

System/Networking performance analytics with perf

Hannes Frederic Sowa
<hannes@stressinduktion.org>

Prerequisites

- Recent Linux Kernel
 - CONFIG_PERF_*
 - CONFIG_DEBUG_INFO
- Fedora:
 - debuginfo-install kernel for vmlinux
 - Dwarves package for pahole

Debuginfo

- BuildID is mostly a SHA-1 checksum which gets placed into its own ELF section
- Mapping from buildid to binary via
/usr/lib/debug/.build-id/xx/xxxxxxxxxxxxx
or
~/.debug/.build-id/xx/xxxxxxxxxxxxx

Basic usage of perf

- perf top
- perf stat
 - count events happening during specific workload
- perf record
 - generate perf.data file in current directory with samples of the measurements
- Event descriptor
 - use 'perf list' to view possible events
 - perf evlist extracts perf events from perf.data

perf trace

- Live tracing
 - Replacement of strace
 - Much higher performance, as we don't need to do multiple kernel ↔ user space transitions anymore
 - Like strace:
 - `perf trace -e read,write <program>`

Analyzing data

- perf report
 - GUI to browse, inspect and annotate samples
 - Can inspect call graphs
- perf script
 - Without arguments presents easy to grep data
 - Can later on be used to process perf data with perl or python
- perf annotate
 - Source listing with annotated performance profiles

Transferring perf data

- `perf archive perf.data`

Now please run:

```
$ tar xvf perf.data.tar.bz2 -C ~/.debug
```

wherever you need to run 'perf report' on.

!root access to perf

- `/proc/sys/kernel/perf_event_paranoid`:

The `perf_event_paranoid` file can be set to restrict access to the performance counters.

2 only allow user-space measurements.

1 allow both kernel and user measurements (default).

0 allow access to CPU-specific data but not raw tracepoint samples.

-1 no restrictions.

- `echo -1 > /proc/sys/kernel/perf_event_paranoid`

Reasoning about performance

- Algorithmic complexity
- Memory access behaviour
 - Parallelism
 - Cross cpu Memory access
- Assumptions on networking traffic
 - Caching
 - Fast paths
- Raw instruction throughput
 - e.g. CPI (cycles per instructoin)

perf_event_open

- open fd to measure one particular event
 - Sampling
 - Data mostly gathered via mmap
 - Counting
 - Data mostly gathered via read
- Event types are either
 - Hardware (cycles, instructions, ...)
 - Software (cpu clock, context switches)
 - Integration into tracing
 - Caching events
 - Raw
 - Breakpoint events

Hardware events

- perf list hw
 - Sampling based on counters or frequency
 - -c specifies period to sample
 - -F specifies the frequency
 - perf evlist -v shows details about the perf_event_attr, like sample_freq
- Event / Interrupt is triggered and a sample is captured

Reasoning about performance

- Algorithmic complexity
- Assumptions on networking traffic
 - Caching
 - Fast paths
- Memory access behaviour
 - Parallelism
 - Cross cpu Memory access
- Raw instruction throughput
 - e.g. CPI (cycles per instruction)

Algorithmic complexity

- How can perf help?
 - Find given workload or benchmark
 - Can easily pinpointed by perf top or simple cycle counting in the kernel:
 - cycles:kpp
 - k enables kernel only counting (u for user space)
 - Additional p modifier change precise level
 - Intel: PEBS – Precise Event Based Sampling
 - AMD: IBS – Instruction Based Sampling
 - perf top -e cycles:kpp
 - If the region of code is identified proceed with analyzing the source
 - Pinpointing additional recurrences with perf can help, too (see “Assumptions on networking traffic”)
- Mostly needs to be solved by enhancing the algorithms / code
 - e.g. fib trie neatening
- Find best and worse case and optimize accordingly
- Sometimes can be worked around by caching
 - See section on Assumptions on networking traffic / patterns

Reasoning about performance

- Algorithmic complexity
- Assumptions on networking traffic
 - Caching
 - Fast paths
- Memory access behavior
 - Parallelism
 - Cross cpu Memory access
- Raw instruction throughput
 - e.g. CPI (cycles per instructoin)

Memory access behavior

Eric Dumazet's mlx4 memory access optimizations:

```
+  /* fetch ring->cons far ahead before needing it to avoid stall */  
+  ring_cons = ACCESS_ONCE(ring->cons);  
...  
+  /* we want to dirty this cache line once */  
+  ACCESS_ONCE(ring->last_nr_txbb) = last_nr_txbb;  
+  ACCESS_ONCE(ring->cons) = ring_cons + txbbbs_skipped;
```

Memory access behavior II

- Compilers do not tend to optimize memory access in e.g. large functions and optimize for CPU cache behavior
- Manual guidance is often needed
 - Pacing memory access across functions to allow CPU to access memory more parallel
 - Avoid RMW instructions

Memory access behavior III - struct ordering

- Code which access large structures tend to write to multiple cache lines
- Group members of structs so that specific code has to touch the least amount of cache lines
- Critical Word First / Early Restart
 - CPUs tend to read complete cache lines
 - Early Restart signals the CPU that data is available before complete cache line is read
 - Critical Word First allows the CPU to fetch the wanted data, even it is in the end of a cache line

Memory access behavior IV

- How can perf help?

perf list cache

- | | |
|-----------------------------|------------------------|
| - L1-dcache-loads | [Hardware cache event] |
| - L1-dcache-load-misses | [Hardware cache event] |
| - L1-dcache-stores | [Hardware cache event] |
| - L1-dcache-store-misses | [Hardware cache event] |
| - L1-dcache-prefetch-misses | [Hardware cache event] |
| - L1-icache-load-misses | [Hardware cache event] |
| - LLC-loads | [Hardware cache event] |
| - LLC-stores | [Hardware cache event] |
| - LLC-prefetches | [Hardware cache event] |
| - dTLB-loads | [Hardware cache event] |
| - dTLB-load-misses | [Hardware cache event] |
| - dTLB-stores | [Hardware cache event] |
| - dTLB-store-misses | [Hardware cache event] |
| - iTLB-loads | [Hardware cache event] |
| - iTLB-load-misses | [Hardware cache event] |
| - branch-loads | [Hardware cache event] |
| - branch-load-misses | [Hardware cache event] |
| - | |

Memory access behavior V

- `perf mem record <workload>`
- `perf mem report`
 - uses `'cpu/mem-loads/pp'` event by default
 - `perf mem -t store` record switches to `'cpu/mem-stores/pp'`
- (those kinds of events aren't documented properly:
basically you can find them in
`/sys/devices/cpu/events`)

Analyzing lock behaviour

- Needs kernel compiled with lockdep
 - perf lock record <workload>
 - Perf lock report

Raw counters

- Andi Kleen's pmu-utils test suite
- <https://github.com/andikleen/pmu-tools>
- Offcore events

- On AMD these are available via

```
perf record -e amd_nb/
```

Reasoning about performance

- Algorithmic complexity
- Memory access behaviour
 - Parallelism
 - Cross cpu Memory access
- Assumptions on networking traffic
 - Caching
 - Fast paths
- Raw instruction throughput
 - e.g. CPI (cycles per instruction)

Assumptions on networking traffic

- Examples:
 - xmit_more finally allows the kernel to achieve line rate speeds but the feature must be triggered
 - Receive offloading needs to see packet trains to aggregate sk_buffs
 - Routing caches should not be evicted on every processed data frame
 - Flow-sensitive packet steering should not cause increased cross-CPU memory traffic

Kprobes arguments

GRP : Group name. If omitted, use "kprobes" for it.
EVENT : Event name. If omitted, the event name is generated based on SYM+offs or MEMADDR.
MOD : Module name which has given SYM.
SYM[+offs] : Symbol+offset where the probe is inserted.
MEMADDR : Address where the probe is inserted.

FETCHARGS : Arguments. Each probe can have up to 128 args.
%REG : Fetch register REG
@ADDR : Fetch memory at ADDR (ADDR should be in kernel)
@SYM[+|-offs] : Fetch memory at SYM +|- offs (SYM should be a data symbol)
\$stackN : Fetch Nth entry of stack (N >= 0)
\$stack : Fetch stack address.
\$retval : Fetch return value.(*)
+|-offs(FETCHARG) : Fetch memory at FETCHARG +|- offs address.(**)
NAME=FETCHARG : Set NAME as the argument name of FETCHARG.
FETCHARG:TYPE : Set TYPE as the type of FETCHARG. Currently, basic types (u8/u16/u32/u64/s8/s16/s32/s64), "string" and bitfield are supported.

Examples to pinpoint xmit_more

- Find an applicable function candidate, guess:
 - `perf probe -F --filter dev*xmit*`
- We need to get hold on to the xmit_more flag:
 - `perf probe -L dev_hard_start_xmit`
- Which variables are available at that location?
 - `perf probe -V dev_hard_start_xmit:17`
- Finally adding the probe point:
 - `perf probe -a 'dhsx=dev_hard_start_xmit:17 ifname=dev->name:string xmit_more=next'`
- Record test and view results:
 - `perf record -e probe:* -aRg <workload>`
 - `perf script -G`

Examples to pinpoint xmit_more II

- Example for use with modules:

- `perf probe -v -m \`
`/usr/lib/debug/lib/modules/3.18.5-201.fc21.x86_64/kernel/net/mac80211/mac80211.ko.debug \`
`-a 'ieee80211_xmit ifname=skb->dev->name:string xmit_more=skb->xmit_more'`

- Return value probing

- `perf probe -a 'dev_hard_start_xmit%return retval=$retval'`

- User space probing works, too:

- `debuginfo-install <binary>`
 - `perf probe -x <binary>` should give same results

Gotchas

- Lot's of inlining:
 - noline define in kernel or `__attribute__((noinline))`
 - Sometimes needs a bit more code rearrangement
 - move code out of header
 - noninline per-object file wrapper
 - `EXPORT_SYMBOL`
- Add volatile variables to ease access to certain data

- Can also be achieved via registers

```
perf probe -a 'dev_hard_start_xmit+332 ifname=dev->name:string more txq'
```

Failed to find the location of more at this address.

Perhaps, it has been optimized out.

Error: Failed to add events.

Narf! So...

- `perf probe -a 'napi_gro_complete ifindex=skb->dev->name:string gro_count=+0x3c(%di):u16'`

results in:

perf script

```
irq/30-iwlwifi 498 [000] 6462.790978: probe:napi_gro_complete: (ffffff81649b30) ifindex="wlp3s0" gro_count=0x2
```

```
irq/30-iwlwifi 498 [000] 6462.795096: probe:napi_gro_complete: (ffffff81649b30) ifindex="wlp3s0" gro_count=0x2
```

Raw CPU counter

- Tables in CPU manuals
 - e.g. `cpu/event=xx,umask=xx/flags`
 - `/sys/bus/event_source/devices/<...>`
- Raw events: `r<event><umask>`
- Look up AMD manual to trace amd northbridge events with `-e 'amd_nb/event=0xxx,umask=0xxxx/flags'`

Routing cache expunge

- `print &init_net.ipv6.fib6_sernum`
 - `→ 0xffffffff81f1a5ec`
- `perf record -e \mem:0xffffffff81f1a5ec:rw ip -6 route add 2002::/64 dev lo`

perf script

- Sample scripts available
 - `perf script -l`
- Generate script based on perf.data file
 - `perf script -g perl/python`
- Run script on perf.data file
 - `perf script -s`

Reasoning about performance

- Algorithmic complexity
- Memory access behaviour
 - Parallelism
 - Cross cpu Memory access
- Assumptions on networking traffic
 - Caching
 - Fast paths
- Raw instruction throughput
 - e.g. CPI (cycles per instruction)

Raw instruction throughput

- Mostly compiler and architecture dependent
- Often marginal effects in performance improvement
 - Still, hot code can benefit a lot, e.g. crypto, hashing
- CPI cycles per instruction should be increased
- Mostly dependent on memory accesses
- Open up missed-optimization in gcc bugzilla?
- Architecture specific inline assembly?

Optimizing instruction throughput

- Balance code size (L1i cache pressure) vs. improvement
 - Performance decrease in other code possible
 - Even not related at all to the optimized code
- Mostly case by case optimizations, but some help of tools is possible
- If needed, provide assembly implementation but add `__builtin_constant_p` wrappers so gcc can still do constant folding if possible

Backup: Intel® Architecture Code Analyzer

Intel® Architecture Code Analyzer

- Static analyzer on assembly code
- Allows to analyze code for smaller functions in regard to CPU port exhaustion, stalls and latency
 - Reordering instructions
 - Picking different ones / missing optimization in compiler?
- Does not model memory access
- Some instructions cannot being modeled correctly, e.g. div
- Markers mark beginning and end of section to be analyzed:

`.byte 0x0F, 0x0B`

`movl $111, %ebx`

`.byte 0x64, 0x67, 0x90`

`movl $222, %ebx`

`.byte 0x64, 0x67, 0x90`

`.byte 0x0F, 0x0B`

Intel® Architecture Code Analyzer II

- Throughput mode (-analysis THROUGHPUT):

Throughput Analysis Report

.....

Block Throughput: 18.00 Cycles

Throughput Bottleneck: Port1

Port Binding In Cycles Per Iteration:

Port	0	-	DV	1		2	-	D	3	-	D	4		5		6		7
------	---	---	----	---	--	---	---	---	---	---	---	---	--	---	--	---	--	---

Cycles	7.3	0.0	18.0	9.0	9.0	9.0	9.0	0.0	7.3	7.3	0.0
--------	-----	-----	------	-----	-----	-----	-----	-----	-----	-----	-----

Intel® Architecture Code Analyzer III

	Num Of	Ports pressure in cycles											
•	Uops	0 - DV	1	2 - D	3 - D	4	5	6	7				
•	-----												
•	0*										mov rax, rdx		
•	1	0.7						0.3			cmp rsi, 0x40		
•	0F										jb 0x4b		
•	1	0.3					0.3	0.3			sub rsi, 0x40		
•	1	0.3					0.3	0.3			sub rsi, 0x40		
•	2^		1.0	1.0 1.0						CP	crc32 rax, qword ptr [rdi]		
•	2^		1.0		1.0 1.0					CP	crc32 rax, qword ptr [rdi+0x8]		
•	2^		1.0	1.0 1.0						CP	crc32 rax, qword ptr [rdi+0x10]		
•													

Intel® Architecture Code Analyzer III

- Latency Mode (-analysis LATENCY):

Latency Analysis Report

.....

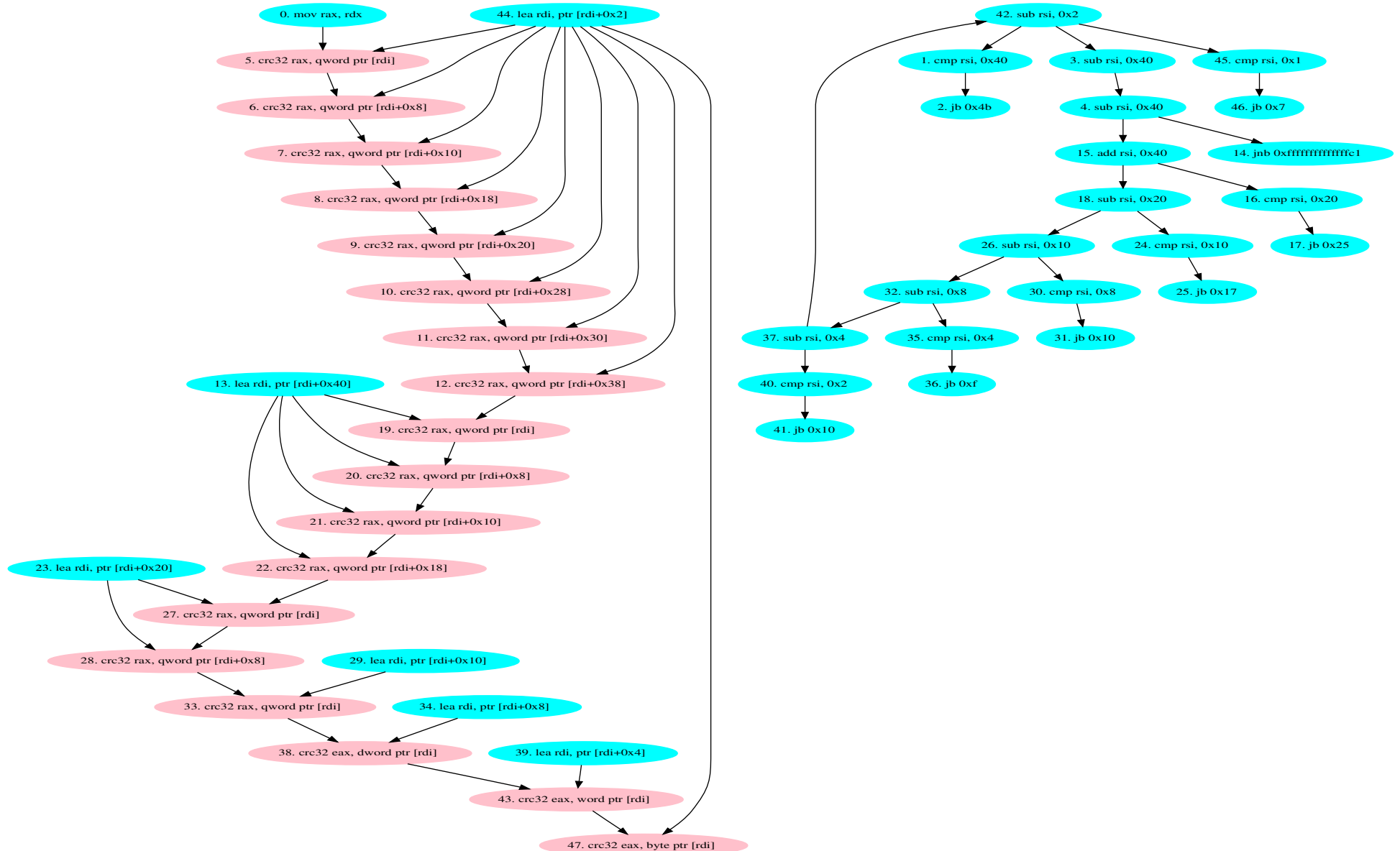
Latency: 60 Cycles

[...]

	3																sub rsi, 0x40
	4																sub rsi, 0x40
	5												1		CP		crc32 rax, qword ptr [rdi]
	6												1		CP		crc32 rax, qword ptr [rdi+0x8]
	7				1								1		CP		crc32 rax, qword ptr [rdi+0x10]
	8						1						1		CP		crc32 rax, qword ptr [rdi+0x18]
	9				1								2		CP		crc32 rax, qword ptr [rdi+0x20]

- Reports resource conflicts on critical paths (CP) and list of delays

Intel® Architecture Code Analyzer III



Thanks!

Questions?