Evaluate performance for Linux on POWER

Analyze performance using Linux tools

Adhemerval Zanella Netto
Ryan S. Arnold

June 12, 2012

Learn to evaluate Linux on POWER® performance issues that focus on compiled language (such as C or C++) environments. This article explains the POWER7® CPI model and demonstrates the use of commonly available Linux® tools to show potential CPU stalls, pipeline hazards, and performance issues. Analyze and optimize an algorithm for POWER7 in the final section.

Introduction

Application performance evaluation can be a complex task on modern machines. Commonly available tools hardly handle all the performance variables. Each workload differs in which computer subsystem it stresses. Measuring and tuning a CPU-bound program is quite different from tuning an IO-bound, or memory-bound program. In this article we focus on CPU-bound and memory-bound programs in compiled language environments (C, C++, and others). We demonstrate how to:

• Find program *hotspots* (a region, function, method in a program where a high proportion of the executed instructions occur)
• Measure how the program behaves on POWER7 using the hardware performance counter available on the processor
• Identify the tools to use for performance evaluation on Linux.

CPI model for POWER7

Understanding application performance analysis begins with a discussion of CPI Metrics. The Cycles per Instruction (CPI) metric is the number of processor cycles needed to complete an instruction. Each instruction is decomposed into multiple stages: a classic RISC pipeline will have an instruction fetch stage followed by instruction decode/register fetch, execution, an optional memory access and finally the writeback. A CPU can improve its CPI metric (measured by a lower CPI value) by exploiting instruction level parallelism: Each stage will handle different instructions in different stages. When optimizing, try to minimize the CPI value to maximize system utilization. Figure 1 shows an optimal instruction flow in a pipelined processor.
Sometimes one stage is not fully independent of other stages or it issues an instruction with dependencies that force the processor to meet that demand before continuing its execution. For instance, a memory load followed by an arithmetic instruction makes the processor first fetch the data into cache or memory only then issuing the arithmetic instruction. When this happens the processor pipeline is said to encounter a stall, which stalls the pipeline. Figure 2 shows what a stalled pipeline might look like.

In the examples in Figure 1 and Figure 2, consider that during 11 cycles of operation on a fully populated pipeline (where one instruction is completed per cycle) the processor can execute eight instructions. However, when a three-cycle stall occurs only five instructions were executed in the same number of cycles. The performance loss is about 40%. Depending on the algorithm, some stalls are unavoidable; however, careful analysis can provide hints and advice on how to rewrite or adjust some pieces of code to avoid such stalls. Find a more complete and didactic explanation on modern CPU pipelining and instruction-level parallelism in the article "Modern Microprocessors - a 90 minute guide" (see Related topics).

The CPI Breakdown Model (CBM) relates functional processor stages with performance counters to show which CPU functional unit is generating stalls. The CBM is dependent on the CPU architecture and processor model; The Power Architecture and Intel Architecture have completely different CBMs. POWER5 CBM, although similar, is different from the POWER7 CBM. Figure 3 shows a part of the POWER7 CBM. (See a text version of this information.)
In the Power Architecture, hardware performance counters are a set of special-purpose registers whose contents are updated when a certain event occurs within the processor. The POWER7 processor has a built-in Performance Monitoring Unit (PMU) with six thread-level Performance Counter Monitors (PCMs) per PMU. Four of these are programmable, meaning it is possible to monitor four events at the same time, and there are more than 500 possible performance events. POWER7 performance counters are defined by groups and the PMU can only watch events of the same groups at one time. Figure 3 shows a subset of the performance counters used to define the POWER7 CBM. The counters in Figure 3 following a profile, are used to denote which CPU functional unit is causing processor stalls and provide possible hints on how to tune the algorithm to eliminate them.
In **Figure 3**, white boxes are specific POWER7 PCMs watched in a profile. Based on their values the gray boxes [each marked with an asterisk (*)] are calculated (these metrics have no specific hardware counters).

Note: Find a comprehensive PMU reference for POWER7 in the paper, "Comprehensive PMU Event Reference POWER7" (See Related topics).

## Tools on Linux

How can you use the PCM found in POWER7 processors? Although you can use various profiling methods on POWER, like hardware interrupts, code instrumentation (such as gprof), operational system hooks (systemtap); PCM provides an extensive set of counter that work directly with processor functionality. The PCM profiler constantly samples the processor register values at regular intervals using operating system interrupts. Although sample profiling might result in less numerically accurate results than instruction trace results, it has less impact in overall system performance and allows the target benchmark to run at nearly full speed. The resulting data is not exact; it is an approximation with an error margin.

The two most commonly-used tools for PCM profiling on Linux are **OProfile** and **perf** (see Related topics). Although both use the same principle, constantly sampling the special hardware register (through a syscall) along a workload's backtrace, each is configured and used in a different way.

The **OProfile** tool is a system-wide profiler for Linux systems, capable of profiling all running code at low overhead. It consists of a kernel driver and daemon for collecting sample data, and several post-profiling tools for turning data into information. Debug symbols (-g option to gcc) are not necessary unless you want annotated source. With a recent Linux 2.6 kernel, **OProfile** can provide gprof-style call-graph profiling information. **OProfile** has a typical overhead of 1-8%, depending on sampling frequency and workload.

On POWER, **OProfile** works by watching groups of performance hardware counters and performance counters, though different groups can not be used together. It means that getting different performance counters from the same workload requires running it multiple times with different **OProfile** event configurations. This also means that you cannot watch the entire POWER7 CBM at the same time. The available groups are defined in the aforementioned "POWER7 PMY Detailed Event Description" document, or by running the command in **Listing 1**:  

### Listing 1. OProfile groups listing

```
# opcontrol -l
```

The commands in **Listing 2** demonstrate a simple **OProfile** configuration and invocation:

### Listing 2. OProfile POWER7 CPU cycles configuration

```
# opcontrol -l
# opcontrol --no-vmlinux
# opcontrol -e PM_CYC_GRP1:500000 -e PM_INST_CMPL_GRP1:500000 -e PM_RUN_CYC_GRP1:500000
   -e PM_RUN_INST_CMPL_GRP1:500000
# opcontrol --start
```
Run the workload in Listing 3.

**Listing 3. OProfile run command sequence**

```bash
# opcontrol --dump
# opcontrol --stop
# opcontrol --shutdown
```

To get the performance counter report, issue the command in Listing 4:

**Listing 4. OProfile report generation**

```bash
# opreport -l > workload_report
```

Note: Find comprehensive guide to OProfile (although not updated for POWER7) in the developerWorks article "Identify performance bottlenecks with OProfile for Linux on POWER" (see Related topics).

The `perf` tool, introduced in Linux kernel 2.6.29, analyzes performance events at both hardware and software levels. The `perf` tool has the advantage of being program oriented, instead of system oriented like OProfile. It has some preset performance counter lists, like 'cpu-cycles OR cycles', 'branch-misses', or 'L1-icache-prefetch-misses' and it has the ability to multiplex the PMU groups to allow gathering of multiple performance counters from different groups at same time at the cost of sample precision.

One drawback is that, although it allows gathering of hardware performance counters directly, `perf` does not recognize the counter name denoted by the POWER7 CBM; it needs to use raw hexadecimal numbers instead. **Table 1** is a mapping of OProfile events to hexadecimal numbers which you can use with `perf` (using the record raw events options) to utilize the CBM for POWER7.

**Table 1. POWER7 perf events raw codes**

<table>
<thead>
<tr>
<th>Counter</th>
<th>Raw code</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM_RUN_CYC</td>
<td>200f4</td>
</tr>
<tr>
<td>PM_CMPLUSTALL</td>
<td>4000a</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_FXU</td>
<td>20014</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_DIV</td>
<td>40014</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_SCALAR</td>
<td>40012</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_SCALAR_LONG</td>
<td>20018</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_VECTOR</td>
<td>2001c</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_VECTOR_LONG</td>
<td>4004a</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_LSU</td>
<td>20012</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_REJECT</td>
<td>40016</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_ERAT_MISS</td>
<td>40018</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_DCACHE_MISS</td>
<td>20016</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_STORE</td>
<td>2004a</td>
</tr>
</tbody>
</table>
PM_CMPLU_STALL_THRD 1001c
PM_CMPLU_STALL_IFU 4004c
PM_CMPLU_STALL_BRU 4004e
PM_GCT_NOSLOT_CYC 100f8
PM_GCT_NOSLOT_IC_MISS 2001a
PM_GCT_NOSLOT_BR_MPRED 4001a
PM_GCT_NOSLOT_BR_MPRED_IC_MISS 4001c
PM_GRP_CMPL 30004
PM_1PLUS_PPC_CMPL 100f2

Note: Find a comprehensive guide to `perf` (although not updated for POWER7) in the IBM Wiki "Using perf on POWER7 systems" (see Related topics).

You can get the raw codes used with `perf` that correspond to the POWER7 events defined in `OProfile` from the `libpfm4` project (see Related topics): They are defined in the POWER7 specific header (`lib/events/power7_events.h`). The example program `examples/showevtinfo` also shows the event names and corresponding raw hexadecimal codes.

To obtain counter information, profiling is a common approach. Profiling allows a developer to identify hotspots in code execution and data access, find performance sensitive areas, understand memory access patterns, and more. Before starting to profile, it is necessary to work out a performance evaluation strategy. The program might be composed of various modules and/or dynamic shared objects (DSO), it might intensively utilize the kernel, it might depend more on data pattern access (high pressure on L2 or L3 cache access) or might focus on the vector operation units. The next section will focus on possible performance evaluation strategies.

Strategies for Performance Evaluation

An initial performance evaluation is to find program hotspots by inspecting the CPU cycle utilization counter. To do this on POWER7, watch the events listed in Table 2:

Table 2. POWER7 CPU cycle utilization counters

<table>
<thead>
<tr>
<th>Counter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM_CYC</td>
<td>Processor Cycles</td>
</tr>
<tr>
<td>PM_INST_CMPL</td>
<td>Number of PowerPC Instructions that completed</td>
</tr>
<tr>
<td>PM_RUN_CYC</td>
<td>Processor Cycles gated by the run latch. Operating systems use the run latch to indicate when they are doing useful work. The run latch is typically cleared in the OS idle loop. Gating by the run latch filters out the idle loop.</td>
</tr>
<tr>
<td>PM_RUN_INST_CMPL</td>
<td>Number of run instructions completed</td>
</tr>
</tbody>
</table>

Running `OProfile` with these events will show the overall time the processor spent in a symbol. Below is an example profile output for the 403.gcc component from the SPECcpu2006 benchmark suite compiled with IBM Advance Toolchain 5.0 for POWER (see Related topics). The following is the output from the command `opreport -l`.

Evaluate performance for Linux on POWER
Listing 5. Output from 'opreport -' for 403.gcc benchmark (counter PM_CYC_GRP1 and PM_INST_CMPL_GRP1)

CPU: ppc64 POWER7, speed 3550 MHz (estimated)
Counted PM_CYC_GRP1 events ((Group 1 pm_utilization) Processor Cycles) with a unit mask of 0x00 (No unit mask) count 500000
Counted PM_INST_CMPL_GRP1 events ((Group 1 pm_utilization) Number of PowerPC Instructions that completed.) with a unit mask of 0x00 (No unit mask) count 500000

<table>
<thead>
<tr>
<th>samples</th>
<th>%</th>
<th>samples</th>
<th>%</th>
<th>image name</th>
<th>app name</th>
<th>symbol name</th>
</tr>
</thead>
<tbody>
<tr>
<td>204528</td>
<td>7.9112</td>
<td>32132</td>
<td>1.3848</td>
<td>gcc_base.none</td>
<td>gcc_base.none</td>
<td>reg_is_remote_constant_p</td>
</tr>
<tr>
<td>125218</td>
<td>4.8434</td>
<td>246710</td>
<td>10.6324</td>
<td>gcc_base.none</td>
<td>gcc_base.none</td>
<td>bitmap_operation</td>
</tr>
<tr>
<td>113199</td>
<td>4.3782</td>
<td>50950</td>
<td>2.1958</td>
<td>libc-2.13.so</td>
<td>libc-2.13.so</td>
<td>memset</td>
</tr>
<tr>
<td>90316</td>
<td>3.4934</td>
<td>22193</td>
<td>0.9564</td>
<td>gcc_base.none</td>
<td>gcc_base.none</td>
<td>compute_transp</td>
</tr>
<tr>
<td>89978</td>
<td>3.4804</td>
<td>11753</td>
<td>0.5065</td>
<td>vmlinux</td>
<td>vmlinux</td>
<td>.pseries_dedicated_idle_sleep</td>
</tr>
<tr>
<td>88429</td>
<td>3.4264</td>
<td>130166</td>
<td>5.6097</td>
<td>gcc_base.none</td>
<td>gcc_base.none</td>
<td>bitmap_element_allocate</td>
</tr>
<tr>
<td>67729</td>
<td>2.6184</td>
<td>41749</td>
<td>1.7876</td>
<td>gcc_base.none</td>
<td>gcc_base.none</td>
<td>ggc_set_mark</td>
</tr>
<tr>
<td>56613</td>
<td>2.1898</td>
<td>89418</td>
<td>3.8536</td>
<td>gcc_base.none</td>
<td>gcc_base.none</td>
<td>canon_rtx</td>
</tr>
<tr>
<td>53949</td>
<td>2.0868</td>
<td>6985</td>
<td>0.3019</td>
<td>gcc_base.none</td>
<td>gcc_base.none</td>
<td>delete_null_pointer_check</td>
</tr>
<tr>
<td>51587</td>
<td>1.9954</td>
<td>26000</td>
<td>1.1205</td>
<td>gcc_base.none</td>
<td>gcc_base.none</td>
<td>ggc_mark_rtx_children_1</td>
</tr>
<tr>
<td>48050</td>
<td>1.8586</td>
<td>16086</td>
<td>0.6933</td>
<td>gcc_base.none</td>
<td>gcc_base.none</td>
<td>single_set_2</td>
</tr>
<tr>
<td>47115</td>
<td>1.8224</td>
<td>33772</td>
<td>1.4555</td>
<td>gcc_base.none</td>
<td>gcc_base.none</td>
<td>note_stores</td>
</tr>
</tbody>
</table>

Listing 6. Output from 'opreport -' for 403.gcc benchmark (counter PM_RUN_CYC_GRP1 and PM_RUN_INST_CMPL_GRP1)

Counted PM_RUN_CYC_GRP1 events ((Group 1 pm_utilization) Processor Cycles gated by the run latch. Operating systems use the run latch to indicate when they are doing useful work. The run latch is typically cleared in the OS idle loop. Gating by the run latch filters out the idle loop.) with a unit mask of 0x00 (No unit mask) count 500000
Counted PM_RUN_INST_CMPL_GRP1 events ((Group 1 pm_utilization) Number of run instructions completed.) with a unit mask of 0x00 (No unit mask) count 500000

<table>
<thead>
<tr>
<th>samples</th>
<th>%</th>
<th>samples</th>
<th>%</th>
<th>samples</th>
<th>%</th>
<th>app name</th>
<th>symbol name</th>
</tr>
</thead>
<tbody>
<tr>
<td>204538</td>
<td>8.3658</td>
<td>32078</td>
<td>1.3965</td>
<td>gcc_base.none</td>
<td>gcc_base.none</td>
<td>reg_is_remote_constant_p</td>
<td></td>
</tr>
<tr>
<td>124596</td>
<td>5.0961</td>
<td>25227</td>
<td>10.9809</td>
<td>gcc_base.none</td>
<td>gcc_base.none</td>
<td>bitmap_operation</td>
<td></td>
</tr>
<tr>
<td>112326</td>
<td>4.5943</td>
<td>50890</td>
<td>2.1555</td>
<td>libc-2.13.so</td>
<td>libc-2.13.so</td>
<td>memset</td>
<td></td>
</tr>
<tr>
<td>90312</td>
<td>3.6939</td>
<td>21882</td>
<td>0.9527</td>
<td>gcc_base.none</td>
<td>gcc_base.none</td>
<td>compute_transp</td>
<td></td>
</tr>
<tr>
<td>48054</td>
<td>1.8566</td>
<td>16086</td>
<td>0.6933</td>
<td>gcc_base.none</td>
<td>gcc_base.none</td>
<td>.pseries_dedicated_idle_sleep</td>
<td></td>
</tr>
<tr>
<td>47115</td>
<td>1.8224</td>
<td>33772</td>
<td>1.4555</td>
<td>gcc_base.none</td>
<td>gcc_base.none</td>
<td>bitmap_element_all_allocate</td>
<td></td>
</tr>
<tr>
<td>67995</td>
<td>2.7811</td>
<td>41331</td>
<td>1.7994</td>
<td>gcc_base.none</td>
<td>gcc_base.none</td>
<td>ggc_set_mark</td>
<td></td>
</tr>
<tr>
<td>56460</td>
<td>2.3093</td>
<td>89484</td>
<td>3.8958</td>
<td>gcc_base.none</td>
<td>gcc_base.none</td>
<td>canon_rtx</td>
<td></td>
</tr>
<tr>
<td>54076</td>
<td>2.1188</td>
<td>6665</td>
<td>0.3832</td>
<td>gcc_base.none</td>
<td>gcc_base.none</td>
<td>delete_null_pointer_check</td>
<td></td>
</tr>
<tr>
<td>51228</td>
<td>2.0953</td>
<td>26057</td>
<td>1.1344</td>
<td>gcc_base.none</td>
<td>gcc_base.none</td>
<td>ggc_mark_rtx_children_1</td>
<td></td>
</tr>
<tr>
<td>48057</td>
<td>1.9656</td>
<td>16005</td>
<td>0.6968</td>
<td>gcc_base.none</td>
<td>gcc_base.none</td>
<td>single_set_2</td>
<td></td>
</tr>
<tr>
<td>47160</td>
<td>1.9289</td>
<td>33766</td>
<td>1.4700</td>
<td>gcc_base.none</td>
<td>gcc_base.none</td>
<td>note_stores</td>
<td></td>
</tr>
</tbody>
</table>

Each watched event is represented by a pair of columns in the output. The first column shows the sample numbers collected from a PCM for the specified event and the second shows the percentage of the total sample numbers it presents. As seen in this report, the symbol reg_is_remote_constant_p is the one which consumes most of the processor cycles and is a good
candidate for code optimization. This profile only identifies which symbols consume the most CPU cycles, but not if the processor pipeline is fully utilized. You can investigate pipeline utilization by comparing the counters results.

Consider the counter **PM INST CMPL GRP1** (the second pair of columns); the symbol `bitmap_operation` shows a higher percentage than the symbol `reg_is_remote_constant_p`. This performance counter is incremented for each processor instruction completed, whereas **PM CYC GRP1** only means the number of CPU cycles utilized. Without further analysis, this might indicate that the symbol `reg_is_remote_constant_p` contains more CPU stalls than the symbol `bitmap_operation` since the number of instructions completed for the symbol `reg_is_remote_constant_p` is significantly lower. This profile provides an initial hint on which symbol to focus subsequent optimization efforts.

Before you start to dig in and crack up the code it is wise to understand if the workload is CPU or memory bound. This is important because optimization approaches are quite different for each workload type. For example, most often memory accesses come from cache or main memory (as opposed to NUMA remote node memory access) and performance depends almost entirely on the algorithms and data structures used. To investigate memory access patterns, watch the following two performance counters in Table 3:

<table>
<thead>
<tr>
<th>Counter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM_MEM0_RQ_DISP</td>
<td>Read requests dispatched for main memory</td>
</tr>
<tr>
<td>PM_MEM0_WQ_DISP</td>
<td>Write requests dispatched for main memory</td>
</tr>
</tbody>
</table>

These two counters can indicate whether a memory access pattern is mainly from memory reads, writes, or both. Using the same benchmark as before (403.gcc from SPECcpu2006), the profile shows:

**Listing 7. Output from 'opreport -' for 403.gcc benchmark (counter PM_MEM0_RQ_DISP and PM_MEM0_WQ_DISP)**

```
CPU: ppc64 POWER7, speed 3550 MHz (estimated)
Counted PM_MEM0_RQ_DISP_GRP59 events ((Group 59 pm_nest2) Nest events (MC0/MC1/PB/GX), Pair0 Bit1) with a unit mask of 0x00 (No unit mask) count 1000
Counted PM_MEM0_WQ_DISP_GRP59 events ((Group 59 pm_nest2) Nest events (MC0/MC1/PB/GX), Pair3 Bit1) with a unit mask of 0x00 (No unit mask) count 1000
samples %        samples %        app name                 symbol name
225841  25.8000  289       0.4896  gcc_base.none            reg_is_remote_constant_p.\isra.3.part.4
90868   10.2893  2183      3.0862  gcc_base.none            compute_transp
54838   6.1733   388       0.4354  gcc_base.none            single_set_2
32660   3.7311  2006      2.8359  gcc_base.none            delete_null_pointer_checks
26362   3.0104  2006      2.8359  gcc_base.none            delete_null_pointer_checks
21306   2.4340  1950      2.7568  vmlinux                  .pseries_dedicated_idle_sleeep
18059   2.0631  9186     12.9865  libc-2.13.so             memset
15867   1.8126   659       0.9316  gcc_base.none            init_alias_analysis
```

Another interesting set of performance counters to observe is the access pressure over the cache, both L2 and L3. The following example uses **perf** to profile the SPECcpu2006 483.xalancbmk
component (see Related topics) that is built using RHEL6.2 Linux system GCC. This component uses memory allocation routines heavily so expect a lot of pressure on the memory subsystem. To accomplish this, watch the following counters in Table 4 with OProfile:

Table 4. POWER7 cache/memory access counters

<table>
<thead>
<tr>
<th>Counter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM_DATA_FROM_L2</td>
<td>The processor's Data Cache was reloaded from the local L2 due to a demand load</td>
</tr>
<tr>
<td>PM_DATA_FROM_L3</td>
<td>The processor's Data Cache was reloaded from the local L3 due to a demand load</td>
</tr>
<tr>
<td>PM_DATA_FROM_LMEM</td>
<td>The processor's Data Cache was reloaded from memory attached to the same module this processor is located on</td>
</tr>
<tr>
<td>PM_DATA_FROM_RMEM</td>
<td>The processor's Data Cache was reloaded from memory attached to a different module than this processor is located on</td>
</tr>
</tbody>
</table>

The profile output shows the following:

Listing 8. Output from 'opreport -' for 489.Xalancbmk benchmark (counter PM_DATA_FROM_L2_GRP91 and PM_DATA_FROM_L3_GRP91)

CPU: ppc64 POWER7, speed 3550 MHz (estimated)
Counted PM_DATA_FROM_L2_GRP91 events ((Group 91 pm_dsource1) The processor's Data Cache was reloaded from the local L2 due to a demand load.) with a unit mask of 0x00 (No unit mask) count 1000
Counted PM_DATA_FROM_L3_GRP91 events ((Group 91 pm_dsource1) The processor's Data Cache was reloaded from the local L3 due to a demand load.) with a unit mask of 0x00 (No unit mask) count 1000

<table>
<thead>
<tr>
<th>samples %</th>
<th>samples %</th>
<th>image name</th>
<th>app name</th>
<th>symbol name</th>
</tr>
</thead>
<tbody>
<tr>
<td>767827</td>
<td>25.5750</td>
<td>gcc_base.none</td>
<td>gcc_base.none</td>
<td>bitmap_element_allocate</td>
</tr>
<tr>
<td>377138</td>
<td>12.5618</td>
<td>gcc_base.none</td>
<td>gcc_base.none</td>
<td>bitmap_operation</td>
</tr>
<tr>
<td>93334</td>
<td>3.1088</td>
<td>gcc_base.none</td>
<td>gcc_base.none</td>
<td>bitmap_bit_p</td>
</tr>
<tr>
<td>70278</td>
<td>2.3468</td>
<td>libc-2.13.so</td>
<td>libc-2.13.so</td>
<td>_int_free</td>
</tr>
<tr>
<td>56851</td>
<td>1.8936</td>
<td>oprofile</td>
<td>oprofile</td>
<td>/oprofile</td>
</tr>
<tr>
<td>47570</td>
<td>1.5845</td>
<td>gcc_base.none</td>
<td>gcc_base.none</td>
<td>rehash_using_reg</td>
</tr>
<tr>
<td>41441</td>
<td>1.3803</td>
<td>libc-2.13.so</td>
<td>libc-2.13.so</td>
<td>_int_malloc</td>
</tr>
</tbody>
</table>

Listing 9. Output from 'opreport -' for 489.Xalancbmk benchmark (counter PM_DATA_FROM_LMEM_GRP91 and PM_DATA_FROM_RMEM_GRP91)

Counted PM_DATA_FROM_LMEM_GRP91 events ((Group 91 pm_dsource1) The processor's Data Cache was reloaded from memory attached to the same module this processor is located on.) with a unit mask of 0x00 (No unit mask) count 1000
Counted PM_DATA_FROM_RMEM_GRP91 events ((Group 91 pm_dsource1) The processor's Data Cache was reloaded from memory attached to a different module than this processor is located on.) with a unit mask of 0x00 (No unit mask) count 1000

<table>
<thead>
<tr>
<th>samples %</th>
<th>samples %</th>
<th>image name</th>
<th>app name</th>
<th>symbol name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1605</td>
<td>0.3344</td>
<td>gcc_base.none</td>
<td>gcc_base.none</td>
<td>bitmap_element_allocate</td>
</tr>
<tr>
<td>1778</td>
<td>0.3784</td>
<td>gcc_base.none</td>
<td>gcc_base.none</td>
<td>bitmap_operation</td>
</tr>
<tr>
<td>1231</td>
<td>0.2564</td>
<td>gcc_base.none</td>
<td>gcc_base.none</td>
<td>bitmap_bit_p</td>
</tr>
<tr>
<td>285</td>
<td>0.0427</td>
<td>libc-2.13.so</td>
<td>libc-2.13.so</td>
<td>_int_free</td>
</tr>
<tr>
<td>583</td>
<td>0.1215</td>
<td>100.000</td>
<td>oprofile</td>
<td>/oprofile</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>gcc_base.none</td>
<td>gcc_base.none</td>
<td>rehash_using_reg</td>
</tr>
<tr>
<td>225</td>
<td>0.0469</td>
<td>libc-2.13.so</td>
<td>libc-2.13.so</td>
<td>_int_malloc</td>
</tr>
</tbody>
</table>

Interpreting the profile output shows that most of the cache pressure came from L2 access with almost no L3 reload from demand, since the total and relative counter sample value for L2 access
(PM_DATA_FROM_L2) is much higher than L3 demand reload (PM_DATA_FROM_L3). You can only obtain further information, like if L2 access is causing CPU stalls due to cache misses, with more comprehensive analysis (by watching more counters). A conclusion that can be drawn from this example profile is that the main memory access (PM_DATA_FROM_LMEM event) is quite low compared to cache access and there is no remote access (event PM_DATA_FROM_RMEM) indicating no remote NUMA node memory access. Analysis of hotspots and memory access patterns can give direction to optimization efforts; in this case, further analysis is required to identify what really causes CPU stalls because simple identifying the workload hotspots and memory access pattern is not enough to correctly identify CPU stalls.

To come up with better strategies for performance optimization further analysis will require using the perf tool rather than OProfile since many POWER7 CBM counters need to be watched simultaneously the 22 presented in Figure 3 and to come with better strategies for performance optimization. Many of these events are in different groups, meaning that using OProfile requires many runs of the same workload. The perf tool will multiplex the watching of hardware counters when the specified counters are in more than one group. Although this results in a less accurate outcome, the overall result tends to be very similar to the expected with the advantage of less time spent profiling.

The following example uses perf to profile the same SPECcpu2006 483.xalancbmk component. To profile this component, issue the command in Listing 10:

**Listing 10. perf command to generated POWER7 CBM**

```
$ /usr/bin/perf stat -C 0 -e r100f2,r4001a,r100f8,r4001c,r2001a,r200f4,r2004a,r4004a,
    r4004e,r4004c,r20016,r40018,r20012,r40016,r40012,r20018,r4000a,r2001c,r1001c,r20014,
    r40014,r30004 taskset -c 0 ./Xalan_base.none -v t5.xml xalanc.xsl > power7_cbm.dat
```

This command will cause perf to watch the raw events defined by the -e argument on the CPU specified by -c. The taskset call ensures that the component will run exclusively on CPU number 0. The workload ./Xalan_base.none -v t5.xml xalanc.xsl can be replaced by another application to profile. After the profile is complete, the perf command will output a simple table of the total count for each raw event with the total number of elapsed seconds:

**Listing 11. Output from 'perf stat' for 489.Xalancbmk benchmark**

```
Performance counter stats for 'taskset -c 0 ./Xalan_base.none -v t5.xml xalanc.xsl':

<table>
<thead>
<tr>
<th>Event</th>
<th>Count</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>366,860,486,404 r100f2</td>
<td>[18.15%]</td>
<td></td>
</tr>
<tr>
<td>8,690,500,758 r4001a</td>
<td>[13.65%]</td>
<td></td>
</tr>
<tr>
<td>50,655,176,004 r100f8</td>
<td>[ 9.13%]</td>
<td></td>
</tr>
<tr>
<td>11,358,043,420 r4001c</td>
<td>[ 9.11%]</td>
<td></td>
</tr>
<tr>
<td>10,318,533,758 r2001a</td>
<td>[13.68%]</td>
<td></td>
</tr>
<tr>
<td>1,301,183,175,870 r200f4</td>
<td>[18.22%]</td>
<td></td>
</tr>
<tr>
<td>2,150,935,303 r2004a</td>
<td>[ 9.10%]</td>
<td></td>
</tr>
<tr>
<td>0 r4004a</td>
<td>[13.65%]</td>
<td></td>
</tr>
<tr>
<td>211,224,577,427 r4004e</td>
<td>[ 4.54%]</td>
<td></td>
</tr>
<tr>
<td>212,033,138,844 r4004c</td>
<td>[ 4.54%]</td>
<td></td>
</tr>
<tr>
<td>264,721,636,785 r20016</td>
<td>[ 9.09%]</td>
<td></td>
</tr>
<tr>
<td>22,176,093,590 r40018</td>
<td>[ 9.11%]</td>
<td></td>
</tr>
<tr>
<td>510,728,741,936 r20012</td>
<td>[ 9.10%]</td>
<td></td>
</tr>
</tbody>
</table>
```
To analyze the `perf` output against the POWER7 CBM, a Python script is provided (check the `power7_cbm.zip` in Downloadable resources), which composes the counter metrics from the collected virtual and hardware counters. To create a report issue the command in Listing 12:

**Listing 12. POWER7 CBM python script invocation**

```bash
$ power7_cbm.py power7_cbm.dat
```

Output similar to Listing 13 will be printed:

**Listing 13. Output from 'power7_cbm.py' for 489.Xalancbmk benchmark**

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value :</th>
<th>Percent :</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM_CMPLUSTALL_DIV</td>
<td>49802421337.0</td>
<td>0.0</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_FXU_OTHER</td>
<td>67578558649.0</td>
<td>5.2</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_SCALAR_LONG</td>
<td>2011413.0</td>
<td>0.0</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_SCALAR_OTHER</td>
<td>7195240464.0</td>
<td>0.6</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_VECTOR_LONG</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_VECTOR_OTHER</td>
<td>1290603592.0</td>
<td>0.1</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_ERAT_MISS</td>
<td>22193968056.0</td>
<td>1.7</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_REJECT_OTHER</td>
<td>18190293594.0</td>
<td>1.4</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_DCACHE_MISS</td>
<td>261865838255.0</td>
<td>20.3</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_STORE</td>
<td>2001549895.0</td>
<td>0.2</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_ISSU_OTHER</td>
<td>20231206181.0</td>
<td>15.7</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_THRD</td>
<td>2025705.0</td>
<td>0.0</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_BRU</td>
<td>208356542821.0</td>
<td>16.2</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_IFU_OTHER</td>
<td>2171796336.0</td>
<td>0.2</td>
</tr>
<tr>
<td>PM_CMPLUSTALL_OTHER</td>
<td>30895294057.0</td>
<td>2.4</td>
</tr>
<tr>
<td>PM_GCT_NOSLOT_IC_MISS</td>
<td>9885421042.0</td>
<td>0.8</td>
</tr>
<tr>
<td>PM_GCT_NOSLOT_BR_MPRED</td>
<td>7823508357.0</td>
<td>0.6</td>
</tr>
<tr>
<td>PM_GCT_NOSLOT_BR_MPRED_IC_MISS</td>
<td>11059314150.0</td>
<td>0.9</td>
</tr>
<tr>
<td>PM_GCT_EMPTY_OTHER</td>
<td>20292049774.0</td>
<td>1.6</td>
</tr>
<tr>
<td>PM_1PLUS_PPC_CMPL</td>
<td>365158978504.0</td>
<td>28.3</td>
</tr>
<tr>
<td>OVERHEAD_EXPANSION</td>
<td>590057044.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Total</td>
<td>961.05870036</td>
<td>96.1</td>
</tr>
</tbody>
</table>

This report is based on statistical values within an error margin, so final percentages are not entirely accurate. Even with a high error margin, about 20% of total CPU stalls are due to data cache misses (`PM_CMPLUSTALL_DCACHE_MISS`). The final instruction completion percentage (`PM_1PLUS_PPC_CMPL`) is about 28%.

Future optimizations should try to maximize this number by decreasing CPU stalls and/or GCT (Global Completion Table) percentages. Based on this report, another avenue for analysis is to identify the code where the stalls are happening. To accomplish this by using
the `perf record` command. It will trace the performance of a raw counter and create a map with a process backtrace allowing identification of which symbol generated the most hardware events. This is similar to the way OProfile works. In this example, to trace the `PM_CMPLU_STALL_DCache_MISS` events, issue the command in **Listing 14:**

**Listing 14. perf record for PM_CMPLU_STALL_DCache_MISS event**

```bash
$ /usr/bin/perf record -C 0 -e r20016 taskset -c 0 ./Xalan_base.none -v t5.xml xalanc.xsl
```

The `perf` command will create a data file (usually "perf.dat") with the results. It can be read interactively using the perf report command as in **Listing 15:**

**Listing 15. Output from 'perf report' for 489.XalancBmk benchmark**

```
Events: 192  raw 0x20016
  39.58% Xalan_base.none Xalan_base.none [.] xercesc_2_5::ValueStore::contains
  11.46% Xalan_base.none Xalan_base.none [.] xalanc_1_8::XStringCachedAllocator
  11.46% Xalan_base.none Xalan_base.none [.] xalanc_1_8::XStringCachedAllocator
  11.46% Xalan_base.none Xalan_base.none [.] xalanc_1_8::XStringCachedAllocator
  11.46% Xalan_base.none Xalan_base.none [.] xalanc_1_8::XStringCachedAllocator
  11.46% Xalan_base.none Xalan_base.none [.] xercesc_2_5::ValueStore::isDuplicate
  9.90% Xalan_base.none Xalan_base.none [.] xalanc_1_8::XStringCachedAllocator
  9.90% Xalan_base.none Xalan_base.none [.] xalanc_1_8::XStringCachedAllocator
  7.29% Xalan_base.none Xalan_base.none [.] xercesc_2_5::ValueStore::isDuplicate
  5.21% Xalan_base.none libc-2.13.so [.] __int_malloc
  5.21% Xalan_base.none Xalan_base.none [.] __gnu_cxx::__normal_iterator<xa
  4.17% Xalan_base.none libc-2.13.so [.] __GI___libc_malloc
  4.17% Xalan_base.none libc-2.13.so [.] __GI___libc_malloc
  2.08% Xalan_base.none libc-2.13.so [.] malloc_consolidate.part.4
  1.56% Xalan_base.none Xalan_base.none [.] xalanc_1_8::ReusableArenaBlock<xa
  1.56% Xalan_base.none Xalan_base.none [.] xalanc_1_8::ReusableArenaBlock<xa
  1.56% Xalan_base.none Xalan_base.none [.] xalanc_1_8::ReusableArenaBlock<xa
  1.04% Xalan_base.none libc-2.13.so [.] __free

[...]
```

With this analysis using POWER7 CBM counter and the `perf report` tool, your optimization effort might concentrate on optimizing memory and cache access on the symbol `xercesc_2_5::ValueStore::contains(xercesc_2_5::FieldMapView<const*)`.

This example is just a subset of possible analysis. The POWER7 CBM shows you that although data cache stalls show up as being the higher cause of CPU stalls, the load and store unit (`PM_CMPLU_STALL_LSU`) and branch unit (`PM_CMPLU_STALL_BRU`) are both source of stalls. Further analysis can address these counters.

**Case Study**

The following case study applies these performance evaluation strategies to analyze a trigonometric math function implementation. Based on analysis results, optimization opportunities will be identified. The function used in this case study is the ISO C `hypot` function, defined as length of the hypotenuse of a right-triangle. The function is defined by C99, POSIX.1-2001 as:

```c
double hypot(double x, double y);
```

The `hypot()` function returns \(\sqrt{x^2+y^2}\). On success, this function returns the length of a right-angled triangle with sides of length \(x\) and \(y\). If \(x\) or \(y\) is an infinity, positive infinity is returned. If \(x\) or \(y\) is a NaN, and the other argument is not an infinity, a NaN is returned. If the result overflows, a range error occurs, and the functions return `HUGE_VAL`, `HUGE_VALF`, or `HUGE_VALL`, respectively. If both arguments...
are subnormal, and the result is subnormal, a range error occurs, and the correct result is returned.

Although the algorithm seems simple, the Floating-Point (FP) argument handling of Infinity and NaN and the overflow/underflow related to FP operations impose some challenges with performance impacts. The GNU C Library (see Related topics) provides an implementation of `hypot` located in the source tree at `sysdeps/ieee754/dbl-64/e_hypot.c`:

**Note:** The license information for this code sample is included in Appendix.

### Listing 16. Default GLIBC hypot source code

```c
double __ieee754_hypot(double x, double y)
{
    double a,b,t1,t2,y1,y2,w;
    int32_t j,k,ha,hb;

    GET_HIGH_WORD(ha,x);
    ha &= 0x7fffffff;
    GET_HIGH_WORD(hb,y);
    hb &= 0x7fffffff;
    if(hb > ha) {a=y;b=x;j=ha; ha=hb;hb=j;} else {a=x;b=y;}
    SET_HIGH_WORD(a,ha); /* a <- |a| */
    SET_HIGH_WORD(b,hb); /* b <- |b| */
    if((ha-hb)>0x3c00000) {return a+b;} /* x/y > 2**60 */
    k=0;
    if(ha > 0x5f300000) { /* a>2**500 */
        if(ha >= 0x7ff00000) { /* Inf or NaN */
            u_int32_t low;
            w = a+b;
            /* for sNaN */
            GET_LOW_WORD(low,a);
            if(((ha&0xfffff)|low)==0) w = a;
            GET_LOW_WORD(low,b);
            if(((hb&0x7ff00000)|low)==0) w = b;
            return w;
        }
        /* scale a and b by 2**-600 */
        ha -= 0x25800000; hb -= 0x25800000;  k += 600;
        SET_HIGH_WORD(a,ha);
        SET_HIGH_WORD(b,hb);
    }
    if(hb < 0x20b00000) { /* b < 2**-500 */
        if(hb <= 0x00000000) { /* subnormal b or 0 */
            u_int32_t low;
            GET_LOW_WORD(low,b);
            if((hb|low)==0) return a;
            t1=0;
            SET_HIGH_WORD(t1,0x7fd00000); /* t1=2^1022 */
            b *= t1;
            a *= t1;
            k = 1022;
        } else { /* scale a and b by 2^600 */
            ha += 0x25800000; /* a *= 2^600 */
            hb += 0x25800000; /* b *= 2^600 */
            k -= 600;
            SET_HIGH_WORD(a,ha);
            SET_HIGH_WORD(b,hb);
        }
    }
    /* medium size a and b */
    w = a-b;
    if (w>b) {
        t1 = 0;
```

Evaluate performance for Linux on POWER
This implementation is quite complex mainly because the algorithm executes many of bit-by-bit FP to INT conversions. It assumes that certain FP operations, like compares and multiplications, are more costly when using float-point instructions than when using fixed-point instructions. This is true on some architectures, but not on the Power Architecture.

Your first step to evaluate this implementation is to create a benchmark which can be profiled. In this case, since it is simply a function with two arguments and a straightforward algorithm (no internal function calls or additional paths) a simple benchmark may be created to evaluate it (check the hypot_bench.tar.gz in Downloadable resources). The benchmark is part of the performance evaluation; optimizations should speed up algorithms or critical parts of algorithms that leverage the total workload performance. Synthetic benchmarks, like this one, should represent normal utilization for this function. Since optimization efforts tend to be resource and time consuming one needs to focus on the most common usage cases or expected behavior. Trying to optimize code that represents low total program usage tends to be a waste of resources.

Since this is a performance analysis on a single function, you can skip hotspot analysis and focus on CBM analysis. Using the benchmark in hypot_bench.c along with perf, the CBM information in Listing 17:

Listing 17. Output from 'power7_cbm.py' for hypot benchmark

```
CPI Breakdown Model (Complete)

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM_CMPLUSTALL_DIV           : 8921688.0  :  8.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PM_CMPLUSTALL_FXU_OTHER     : 13953382275.0:  5.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PM_CMPLUSTALL_SCALAR_LONG   : 24380128688.0 :  8.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PM_CMPLUSTALL_SCALAR_OTHER  : 33862492798.0 : 12.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PM_CMPLUSTALLVECTOR_LONG    : 0.0       :  0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PM_CMPLUSTALLVECTOR_OTHER   : 275057010.0 :  0.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PM_CMPLUSTALLERAT_MISS      : 173439.0  :  0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PM_CMPLUSTALLREJECTOTHER    : 902838.0  :  0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PM_CMPLUSTALLDCACHE_MISS    : 15200163.0 :  0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PM_CMPLUSTALLSTORE          : 1837414.0 :  0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PM_CMPLUSTALLLSU_OTHER      : 94866270200.0: 33.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PM_CMPLUSTALLTHIRD         : 560936.0  :  0.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
The profile analysis shows that most of CPU stalls and hence performance loss comes from the Load and Store Unit (LSU - counter `PM_CMPLU_STALL_LSU_OTHER`). The LSU has various counters associated with it, however during CPU stall analysis your focus is the counters that are associated with performance degradations. The ones that show performance degradations on POWER are associated with the Load-Hit-Store (LHS) hazards. This is a large stall that occurs when the CPU writes data to an address and then tries to load that data again too soon afterward. The next step is to check if this is happening on this particular algorithm by first checking the event `PM_LSU_REJECT_LHS` (raw code "rc8ac") as shown in Listing 18.

Listing 18. perf record of PM_LSU_REJECT_LHS POWER7 event

```
$ perf record -C 0 -e rc8ac taskset -c 0 ./hypot_bench_glibc
$ perf report
Events: 14K raw 0xc8ac
  79.19%  hypot_bench_gli  libm-2.12.so  [.] __ieee754_hypot
  10.38%  hypot_bench_gli  libm-2.12.so  [.] __hypot
  6.34%  hypot_bench_gli  libm-2.12.so  [.] __GI___finite
```

The profile output shows the symbol __ieee754_hypot is the one generating most of `PM_LSU_REJECT_LHS` events. Investigating the assembly code generated by the compiler to identify which instructions are generating the event. Expand the symbol __ieee754_hypot to annotate the assembly by iterating on the `perf report` screen and selecting the __ieee754_hypot symbol, which shows the output in Listing 19.

Listing 19. perf report of PM_LSU_REJECT_LHS POWER7 event

```
:        00000080fc38b730 <__ieee754_hypot>:
 0.00 :     80fc38b730:  7c 08 02 a6     mflr    r0
 0.00 :     80fc38b734:  fb c1 ff f0     std     r30,-16(r1)
 0.00 :     80fc38b738:  fb e1 ff f8     std     r31,-8(r1)
 13.62 :    80fc38b73c:  f8 01 00 10     std     r0,16(r1)
 0.00 :    80fc38b740:  f8 21 00 70     stfd    f1,112(r1)
 10.82 :    80fc38b744:  d8 21 00 70     stfd    f1,112(r1)
 0.23 :    80fc38b748:  e9 21 00 70     ld      r9,112(r1)
 17.54 :    80fc38b74c:  d8 41 00 70     stfd    f2,112(r1)
 0.00 :    80fc38b750:  79 29 00 62     rldicl  r9,r9,32,33
 0.00 :    80fc38b754:  e9 61 00 70     ld      r11,112(r1)
 0.00 :    80fc38b758:  e8 01 00 70     ld      r0,112(r1)
 8.46 :    80fc38b75c:  d8 21 00 70     stfd    f1,112(r1)
[...]
```

Early in the code the implementation uses the macro `GET_HIGH_WORD` to transform a `float` to an `integer` for posterior bit-wise operations. GLIBC’s math/math_private.h defines the macro using the code in Listing 20.
Listing 20. GET_HIGH_WORD macro definition

```c
#define GET_HIGH_WORD(i,d)                                      
  do {                                                            
    ieee_double_shape_type gh_u;                                  
    gh_u.value = (d);                                             
    (i) = gh_u.parts.msw;                                         
  } while (0)
```

A possible culprit causing a LHS stall in this macro is the operation that reads the attributes of the float to internal value and then reads it to the variable i. The POWER7 processor does not have a native instruction to move the contents of a floating-point register, bit-by-bit, to a Fixed-point register. The way this is accomplished on POWER is to store the FP number in the floating-point register to memory using a store operation and to then load the same memory location into a fixed-point (general-purpose). Since memory access is slower than register operations (even when accessing L1 data cache), the CPU is stalled during the store to complete the subsequent load.

**Note:** The document, "POWER ISA 2.06 (POWER7)" (see Related topics), contains more information.

Most often performance counter events trigger interrupts that save a PC address of an instruction close to the executing instructions. This can lead to assembly annotation which is not completely accurate. To mitigate this behavior POWER4 and later have a limited set of performance counters named marked. Marked instructions will generate less events per time frame; however, the PC instruction will be exact, resulting in an accurate assembly annotation. Marked events have the PM_MRK prefix in OProfile counter list obtained by opcontrol -l.

To double check the analysis, watch the `PM_MRK_LSU_REJECT_LHS` counter. Both counters, `PM_MRK_LSU_REJECT_LHS` and `PM_LSU_REJECT_LHS`, watch for the same performance event. However the marked counter (`PM_MRK_LSU_REJECT_LHS`) will generate less events per time frame but with a more accurate assembly annotation. (See Listing 21.)

Listing 21. perf record of PM_MRK_LSU_REJECT_LHS POWER7 event

```
$ perf record -C 0 -e rd082 taskset -c 0 ./hypot_bench_glibc
$ perf report
Events: 256K raw 0xd082
  64.61%  hypot_bench_gli  libm-2.12.so       [.] __ieee754_hypot
  35.33%  hypot_bench_gli  libm-2.12.so       [.] __GI___finite
```

This generates the assembly annotation in Listing 22.

Listing 22. perf report of PM_MRK_LSU_REJECT_LHS POWER7 event

```
:        00000080fc38b730 <.__ieee754_hypot>:
[...]
  1.23 :          80fc38b7a8:   c9 a1 00 70     lfd     f13,112(r1)
  0.00 :          80fc38b7ac:   f8 01 00 70     std     r0,112(r1)
 32.66 :          80fc38b7b0:   c8 01 00 70     lfd     f0,112(r1)
[...]
  0.00 :          80fc38b954:   f8 01 00 70     std     r0,112(r1)
  0.00 :          80fc38b958:   e8 0b 00 00     ld      r0,0(r11)
  0.00 :          80fc38b95c:   79 00 00 0e     rldimi  r0,r8,32,0
  61.72 :          80fc38b960:   c9 61 00 70     lfd     f11,112(r1
[...]
```
Another symbol shows about 35% of the generated events with similar behavior, in Listing 23.

**Listing 23. More highlights of the perf report**

```plaintext
0.00 : 80fc3a2610:   d8 21 ff f0     stfd  f1,-16(r1)
100.00 : 80fc3a2614:   e8 01 ff f0     ld      r0,-16(r1)
```

Based on this information, your optimization effort might be to eliminate these stalls by removing the FP to INT conversions. The POWER processor has a fast and efficient Float-Point execution unit so there is no need to perform these calculations with Fixed-Point instructions. The algorithm that POWER currently uses in GLIBC (sysdeps/powerpc/fpu/e_hypot.c) has removed all of the LHS stalls by using FP operations only. The result is the much simpler algorithm, in Listing 24.

**Listing 24. PowerPC GLIBC hypot source code**

```c
double __ieee754_hypot (double x, double y)
{
 x = fabs (x);
 y = fabs (y);

 TEST_INF_NAN (x, y);

 if (y > x)
  {
   double t = x;
   x = y;
   y = t;
  }

 if (y == 0.0 || (x / y) > two60)
  {
   return x + y;
  }

 if (x > two500)
  {
   x *= twoM600;
   y *= twoM600;
   return __ieee754_sqrt (x * x + y * y) / twoM600;
  }

 if (y < twoM500)
  {
   if (y <= pdnum)
    {
     x *= two1022;
     y *= two1022;
     return __ieee754_sqrt (x * x + y * y) / two1022;
    }
   else
    {
     x *= two600;
     y *= two600;
     return __ieee754_sqrt (x * x + y * y) / two600;
    }
  }

 return __ieee754_sqrt (x * x + y * y);
}
```

The `TEST_INF_NAN` macro is a further small optimization which tests if a number is NaN or INFINITY before starting further FP operations (this is due to the fact that operations on NaN and INFINITY can raise FP exceptions and the function specification does not allow that). On POWER7 the
isinf and isnan function calls are optimized by the compiler to FP instructions and do not generate extra function calls, while on older processors (POWER6 and older) it will generate a call to the respective functions. The optimization is basically the same implementation, but inlined to avoid function calls.

Finally to compare both implementations, perform the following simple test. Recompile GLIBC with and without the new algorithm and compare the total time for each benchmark run. The default GLIBC implementation results are in Listing 25:

**Listing 25. Benchmark with default GLIBC hypot**

```bash
$ /usr/bin/time ./hypot_bench_glibc
INF_CASE : elapsed time: 14:994339
NAN_CASE : elapsed time: 14:707085
TW060_CASE : elapsed time: 12:983906
TW0500_CASE : elapsed time: 10:589746
TWOM500_CASE : elapsed time: 11:215079
NORMAL_CASE : elapsed time: 15:325237
```

79.80user 0.01system 1:19.81elapsed 99%CPU (0avgtext+0avgdata 151552maxresident)k
0inputs+0outputs (0major+48minor)pagefaults 0swaps

The optimized version results are in Listing 26:

**Listing 26. Benchmark with optimized GLIBC hypot**

```bash
$ /usr/bin/time ./hypot_bench_glibc
INF_CASE : elapsed time: 4:667043
NAN_CASE : elapsed time: 5:100940
TW060_CASE : elapsed time: 6:245313
TW0500_CASE : elapsed time: 4:838627
TWOM500_CASE : elapsed time: 8:946053
NORMAL_CASE : elapsed time: 6:245218
```

36.03user 0.00system 0:36.04elapsed 99%CPU (0avgtext+0avgdata 163840maxresident)k
0inputs+0outputs (0major+50minor)pagefaults 0swaps

This is a final performance improvement of more than 100%, cutting the benchmark time by half.

**Conclusion**

Performance evaluation with hardware counter profiling is a powerful tool to understand how a workload behaves on a certain processor and to give hints of where to work on performance optimizations. The latest POWER7 processor has hundreds of performance counters available so we presented a simple model of how to map the workload to CPU stalls. Understanding the POWER7 CBM is somewhat complicated so we also explained tools for Linux that simplify it. Strategies for performance evaluation focused on how to find hotspots, how to understand the memory pattern of an application, and how to use the POWER7 CBM. Finally, we used a recent optimization done on a trigonometric function within GLIBC to explain the performance analysis that was used to result in the optimized code.

**Appendix**

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3; with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLIBC hypot benchmark</td>
<td>hypot_bench.tar.gz</td>
<td>6KB</td>
</tr>
<tr>
<td>Python script to format perf output</td>
<td>power7_cbm.zip</td>
<td>2KB</td>
</tr>
</tbody>
</table>
Related topics

- Read more about the operation of processors in Modern Microprocessors - a 90 minute guide.
- Learn more about OProfile on the project website.
- Learn more about the perf tool—the code is maintained within the Linux kernel source.
- Read Identify performance bottlenecks with OProfile for Linux on POWER (John Engel, developerWorks, May 2005) for a comprehensive guide to OProfile (although not updated for POWER7).
- Explore the IBM Wiki Using perf on POWER7 systems. a comprehensive guide to perf (although not updated for POWER7).
- Find raw POWER7 codes for perf on the libpfm4 project site.
- Read the description of the SPECcpu2006 483.xalancbmk component.
- See the GNU C Library at the project page.
- Check out a didactic and informative article on modern processors pipelining and instruction-level parallelism, instruction dependencies, branches prediction and other topics related to CPU architecture.
- Identify performance bottlenecks with OProfile for Linux on POWER.
- Read more on how to use perf on POWER7 systems.
- Dig into the GLIBC source code.
- Review the SPECCPU2006 documentation for some examples used in this article.
- In the developerWorks Linux zone, find hundreds of how-to articles and tutorials, as well as downloads, discussion forums, and a wealth of other resources for Linux developers and administrators.
- Start developing with product trials, free downloads, and IBM Bluemix services.