Identify performance bottlenecks with OProfile for Linux on POWER

John Engel

May 17, 2005

Learn about OProfile and how to use it on IBM® POWER™ processor-based servers running Linux™. First, get a high-level overview of OProfile and its implementation on Linux on POWER and then follow along as the author presents two examples of how to profile code and analyze the results by using OProfile on a Linux on POWER platform.

Introduction
Identifying performance bottlenecks is one of the most difficult tasks you, as a developer, face when trying to improve code efficiency. Code profiling is one method that can make this task easier. Code profiling involves analyzing samples of data that represent certain processor activities on a running system. OProfile provides this solution for Linux on POWER. OProfile is included in the latest IBM® supported Linux for POWER distributions: Red Hat Enterprise Linux 4 (RHEL4) and SUSE LINUX Enterprise Server 9 (SLES9). This paper introduces OProfile for Linux on POWER and provides two examples of how you can use it to find performance bottlenecks.

Profiling overview
OProfile for Linux on POWER uses a kernel module that has access to performance counter registers and a user-space daemon that runs in the background, collecting the data from these registers. Before the daemon starts, OProfile configures the type of events and the sample count of each event. If no events are configured, OProfile uses the default event for Linux on POWER, called CYCLES, which counts processor cycles. The sample count for events determines how often a counter is incremented for each occurrence of the event. OProfile was designed to run with low overhead so the daemon running in the background does not disrupt system performance.

OProfile has kernel support for POWER4™, POWER5™, and PowerPC® 970 processors. The PowerPC 970 and POWER4 processors have eight counter registers, while the POWER5 processor has six counter registers. Timer mode is used on architectures that do not have OProfile kernel support. OProfile uses a timer interrupt in this mode and cannot profile any code that has interrupts disabled.

OProfile tools
Along with the kernel support for OProfile, there are user-space tools that interact with the kernel and tools that analyze the collected data. As previously mentioned, the OProfile daemon collects
the sample data. The tool that controls this daemon is called opcontrol. Table 1 lists some of the command line options for opcontrol. Some other tools that are described later in this paper are oreport and opannotate, both of which are used to analyze the collected data. An overview of all the OProfile tools can be found in Section 2.2 of the OProfile manual. (See Related topics.)

The types of processor events supported on RHEL4 and SLES9 will vary, as will the types of events supported on the different POWER processors. Use the option --list-events, with opcontrol to get a list of events supported on your platform.

### Table 1. opcontrol command line options

<table>
<thead>
<tr>
<th>opcontrol option</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--list-events</td>
<td>List processor events and unit masks</td>
</tr>
<tr>
<td>--vmlinux=&lt;kernel image&gt;</td>
<td>The kernel image file that will be profiled</td>
</tr>
<tr>
<td>--no-vmlinux</td>
<td>Do not profile the kernel</td>
</tr>
<tr>
<td>--reset</td>
<td>Clears data from current session</td>
</tr>
<tr>
<td>--setup</td>
<td>Sets up the daemon before running it</td>
</tr>
<tr>
<td>--event=&lt;processor event&gt;</td>
<td>Monitor the given processor event</td>
</tr>
<tr>
<td>--start</td>
<td>Start sampling</td>
</tr>
<tr>
<td>--dump</td>
<td>Flush the data to the daemon</td>
</tr>
<tr>
<td>--stop</td>
<td>Stop data sampling</td>
</tr>
<tr>
<td>-h</td>
<td>Kill the daemon</td>
</tr>
</tbody>
</table>

### OProfile examples

You can use OProfile to analyze such events as processor cycles, translation look-aside buffer (TLB) misses, memory references, branch mis-predictions, cache misses, interrupt handlers, and many more. Again, use the --list-events option of opcontrol to provide a comprehensive list of events that can be monitored on a specific processor.

The following two examples demonstrate how to use OProfile for Linux on POWER. The first example monitors processor cycles to identify a poorly written algorithm that creates a potential performance bottleneck. While this example is trivial, you can use the same approach when profiling an application to identify where most of the processor cycles are being used. You can then further analyze this portion of code to see if it can be optimized.

The second example is more complex -- it demonstrates how to identify level 2 (L2) data cache misses and provides a couple of solutions for reducing the number of data cache misses.

### Example 1: Analyzing poorly written code

The purpose of this example is to show you how to compile and profile a poorly written code sample to analyze which function shows poor performance. The example is trivial and uses two functions -- slow_multiply() and fast_multiply() -- to multiply two numbers, as shown in Listing 1 below.
Listing 1. Two functions performing multiplication

```c
int fast_multiply(x, y)
{
    return x * y;
}

int slow_multiply(x, y)
{
    int i, j, z;
    for (i = 0, z = 0; i < x; i++)
        z = z + y;
    return z;
}

int main()
{
    int i,j;
    int x,y;
    for (i = 0; i < 200; i++) {
        for (j = 0; j < 30 ; j++) {
            x = fast_multiply(i, j);
            y = slow_multiply(i, j);
        }
    }
    return 0;
}
```

Profile this code and analyze it with opannotate, which allows you to view the source code with OProfile annotation. You must first compile the source with debug information, which opannotate requires to add its annotation. Use the Gnu Compiler Collections C compiler, gcc, to compile the example in Listing 1 by running the following command. Note that the `-g` flag adds the debug information.

```
gcc -g multiply.c -o multiply
```

Next, use the commands in Listing 2 to profile the code, and then analyze the results by using the CYCLES event to calculate processor cycles.

Listing 2. Commands for profile multiplication example

```
# opcontrol --vmlinux=/boot/vmlinux-2.6.5-7.139-pseries64
# opcontrol --reset
# opcontrol --setup --event=CYCLES:1000
# opcontrol --start
Using 2.6+ OProfile kernel interface.
Reading module info.
Using log file /var/lib/oprofile/oprofiled.log
Daemon started.
Profiler running.
# ./multiply
# opcontrol --dump
# opcontrol --stop
Stopping profiling.
```

Identify performance bottlenecks with OProfile for Linux on POWER
Finally, use the opannotate tool to generate source code with the **--source** option or the assembly code with the **--assembly** option. Whether you use one or both of these options depends on the level of detail you want to profile. For this example, use the **--source** option to identify where most of the processor cycles are occurring.

**Listing 3. Analysis of opannotate results from multiplication example**

```c
# opannotate --source ./multiply
/*
 * Command line: opannotate --source ./multiply
 * Interpretation of command line:
 * Output annotated source file with samples
 * Output all files
 * CPU: ppc64 POWER5, speed 1656.38 MHz (estimated)
 * Counted CYCLES events (Processor cycles) with a unit mask of 0x00 (No unit mask) count 1000
 */
/*
 * Total samples for file : "/usr/local/src/badcode/multiply.c"
 */
6244 100.000

:int fast_multiply(x, y)
36 0.5766 :{ /* fast_multiply total:  79 1.2652 */
26 0.4164 :    return x * y;
17 0.2723 :
}

:int slow_multiply(x, y)
50 0.8008 :{ /* slow_multiply total:  6065 97.1332 */
: int i, j, z;
2305 36.9154 :    for (i = 0, z = 0; i < x; i++)
3684 59.0006 :        z = z + y;
11 0.1762 :
15 0.2402 :
}

:int main()
*: /* main total:  100 1.6015 */
: int i,j;
: int x,y;
:
: 1 0.0160 : for (i = 0; i < 200; i++) {
: 6 0.0961 :    for (j = 0; j < 30 ; j++) {
75 1.2812 :
18 0.2883 :
:
:    } return 0;
:
The following lines from Listing 3 show the number of CYCLES used in both multiplication functions:
As you can see, `fast_multiply()` only used 79 samples, while `slow_multiply()` used 6065 samples. While this is a trivial example and would not likely occur in the real world, it demonstrates how you could profile your code and analyze it for performance bottlenecks.

**Example 2: Identifying level 2 data cache misses**

This example is more complex than the first one and involves identifying level 2 (L2) data cache misses. POWER processors contain on-chip L2 cache, which is high-speed memory located close to the processor. The processor looks in the L2 cache for frequently modified data. A problem can arise when two processors share a data structure and modify that shared data simultaneously. CPU1 contains a copy of the data in its L2 cache, while CPU2 modifies the shared data structure. The copy in CPU1's L2 cache is now invalid and must be updated. CPU1 must perform the expensive step of retrieving the data from main memory which costs extra processor cycles. Figure 1 shows two processors that each have a copy of a shared data structure in their own L2 cache.

![Figure 1. Two processors sharing a data structure](image)

In this example, you will look at the data structure (shown in **Listing 4**) and analyze what happens when two processors modify this data structure simultaneously. You will then observe the data cache misses and review two possible solutions to correct the problem.

**Listing 4. Shared data structure**

```c
struct shared_data_struct {
    unsigned int data1;
    unsigned int data2;
}
```
The program in **Listing 5** spawns a child process using the `clone()` system call, passing in the `VM_CLONE` flag. The `VM_CLONE` flag causes the child process to run in the same memory space as the parent. The parent thread modifies the first element of the data structure, while the child thread modifies the second element.

**Listing 5. Example code to demonstrate L2 data cache misses**

```c
#include <stdlib.h>
#include <sched.h>

struct shared_data_struct {
    unsigned int data1;
    unsigned int data2;
};

struct shared_data_struct shared_data;

static int inc_second(struct shared_data_struct *);

int main(){
    int i, j, pid;
    void *child_stack;

    /* allocate memory for other process to execute in */
    if((child_stack = (void *) malloc(4096)) == NULL) {
        perror("Cannot allocate stack for child");
        exit(1);
    }

    /* clone process and run in the same memory space */
    if ((pid = clone((void *)&inc_second, child_stack,
        CLONE_VM, &shared_data)) < 0) {
        perror("clone called failed.");
        exit(1);
    }

    /* increment first member of shared struct */
    for (j = 0; j < 2000; j++) {
        for (i = 0; i < 100000; i++) {
            shared_data.data1++;
        }
    }

    return 0;
}

int inc_second(struct shared_data_struct *sd){
    int i,j;

    /* increment second member of shared struct */
    for (j = 1; j < 2000; j++) {
        for (i = 1; i < 100000; i++) {
            sd->data2++;
        }
    }
}
```

Using the gcc compiler, run the command in **Listing 6** to compile the example program with no optimization.
Listing 6. Command to compile example code in Listing 5

```bash
gcc -o cache-miss cache-miss.c
```

You can now analyze L2 data cache misses that occur in this program with OProfile.

For this example, the author executed and profiled the program on an IBM eServer™ OpenPower™ 710 with two POWER5 processors running SLES9 Service Pack 1 (SLES9SP1). The `--list-events` flag is passed to opcontrol to determine which event monitors L2 data cache misses. For a POWER5 processor-based system running SLES9SP1, the `PM_LSU_LMQ_LHR_MERGE_GP9` event monitors L2 data cache misses. If you set the sample count to 1000, as in this example, OProfile will take a sample for every 1000th hardware event being counted. If you are using a different platform, such as a POWER4 processor-based server, this event will be different.

Use the commands in Listing 7 to profile the example code as follows:

Listing 7. Commands to profile L2 data cache misses in the example in Listing 5

```bash
# opcontrol --vmlinux=/boot/vmlinux-2.6.5-7.139-pseries64
# opcontrol --reset
# opcontrol --setup "event=PM_LSU_LMQ_LHR_MERGE_GP9:1000"
# opcontrol --start
Using 2.6+ OProfile kernel interface.
Reading module info.
Using log file /var/lib/oprofile/oprofiled.log
Daemon started.
Profiler running.
# ./cache-miss
# opcontrol --dump
# opcontrol -h
Stopping profiling.
Killing daemon.
# opreport -l ./cache-miss
CPU: ppc64 POWER5, speed 1656.38 MHz (estimated)
Counted PM_LSU_LMQ_LHR_MERGE_GP9 events (Dcache miss occurred for the same real cache line as earlier req, merged into LMQ) with a unit mask of 0x00 (No unit mask) count 1000 samples % symbol name
47897 58.7470 main
33634 41.2530 inc_second
```

When you analyze the results from opreport, you will see that there are a number of cache misses in the functions `main()` and `inc_second()`. The `-l` option to opreport prints out symbol information, instead of just the binary image name. Again, the cache misses occur because there are two processors that modify a shared data structure that is 8 bytes in size and fits into a single 128 byte cache line.
One way to eliminate data cache misses is to pad the data structure so that each element is stored in its own cache line. **Listing 8** contains a modified structure with a 124 byte pad.

**Listing 8. Shared data structure with padding that forces elements into different cache lines**

```c
struct shared_data_struct {
    unsigned int data1;
    char pad[124];
    unsigned int data2;
}
```

**Figure 2** shows how each data element is stored in its own cache line on each processor now that the structure contains this pad.

**Figure 2. Two processors sharing padded data structure**

Recompile the program as you did before, but this time use the modified data structure. Then analyze the results again by using the commands in **Listing 9**.

**Listing 9. Commands to profile L2 data cache misses with padded data structure**

```bash
# opcontrol --vmlinux=/boot/vmlinux-2.6.5-7.139-pseries64
# opcontrol --reset
# opcontrol --setup â##event=PM_LSU_LMQ_LHR_MERGE_GP9:1000
# opcontrol --start
Using 2.6+ OProfile kernel interface.
Reading module info.
Using log file /var/lib/oprofile/oprofiled.log
Daemon started.
Profiler running.
# ./cache-miss
# opcontrol --dump
```
Opreport suggests that there is a possible error since no sampling data was found. However, with the modification to the shared data structure, this is expected since there are no L2 cache misses because each data element is in its own cache line.

You can now examine how expensive the L2 cache misses are in terms of processor cycles. First, profile the code with the original shared data structure without the padding (Listing 4). The event you will sample is CYCLES. Use the commands in Listing 10 to profile the example for the CYCLES event.

**Listing 10. Commands to profile number of processor cycles in the example in Listing 5**

```
# opcontrol --vmlinux=/boot/vmlinux-2.6.5-7.139-pseries64
# opcontrol --reset
# opcontrol --setup event=CYCLES:1000
# opcontrol --start
Using 2.6+ OProfile kernel interface.
Reading module info.
Using log file /var/lib/oprofile/oprofiled.log
Daemon started.
Profiler running.
# ./cache-miss
# opcontrol --dump
# opcontrol -h
Stopping profiling.
Killing daemon.
```

```
# opreport -l ./cache-miss
error: no sample files found: profile specification too strict?
```

Now, profile the example code with the padded data structure (Listing 8) using the commands in Listing 11.

**Listing 11. Commands to profile number of processor cycles in example with padded data structure**

```
# opcontrol --vmlinux=/boot/vmlinux-2.6.5-7.139-pseries64
# opcontrol --reset
```

```
# opreport -l ./cache-miss
CPU: ppc64 POWER5, speed 1656.38 MHz (estimated)
Counted CYCLES events (Processor cycles) with a unit mask of 0x00
(No unit mask) count 1000
samples % symbol name
121166  53.3853 inc_second
105799  46.6147 main
```

```
# opcontrol -h
Stopping profiling.
Killing daemon.
```

```
# opreport -l ./cache-miss
error: no sample files found: profile specification too strict?
```
As expected, the number of processor cycles increases with the number of L2 cache misses. The main reason for this is the expensive operation of retrieving data from main memory, rather than from L2 cache.

Another way to avoid cache misses between the two processors is to run both threads on the same processor. The following example will use Cpu affinity, which binds a process to a particular processor, to demonstrate this. On Linux, the `sched_setaffinity()` system call runs both threads on one processor. Listing 12 provides another variation of the original example program that uses the `sched_setaffinity()` call to perform this operation.

**Listing 12. Example code that uses cpu affinity to avoid L2 cache misses**

```c
#include <stdlib.h>
#include <sched.h>

struct shared_data_struct {
    unsigned int data1;
    unsigned int data2;
};

struct shared_data_struct shared_data;

static int inc_second(struct shared_data_struct *);

int main(){
    int i, j, pid;

    cpu_set_t cmask;
    unsigned long len = sizeof(cmask);
    pid_t p = 0;
    void *child_stack;

    __CPU_ZERO(&cmask);
    __CPU_SET(0, &cmask);
```
/* allocate memory for other process to execute in */
if ((child_stack = (void *) malloc(4096)) == NULL) {
    perror("Cannot allocate stack for child");
    exit(1);
}

/* clone process and run in the same memory space */
if ((pid = clone((void *)&inc_second, child_stack,
CLONE_VM, &shared_data)) < 0) {
    perror("clone called failed");
    exit(1);
}
if (!sched_setaffinity(0, len, &cmask)) {
    printf("Could not set cpu affinity for current
process.\n");
    exit(1);
}
if (!sched_setaffinity(pid, len, &cmask)) {
    printf("Could not set cpu affinity for cloned
process.\n");
    exit(1);
}

/* increment first member of shared struct */
for (j = 0; j < 2000; j++) {
    for (i = 0; i < 100000; i++) {
        shared_data.data1++;
    }
}
return 0;
}

int inc_second(struct shared_data_struct *sd)
{
    int i, j;

    /* increment second member of shared struct */
    for (j = 1; j < 2000; j++) {
        for (i = 1; i < 100000; i++) {
            sd->data2++;
        }
    }
}

This example runs both threads on the same processor with the shared data structure in one L2 cache line on one processor. This should result in zero cache misses. Use the steps described earlier to profile cache misses to verify that there are no L2 cache misses when you run both processes on one processor. A third solution to the data cache miss problem is to use compiler optimizations that might reduce the number of cache misses. However, there may be circumstances when this is not an option and you must profile the code and make corrections for poor performance.

Summary

Profiling is one of the most difficult tasks in the development process. Having good tools is essential when trying to get the optimal performance out of your code. OProfile is one such tool that currently provides this function for Linux on POWER. There are many other performance
and debug tools available for Linux on other platforms that are rapidly being ported to Linux on POWER, as well. With the exception of the different types of processor events, running OProfile on POWER processor-based Linux platforms is similar to running OProfile on other architectures. So, if you have used OProfile on other platforms you should be up and running on Linux on POWER in a short amount of time.

Acknowledgments

I would like to acknowledge Linda Kinnunen for her document template and review and Maynard Johnson for his technical review of this document.
Related topics

- A list of all the OProfile tools can be found in Section 2.2 of the OProfile manual.
- Visit OProfile for the latest information.
- Visit the SUSE SLES9 Software Developer Kit Web page for more information on the SUSE LINUX SDK 9.
- The article "Smashing performance with OProfile" (developerWorks, October 2003) provides information on the OProfile profiling tool.
- See the GCC home page for information on the GNU Compiler Collection C compiler.