144 ms
64 bytes from pserver1.boeblingen.de.ibm.com (9.152.140.6): icmp_seq=3 ttl=64 time=0.143 ms
64 bytes from pserver1.boeblingen.de.ibm.com (9.152.140.6): icmp_seq=4 ttl=64 time=0.140 ms
64 bytes from pserver1.boeblingen.de.ibm.com (9.152.140.6): icmp_seq=5 ttl=64 time=0.141 ms
64 bytes from pserver1.boeblingen.de.ibm.com (9.152.140.6): icmp_seq=6 ttl=64 time=0.146 ms

– pserver1.boeblingen.de.ibm.com ping statistics –
6 packets transmitted, 6 received, 0% packet loss, time 5095ms
rtt min/avg/max/mdev = 0.140/0.210/0.547/0.150 ms
```

With the present `perf.data` and the `report` tool one can easily e.g. find out in which source file the function `icmp_rcv` is implemented.

```
# show the source file
#perf report -s srcfile –stdio 2>&1 –stdio-color=never
# To display the perf.data header info, please use –header/–header-only options.
#
#
# Total Lost Samples: 0
#
# Samples: 6  of event 'probe:icmp_rcv'
# Event count (approx.): 6
#
# Overhead  Source File
# ........  ...........
#
  100.00% icmp.c
```

Additionally one can gather the source line of the file reported above where the *Tracepoint* is located.

```
# show the source line
# perf report -s srcline –stdio 2>&1 –stdio-color=never
# To display the perf.data header info, please use –header/–header-only options.
#
#
# Total Lost Samples: 0
#
# Samples: 6  of event 'probe:icmp_rcv'
# Event count (approx.): 6
#
# Overhead  Source:Line
# ........  ...........
#
  100.00% icmp.c:973
```

```
# vi ../6ddc/kernel/linux/net/ipv4/icmp.c 973
```

FLAMEGRAPH 10

10.1 CALL-STACKS

In 1.2.2 FRAME POINTER it was described what a stack is and how it works. In this section one will use *perf* to capture or sample these *call-stacks* and visualize the in a very specific way, namely in a *flamegraph*.

But first lets print out some example call-stacks from a running application and deduce how to interpret a *flamegraph*. For this the `eu-stack`[1] tool will be used for examination. It shows call-stacks from an application currently running.

10.1 For Java use `jstacks`

The application used here is `pi` from the *gmpbench* suite. It calculates the number π up to the, as an argument given, decimal places.

```
#  LD_SHOW_AUXV=1 ./pi 1000
AT_SYSINFO_EHDR: 0x3ff8e6fe000
AT_HWCAP:    esan3 zarch stfle msa ldisp eimm dfp edat etf3eh highgprs te vx
AT_PAGESZ:       4096          /* System page size              */
AT_CLKTCK:       100           /* Program headers for program   */
AT_PHDR:         0x2aa0cc80040 /* Frequency of times()          */
AT_PHENT:        56            /* Size of program header entry  */
AT_PHNUM:        9             /* Size of program header entry  */
AT_BASE:         0x3ff8e680000 /* Size of program header entry  */
AT_FLAGS:        0x0           /* Flags                         */
AT_ENTRY:        0x2aa0cc81c58 /* Flags                         */
AT_UID:          0             /* Real uid                      */
AT_EUID:         0             /* Effective uid                 */
AT_GID:          0             /* Real gid                      */
AT_EGID:         0             /* Effective gid                 */
AT_SECURE:       0             /* Treat executable securely     */
AT_RANDOM:       0x3ffca77e3bc /* Address containing a random value */
AT_EXECFN:       ./pi          /* Pathname used to execute program  */
AT_PLATFORM:     z13           /* Hardware platform             */
```

Calculate 1000 decimal places for the number π using Chudnovsky's algorithm and show info about the ELF auxiliary vector. ELF auxiliary vectors are a mechanism to transfer certain kernel level information to the user processes

During the runtime of `pi` one can call `` eu-stack -p `pidof pi` `` periodically e.g. every 1/2 seconds to simulate the call-stack sampling of `perf record -F2` with a frequency of *2 Hz*. This way one can manually collect 2 call-stacks per second.

Here are 3 call-stack samples from the beginning of `pi`.

```
PID 18770 - process                       PID 18770 - process                       PID 18770 - process
TID 18770:                                TID 18770:                                TID 18770:
#0  0x00000000000403bcc build_sieve        #0  0x00000000000403bc3 build_sieve        #0  0x00000000000403bff build_sieve
#1  0x00000000000403d59 picomp             #1  0x00000000000403d59 picomp             #1  0x00000000000403d59 picomp
#2  0x000000000004007db main               #2  0x000000000004007db main               #2  0x000000000004007db main
#3  0x000000000042cbe6 generic_start_main #3  0x000000000042cbe6 generic_start_main #3  0x000000000042cbe6 generic_start_main
#4  0x000000000042ce6e __libc_start_main  #4  0x000000000042ce6e __libc_start_main  #4  0x000000000042ce6e __libc_start_main
#5  0x0000000000400dea _start             #5  0x0000000000400dea _start             #5  0x0000000000400dea _start

# Use addr2line to find out at which source code line the instruction pointer pointed
# to during execution when pi was sampled

addr2line -a 0x00000000000403bcc -e ./pi  addr2line -a 0x00000000000403bc3 -e ./pi  addr2line  -a 0x00000000000403bff -e ./pi
0x00000000000403bcc                       0x00000000000403bc3                       0x00000000000403bff
gmpbench-0.2/pi.c:485                     gmpbench-0.2/pi.c:484                     gmpbench-0.2/pi.c:488

485:   s[i/2].fac = i;                    484:  if (s[i/2].fac == 0) {              488:for (j=i*i, k=i/2; j<=n; j+=i+i, k++) {
```

The addresses for `build_sieve` can be and are different, because this was the active stack frame, when sampled, and the address is the *instruction pointer* at that time. To know where in the code one was exactly when sampled, `addr2line` can be used

Now to visualize the call-stacks one needs additional software that is available on *github*.

```
git clone https://github.com/brendangregg/FlameGraph.git
```

There are several tools to convert *call-stacks* from different source (*perf,dtrace,...*) to a format that `flamegraphs.pl` understands. First it will be done manually and then shown how to automate these steps with *perf*.

The input file for `flamegraphs.pl` is as follows, each line defines exactly one unique *call-stack* with the count of it. So e.g. in the case above the *folded call-stack* file should look like this.

```
# cat pi.man.folded
_start;__libc_start_main;generic_start_main;main;picomp;build_sieve 3
```

This file can now be used to generate the *flamegraph*.

```
cat pi.man.folded > flamegraph.pl > pi.man.svg
```

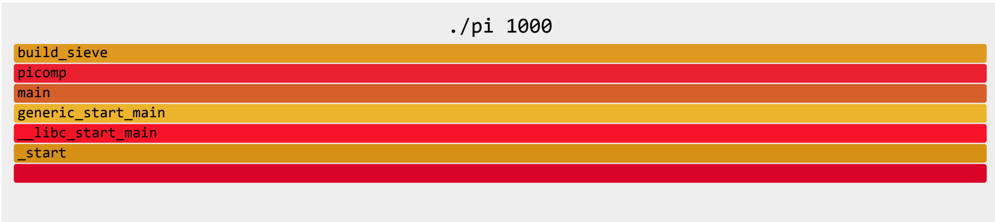


FIGURE 10.1 Flamegraph of 3 *call-stack* samples

10.2 PERF FLAMEGRAPH

The previous chapter showed how one can generate a *flamegraph* manually. Now let's see how to automate or generate a *flamegraph* with *perf*.

First, sample the workload with the `-call-graph=dwarf` switch.

Limit the frequency of sampling to your needs, otherwise *perf* could slow down your system to uselessness, there is a lot of CPU and memory load.

```
# perf record -F99 -call-graph=dwarf - ./pi 1000
```

Combine all the steps explained above to one line and generate the *flamegraph*.

```
# perf script | stackcollapse-perf.pl | flamegraph.pl > pi.svg
```

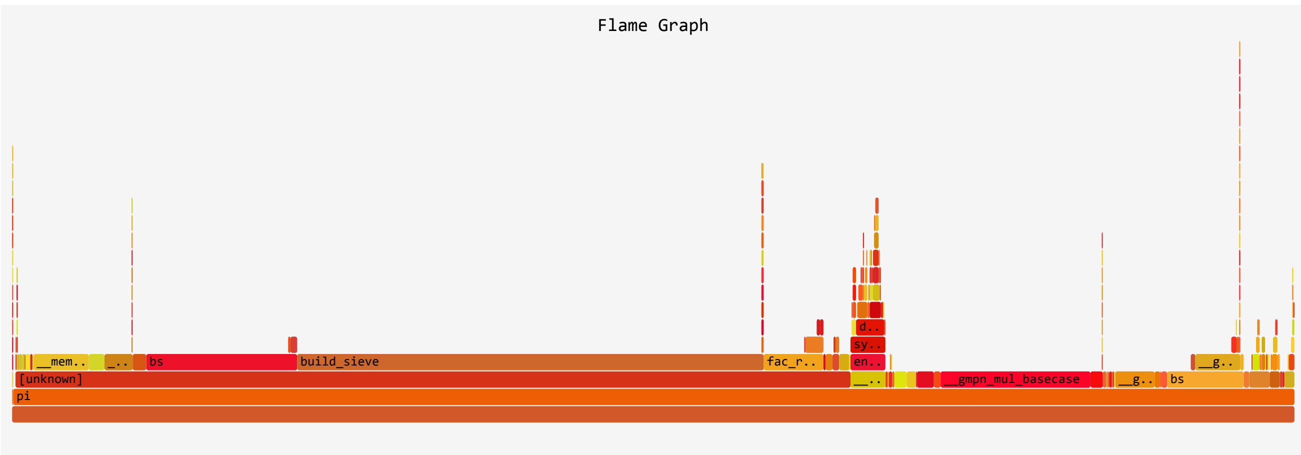


FIGURE 10.2 Complete Flamegraph of `pi`

The generated SVG is a interactive graphic, where one can search for a symbol or just activate a specific *call-stack* by clicking on it. The broader a bar is the more samples where gathered for this *call-stack* and hence more time was spent in it. Beware that *perf* was in sampling mode, which means, that *call-stack* that were really small in time, can but not necesserialy have to be sampled. If *call-stacks* are missing try to increase the sampling frequency.

10.3 DIFFERENTIAL FLAMEGRAPHS

Differential flame graphs visualize the differences between two performance profiles. It can be used e.g. for regression runs or to identify possible performance enhancement like compiler switches, tuning parameters and so on. Let's try to examine how the different optimization levels affect performance of the workload `pi`. Therefore the workload will be compiled with `-O1, -O2, -O3` and the corresponding *flamegraphs* for each run will be generated. The last step is to generate the *differential flamegraphs* to see the differences.

First generate the workloads.

```
# gcc -O3 -g pi.c -o pi.3 -static -lgmp -lm
# gcc -O2 -g pi.c -o pi.2 -static -lgmp -lm
# gcc -O1 -g pi.c -o pi.1 -static -lgmp -lm
```

Now for each optimization level sample the *call-stacks*

```
# perf record -F99 -call-graph=dwarf -o pi.1.data – ./pi.1 1000
# perf record -F99 -call-graph=dwarf -o pi.2.data – ./pi.2 1000
# perf record -F99 -call-graph=dwarf -o pi.3.data – ./pi.3 1000
```

The next step is to generate the folded stacks.

```
# perf script -i pi.1.data | stackcollapse-perf.pl > pi.1.folded
# perf script -i pi.2.data | stackcollapse-perf.pl > pi.2.folded
# perf script -i pi.3.data | stackcollapse-perf.pl > pi.3.folded
```

And as a last step the *differential flamegraphs* can be created.

```
difffolded.pl -n pi.1.folded pi.2.folded | flamegraph.pl –negate > pi.diff12.svg
difffolded.pl -n pi.2.folded pi.3.folded | flamegraph.pl –negate > pi.diff23.svg
```

Functions that are performing better are coloured blue and functions that perform worse are coloured red. The intensity shows how much better or worse the functions get in terms of performance.

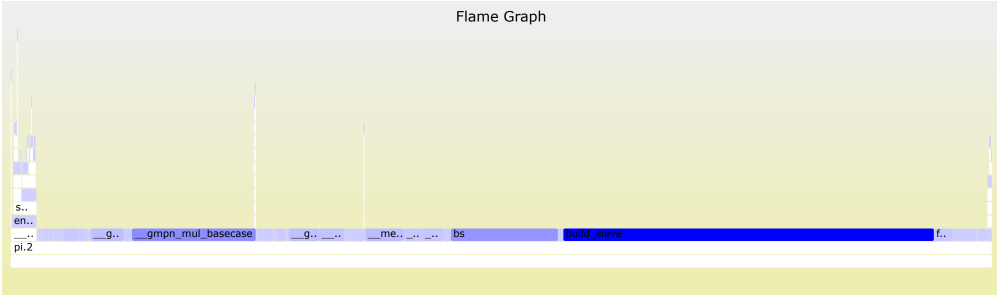
10.4 JAVA FLAMEGRAPH

The nice thing about *perf* with *Java* is that it's possible to not only sample *Java* like *Jinsight* but it can also sample the JVM with the *garbage collector*, userspace and the kernel. The complete stack can be examined and analyzed.

Since *Java* has a deep call hierarchy one should increase the maximum stack-frame count so that the complete *call-stack* is recorded.

```
# sysctl -w kernel.perf_event_max_stack=512
```

For the Just-in-time (JIT) compiled methods *perf* needs additional information to map the samples correctly to the symbols. *Perf* has a nifty mechanism for finding missing mappings (from instruction addresses to symbols/methods). When `perf report` encounters missing mappings it searches for maps in `/tmp/perf-<pid>.map`. This allows runtimes that generate code on the fly to supply dynamic symbol mappings to be used with *perf*. That is the file where the next tool will create mappings for JITcompiled methods.



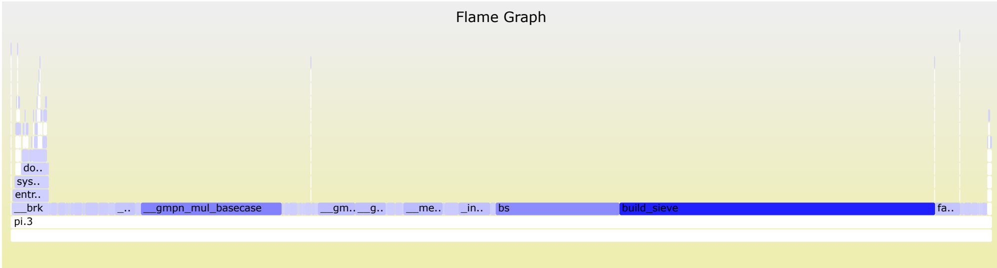


FIGURE 10.3 Differential Flamegraphs between compiler optimization levels

```
# git clone  https://github.com/jvm-profiling-tools/perf-map-agent.git
cd perf-map-agent
cmake .
make
```

There is another issue with *Java* namely the *frame-pointer*. The JVM compiler is ommiting the *frame-pointer* and not generating DWARF debuginfo to unwind the stack. Luckily since JDK Version 8 there is a new option to preserve the *frame-pointer*. Beware that preserving the *frame-pointer* the performance can degrade by up to 2%. A prerequisite for sampling *Java call-stacks* is the following option.

This option is not (yet) available in the IBM JDK

```
-XX:+PreserveFramePointer
```

For the next steps one will use the workload *SPECjbb2005* with the modified *Java* command line. To record and get the right maps for the workload the scripts in `perf-map-agent/bin` can be used for easy of use. But first export `JAVA_HOME` to the correct JDK .

```
export JAVA_HOME=/usr/lib/jvm/java-1.9.0-openjdk-amd64
```

First start the workload.

```
./SPECjbb2005Run 1073741824  /usr/lib/jvm/java-1.9.0-openjdk-amd64/bin  4096m   4096m   jitc  0   0 1  ""   0  0 "keep"
```

Now collect the *call-stacks*.

```
# ./perf-java-record-stack 10313
+++ readlink -f ./perf-java-record-stack
++ dirname /home/zkosic/git/perf-map-agent/bin/perf-java-record-stack
+ PERF_MAP_DIR=/home/zkosic/git/perf-map-agent/bin/..
+ PID=10313
+ '[' -z '' ']'
+ PERF_JAVA_TMP=/tmp
+ '[' -z '' ']'
+ PERF_RECORD_SECONDS=15
```

```
+ ’[’ -z ’’ ’]’
+ PERF_RECORD_FREQ=99
+ ’[’ -z ’’ ’]’
+ PERF_DATA_FILE=/tmp/perf-10313.data
+ echo ’Recording events for 15 seconds (adapt by setting PERF_RECORD_SECONDS)’
Recording events for 15 seconds (adapt by setting PERF_RECORD_SECONDS)
+ sudo perf record -F 99 -o /tmp/perf-10313.data -g -p 10313 - sleep 15
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.681 MB /tmp/perf-10313.data (3555 samples) ]
```

```
# perf script -i /tmp/perf-10313.data | stackcollapse-perf.pl -all | flamegraph.pl -color=java -hash > jbb.jsvg
```

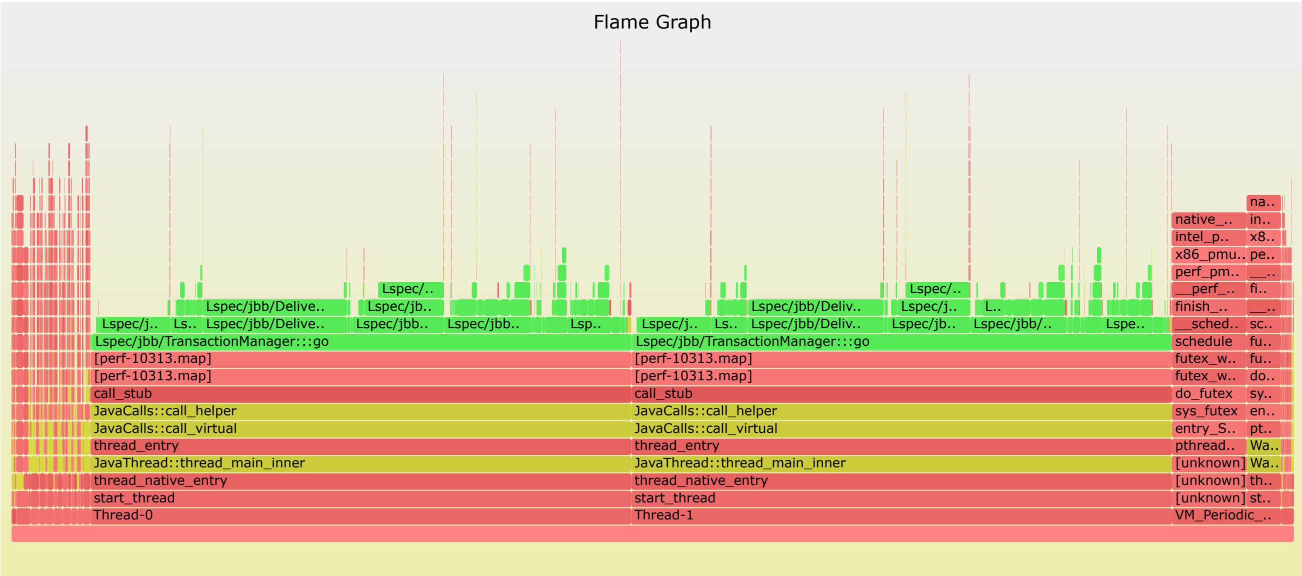


FIGURE 10.4 Flamegraph of *Java*, *JDK* and Kernel

The *green* colour represents *Java* functions the *yellowish* are functions from the *JDK* and last but not least the *red* ones are from the *Kernel*. Java code is inlined at a high degree and for that purpose one can supply `unfoldall` as an option to see even the inlined functions.

```
# export PERF_MAP_OPTIONS=unfoldall
# perf-map-agent/bin# ./perf-java-record-stack 11868
+++ readlink -f ./perf-java-record-stack
++ dirname /home/zkosic/git/perf-map-agent/bin/perf-java-record-stack
+ PERF_MAP_DIR=/home/zkosic/git/perf-map-agent/bin/..
+ PID=11868
+ ’[’ -z ’’ ’]’
+ PERF_JAVA_TMP=/tmp
+ ’[’ -z ’’ ’]’
+ PERF_RECORD_SECONDS=15
+ ’[’ -z ’’ ’]’
+ PERF_RECORD_FREQ=99
+ ’[’ -z ’’ ’]’
+ PERF_DATA_FILE=/tmp/perf-11868.data
+ echo ’Recording events for 15 seconds (adapt by setting PERF_RECORD_SECONDS)’
Recording events for 15 seconds (adapt by setting PERF_RECORD_SECONDS)
+ sudo perf record -F 99 -o /tmp/perf-11868.data -g -p 11868 - sleep 15
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.653 MB /tmp/perf-11868.data (3414 samples) ]
```

The result is stunning, additional to the visualization of the complete execution stack one has the inlined *Java* functions. Newer *Perf* version can handle even inlined C/C++ code.

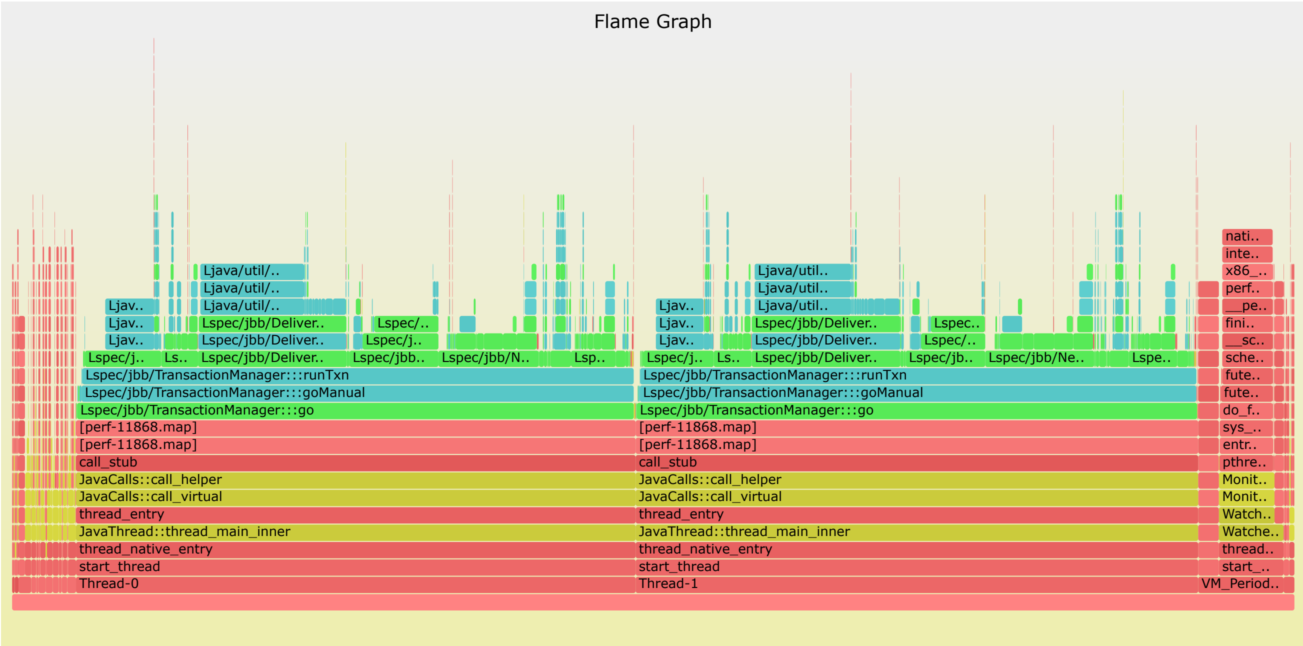


FIGURE 10.5 Flamegraph of *Java* (with inlined functions), *JDK* and Kernel

A PERLIMINARY FEATURES

A.1 BUILD TOP-NOTCH PERF

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/acme/linux -b perf/core
cd linux/tools/perf
make
./perf report –inline ...
```