

# Contrasting Linux on POWER Profilers

Linux Performance Considerations

*IBM's Linux Technology Center*

Rob Nava, Bill Buros

## Abstract

The goal of this report is to compare and contrast three commonly used Linux™ on POWER™ profilers: OProfile, gprof and Tprof. These open-source profiler tools are available to end users and programmers for both SUSE™ and Red Hat® Linux distributions.

These profilers are used to help pinpoint common performance problems in an application. In this report, a simple sort program incorporating three different sorting algorithms is profiled in an exercise to demonstrate each profiler's strengths and weaknesses, including any overhead the profilers add during run-time.

## 1 Introduction

There are several open-source profilers available for Linux, each offering different features. Three popular profilers for Linux on POWER systems are OProfile, gprof, and Tprof. OProfile allows cycle event profiling as well as profiling based on any of the hardware performance counters. The gprof profiler has the ability to use profiling hooks at compile time and is also the only profiler of these three that profiles the application itself (as opposed to profiling system-wide.) Additionally, the gprof profiler also has a working call-graph function. The Tprof profiler is a system-wide profiler like OProfile, but it also adds the ability to profile Java code.

For this paper, each of the three profilers was run on a simple sorting application that implements three classic sorting algorithms. The sort program sorts a set of 15 million random numbers (generated using the same seed so that the same numbers are used consistently throughout the exercise) using the Merge, Quick, and Heap sorting algorithms. Each of the algorithms was implemented in its own function.

The sort application was written in C and was compiled with the gcc compiler using the -O2 option in every case. The -O2 optimization was chosen over the -O3 optimization because the -O3 optimization inlines the Heap sorting algorithm, and the inlining of small functions can make the calling function appear more busy than it is and conceal the true hot functions. After compiling,

the sort application ran five times on a 4-way POWER5™ box with SMT turned off and only one processor enabled. This was done because the sort application is a single process program and multiple processors would skew the CPU utilization data.

The results were consistent running on both SLES9 SP1 and RHEL4 U2, but only the SLES9 SP1 data is presented in this report. The results for each of the sorts were within 2% of each other with the median result of the five being recorded.

For example, the basic results of the Merge, Quick and Heap sorting of 15 million random numbers is shown in the following table:

Sorting Algorithm	Number of Elements	Time to Complete
Quick Sort	15,000,000	<b>3.84 seconds</b>
Heap Sort	15,000,000	9.46 seconds
Merge Sort	15,000,000	11.68 seconds

It is clearly evident that sorting with the Quick Sort algorithm was significantly faster at sorting the random numbers. Because each of the sorting algorithms is encapsulated in its own function, it is easy for a profiler to illustrate the differences in the sorting functions. The differences between the three sort algorithms are analyzed in this report in order to identify and understand some of the reasons behind the differences in performance.

OProfile and gprof are the more commonly used profilers in the Linux community and the simple approach for using these are covered first. Tprof comes with the more comprehensive Performance Inspector toolkit, providing low-overhead profiling capabilities and in particular the capability to profile Java applications.

OProfile offers a significantly more sophisticated approach where different hardware event counters can be used to trigger sampling of data, which is discussed in Section 5.

## 2 OProfile

OProfile is a popular Linux profiler available from Source Forge ([oprofile.sourceforge.net](http://oprofile.sourceforge.net)) [1]. It is relatively easy to use and does not require kernel patching on SLES 9-based or RHEL 4-based POWER kernels. It is unique in that it offers profiling based on user-specified hardware counter events.

John Engel has provided a report [2] on how to identify performance bottlenecks with OProfile for Linux on POWER.

## 2.1 Installing OProfile

Installing OProfile requires the following process:

1. Download from the following URL: <http://oprofile.sourceforge.net/news/oprofile-0.9.1.tar.gz> is the latest version as of the writing of this report)
2. Unarchive the tar ball with the following commands:

```
# tar -xzvf oprofile-0.9.1.tar.gz
# cd oprofile-0.9.1/
```

3. Configure, make and install with the following commands:

```
# ./configure --with-kernel-support
# make
# make install
```

The installation should put the OProfile tools in the `/usr/local/bin/` directory.

An important note during the configuration step is to use the `--with-kernel-support` flag when running on a 2.6 kernel to indicate that the kernel provides the OProfile device driver. Without this flag, the configure step will fail with the message: "unsupported kernel version."

A normal (expected) warning on the configure step is: "a working Qt not found, no GUI will be built". Many users do not use the OProfile GUI, and this is not covered in this paper.

## 2.2 Running OProfile

The default event for OProfile is CYCLES, which is simply the processor frequency. For example, a 1.90 GHz processor will go through 1.9 billion CYCLES events every second.

Note: The user must have 'root' authority to run the following commands.

**EXAMPLE:** Running with the CYCLES event on the sort application:

```
opcontrol --init # Loads the OProfile module and oprofilefs
opcontrol --vmlinux=/boot/vmlinux # Point to running (booted) kernel
opcontrol --reset # clear the counters
opcontrol --event=CYCLES:1000000 # Sample every 1 million CYCLES
opcontrol --start # Start the profiling
./sort -c 1500000 -s 1 -s 2 -s 4 -r 1 # Run the sort application
```

```

opcontrol --dump # Flush the collected data into the daemon
opcontrol --stop # Stop collecting data
opcontrol --shutdown # Kill the daemon
opreport --symbols > opreport.sym_out # List symbol information
opreport --long-filenames > opreport.filenames_out # Show full path to files
opannotate --source > opannotate.source_out # Annotate source code
opannotate --assembly > opannotate.assembly_out # Annotate assembly code

```

## 2.3 OProfile Output

**EXAMPLE:** Output of OProfile with the CYCLES event on the sort application:

```

CPU: ppc64 POWER5, speed 1904.44 MHz (estimated)
Counted CYCLES events (Processor cycles) with a unit mask of 0x00 \
      (No unit mask) count 1000000

```

samples	%	app name	symbol name
17716	30.9818	sort	heap_sort
8666	15.1551	sort	merge_sort
5937	10.3826	sort	quick_sort
5684	9.9402	libc.so.6	random
4569	7.9903	libc.so.6	cfree
3742	6.5440	libc.so.6	malloc
2433	4.2548	libc.so.6	_int_malloc
2175	3.8036	libc.so.6	random_r
1380	2.4133	libc.so.6	_int_free
1375	2.4046	sort	swap
1075	1.8800	sort	array_populate
.			
.		[SNIPPED DATA]	
.			

The OProfile output shows that `heap_sort` had the majority of the samples with 30.98%, `merge_sort` had 15.36% of the samples, and the `quick_sort` function had 11.56% of the samples. While the percentages of samples are not directly proportional to the results (Merge completed in 11.68 seconds; Quick in 3.84 seconds, and Heap in 9.46 seconds), the 'hotness order' of the functions are in line with the results. OProfile is generally a good starting point for deeper analysis.

## 2.4 Annotated source code profiling

```

/*
 * Command line: opannotate --source
 *
 * Interpretation of command line:
 * Output annotated source file with samples
 * Output all files

```

```

*
* CPU: ppc64 POWER5, speed 1904.44 MHz (estimated)
* Counted CYCLES events (Processor cycles) with a unit mask of 0x00 (No unit mask) coun
*/
/*
* Total samples for file : "/profdata/sort.c"
*
* 33794 60.2281
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
.
. [SNIPPED DATA]
.
/* code for quick sort algorithm */
#define QUICK_SORT 2
:
88 0.1539 :void quick_sort(bit_size *A, bit_size beg, bit_size end) { /* quick_sort
: if (end > (beg + 1)) {
28 0.0490 : bit_size piv = A[beg];
: bit_size l = beg + 1;
: bit_size r = end;
:
73 0.1277 : while (l < r) {
2640 4.6168 : if(A[l] <= piv) {
1304 2.2804 : l++;
:
: else {
1409 2.4641 : swap(&A[l], &A[--r]);
:
: }
:
177 0.3095 : swap(&A[--l], &A[beg]);
48 0.0839 : quick_sort(A, beg, l);
59 0.1032 : quick_sort(A, r, end);
: }
111 0.1941 :}
/* end of quick sort algorithm code */
:
:
/* code for heap sort algorithm */
#define HEAP_SORT 4
:
: void heap_sort(bit_size *A, bit_size N) { /* heap_sort total: 17716 30.
: bit_size n = N, i = n/2, j, w;
: bit_size t;
:
: for (;;) {
: if (i > 0) {
34 0.0595 : i--;
27 0.0472 : t = A[i];
: } else {

```

```

:      n--;
179  0.3130 :      if (n == 0) return;
:      t = A[n];
338  0.5911 :      A[n] = A[0];
:      }
:
:      j = i;
185  0.3235 :      w = (i * 2) + 1;
:
:      while (w < n) {
6178 10.8041 :          if (((w + 1) < n)  &&  (A[w + 1] > A[w])) {
2045  3.5763 :              w++;
:          }
1717  3.0027 :          if (A[w] > t) {
4313  7.5426 :              A[j] = A[w];
:              j = w;
619   1.0825 :              w = (j * 2) + 1;
1654  2.8925 :          } else {
:              break;
:          }
:      }
411  0.7188 :      A[j] = t;
:  }
: }
: /* end of heap sort algorithm */
.
. [SNIPPED DATA]
.

```

[The number of samples is in the first column while the percentage of the total samples that number represents is in the second column.]

The C source code is annotated with the number of samples (in this case the CYCLES events) attributed to the corresponding portions of the source code.

The Quick and Heap sort algorithms are both presented in their entirety above. It is evident that the Quick sort (5,937 samples) encountered many fewer CYCLES events than the Heap sort (17,716 samples), which is consistent with the summary.

## 2.5 OProfile Overhead

The overhead of OProfile is dependent on a couple of things:

- the workload that is being profiled
- how often samples are being taken

Some workloads are more sensitive to OProfile than others. For a simple application like this sorting application, OProfile does not impact performance very much at all, in this case adding about 1% of time.

The OProfile documentation provides some good guidance and suggestions for determining how often to sample, which can affect the overhead seen. Obviously, the more often samples that are taken, the more overhead that will be added.

Overhead of running OProfile while Merge, Quick and Heap sorting (five runs, Median recorded):

Algorithm	Number of Elements	Time to Complete	Time with OProfile	Overhead
Quick Sort	15,000,000	3.84 seconds	3.88 seconds	+1%
Heap Sort	15,000,000	9.46 seconds	9.57 seconds	+1%
Merge Sort	15,000,000	11.68 seconds	11.79 seconds	+1%

Currently, OProfile is not capable of supplying call graphs, which keep track of what functions are called by other functions, on POWER architecture. There is a patch in the works, but it has not been accepted into the main release. The issue is with the kernel supporting the POWER architecture call graphs and not the OProfile code. For more details, Carl Love and Maynard Johnson have provided a report [3] that describes the implementation details of OProfile on the POWER base.

OProfile must be enabled in the kernel configuration in order for the function names to show up in the output report. If OProfile is not enabled, everything is attributed to vmlinux with no function name breakdown. Currently, this is turned on by default in both SUSE and Red Hat Linux on POWER releases.

### 3 gprof

Like all profilers, gprof helps identify parts of a program that take up most of the run time. The gprof profiler has been around for a long time as an aspect of the gcc tool-chain.

A unique property of gprof is that the profiling is done specifically on the application, and not the entire running system.

Another characteristic of gprof is that in addition to breaking down a program into the functions that take up the most processing time, it can also provide a call graph feature. The call graph feature illustrates which functions call which other functions. This is useful in tracking the flow of a program and determining if a function is being called more or less often than expected. This can help identify hot spots.

Like OProfile, gprof also does not require kernel patching. However, unlike OProfile, it does require that the program be recompiled to enable gprof functionality. If the program cannot be recompiled, gprof cannot be used. In general, it is assumed here that the program is being recompiled with gcc. However, IBM®VisualAge®C/C++, and Fortran compilers as well as other compilers support the gprof option as well as gcc.

### 3.1 Installing gprof

One of gprof's strengths is that it is often times already installed on a system as part of a developer's package. If it is not installed, it can be found on the install media. Simply search for a gprof rpm package and install it.

### 3.2 Running gprof

Running gprof is the easiest of the three featured profilers. A simple recompiling of the application is all that is required. While this makes gprof profiling easy to do, unfortunately, when recompilation is not possible (no access to the application source code), gprof cannot be used. And unlike OProfile, gprof does not require that it be run with 'root' authority.

**EXAMPLE:** Running gprof on the sort application:

```
gcc -O2 -m32 -g -pg -o sort sort.c # Compile with the -pg option
./sort -c 15000000 -s 1 -s 2 -s 4 -r 1 # Run the sort application
gprof sort > gprof.out # Run 'gprof' against the executable.
```

The -pg option is only necessary at compile time if call graph output is desired. If only timing information is needed, the -pg option can be omitted at compiler time, but it will still be necessary at link time. Most of the time, the call graph is desired.

After the application is recompiled with the -pg option, the application runs as usual. After the application completes, a gmon.out file, which contains the raw prof performance data, is automatically created in the directory where the application ran. Then, gprof can be run with any number of options (execute gprof -h to see the list of options) to create a gprof.out report.

### 3.3 gprof Output

**EXAMPLE:** The flat profile Output of gprof on the sort application:

```
Each sample counts as 0.01 seconds.
%   cumulative   self           self   total
time  seconds    seconds   calls  s/call  s/call  name
41.63    94.71    94.71         1    94.71   94.71  heap_sort
23.04   147.13    52.42         1    52.42   78.49  quick_sort
22.73   198.85    51.72         1    51.72   51.72  merge_sort
11.46   224.92    26.07 231276262     0.00    0.00  swap
 1.13   227.50     2.58         3     0.86    0.86  array_populate
```

One notable difference in gprof output is that there is no reference to glibc as there is in the OProfile output. The glibc data is not accounted for because it is not part of the application. This is why the percentage of time allocated to the heap\_sort(), quick\_sort(), and merge\_sort() functions will not match up with system-wide profilers like OProfile or Tprof.

The gprof profiler only profiles the application itself. This can be useful if the user wants to filter out system 'noise.'

Looking at the timing information for gprof suggests that Heap sort is the hot function. Given the data (without profilers running the background,) this is expected. However, this profile data also suggests that the quick\_sort() and merge\_sort() sorting functions are about equal with each other in terms of percentage of time taken. Again, looking at the non-profiled runs suggest the two functions are not equal. The answer as to why this is the case lies in the call graph.

**EXAMPLE:** The call graph output of gprof on the sort application:

```

Call graph (explanation follows)

granularity: each sample hit covers 4 byte(s) for 0.00\% of 227.50 seconds

index \% time    self  children    called    name
-----
[1]    100.0      0.00  227.50          <spontaneous>
      94.71    0.00    1/1      main [1]
      52.42   26.07    1/1      heap_sort [2]
      51.72    0.00    1/1      quick_sort [3]
      2.58    0.00    3/3      merge_sort [4]
      array_populate [6]
-----
      94.71    0.00    1/1      main [1]
[2]    41.6      94.71    0.00      1      heap_sort [2]
-----
      9999721          quick_sort [3]
      52.42   26.07    1/1      main [1]
[3]    34.5      52.42   26.07    1+9999721 quick_sort [3]
      26.07    0.00  231276262/231276262 swap [5]
      9999721          quick_sort [3]
-----
      29999998          merge_sort [4]
      51.72    0.00    1/1      main [1]
[4]    22.7      51.72    0.00    1+29999998 merge_sort [4]
      29999998          merge_sort [4]
-----
      26.07    0.00  231276262/231276262 quick_sort [3]
[5]    11.5      26.07    0.00  231276262 swap [5]
-----
      2.58    0.00    3/3      main [1]
[6]    1.1      2.58    0.00      3      array_populate [6]
-----

```

### 3.4 gprof Overhead

The overhead associated with running gprof on an application varies greatly. In the case of the sort application, Heap sorting took no performance hit, Merge sort took a small performance hit, and Quick sort took a big performance hit. The Quick sort was so adversely affected that it went from the fastest to by far the slowest sorting method.

The reason for these results lies with the call graph feature of gprof.

The call graph in the previous section shows that while the Merge and Heap functions do not call other functions, the Quick Sort function calls a function called 'swap'. Because the merge\_sort() and heap\_sort() functions are self contained, only the Quick function was heavily affected by the extra code that gprof adds to create the call graph. In order to illustrate this point further, one simply has to compile without the -pg option, which eliminates the call graph. This is illustrated below in the Overhead section.

The overhead of running gprof with and without call graph while Merge, Quick and Heap sorting (5 runs, Median recorded) is shown in the following table:

Algorithm	Elements	Time to Complete (Baseline)	Time to Complete with gprof	Time to Complete with gprof NO C.G.
Quick Sort	15,000,000	3.84 seconds	<b>21.77 seconds (466%)</b>	4.06 seconds (6%)
Heap Sort	15,000,000	9.46 seconds	9.45 seconds (0%)	9.58 seconds (1%)
Merge Sort	15,000,000	11.68 seconds	14.18 seconds (21%)	11.78 seconds (1%)

## 4 Tprof

Tprof profiling can be done by using the Performance Inspector toolkit. It is not a single tool, but a collection of tools working together:

- kernel instrumentation (generate interrupts)
- trace facility (to record interrupt data)
- symbol resolution facility (translate addresses to names)
- post processor (coordinate post-processing.)

While counter-based sampling (sampling every  $n$  number of events) with Tprof exists on other platforms, only time-based sampling (sampling at constant intervals) is available on the POWER architecture at this time.

Older versions of Tprof required a bit of effort to run as it required patching and building a kernel and loading a special driver. Custom hooks could also be inserted into the kernel.

Tprof interrupted the system periodically, determined the address of the interrupted code, and with the process id (pid) and thread id (tid), recorded the Tprof hook in the software trace buffer and then returned to the interrupted code.

The Tprof hooks were stored in a trace file using the swtrace facility. The "post" command was used to post process the sampling information. "post" produced the Tprof.out report which included several sub-reports within it with regard to processes, threads and subroutines.

The newer version of Performance Inspector (06-23-05) is designed to work without patching the kernel. It uses the same system hooks used in OProfile when using 2.6 kernels. This version registers with the OProfile hooks to get timer ticks and task exits. However, the new version only supports SMP versions of 2.4 kernels which still must be patched.

With either version, the **run.tprof script** is used to obtain the Tprof trace and generate the subsequent report.

Tprof is the only profiler featured in this report that can profile Java applications. Indeed, a great deal of the Tprof focus is on Java profiling.

#### **4.1 Installing Tprof**

Installing either the older or newer version of Performance Inspector is more difficult on 2.4 kernels because of the need for patching the kernel. In either case, the included documentation is thorough and should not present many problems. Here are the basic steps involved:

1. Download the Performance Inspector package from the following URL:  
<http://sourceforge.net/projects/perfinsp>
2. Untar the package.
3. Patch the kernel. (2.4 kernels only)
4. Build kernel and driver. (2.4 kernels only)
5. Reboot with new kernel. (2.4 kernels only)
6. Install the tools.

Follow the instructions included with the package documentation for specifics on patching/building kernels and installing tools.

## 4.2 Running Tprof

Since there is not actual tool called "tprof", the profiling is done by a series of commands using the `/piperf/bin/run.tprof` script.

While the provided script standardizes the profiles taken and reports generated, it can cause problems because it requires that the `run.tprof` script be run in one window while running the workload in another window. This might be fine for a lot of applications, but it makes capturing only the workload, from beginning to end, a bit more difficult because it relies on human interaction.

For the sake of this report, the `run.tprof` script was modified so that it could take a workload to be profiled as the first argument to the script. The modified script is provided later in this report.

**EXAMPLE:** Running Tprof on the sort application:

```
run.tprof.prog "./sort -c 1500000 -s 1 -s 2 -s 4 -r 1" # Run modified run.tprof script
```

The changes made to the `run.tprof` script include removing the need for user input when starting and ending the profiling. Instead, the script takes the workload as an argument (in double quotes) and begins profiling immediately before the workload is started. Then the script stops the profiling after the workload completes. For a complete listing of the modified `run.tprof.prog` script, please see the modified script in Section 6.

## 4.3 Tprof Output

Tprof counts "ticks". It then generates multiple lists of functions that take the most ticks (TKS). The following data is found within the `tprof.out` report.

**EXAMPLE:** The Module output of Tprof on the sort application:

```
=====
)) Module
=====

LAB      TKS      %%%      NAMES

MOD  17581  58.72    /profdata/tprof_out-02/sort
MOD  11680  39.01    /lib/tls/libc.so.6
MOD    666   2.22     vmlinux
MOD     7   0.02     /lib/ld-2.3.3.so
MOD     2   0.01     NoModule
MOD     2   0.01     /bin/bash
```

**EXAMPLE:** The Module\_Symbol Output of Tprof on the sort application:

```

=====
)) Module_Symbol
=====

LAB      TKS      %%%      NAMES

MOD  17581  58.72   /profdata/tprof_out-02/sort
SYM   9460  31.60   heap_sort
SYM   4077  13.62   merge_sort
SYM   3387  11.31   quick_sort
SYM    455   1.52   swap
SYM    118   0.39   array_populate
SYM     84   0.28   <plt>

MOD  11680  39.01   /lib/tls/libc.so.6
SYM   4117  13.75   __random
SYM   3061  10.22   __cfree
SYM   2769   9.25   __GI___libc_malloc
SYM    845   2.82   _int_malloc
SYM    377   1.26   __random_r
SYM    283   0.95   _int_free
SYM    107   0.36   NoSymbolFound
SYM     94   0.31   rand
SYM     10   0.03   malloc_consolidate
SYM     3   0.01   __gconv_transform_utf8_internal
SYM     2   0.01   __GI___mbrtowc
SYM     1   0.00   __GI__dl_mcount_wrapper_check
SYM     1   0.00   __GI_strncpy
SYM     1   0.00   _int_realloc
SYM     1   0.00   __getmntent_r_internal
SYM     1   0.00   __brk
SYM     1   0.00   __cfree
SYM     1   0.00   __GI_open
SYM     1   0.00   __mbsinit
SYM     1   0.00   NoSymbolFound
SYM     1   0.00   __GI___lxstat
SYM     1   0.00   __GI___mmap
SYM     1   0.00   strncat

MOD    666   2.22   vmlinux
SYM    102   0.34   .plpar_hcall
SYM     96   0.32   .update_mmu_cache
SYM     75   0.25   .save_remaining_regs_SystemCall
SYM     43   0.14   ._raw_spin_lock
.
. [SNIPPED DATA]
.

```

Because Tprof profiles system-wide, like OProfile, the results are more consistent with OProfile results with Heap sort taking about 31% of the time, Merge sort taking 13% of the time, and Quick Sort taking 11% of the time.

## 4.4 Tprof Overhead

The overhead on Tprof is nearly negligible for the sort application.

The overhead of running Tprof while Merge, Quick and Heap sorting (5 runs, Median recorded) is shown in the following table:

Algorithm	Number of Elements	Time to Complete	Time to Complete (w/Tprof)	Overhead
Quick Sort	15,000,000	3.84 seconds	3.84 seconds	0%
Heap Sort	15,000,000	9.46 seconds	9.44 seconds	0%
Merge Sort	15,000,000	11.68 seconds	11.68 seconds	0%

## 5 OProfile - hardware event triggers

This special section begins to introduce a powerful feature of OProfile that allows the sampling trigger to be changed to any hardware event counter supported on POWER systems.

Every  $n$  number of times an OProfile-specified event occurs, OProfile samples the system to see what application and what functions are currently running. This is useful to see where the system is spending its time during these events.

The events include such things as the default processor cycles-based event CYCLES, the data TLB miss event PM\_DTLB\_MISS\_G43, the data loaded from memory event PM\_DATA\_FROM\_LMEM\_G49, and many others. In fact, as of version 0.9.1, there are nearly 100 events available for POWER5 architecture.

The list of available events can be seen by running **opreport --list-events**.

**NOTE:** The events listed will often appear with similar descriptions in multiple groups.

**opcontrol --list-events** # List the available OProfile events.

```
1 oprofile: available events for CPU type "ppc64 POWER5"
2
3 Obtain PowerPC64 processor documentation at:
4 http://www-306.ibm.com/chips/techlib/techlib.nsf/productfamilies/PowerPC
5 CYCLES: (counter: 2)
6     Processor cycles (min count: 10000)
--> 7 PM_DTLB_MISS_GP9: (counter: 0)
8     A TLB miss for a data request occurred (min count: 1000)
9 PM_DC_PREF_L2_CLONE_L3_GP9: (counter: 1)
10    L2 prefetch cloned with L3 (min count: 1000)
```

```

11 PM_LSU_LMQ_LHR_MERGE_GP9: (counter: 2)
12     Dcache miss occured for the same real cache line as earlier req, merged
    into LMQ (min count: 1000)
13 PM_LSU_SRQ_EMPTY_CYC_GP9: (counter: 3)
14     Cycles Store Req Queue empty (min count: 1000)
15 PM_INST_CMPL_GP9: (counter: 4)
16     Number of PPC inst completed (min count: 10000)
17 PM_RUN_CYC_GP9: (counter: 5)
18     Proc cycles gated by the run latch (min count: 10000)
.
. [SNIPPED DATA]
.
39 PM_INST_CMPL_G14: (counter: 4)
40     Instructions completed (min count: 10000)
41 PM_RUN_CYCLES_G14: (counter: 5)
42     Processor cycles gated by run latch (min count: 10000)
--> 43 PM_DTLB_MISS_G43: (counter: 1)
44     Data TLB miss occurred (min count: 1000)
45 PM_LD_MISS_L1_G43: (counter: 2)
46     L1 D cache load miss (min count: 1000)
47 PM_LD_REF_L1_G43: (counter: 3)
48     L1 D cache load references (min count: 1000)
49 PM_INST_CMPL_G43: (counter: 4)
50     PPC instructions completed (min count: 10000)
51 PM_RUN_CYC_G43: (counter: 5)
52     Processor cycles gated by run latch (min count: 10000)
53 PM_DATA_FROM_L2_G44: (counter: 0)
.
. [SNIPPED DATA]
.

```

In the abbreviated list above, there are two events that have very similar descriptions: PM\_DTLB\_MISS\_GP9 (line 7) and PM\_DTLB\_MISS\_G43 (line 43). Both appear to point out Data TLB misses. And in fact, both do. The difference is that they are part of different event groups and therefore have different group numbers.

Because POWER5 architecture has six performance counters, it is possible to sample up to six events at once; however, the events must all be in the same event group. These groups of events are pre-determined and limited to six events per group. A problem arises then, if an event like a Data TLB miss is to be sampled with an event from a different group. To work around this, the same event is placed in a different group and given a different group number. In the example above, PM\_DTLB\_MISS is both in group GP9 and group GP43 so it could be sampled with events from either group.

For example, if there was a need to take samples on L1 Data cache load misses and Data TLB misses, the PM\_LD\_MISS\_L1\_G43 and PM\_DTLB\_MISS\_G43 events would be used.

However, if there was a need for sampling when the L2 prefetch was cloned with L3 and Data TLB misses, the PM\_DC\_PREF\_L2\_CLONE\_L3\_GP9 and PM\_DTLB\_MISS\_GP9 events would be used.. The PM\_DTLB\_MISS\_G43 and PM\_DTLB\_MISS\_GP9 events are the same.

So, in effect, the number of events listed in OProfile can be a bit misleading because some events are listed multiple times as part of different groups.

Commonly used groups on POWER5 are the following:

Groups	Description
1	CPI Cycles/Instruction
50	Mem/L2/L3 Cache
41, 47	TLB/L1 Cache
41, 42, 43	Mispredicted Branch
19	LHS (Load-Hit-Store Hazard)
25	Misaligned Loads/Stores
44	Data TLB
78, 79, 80, 83	FPU

## 6 Tprof run script

### Modified run.tprof.prog script.

This script takes the workload as the first argument to the script and starts the profiler right before the workload begins and stops the profiler immediately after the workload completes.

```

echo "Tprof Measurement Procedure."
echo "Tprof Measurement Procedure." >run.tprof.log
date >>run.tprof.log

if [ "$1" = "-h" ] ; then
echo " Command syntax: "
echo " "
echo "   run.tprof.prog [Program To Trace, Buffer Size, Perf Counter ID, No. of Events,
echo " "
echo " Parameters: "
echo "           Buffer Size           - Size of the trace buffer to user"
echo "                               Default is 4 Megbytes"
#echo "/*****/"
#echo "If you are profiling Java, then you need to perform one of the following:"
#echo " "
#echo " "
#echo " 1. Specify the -Xrunjprof:jita2n,threadinfo,fnm=$PERFTPROF/log option on the ja
#echo " 2. Set the IBM_JAVA_OPTIONS environment variable as follows:"
#echo "   IBM_JAVA_OPTIONS=-Xrunjprof:jita2n,threadinfo,fnm=$PERFTPROF/log"
#echo " "
#echo "/*****/"
exit
fi

```

```

#echo " "
#echo "If you are profiling Java, then you need to perform one of the following:"
#echo " "
#echo "1. Specify the -Xrunjprof:jita2n,threadinfo,fnm=$PWD/log option on the java start
#echo "2. Set the IBM_JAVA_OPTIONS environment variable as follows:"
#echo "   export IBM_JAVA_OPTIONS=-Xrunjprof:jita2n,threadinfo,fnm=$PWD/log"

export PATH=$PATH:/piperf/bin/

read JA2N<perftprof.dir
rmja2n $JA2N >>run.tprof.log

if [ "$1" == "" ] ; then
    echo "Need something to trace!"
    exit
fi

if [ "$2" != "" ] ; then
    swtrace init -s $2
else
    swtrace init -s 5
fi

swtrace disable >>run.tprof.log
swtrace enable 17 25 >>run.tprof.log
if [ "$3" != "" ] ; then
    echo "User Perf counter for tprof"
    swtrace pcnt $3 $4
fi
swtrace on >>run.tprof.log 2>>run.tprof.log
echo " "
#echo "Start the program to be tprof'ed in another window."
#read -p "Press enter in this window when you want to start collecting tprof data."
swtrace enable 16 >>run.tprof.log 2>>run.tprof.log
echo " "
#read -p "Data being collected. Press any key to stop tprof data collection."
echo "Collecting Data..."

./$1

swtrace off >>run.tprof.log 2>>run.tprof.log
swtrace get >>run.tprof.log 2>>run.tprof.log
swtrace disable >>run.tprof.log 2>>run.tprof.log
echo "Trace data collected. Producing reports."
    post -clip 0 -nv -kmap /boot/System.map >>run.tprof.log 2>>run.tprof.log
echo " "
echo "tprof reports produced in tprof.out. Detail run information in run.tprof.log"

```

## 7 sort application source

### The source code, in C, for the sort application.

```
/* 'sort.c' written by Karl Rister (krister@us.ibm.com) */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>

/* generic defines and other stuff */
#define RAND_REPRO 1
#define RAND_KERNEL 2
// #define DATA_WIDTH_64_BIT 1
#define NUM_SORTS 3

#ifdef _DATA_WIDTH_64_BIT_
typedef long long bit_size;
#define BIT_FLAG "%lld"
#define BIT_WIDTH "64"
#else
typedef long bit_size;
#define BIT_FLAG "%ld"
#define BIT_WIDTH "32"
#endif
/* end of generic defines and other stuff */

/* generic functions */
void usage() {
    printf("usage: sort -c NUMBER -s SORT_INDEX\n");
    printf("-c NUMBER\t\tNUMBER is the amount of array elements to use\n");
    printf("-s SORT_INDEX\t\tSORT_INDEX is either 1 for merge sort, 2 for quick sort, or 4 for radix sort\n");
}

void swap (bit_size *a, bit_size *b) {
    bit_size t;
    t = *a;
    *a = *b;
    *b = t;
}

void * array_populate(bit_size size, int rand_mode) {
    bit_size *array;
    bit_size i;
    FILE *fp;
    bit_size num_read = 0;

    /* allocate memory to be the number of elements requested multiplied by the size of each element
    array = malloc(sizeof(bit_size) * size);

```

```

if (array == NULL)
    return NULL;

switch(rand_mode) {
case RAND_REPRO:
    srand(1);
    for(i=0; i<size; i++) {
        array[i] = rand();
    }
    break;
case RAND_KERNEL:
    fp = fopen("/dev/urandom", "r");
    if (fp) {
        num_read = fread(array, sizeof(bit_size), size, fp);
        fclose(fp);
        if (num_read != size) {
            printf("ERROR: only read " BIT_FLAG " elements (not " BIT_FLAG ").\n", num_read,
                free(array);
                exit(1);
            }
        }
    }
    else {
        printf("ERROR: could not open /dev/urandom for kernel random mode.\n");
        return NULL;
    }
    break;
}

return array;
}

void array_print(bit_size *array, bit_size size) {
    bit_size i;

    for(i=0; i<size; i++) {
        printf(BIT_FLAG ". " BIT_FLAG "\n", i, array[i]);
    }
}
/* end of generic functions */

/* code for merge sort algorithm */
#define MERGE_SORT 1

void merge_sort(bit_size *A, bit_size n) {
    bit_size i, j, P1, P2;
    bit_size r = n & 0x1; /* optimized version of n % 2 */
    bit_size *A1;
    bit_size *A2;

    if (n <= 1)
        return;

    /* split the array into two smaller arrays, first we must allocate memory to do so */

```

```

A1 = malloc(sizeof(bit_size) * (n / 2));
if (A1 == NULL) {
    printf("Failed to malloc A1\n");
    exit(1);
}
A2 = malloc(sizeof(bit_size) * ((n / 2) + r));
if (A2 == NULL) {
    printf("Failed to malloc A2\n");
    exit(1);
}

/* copy the values into the new arrays */
for(i=0; i<(n / 2); i++) {
    A1[i] = A[i];
    A2[i] = A[i + (n / 2)];
}
if (r) {
    A2[(n / 2)] = A[n-1];
}

/* recursively call this function on the new arrays */
merge_sort(A1, n / 2);
merge_sort(A2, (n / 2) + r);

/* merge A1 and A2 and put the result in A */
P1 = P2 = 0;
for(i=0; i<n; i++) {
    if (A1[P1] < A2[P2])
        A[i] = A1[P1++];
    else
        A[i] = A2[P2++];

    if (P1 >= (n / 2)) {
        for(j=P2; j<((n / 2) + r); j++) {
            i++;
            A[i] = A2[j];
        }
        break;
    }

    if (P2 >= ((n / 2) + r)) {
        for(j=P1; j<(n / 2); j++) {
            i++;
            A[i] = A1[j];
        }
        break;
    }
}

free(A1);
free(A2);

//printf("ending merge sort -- side = %d\n", side);
//array_print(A, n);

```

```

    return;
}
/* end of merge sort algorithm code */

/* code for quick sort algorithm */
#define QUICK_SORT 2

void quick_sort(bit_size *A, bit_size beg, bit_size end) {
    if (end > (beg + 1)) {
        bit_size piv = A[beg];
        bit_size l = beg + 1;
        bit_size r = end;

        while (l < r) {
            if(A[l] <= piv) {
                l++;
            }
            else {
                swap(&A[l], &A[--r]);
            }
        }

        swap(&A[--l], &A[beg]);
        quick_sort(A, beg, l);
        quick_sort(A, r, end);
    }
}
/* end of quick sort algorithm code */

/* code for heap sort algorithm */
#define HEAP_SORT 4

void heap_sort(bit_size *A, bit_size N) {
    bit_size n = N, i = n/2, j, w;
    bit_size t;

    for (;;) {
        if (i > 0) {
            i--;
            t = A[i];
        } else {
            n--;
            if (n == 0) return;
            t = A[n];
            A[n] = A[0];
        }

        j = i;
        w = (i * 2) + 1;

        while (w < n) {

```

```

        if (((w + 1) < n) && (A[w + 1] > A[w])) {
            w++;
        }
        if (A[w] > t) {
            A[j] = A[w];
            j = w;
            w = (j * 2) + 1;
        } else {
            break;
        }
    }
    A[j] = t;
}
}
/* end of heap sort algorithm */

int main (int argc, char **argv) {

    int i;
    bit_size array_size = 0;
    int sorts = 0;
    bit_size *array = NULL;
    int rand_mode = RAND_KERNEL;
    struct timeval starttime, endtime, difftime;

    if (argc < 2) {
        usage();
        return 1;
    }

    printf("Array Elements are unsigned " BIT_WIDTH " bit wide values\n");

    for(i=1; i<argc; i++) {
        if (strcmp(argv[i], "-c") == 0) {
            array_size = atoll(argv[++i]);
            printf("Setting array size to " BIT_FLAG "\n", array_size);
        }
        if (strcmp(argv[i], "-s") == 0) {
            switch(atoi(argv[++i])) {
                case MERGE_SORT:
                    printf("Adding merge sort\n");
                    sorts |= MERGE_SORT;
                    break;
                case QUICK_SORT:
                    printf("Adding quick sort\n");
                    sorts |= QUICK_SORT;
                    break;
                case HEAP_SORT:
                    printf("Adding heap sort\n");
                    sorts |= HEAP_SORT;
                    break;
            }
        }
    }
}

```

```

if (strcmp(argv[i], "-r") == 0) {
    rand_mode = atoi(argv[++i]);
    switch(rand_mode) {
    case RAND_REPRO:
        printf("Using reproducible random mode for benchmarking (srand is seeded with va
        break;
    case RAND_KERNEL:
        printf("Using kernel random mode (opening and reading from /dev/urandom).\n");
        break;
    }
}

if (array_size < 1) {
    printf("Array size must be greater than 0, use -c\n");
    goto out;
}

if (sorts == 0) {
    printf("You must select a sort, use -s with 1, 2, or 4\n");
    goto out;
}

if (sorts & MERGE_SORT) {
    switch(rand_mode) {
    case RAND_REPRO:
        printf("Restoring array from backup...");
        break;
    case RAND_KERNEL:
        printf("Generating kernel random numbers...");
        break;
    }

    fflush(stdout);
    array = array_populate(array_size, rand_mode);
    if (array == NULL) {
        printf("Failed to malloc array\n");
        goto out;
    }
    printf("done\n");

    printf("Performing merge sort test...");
    fflush(stdout);
    gettimeofday(&starttime, NULL);

    merge_sort(array, array_size);

    gettimeofday(&endtime, NULL);
    timersub(&endtime, &starttime, &difftime);
    printf("finished in %lf seconds\n", ((float)difftime.tv_sec + (float)difftime.tv_usec
}

if (sorts & QUICK_SORT) {
    switch(rand_mode) {

```

```

case RAND_REPRO:
    printf("Restoring array from backup...");
    break;
case RAND_KERNEL:
    printf("Generating kernel random numbers...");
    break;
}

fflush(stdout);
array = array_populate(array_size, rand_mode);
if (array == NULL) {
    printf("Failed to malloc array\n");
    goto out;
}
printf("done\n");

printf("Performing quick sort test...");
fflush(stdout);
gettimeofday(&starttime, NULL);

quick_sort(array, 0, array_size);

gettimeofday(&endtime, NULL);
timersub(&endtime, &starttime, &difftime);
printf("finished in %lf seconds\n", ((float)difftime.tv_sec + (float)difftime.tv_us

}

if (sorts & HEAP_SORT) {
    switch(rand_mode) {
    case RAND_REPRO:
        printf("Restoring array from backup...");
        break;
    case RAND_KERNEL:
        printf("Generating kernel random numbers...");
        break;
    }

    fflush(stdout);
    array = array_populate(array_size, rand_mode);
    if (array == NULL) {
        printf("Failed to malloc array\n");
        goto out;
    }
    printf("done\n");

    printf("Performing heap sort test...");
    fflush(stdout);
    gettimeofday(&starttime, NULL);

    heap_sort(array, array_size);

    gettimeofday(&endtime, NULL);
    timersub(&endtime, &starttime, &difftime);
    printf("finished in %lf seconds\n", ((float)difftime.tv_sec + (float)difftime.tv_us

```

```
}  
  
return 0;  
  
out:  
if (array != NULL)  
    free(array);  
return 1;  
}
```

## 8 Summary

System and application profilers are a fact of life in performance analysis. This report compared the profilers across functionality, overhead, and comparative ease of use.

Comparing the time-based profiling output for each of the three profilers shows that Tprof and OProfile are very close and consistently while gprof differs significantly due to the call graph overhead and the fact that it is only concerned with the application itself and not the complete system.

Profiler	Scope	Heap Sort	Merge Sort	Quick Sort
OProfile	system-wide	30.98%	15.16%	10.38%
Tprof	system-wide	31.60%	13.62%	11.31%
gprof w/call graph	application only	41.63%	22.75%	23.04%

In order to simplify the comparison, some basics of the profilers are shown below:

Profiler	Pros	Cons	Notes
OProfile	<ul style="list-style-type: none"><li>- No patching</li><li>- No recompiling of app</li><li>- Many sampling events</li><li>- Can annotate code</li><li>- Standard/Wide use</li></ul>	<ul style="list-style-type: none"><li>- No call graph on POWER (yet)</li><li>- Lots of commands</li><li>- 'root' access required</li></ul>	General purpose. System-wide profiling.
gprof	<ul style="list-style-type: none"><li>- Call graphs</li><li>- Easy to run</li><li>- 'root' not needed</li><li>- Application focus</li></ul>	<ul style="list-style-type: none"><li>- Recompiling necessary</li><li>- Heavy call graph overhead</li></ul>	Only option for call graphs and application-only profiling.
Tprof	<ul style="list-style-type: none"><li>- Can handle Java</li><li>- Low overhead</li></ul>	<ul style="list-style-type: none"><li>- Rebuild kernel (on 2.4 only)</li><li>- Build drivers</li><li>- Runs in separate window by default</li></ul>	Only option for Java profiling.

## 9 Trademarks

POWER and POWER5 are trademarks and IBM and VisualAge are registered trademarks of IBM, Inc. in the United States and other countries.

Red Hat, the Red Hat "Shadow Man" logo, and all Red Hat-based trademarks and logos are trademarks or registered trademarks of Red Hat, Inc., in the United States and other countries.

SUSE is a trademark of Novell, Inc. in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

## 10 References

[1] OProfile system-wide profile for Linux

<http://oprofile.sourceforge.net/news/>

[2] "Identify performance bottlenecks with OProfile for Linux on POWER"

<http://www.ibm.com/developerworks/library/l-pow-oprofile/>

John Engel (engel@us.ibm.com), May 17th, 2004

[3] POWER Support for the OProfile tool

<http://www.ibm.com/developerworks/library/l-pow-oprofile-internals/>

A paper written by Carl Love and Maynard Johnson on the implementation considerations with OProfile on POWER systems.