Enabling Raspberry Pi Performance Counter Support on Linux perf_event

Chad Paradis
University of Maine
chad.paradis@umit.maine.edu

Vincent M. Weaver
University of Maine
vincent.weaver@maine.edu

ABSTRACT

The Raspberry Pi is a low-cost, low-power, embedded ARM platform designed for use as an educational tool. The ARMv6 processor core included on the Raspberry Pi includes support for hardware performance counters (low-overhead registers that can provide detailed architectural performance measurements). Support for these counters is available for ARM Linux via the perf_event interface, but not enabled by default for the Raspberry Pi.

In this paper we investigate why the counters were not enabled, describe what steps are needed to enable them, and then validate the results to ensure they are working. We contributed the patches needed to enable the counters to upstream Linux maintainers so that support will be available by default for all users.

1. INTRODUCTION

The Raspberry Pi is a low-cost, low-power, ARM-based computer capable of running many distributions of the Linux operating system. At $35, the Raspberry Pi is an inexpensive introduction to modern embedded systems, and widely used as an educational tool.

Understanding performance and power on modern systems is difficult, as advanced processors do all sorts of extra work behind the scenes to obtain maximum performance. Even moderately complex processor models have difficulty capturing this. To truly understand a program's behavior, actual measurements must be taken. Most modern processors have hardware performance counters that provide low-overhead access to system performance (including information such as elapsed cycles, total instruction counts, cache misses, and branch mispredictions).

Although the BCM2835 CPU included in the Raspberry-Pi has ARM1176 ARMv6 compatible performance counters, by default they are not exported by the Linux perf_event performance counter subsystem. There are two barriers to this support: the first is notifying the kernel that such support is there, the second is the lack of a usable overflow interrupt. The lack of overflow interrupt will prevent the use of sampled event measurements, but total aggregate counts can still be measured. It is also possible to gather sampled events using a software workaround described later.

We detail what changes are needed to get support working on modern Linux kernels. We contribute these changes back to the upstream Linux developers so all users can have counter access support. In addition we validate the counters to be sure the results match expected values.

2. BACKGROUND

The Raspberry Pi comes in two flavors: model A and model B. Commonalities between the two models include an HDMI port, Composite RCA, 3.5 mm audio jack, SD card slot, USB port, and a flexible GPIO port. Both models take up an area about the size of a credit card and consume roughly 3 Watts of power under load.

Of greater interest, both models feature a Broadcom BCM2835 SoC which implements ARM's ARM1176JZ-F processor [1]. ARM1176JZ-F is based on the 32-bit RISC ARMv6 architecture. The SoC is clocked at 700MHz (but can be overclocked to 800MHz). A Dual Core VideoCore IV Multimedia Co-Processor is present in both models to relieve the CPU from having to do any graphics processing.

In addition to those features listed above, the model B also includes an Ethernet port and 512 MB of RAM (twice the memory of the model A). Due to the greater set of features, most importantly the Ethernet port, the model B is used in all parts of this paper and is hereafter referred to as Raspberry Pi, Pi, or RPI. The significance of the Ethernet port is simply a matter of convenience and not a necessity. Internet access allows the Pi to run in headless mode (i.e. without a display) and allows easy remote access via SSH (Secure Shell).

2.1 Linux perf_event

Performance counter support is via the perf_event interface which was introduced with the 2.6.31 kernel by Gleixner and Molnar [5]. More on the interface can be found here [6].

The primary tool for accessing performance counters is the `perf` tool that is included with the Linux kernel source.

Running the perf utility on a kernel prior to Linux 3.15:

```
$ perf stat ls
```

does not yield the desired results. What is expected is for useful information such as the number of instructions, branches, and branch-misses to be displayed to the user. Instead, something similar to the following is reported,

Listing 1: Unmodified perf Utility Output

```
 Performance counter stats for 'ls':

          9.418000 task-clock # 0.469 CPUs utilized
                 6 context-switches # 0.637 K/sec
                 0 cpu-migrations # 0.000 K/sec
               170 page-faults # 0.018 M/sec
    <not supported> cycles
    <not supported> stalled-cycles-frontend
    <not supported> stalled-cycles-backend
    <not supported> instructions
    <not supported> branches
    <not supported> branch-misses

       0.020098000 seconds time elapsed
```

Our goal is to get this reporting proper values.

2.2 ARM1176JZ-F Performance Monitoring

The ARM1176JZ-F performance monitoring system is described in the processor's technical reference manual [1] Section 3.2.51, titled *Performance Monitor Control Register*.

The hardware interface involves several registers including three counters and a control register. The three counters are *Cycle Counter Register*, *Count Register 0*, and *Count Register 1*. Each counter is 32 bits wide. The cycle count register simply provides access to core clock cycles of the processor. Count registers zero and one are used to access a large range of performance information including translation lookaside buffer misses, branch prediction information, stalls and more. A complete list of events is available in Table 3-137 of the technical reference manual. The events monitored by the counter registers are determined by bits in the *Performance Monitor Control Register*.

The BCM2835 documentation [2] is unclear about the status of the performance monitoring unit (PMU) interrupt. Dom Cobley (an engineer working for the Raspberry Pi foundation) confirms [3] that the nPMUIRQ PMU interrupt is not exported on the BCM2835 SOC.

3. ADDING KERNEL SUPPORT

To find the source of the missing perf_event support we started with the source code for the `perf` tool that is included with the Linux kernel. Searching for the *<not supported>* string reveals a few useful hits under *tools/perf/*. The errors returned can be traced back to the ARMv6 performance event implementation in the kernel.

The ARM perf_event support resides in the *arch/arm/kernel/* directory in several files with the naming scheme *perf_event*.c*. For an ARMv6 CPU as found in the Raspberry-pi the files of interest are *perf_event_v6.c* and *perf_event_cpu.c*. *perf_event_cpu.c* contains initialization code for the performance event system and *perf_event_v6.c* contains the interrupt handler function dealing with overflows. It is in these two files that most of the changes will be made.

3.1 PMU Discovery

The first task needed to be done is have the kernel detect the ARM 1176 PMU available on the chip.

For older kernels, including the one supported by the Raspberry-pi foundation, this involves hard coding the detection into the kernel via the platform model of detection.

On newer upstream kernels the platform support is being phased out and instead support is added via a device tree [4]. We describe both methods for adding support.

3.1.1 Platform Model

On kernels provided by the raspberry-pi foundation, the pi specific hardware is initialized as part of the mach-bcm2708 support. This code is maintained outside of the main Linux tree, it is not included in upstream kernels.

The file *arch/arm/mach-bcm2708/bcm_2708.c* contains Raspberry Pi initialization routines for things such as the GPIO module, audio, and system clocks. Absent from this is a call to start the performance monitoring system.

The driver platform is organized using structs and function pointers as a simple form of inheritance. A device is described by a struct of type *platform_driver*. The struct specifies several things including the driver name, but more importantly a device probe function. The probe function finds the device and allocates memory for the struct. The *platform_device_register(struct platform_driver)* function is used to initialize the device.

The first file modified is *arch/arm/mach-bcm2708/bcm2708.c*. The code in this file is executed soon after boot and starts and configures many of the Raspberry Pi specific modules. A few necessary changes are made to this file in order to enable the performance monitoring system. A platform_device struct is created specifying the performance monitoring system driver, and then a call to `bcm_register_device()` is added to initialize the performance monitoring unit. The patch is shown in Figure 1.

3.1.2 Device Tree Model

Newer ARM kernels attempt to automate discovery of resources via a *device tree* file, rather than hard-coding platform definitions in a C file. This allows generic kernels which can boot on multiple boards, as long as an appropriate device tree file is available at boot time.

Booting an up-to-date upstream Linux kernel on the Raspberry Pi is not an easy task, although as of the 3.15 kernel it is at least possible to get a minimal system working. The problem is not all hardware support has been contributed

```
diff --git a/arch/arm/mach-bcm2708/bcm2708.c b/arch/arm/mach-bcm2708/bcm2708.c
index 13b91de..71f8447 100644
--- a/arch/arm/mach-bcm2708/bcm2708.c
+++ b/arch/arm/mach-bcm2708/bcm2708.c
@@ -462,6 +462,11 @@ static struct resource bcm2708_powerman_resources[] = {
},
};

+static struct platform_device bcm2708_pmu_device = {
+ .name = "arm-pmu",
+ .id = -1, /* Only one */
+};
+
static u64 powerman_dmamask = DMA_BIT_MASK(DMA_MASK_BITS_COMMON);

struct platform_device bcm2708_powerman_device = {
@@ -728,6 +733,7 @@ void __init bcm2708_init(void)
bcm_register_device(&bcm2708_usb_device);
bcm_register_device(&bcm2708_uart1_device);
bcm_register_device(&bcm2708_powerman_device);
+ bcm_register_device(&bcm2708_pmu_device);

#ifdef CONFIG_MMC_SDHCI_BCM2708
bcm_register_device(&bcm2708_emmc_device);
```

Figure 1: Patch needed to enable platform PMU detection.

upstream, so things like USB and ethernet may not function.

In addition the default Raspberry-pi bootloader does not support device tree files. The Raspberry-pi has a somewhat unusual boot sequence for an embedded processor. The graphics processor (GPU) first boots and loads firmware before passing control to the ARM cpu and loading a kernel. The pi can be configured instead to boot the more common Uboot boot loader, which can then load a custom kernel with a device tree.

Once an upstream kernel is configured and Uboot set to boot, the bcm2835 device tree needs to be updated to know about the ARM1176 PMU. This change, as seen in Figure 2, was contributed by us upstream, and was merged into the Linux 3.15 kernel as git change:
14ac652b67fe08b0dca78995a4298aad38345a31
"ARM: bcm2835: perf_event support for Raspberry-Pi".

3.2 Ignoring the missing Interrupt

Once one of the previously mentioned methods of enabling PMU detection is enabled, then upon boot the following message should be in the changelog:

```
hw perfevents: enabled with v6 PMU driver,
3 counters available
```

However if you try to run `perf`, an error will still be generated and a message "no irqs for PMUs defined" will appear in the system log. It turns out that having a functional PMU interrupt is not strictly needed for perf_event functionality. While sampling events (i.e., `perf record`) will not work, total aggregate event counts should still work. In theory counter overflows could be lost, so if an event managed to overflow a few billion instructions (2^{31}) this could also be a problem, but it appears that this problem would be caught at the next context switch, so as long as the counter you are measuring does not overflow faster than the context switch rate (usually many times a second) things should work fine.

In addition, it is still possible to gather sampled results by using a software event as the sample source, something like `perf record -e cpu-clock`.

The simplest change that disables the interrupt check can be found in Figure 3.

The change that will be in the 3.16 kernel was a more generic change proposed by us that allows no-irq PMU setups to work on any architecture, not just ARM. That patch series was posted to the Linux kernel mailing list in June 2014 with the title "perf: Disable sampled events if no PMU interrupt". See commits:
c184c980de30dc5f6fec4b281928aa6743708da9
edcb4d3c36a6429caa03ddfeab4cbb153c7002b2

3.3 ARM Perf Utility Issues

While debugging the Raspberry-pi perf_event issues, we found some issues with the `perf` tool on ARM processors.

3.3.1 Compilation Error

On our particular setup the `perf` utility would not compile due to various gcc flags being used. This has since been fixed, possibly by git commit:
575bf1d04e908469d26da424b52fc1b12a1db9d8.

```
diff --git a/arch/arm/boot/dts/bcm2835.dtsi b/arch/arm/boot/dts/bcm2835.dtsi
index b021c96..beb8659 100644
--- a/arch/arm/boot/dts/bcm2835.dtsi
+++ b/arch/arm/boot/dts/bcm2835.dtsi
@@ -113,6 +113,10 @@
                         reg = <0x7e980000 0x10000>;
                         interrupts = <1 9>;
                 };
+
+                arm-pmu {
+                        compatible = "arm,arm1176-pmu";
+                };
         };

         clocks {
```

Figure 2: Patch needed to enable device-tree PMU detection.

```
diff --git a/arch/arm/kernel/perf_event_cpu.c b/arch/arm/kernel/perf_event_cpu.c
index d85055c..a74e0cd 100644
--- a/arch/arm/kernel/perf_event_cpu.c
+++ b/arch/arm/kernel/perf_event_cpu.c
@@ -97,8 +97,9 @@ static int cpu_pmu_request_irq(struct arm_pmu *cpu_pmu, irq_handler_t handler)

         irqs = min(pmu_device->num_resources, num_possible_cpus());
         if (irqs < 1) {
-                pr_err("no irqs for PMUs defined\n");
-                return -ENODEV;
+                printk_once("no irqs for PMUs defined, disabling sampled events\n");
+                return 0;
         }

         for (i = 0; i < irqs; ++i) {
```

Figure 3: Patch needed to enable perf_event even if no PMU interrupts are available.

3.3.2 `perf list` *Not Working*

On the Pi, perf 3.12 fails to list any hardware performance events or hardware cache events. It was initially thought that this was a result of the performance monitoring system being configured improperly. Upon running the same perf list command with the newly compiled version, 3.10.22, all expected hardware perf events were listed.

The newer perf utility had changed to iteratively check that each hardware event could indeed be measured before reporting it as available (before all events were shown, whether they were available or not). The problem was the test was run with the .exclude_kernel flag set on the events, but most ARM devices pre-dating the Cortex A15 do not support this mode of operation. This means that all hardware events were listed as unsupported when in fact some of them were.

We contributed a fix for this issue, git commit: 88fee52e58ca14d8465b614774ed0bf08e1a7790 "perf list: Fix checking for supported events on older kernels" which was included in the Linux 3.14 release.

4. VALIDATION

Once perf_event support is enabled, it becomes necessary to validate to make sure it is actually working. We have only done some preliminary testing, more in-depth tests should be conducted.

4.1 10 billion test

An initial validation test is a simple hand-coded assembly language benchmark that runs 10 billion instructions. When run on ARMv6 devices it will return a value above 10 billion; this is due to a hardware limitation where it is not possible to exclude kernel events from the count. The results are shown in Figure 4, where the results seem reasonable with 10 billion instructions plus 67 million from the kernel.

4.2 perf_event validation tests

Weaver has made available perf_event validation tests that check the behavior of perf_event on a given system.

Prior to our changes, the Rasp-pi only passes two of the tests (the ones checking for software only events, not affected by lack of hardware event support).

More pass once our changes are made, but still many fail. This needs to be investigated further.

These tests can be downloaded from:
`git clone https://github.com/deater/perf_event_tests.git`

5. RELATED WORK

Because rasp-pi support was not available, various people have written drivers to directly read the low-level counter values. Since reading these counters is a low-level access, this involved writing custom kernel modules.

Vince Weaver wrote the rasp-pi-pmu module available as part of his uarch-configure.

Paul J. Drongowski wrote rpistat, described at his website `http://sandsoftwaresound.net/raspberry-pi/rpi-perf-event-monito`

6. CONCLUSIONS AND FUTURE WORK

We contribute code that enables Linux perf_event hardware performance counter support for Raspberry-Pi hardware. This will allow users to gather useful performance data without having to hand-patch the kernel, and without using custom written kernel modules. Performance is critical on small embedded boards where every cycle counts, having the `perf` tool work by default will enable performance analysis by a wider range of users.

There still remains various pieces of related work that need to be done.

- Even though the upstream kernel has gained support for counters, we should ensure the Raspberry-pi foundation version also does.
- Further testing should be done that the results obtained from the counters are correct.
- We should ensure the generic no-irq patch gets committed for Linux 3.16. The patches are in the linux-tip tree but have not made it fully upstream yet.
- We should make sure that the generic no-irq patches are extended to work on other platforms (besides ARM) that lack working IRQs.
- Support for other tools that use perf_even (such as PAPI) should have Raspberry-pi support addded.

7. REFERENCES

[1] ARM. *ARM1176JZF-S Techincal Reference Manual*, 2009.

[2] Broadcom Europe Ltd. *BCM2835 ARM Peripherals*, 2012.

[3] D. Cobley. Re: Profiling on the pi? http://www.raspberrypi.org/phpBB3/viewtopic.php?f=33&t=19151, Oct. 2012.

[4] J. Corbet. Platform devices and device trees. *Linux Weekly News*, June 2011.

[5] T. Gleixner and I. Molnar. Performance counters for Linux, 2009.

[6] V. Weaver. perf_event_open manual page. In M. Kerrisk, editor, *Linux Programmer's Manual*. Dec. 2013.

```
$ perf stat -e cycles,instructions ./tenbillion

 Performance counter stats for './tenbillion':

    20,583,314,948 cycles
    10,067,266,891 instructions              #    0.49  insns per cycle

      43.754711581 seconds time elapsed
```

Figure 4: Results of a 10-billion instruction run

APPENDIX

A. ARCH LINUX AND RASP-PI KERNELS

A.1 Compiling the Linux Kernel

This section gives a brief overview of the process to compile a Linux kernel on the Pi. This section expects at least a basic understanding of the Linux command line including how to copy and move files, and edit text files. Command line actions will be prefixed with a $ to indicate a user command or a # to indicate a command that requires root privileges.

A.1.1 Arch Linux

All tests and compilations are done on Arch Linux on the latest version running kernel 3.10.22. Arch Linux is a rolling release so version numbers have little meaning. Arch Linux is freely available from `http://archlinuxarm.org/platforms/armv6/raspberry-pi`. Installation instructions are readily available at the given link. Software requirements can be installed with Arch Linux's package management tool pacman. Installing a dependency is as easy as,

```
# pacman -S make
```

A non-exhaustive list of dependencies is given in Table 1.

Table 1: Software Dependencies

Dependencies	Package Name
GNU Make	make
gcc	gcc
Arbitrary Precision Calculator	bc
GNU Project Parser Generator	bison
Fast Lexical Analyser Generator	flex

Make, gcc, and bc are used to compile the main portion of the Linux kernel. Bison and flex are small utilities used to compile the perf tool from the source available in the Linux kernel.

A.1.2 Getting the Latest Source

The second step after getting a functioning operating system is to obtain the kernel source code. The latest source is available from `http://kernel.org` but in the case of the Raspberry Pi, the source from here is unusable as it is missing a number of device specific patches and drivers. The easiest method of obtaining the source is to download an archive from `https://github.com/raspberrypi/linux/`. Cloning the git repository is difficult as the Raspberry Pi tends to run out of memory while doing so and fails with some less-than-helpful error messages.

A.1.3 Configuring the Source

The next step is to obtain a valid configuration file for the Raspberry Pi. The configuration file has thousands of options and determines many things during the compilation process including which modules to compile, which architecture to compile for, which drivers to include and many more. A configuration file already resides in the Arch Linux installation under /proc/config.gz. This file should be copied to the top-level directory of the kernel source like such,

```
$ zcat /proc/config.gz > .config
```

Configure the kernel with make,

```
$ make oldconfig
```

A.1.4 Compiling

Next, compile the source code. This is done with the GNU make utility. Because compilation can take upwards of nine hours on the Pi it may be necessary to logout during the process or the SSH session may timeout. An easy way to continue the compilation in the background is to use the tmux. It allows you to create, leave, and resume terminal sessions. Simply type *$ tmux* to enter a tmux shell, leave a session by typing <ctrl> + b and then d. Resume the session by typing *$ tmux attach*. After compiling for the first time, future calls to make will only recompile changed files saving a great deal of time.

A.1.5 Installing the New Kernel

After the compilation procedure is finished the kernel needs to be installed. The compiled kernel is the file *arch/arm/boot/Image*. Copy the newly compiled kernel image to */boot/kernel.img*.

A.1.6 Navigating the Kernel

Now that the kernel is successfully compiled, it is time to dive into the source code. This can be daunting at first as the kernel comprises more than 60,000 files. Some useful utilities for navigating the kernel source include grep, find,

and http://lxr.free-electrons.com/. Grep can be used to search the contents of files while find is used to search for file names. The URL given is a useful web-based reference that allows quick and easy navigation by function or variable name. Clicking a function name will bring up a list of all other usages of the given function. This is helpful for traversing the long call chains that are common in the kernel.

A.1.7 Tips

Because the make utility depends on file modification times to determine whether a compiled file is up-to-date it is very important not to manually change these modification times. Doing something like moving the entire kernel source directory to a new backup directory will update the modification times on all the files. This results in the make utility thinking that it has to recompile the entire kernel again from scratch. If moving the kernel source use *cp -p* to preserve all file attributes. Losing nine hours of work to recompile due to a careless move command can be infuriating. Doing it more than once even more so.

B. RASPIAN AND STOCK KERNELS

https://plus.google.com/101715070573995136397/posts/gWkwrfNfYVm

install u-boot-tools

B.1 Install Tools

Similar to above, install

```
apt-get install gcc bc
```

B.2 Get a kernel

Git won't work well on a constrained system. Can get a tarball from http://www.kernel.org/pub/linux/kernel/v3.x/testing in our case, linux-3.13-rc6.tar.xz Uncompress with

```
tar xvf linux-3.13-rc6.tar.xz
```

B.3 Patch kernel

I applied patches 1-5 from https://github.com/notro/rpi-build/blob/master/patches/bcm2835/

B.4 Build Kernel

```
make bcm2835_defconfig
make -j2
```

B.5 Install Kernel

Make sure running newest firmware on the pi.

```
cp arch/arm/boot/Image /boot/kernel_stock.img
```

B.6 Update and copy Device Tree

Modify to have the boot arguments

```
        chosen {
                bootargs = "dwc_otg.lpm_enable=0 console=ttyAMA0,115200 kgdboc=ttyAMA0,115200 console=tty1 root=/dev/mm
        };
```

```
cp ./arch/arm/boot/dts/bcm2835-rpi-b.dtb /boot
```

B.7 Setup boot files

Add to /boot/config.txt

```
kernel=kernel_stock.img
device_tree=bcm2835-rpi-b.dtb
device_tree_address=0x100
kernel_address=0x8000
disable_commandline_tags=1
```