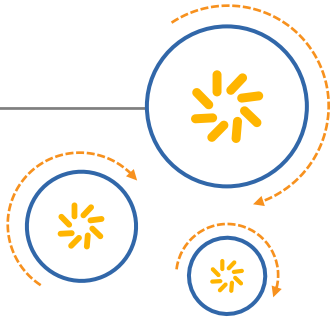
NOTICE REGARDING QUALCOMM ATHEROS, INC.

Effective June 2016, Qualcomm Atheros, Inc. (QCA) transferred certain of its assets, including substantially all of its products and services, to its parent corporation, Qualcomm Technologies, Inc. Qualcomm Technologies, Inc. is a wholly-owned subsidiary of Qualcomm Incorporated. Accordingly, references in this document to Qualcomm Atheros, Inc., Qualcomm Atheros, Atheros, QCA or similar references, should properly reference, and shall be read to reference, Qualcomm Technologies, Inc.

Qualcomm Atheros, Inc.



IPQ40x8 Profiling with perf

Application Note

80-Y9571-2 Rev. B

December 1, 2015

For additional information or to submit technical questions go to https://createpoint.qti.qualcomm.com/

Qualcomm Atheros, Inc.
1700 Technology Drive
San Jose, CA 95110
U.S.A.

Revision history

Revision	Date	Description
A	July 2015	Initial release
B	December 2015	Updated the following sections: ▪ Section 4.1 ▪ Section 4.8

Contents

Tables

1 Introduction

This application note assists software engineers who wish to use **`perf`** to understand the behavior of their software when running on the Cortex CPUs found within the IPQ40x8.

The Linux perf profiling tool can provide a huge amount of detail about the operation of software within a Linux system.

The official documentation for perf can be found at https://perf.wiki.kernel.org/.

This document discusses how perf can be configured and used with the Cortex CPUs found within the IPQ40x8 series of network processors.

perf is the standard software profiling tool that has been shipped with Linux since kernel version 2.6.31. It supports all of the major ISAs to which Linux has been ported and as such is now the standard profiling infrastructure used by most Linux developers.

The profiler provides the ability to analyze the performance of the kernel, kernel loadable modules (LKLMs), and user-space processes. For SMP systems it can profile either specific CPUs or all CPUs.

The perf system comprises several components. It uses the kernel's performance counters infrastructure, which is, in turn, managed by a user-space tool called perf.

2 How perf Works

perf operates by using the hardware performance counters found in almost all modern CPUs. These allow some small number of configured events within the CPU to be counted. When the counts exceed defined thresholds the CPU is interrupted.

The kernel processes the event counter interrupts and captures the state of the CPU. In some instances it will also attempt to back trace the call stack, so that details of the call sequence leading up to the sampled instruction can be understood too. It makes the data available to the user-space process that collects event sample data for either immediate presentation to a user or, more usefully, stores the data for subsequent processing.

The event sampling rate is relatively low so the total CPU load from the event sampling is low (typically less than 1%). The low sampling rate gives this form of profiling a light touch; it does not disrupt the normal execution flow.

3 Profiler Analysis

Profiling software with perf has some subtleties. Some of these are presented here.

3.1 Limitations of Statistical Sample Profiling

perf does not attempt to generate precise instruction traces. Instead it relies on running the same workload for prolonged periods of time in order to build up a statistically useful profile. To generate useful information, the system typically must be sampled for tens of seconds or even several minutes. During this time, any infrequently executed software may not be sampled, so profiling will show where most time is spent but not what the worst-case behavior may be.

To get a complete understanding of the performance of some software, it may be necessary to sample a number of event counters while executing the same workload. perf can only sample a small number of events at any one time, so, for these types of test, it is necessary to run the test workload several times, collecting different event data each time.

3.2 Synchronous and Asynchronous Events

CPU event counters increment when the CPU attempts to perform some specific action. In some cases these actions are synchronous with the execution of a particular instruction that triggers the event, but some events occur asynchronously. It is easy to interpret synchronous events but asynchronous events are trickier.

Examples of events that happen synchronously are CPU clock events (cycle counters), instruction counters, or specific instructions being executed. Examples of asynchronous events may include L2 cache line refill events or unaligned memory accesses.

Asynchronous event counting is made more difficult when the system being profiled has an out-of-order (OoO) execution CPU pipeline. In these pipelines, instructions that can be executed without relying on the results of earlier instructions may be executed before those earlier instructions where those earlier instructions may not have all of the data that they require. One consequence of this is that CPU events that are related to an earlier instruction that could not execute are often recorded as an event affecting some later instruction.

Consider the following pseudo-code for an OoO CPU:

```
load r1, [r10]              ; 1: load data from memory
add r2, r3, r4              ; 2: r2 = r3 + r4
add r5, r2, r1              ; 3: r5 = r2 + r1
store r2, [r11]        ; 4: store data to memory
```

In this example it is quite possible that the first instruction (the load) does not complete before the first add instruction. Instead if the values of r3 and r4 are already established then the add instruction can generate the result r2 without waiting for the load instruction to complete. The

second add instruction cannot execute without the load completing and so the next instruction that may complete executing may be the store. It is possible that the order in which instructions are retired is 2, 4, 1, 3.

For many OoO pipelines this has a consequence for event counting because many of the events that may be associated with the load instruction will actually be recorded against the first instruction that tries to use the result of that load; in this case instruction 3 (the second add). Even though instruction 3 does not access memory it will be likely that events such as cache misses, unaligned accesses or non-cached accesses would instead be recorded against it and not instruction 1.

One way to think about this is that events are often recorded by the victim instruction; the one that cannot make progress until the event completes.

While many event counts are simple to interpret, cycles counts can be the most complex. These are the most useful counters available as they capture how long it takes to execute specific code sequences, but it is possible that more than one event can be combined to reach a cumulative cycles count. Consider a simple variant of the earlier pseudo-code:

```
load r11, [r12]        ; 1: load data from memory
load r1, [r10]         ; 2: load data from memory
add r2, r3, r4         ; 3: r2 = r3 + r4
add r5, r2, r1         ; 4: r5 = r2 + r1
store r2, [r11]        ; 5: store data to memory
```

In this example instruction 3 may not only have to wait for instruction 1 to complete, but possibly also instruction 0 because the memory subsystem will queue the second load behind the first. While the data dependency has not changed the latency may be significantly higher.

Consider also another subtle interaction:

```
load r1, [r10]         ; 1: load data from memory
load r11, [r12]        ; 2: load data from memory
add r2, r3, r4         ; 3: r2 = r3 + r4
add r5, r2, r1         ; 4: r5 = r2 + r1
store r2, [r11]        ; 5: store data to memory
```

This version is the same as the previous but with instructions 1 and 2 swapped. Assuming that both of these load instructions take cache misses then this subtle change can have a dramatic impact on the execution order. The first version may retire the instructions in the order 3, 1, 5, 2, 4 while the second version is more constrained and may retire them in the order 3, 1, 4, 2, 5. This second version will probably execute slightly quicker than the first, however because the cache misses do not stall for the CPU pipeline quite as long.

3.3 Special Instructions

In addition to normal instructions, most CPUs have a number of instructions that handle tasks such as enabling and disabling interrupts or implementing memory barriers that guarantee memory access ordering. These instructions often take special roles as victim instructions when profiling.

Memory barrier instructions typically cause the CPU pipeline to stall until all pending memory accesses of a specific type have completed. In complex CPU systems, there may be many such memory accesses in flight, so the instruction may record the cycle count costs of all of those memory accesses completing.

Some CPUs use non-maskable interrupts to implement the event counter interrupts and therefore can profile interrupt service routines (ISRs). Many more CPUs do not, however, and are therefore not able to profile ISRs. In these CPUs any events that trigger during the ISR will in fact trigger an interrupt when interrupts are re-enabled, potentially with an interrupt enable instruction. In these cases those instructions become “victim instructions” that can see large numbers of events recorded against them.

3.4 Superscalar Execution

Many CPUs now support superscalar execution. In these designs, the CPU is able to execute more than one instruction in a single clock cycle by using multiple parallel instruction pipelines. These designs may actually be able to retire significantly more instructions in a single cycle, because loads and stores may retire along with associated ALU operations.

When the CPU is interrupted for an event, there will generally only be a single program counter (PC) value to represent any of the instructions that were retired in that clock cycle; so, all events are attributed to that one PC, even though (in the case of IPQ40xx) as many as seven instructions might be retired together. Care must be taken to identify which instruction caused any specific event counter to trigger an interrupt.

4 Using perf in QSDK

4.1 Configuring the Profiler

The **perf** infrastructure is not enabled by default within QSDK, however it is easy to configure for use. Run **make menuconfig** and ensure that these options are enabled:

- Global build settings -> Compile the kernel with profiling enabled
- If tracepoint events are required enable the following config option
  - Global build settings-> Compile the kernel with tracing support
- Global build settings -> Compile the kernel with symbol table information
- Development -> perf
- Base system->busybox->coreutils->expand

After rebuilding QSDK, the software will be set up to enable profiling and will incorporate the necessary user-space software components. To copy files to and from a target system it is worth ensuring that `scp` is available too.

4.2 Testing the Profiler

After booting the target system with the new QSDK image, it is useful to perform a quick test.

The following command should run for 5 seconds and then report summary statistics:

```
perf stat –a –e cycles sleep 5
```

4.3 Quirks of QSDK/OpenWrt

QSDK and OpenWrt have a minor quirk that affects the ability to profile kernel-loadable modules (LKLMs). **perf** expects to find all **.ko** files in the directory **/lib/modules/**<kernel version>**/kernel**, but QSDK and OpenWrt place them in **/lib/modules/kernel**.

If any profiling will involve LKLMs, a new directory must be created (e.g. for kernel version 3.4.43 this would be **/lib/modules/3.14.43/kernel**), and the **.ko** files must be copied to that directory.

4.4 Modes of Profiling

perf supports a number of different profiling modes, ranging from simple statistic sumaries, top-like live statistics to detailed statistic generation that enable detailed offline analysis.

The test step above introduced one of the simplest forms of operation using **perf stat**. This collects profiler statistics for one or more event counters for the duration of execution of some user-space

process. By using the application **sleep**, it's possible to profile for a fixed time duration and where the process being used exerts almost no inFluence on the behavior of the rest of the system.

The **perf** command takes a number of standard parameters. (For details, see **man perf**.)
The most commonly used **perf** commands are:

- **perf stat** - Present summary of event counts
- **perf record** - Record event details for later analysis
- **perf report** - Take recorded event data and summarize by function
- **perf annotate** - Take recorded event data and summarize by instruction

4.5 Example of perf stat

```
perf stat –e cycles,instructions –a sleep 120
```

This command takes a capture of the **cycles** and **instructions** events (**-e**) across all (**-a**) of the system for the duration of the **sleep 120** command. The sleep command is used as a no-op process that takes almost no CPU time; it is ideal for profiling kernel operations.

perf stat is important, as it can be used to calibrate the rates of particular events with each other. Typically **cycles** will be the highest count rate, as it counts CPU cycles while almost all other events occur much less frequently. By knowing how many events occur in a specific period of time it is possible to work out how important or unimportant the particular event may be. For example, if a test showed 10 M D-cache misses per second, which would have a significant impact on performance. However, 10 k D-cache misses per second would probably be uninteresting. In general it is strongly recommended to use **perf stat** to establish these things before running more complex forms of profiling.

4.6 Example of perf record

```
perf record –a –g –e cycles sleep 5
```

This command records the **cycles** events **(-e**) across all of the system for the duration of the **sleep 120** command. Backtracing is enabled for each sample (**-g**). This generates a data file, "**perf.data**" in the target system's file system (in the current directory).

4.7 Example perf report

```
perf report > report.cycles.txt
```

This command takes the "perf.data" file from a previous **perf record** command and generates a text file called "**report.cycles.txt**" (although the name can be any valid filename). This gives a function-level breakdown of the system's behavior while **perf record** was running and shows where the cycles events occurred.

A useful variant of this command is:

```
perf report --sort symbol > report.cycles.sort.txt
```

This sorts all symbols, and groups them together rather than presenting each one its own unique process context. For profiling kernel operations, this is often easier to understand.

Then execute the command:

```
cat report.cycles.txt or cat report.cycles.sort.txt
```

An example of part of the output for **perf report** (but without the **–g** option being used for the **perf record**) is shown here:

```
#
# Samples: 11K of event 'cycles'
# Event count (approx.): 1236046824
#
# Overhead                                                  Symbol
# ........  ..................................................
#
    16.90%  [k] LzmaEnc_MemEncode
            |
            --- jffs2_lzma_compress
                jffs2_selected_compress
                jffs2_compress
                jffs2_write_inode_range
                jffs2_write_end
                generic_file_buffered_write
                __generic_file_aio_write
                generic_file_aio_write
                do_sync_write
                vfs_write
                sys_write
                ret_fast_syscall
                0xb6e1254c
                0xbedd7bf4

    15.24%  [k] _memset_io
            |
            --- sps_bam_pipe_connect
                sps_rm_state_change
                sps_connect
                msm_spi_bam_pipe_connect
               |
               |--98.32%-- msm_spi_transfer_one_message
               |          spi_pump_messages
               |          kthread_worker_fn
               |          kthread
               |          ret_from_fork
               |
                --1.68%-- msm_spi_bam_pipe_flush
                          msm_spi_process_transfer
                          msm_spi_transfer_one_message
                          spi_pump_messages
                          kthread_worker_fn
                          kthread
                          ret_from_fork
    10.60%  [k] Bt4_MatchFinder_GetMatches
            |
            |--99.76%-- ReadMatchDistances
            |          LzmaEnc_MemEncode
            |          jffs2_lzma_compress
            |          jffs2_selected_compress
```

```
                |          jffs2_compress
                |          jffs2_write_inode_range
                |          jffs2_write_end
                |          generic_file_buffered_write
                |          __generic_file_aio_write
                |          generic_file_aio_write
                |          do_sync_write
                |          vfs_write
                |          sys_write
                |          ret_fast_syscall
                |          0xb6e1254c
                |          0xbedd7bf4
                 --0.24%-- [...]

      8.67%  [k] msm_spi_process_transfer
                |
                --- msm_spi_transfer_one_message
                    spi_pump_messages
                    kthread_worker_fn
                    kthread
                    ret_from_fork

      7.48%  [k] _raw_spin_unlock_irqrestore
                |
                |--37.99%-- spi_async_locked
                |          spi_sync
                |          |
                |          |--98.31%-- spi_write_then_read
                |          |          |
                |          |          |--99.74%-- read_sr.isra.8
                |          |          |          wait_till_ready
                |          |          |          |
                |          |          |          |--98.39%-- m25p80_read
                |          |          |          |          part_read
                |          |          |          |          mtd_read
                |          |          |          |
jffs2_erase_pending_blocks
                |          |          |          |
jffs2_garbage_collect_pass
                |          |          |          |
jffs2_garbage_collect_thread
                |          |          |          |          kthread
                |          |          |          |          ret_from_fork
                |          |          |          |
                |          |          |           --1.61%-- m25p80_write
                |          |          |                     mtd_writev
                |          |          |                     jffs2_flash_writev
                |          |          |                     jffs2_write_dnode
                |          |          |
jffs2_write_inode_range
                |          |          |                     jffs2_write_end
                |          |          |
```

```
generic_file_buffered_write
            |          |          |
__generic_file_aio_write
            |          |          |
generic_file_aio_write
            |          |          |                      do_sync_write
            |          |          |                      vfs_write
            |          |          |                      sys_write
            |          |          |                      ret_fast_syscall
            |          |          |                      0xb6e1254c
            |          |          |                      0xbedd7bf4
            |          |           --0.26%-- [...]
            |          |
            |          |--1.31%-- m25p80_read
            |          |          part_read
            |          |          mtd_read
            |          |          jffs2_erase_pending_blocks
            |          |          jffs2_garbage_collect_pass
            |          |          jffs2_garbage_collect_thread
            |          |          kthread
            |          |          ret_from_fork
            |           --0.38%-- [...]
            |
            |--21.03%-- msm_spi_transfer_one_message
            |          spi_pump_messages
            |          kthread_worker_fn
            |          kthread
            |          ret_from_fork
            |
            |--5.87%-- __pm_runtime_suspend
            |          msm_spi_unprepare_transfer_hardware
            |          spi_pump_messages
            |          kthread_worker_fn
            |          kthread
            |          ret_from_fork
            |
            |--5.27%-- put_local_resources
            |          msm_spi_transfer_one_message
            |          spi_pump_messages
            |          kthread_worker_fn
            |          kthread
            |          ret_from_fork
            |
            |--5.04%-- sps_register_event
            |          msm_spi_bam_pipe_connect
            |          |
            |          |--96.70%-- msm_spi_transfer_one_message
            |          |          spi_pump_messages
            |          |          kthread_worker_fn
            |          |          kthread
            |          |          ret_from_fork
            |          |
```

```
|            --3.30%-- msm_spi_bam_pipe_flush
|                      msm_spi_process_transfer
|                      msm_spi_transfer_one_message
|                      spi_pump_messages
|                      kthread_worker_fn
|                      kthread
|                      ret_from_fork
|
|--4.12%-- __irq_put_desc_unlock
|          |
|          |--51.95%-- __disable_irq_nosync
|          |           disable_irq
|          |           put_local_resources
|          |           msm_spi_transfer_one_message
|          |           spi_pump_messages
|          |           kthread_worker_fn
|          |           kthread
|          |           ret_from_fork
|          |
|           --48.05%-- enable_irq
|                      get_local_resources
|                      msm_spi_transfer_one_message
|                      spi_pump_messages
|                      kthread_worker_fn
|                      kthread
|                      ret_from_fork
|
|--3.29%-- synchronize_irq
|          |
|          |--60.06%-- sps_rm_state_change
|          |           sps_disconnect
|          |           msm_spi_transfer_one_message
|          |           spi_pump_messages
|          |           kthread_worker_fn
|          |           kthread
|          |           ret_from_fork
|          |
|           --39.94%-- put_local_resources
|                      msm_spi_transfer_one_message
```

As can be seen, the unsorted and without call graph (-g option) version shows the same kernel functions (denoted with a **k**) a number of times, each with a different process context. The sorted version is much easier to understand:

```
#
# Samples: 432K of event 'cycles'
# Event count (approx.): 63812581177
#
# Overhead       Command           Shared Object
Symbol
# ........  ............  ......................
.................................................
#
    31.12%          perf  [kernel.kallsyms]      [k] LzmaEnc_MemEncode
    18.29%          perf  [kernel.kallsyms]      [k]
Bt4_MatchFinder_GetMatches
     8.14%          perf  [kernel.kallsyms]      [k] MatchFinder_Init
     6.38%          spi0  [kernel.kallsyms]      [k] _memset_io
     6.26%          perf  [kernel.kallsyms]      [k]
LitEnc_GetPriceMatched
     3.03%          spi0  [kernel.kallsyms]      [k]
msm_spi_process_transfer
     2.69%          perf  [kernel.kallsyms]      [k] GetPureRepPrice
     2.06%          spi0  [kernel.kallsyms]      [k]
_raw_spin_unlock_irqrestore
     1.71%       swapper  [kernel.kallsyms]      [k] tick_nohz_idle_enter
     1.55%          spi0  [kernel.kallsyms]      [k] bam_pipe_init
     1.35%       swapper  [kernel.kallsyms]      [k] arch_cpu_idle
     1.28%          perf  [kernel.kallsyms]      [k] ReadMatchDistances
     1.27%       swapper  [kernel.kallsyms]      [k] _raw_spin_unlock_irq
     1.22%          perf  [kernel.kallsyms]      [k] RangeEnc_EncodeBit
     0.76%          perf  [kernel.kallsyms]      [k] RcTree_GetPrice
     0.61%          spi0  [kernel.kallsyms]      [k]
msm_spi_transfer_one_message
     0.57%       swapper  [kernel.kallsyms]      [k] tick_nohz_idle_exit
     0.56%          perf  [kernel.kallsyms]      [k]
_raw_spin_unlock_irqrestore
     0.56%          spi0  [kernel.kallsyms]      [k] _raw_spin_unlock_irq
     0.55%          perf  [kernel.kallsyms]      [k] MatchFinder_MovePos
     0.41%          spi0  [kernel.kallsyms]      [k] bam_pipe_exit
     0.40%          perf  [kernel.kallsyms]      [k] LitEnc_GetPrice
     0.30%          spi0  [kernel.kallsyms]      [k] sps_rm_state_change
     0.29%          spi0  [kernel.kallsyms]      [k] __memzero
     0.28%          perf  [kernel.kallsyms]      [k] _raw_spin_unlock_irq
     0.28%          spi0  [kernel.kallsyms]      [k] bam_pipe_set_irq
     0.26%          spi0  [kernel.kallsyms]      [k] __aeabi_uidiv
     0.26%          spi0  [kernel.kallsyms]      [k] sps_bam_pipe_connect
```

This example also demonstrates an aspect of profiling that was described earlier. A number of the (apparently) more expensive functions are actually not responsible for the operations that cause their cycle counts to be so high. Instead, other functions caused memory bus accesses that do not stall the system until the “victim” function executes.

4.8 Example of perf annotate

```
perf annotate –k vmlinux > annotate.cycles.txt
```

This command takes the "perf.data" file from a previous **`perf record`** command and generates a text file called "**`annotate.cycles.txt`**" (although the name can be any valid filename).

This gives a instruction-level breakdown of the system's behavior while **`perf record`** was running and shows where the cycles events occurred.

Note the use of the **`-k`** option here; **`perf annotate`** requires a kernel ELF file in order to operate. The ELF file required is the "**`vmlinux`**" file found in the kernel build directory **`qsdk/bin/ipq806x/debug/vmlinux`**. The easiest way to get this to the target system is to copy it from the build system using **`scp`**.

An example of the output can be seen below:

```
Percent |     Source code & Disassembly of vmlinux
------------------------------------------------
         :
         :
         :
         :    Disassembly of section .text:
         :
         :    c0018914 <arm_dma_unmap_page>:
    1.49 :    c0018914:       ldr     r0, [pc, #48]   ; c001894c
<arm_dma_unmap_page+0x38>
    0.00 :    c0018918:       lsr     ip, r1, #12
    0.00 :    c001891c:       push    {r4}            ; (str r4, [sp, #-4]!)
   78.87 :    c0018920:       ldr     r0, [r0]
    0.00 :    c0018924:       cmp     r0, #0
    0.00 :    c0018928:       lsrne   r4, r1, #28
    0.00 :    c001892c:       lsl     r1, r1, #20
    0.00 :    c0018930:       addne   r0, r0, r4, lsl #3
    0.00 :    c0018934:       lsr     r1, r1, #20
   17.53 :    c0018938:       ldr     r0, [r0]
    0.00 :    c001893c:       bic     r0, r0, #3
    2.11 :    c0018940:       add     r0, r0, ip, lsl #5
    0.00 :    c0018944:       pop     {r4}
    0.00 :    c0018948:       b       c0018868 <__dma_page_dev_to_cpu>
    0.00 :    c001894c:       .word   0xc1afaf40
```

The output demonstrates a number of things described earlier. Many functions appear to take 0.00 cycles, but these may in fact just be paired with other instructions that were executed in the same CPU cycle. It is also clear that some instructions appear to be very expensive, but these are having to deal with long-latency memory operations.

5 Performance Counters

The IPQ40xx features a number of standard ARMv7 performance counters. Standard ARM Cortex-A7 performance counter details can be found in ARMv7 CPU data available from ARM's website. A good source of data is the *Cortex-A7 MPCore Technical Reference Manual*.

The **`perf`** documentation describes how different types of event counter can be selected by using an **`rNN`** or **`rNNN`** value to the **`-e`** parameter. Key additional counters for the Cortex CPU are shown in Table 1.

Table 1 Additional counters for ARM-A7

rNN	Description
r1	Instruction fetch that causes a refill
r2	Instruction fetch that causes a TLB refill
r3	Data read or write operation that causes a refill
r4	Data read or write operation that causes a cache access
r5	Data read or write operation that causes a TLB refill
r6	Data read architecturally executed
r7	Data write architecturally executed
r8	Instruction architecturally executed
r9	Exception taken. Counts the number of exceptions architecturally taken
Ra	Exception return architecturally executed
Rb	Change to ContextID retired
Rc	Software change of PC
Rd	Immediate branch architecturally executed
Re	Procedure return (other than exception returns) architecturally executed
Rf	Unaligned load-store
r10	Branch mispredicted/not predicted
r11	Cycle counter
r12	Branches or other change in program flow that could have been predicted by the branch prediction resources of the processor
r13	Data memory access
r14	Instruction Cache access
r15	Data cache eviction
r16	Level 2 data cache access
r17	Level 2 data cache refill
r18	Level 2 data cache write-back