Smashing performance with OProfile

Identifying performance bottlenecks in real-world systems

Prasanna Panchamukhi

October 16, 2003

Analyzing the performance of the Linux operating system and application code can be difficult due to unexpected interactions between the hardware and the software, but profiling is one way you can identify such performance problems. This article looks at OProfile, a profiling tool for Linux that will be included in the upcoming stable kernel.

Profiling is a formal summary or analysis of data, often in the form of a graph or table, representing distinctive performance features or characteristics. The profiling table provides the percentage and number of samples collected for specified processor events such as the number of cache line misses, Transition Lookaside Buffer (TLB) misses, and so on.

OProfile is one of several profiling and performance monitoring tools for Linux. It works on various architectures, including the IA32, IA64, and AMD Athlon families, has a low overhead, and will be included in the 2.6 version of the kernel.

OProfile can help you identify issues such as loop unrolling, poor cache utilization, inefficient type conversion and redundant operations, branch mispredictions, and so on. It collects information about processor events including TLB misses, stalls, memory references, total lines allocated in the DCU (Data Cache Unit), the number of cycles of a DCU miss, and the number of non-cacheable and cacheable instruction fetches. OProfile is fine-grained and can collect samples for a set of instructions, or for function, system call, or interrupt handlers. OProfile works by sampling, and using the collected profile data, you can easily identify performance problems.

Installing OProfile

OProfile is included in the Linux 2.5 kernel and above and on most newer distributions, including Red Hat 9. You can also download OProfile using the link in the Related topics section later in this article. You'll need to recompile the kernel with OProfile enabled; here is how to do that:

1. Enable OProfile:
   
   ```
   # cd /usr/src/linux
   # make xconfig/config
   ```
Enable OProfile in the profiling menu and set CONFIG_PROFILING=y and CONFIG_OPROFILE=y in the .config file. Also enable Local APIC and IO-APIC in the "Processor type and features" menu.

2. Recompile like so:

```bash
# make dep (use for 2.4 kernel versions)
# make bzImage
```

3. Boot the new kernel.

4. To configure and install the OProfile utilities, enter this:

```bash
# ./configure --with-linux=/usr/src/linux/ --with-qt-dir=/usr/lib/qt/
--with-kernel-support
# make
# make install
```

For information about system requirements and for more detailed installation instructions, see the links in the Related topics section.

**Overview of OProfile tools**

- **op_help**: Lists available events with short descriptions
- **opcontrol**: Controls OProfile data collection
- **oprofpp**: Retrieves useful profile data
- **op_time**: Lists the relative profile values for all images on the system
- **op_to_source**: Produces annotated sources, assembly or mixed source assembly
- **op_merge**: Merges sample files belonging to the same application
- **op_import**: Converts sample database files from the foreign format (abi) to native format

**Three quick steps to start profiling**

1. Start up the profiler:

```bash
# opcontrol --setup --ctr0-event=CPU_CLK_UNHALTED
--ctr0-count=600000 --vmlinux=/usr/src/linux-2.4.20/vmlinux
For RTC mode users, use --rtc-value=2048
# opcontrol --start
```

2. Now the profiler is running; go do what ever you want to profile.

3. Dump the profiled data with this option:

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

**OProfile analysis: Cache memory utilization problem**

Caches are the memory closest to the processor execution unit. Cache memory is much smaller and faster than main memory and can either be internal or external to the processor chip. Caches contain copies of the most frequently used instructions and data. By allowing fast access to frequently used data, software can run much faster than if it had accessed the data from the main memory. In the Intel IA32 P4, data is stored in cache lines of 32 bytes each.

For multiple CPUs, the cache lines in a CPU's cache are invalidated when a CPU modifies data that is shared between CPUs.

If the data or instruction is not present in the cache, or if the cache line is invalidated, the CPU updates its cache by reading the data from the main memory. The processor event that does this
is known as **L2_LINES_IN**. Reading the data from the main memory requires more CPU cycles. OProfile helps you identify a cache memory problem like the one in Listing 1.

### Listing 1. Code with cache memory problem

```c
/*
 * Shared data being modified by two threads running on different CPUs.
 */

/* shared structure between two threads which will be optimized later */
struct shared_data_align {
    unsigned int num_proc1;
    unsigned int num_proc2;
};

/* Shared structure between two threads remains unchanged (non optimized)
 * This is required in order to collect some samples for the L2_LINES_IN event.
 */
struct shared_data_nonalign {
    unsigned int num_proc1;
    unsigned int num_proc2;
};

/* In the example program below, the parent process creates a clone
 * thread sharing its memory space. The parent thread running on one CPU
 * increments the num_proc1 element of the common and common_aln. The cloned
 * thread running on another CPU increments the value of num_proc2 element of
 * the common and common_aln structure.
 */
/* Declare global data */
struct shared_data_nonalign common_aln;

/* Declare local shared data */
struct shared_data_align common;

/* Now clone a thread sharing memory space with the parent process */
if ((pid = clone(func1, buff + 8188, CLONE_VM, &common)) < 0) {
    perror("clone");
    exit(1);
}

/* Increment the value of num_proc1 in loop */
for (j = 0; j < 200; j++)
    for (i = 0; i < 100000; i++) {
        common.num_proc1++;
    }

/* Increment the value of num_proc1 in loop */
for (j = 0; j < 200; j++)
    for (i = 0; i < 100000; i++) {
        common_aln.num_proc1++;
    }

/* The routine below is called by the cloned thread, to increment the num_proc2
 * element of common and common_aln structure in loop.
 */
int func1(struct shared_data_align *com)
{
    int i, j;
    /* Increment the value of num_proc2 in loop */
    for (j = 0; j < 200; j++)
        for (i = 0; i < 100000; i++) {
            com->num_proc2++;
        }
```

Smashing performance with OProfile
/* Increment the value of num_proc2 in loop */
for (j = 0; j < 200; j++)
    for (i = 0; i < 100000; i++) {
        common_aln.num_proc2++;
    }
}

The above program is profiled for the event L2_LINES_IN. Note the samples collected in `func1` and `main`:

Listing 2. OProfile data for L2_LINES_IN

```
# opcontrol --setup --ctr0-event=L2_LINES_IN
    --ctr0-count=500 --vmlinux=/usr/src/linux-2.4.20/vmlinux
#opcontrol --start
#./appln
#opcontrol --stop
#oprofpp -l ./appln
Cpu type: PIII
Cpu speed was (MHz estimation) : 699.57
Counter 0 counted L2_LINES_IN events (number of allocated lines in L2) with a
unit mask of 0x00 (No unit mask) count 500
vma   samples % symbol name
080483d0 0 0 _start
080483f4 0 0 call_gmon_start
08048420 0 0 __do_global_dtors_aux
08048480 0 0 fini_dummy
08048490 0 0 frame_dummy
080484c0 0 0 init_dummy
08048530 0 0 __do_global_ctors_aux
08048550 0 0 init_dummy
08048680 0 0 __fini
080484d0 4107 49.2033 main
080485bc 4240 50.7967 func1
```

Now, profile the same application (executable) with the event CPU_CLK_UNHALTED, which basically collects samples on the number of cycles the CPU runs without halting. The number of samples collected in the routine is proportional to the time spent by the processor in executing instructions. The more samples collected, the more time the processor has spent executing those instructions. Note the number of samples collected in `main` and `func1`:

Listing 3. OProfile data for CPU_CLK_UNHALTED

```
#oprofpp -l ./appln
Cpu type: PIII
Cpu speed was (MHz estimation) : 699.667
Counter 0 counted CPU_CLK_UNHALTED events (clocks processor is not halted) with
a unit mask of 0x00 (No unit mask) count 10000
vma   samples % symbol name
080483d0 0 0 _start
080483f4 0 0 call_gmon_start
08048420 0 0 __do_global_dtors_aux
08048480 0 0 fini_dummy
08048490 0 0 frame_dummy
080484c0 0 0 init_dummy
08048530 0 0 __do_global_ctors_aux
08048550 0 0 init_dummy
08048680 0 0 __fini
080484d0 40317 49.9356 main
080485bc 40421 50.0644 func1
```
We will now optimize the shared data structure to improve performance by separating the two elements of the shared data structure into different cache lines. In an Intel IA32 P4 processor, the size of each L2 cache line is 32 bytes. By padding 28 bytes of the first element in the `shared_data_align` structure, the elements of the structure can be separated into two different cache lines. Now, the parent thread modifies `num_proc1` of `shared_data_align`, resulting in the reading in of `num_proc1` on one cache line of CPU number 1 during the first access. Future access to `num_proc1` by the parent thread results in the reading in of the data from the cache line. The clone thread modifies `num_proc2` of `shared_data_align`, which results in having `num_proc2` on another cache line in CPU number 2. The two threads running in parallel modify the elements `num_proc1` and `num_proc2` respectively, which are on different cache lines. By separating the two elements of the structure into two different cache lines, modification of one cache line does not cause another cache line to be read in again from the memory. In this way the number of cache lines read in are reduced.

**Listing 4. Optimized data structures**

```c
/*
 * The padding is added to separate the two unsigned ints in such a
 * way that the two elements num_proc1 and num_proc2 are on two
 * different cache lines.
 */
struct shared_data_align {
    unsigned int num_proc1;
    char padding[28];
    unsigned int num_proc2;
};

/*
 * This structure remains unchanged, so that some cache lines
 * read in can be seen in profile data.
 */
struct shared_data_nonalign {
    unsigned int num_proc1;
    unsigned int num_proc2;
};
```

Note that `shared_data_nonalign` has not been optimized.

Since you already have OProfile enabled and running (you do, don’t you?), here are some profiles you can try running yourself: collect OProfile data for the event `L2_LINES_IN` and set the count to 500, as shown in **Listing 2**.

Also try collecting OProfile data for the event `CPU_CLK_UNHALTED` with the count set to 10000; compare the data collected with and without optimization and note the performance improvement.

**OProfile analysis: Branch misprediction**

Modern processors practice branch prediction (see Related topics), since the underlying algorithm and data have regularities. If the prediction is correct, the cost of a branch is lower. But branch prediction is not always correct, and some branches are hard to predict. You can approach this by improving branch prediction in your software, and you can also identify the issues by profiling the application and the kernel for the event `BR_MISS_PRED_TAKEN_RET`. 
The code in Listing 5 shows branch misprediction. This example program creates a cloned thread sharing memory space with the parent process. The parent process running on one processor toggles the value of `num_proc1` based on the value of `num_proc2` (and branches based on the value of the variable modified by another process). The compiler simply assumes that the value of `num_proc2` will be equal to 1 every time, and generates the code for that branch by default. If this prediction of `num_proc2` equal to 1 is false, then branch misprediction occurs.

The cloned thread running on another processor toggles the value of `num_proc2` based on the value of `num_proc1` (and branches based on the value of the variable modified by another process). This causes `num_proc2` to not always be equal to 1, and hence branch misprediction occurs in the parent thread. Similarly, toggling the value of `num_proc1` by the parent thread causes branch misprediction in the clone thread.

**Listing 5. Code showing branch misprediction**

```c
/*shared structure between the two processes */
struct share_both {
    unsigned int num_proc1;
    unsigned int num_proc2;
};

/* The parent process clones a thread by sharing its memory space. The parent
 * process just toggles the value of num_proc1 in loop.
 */

/* Declare local shared data */
struct share_both common;

/* clones a thread with memory space same as parent thread*/
if ((pid = clone(func1, buff+8188, CLONE_VM, &common)) < 0) {
    perror("clone");
    exit(1);
}

/* toggles the value of num_proc1 in loop */
for (j = 0; j < 200; j++)
    for(i = 0; i < 100000; i++) {
        if (common.num_proc2)
            common.num_proc1 = 0;
        else
            common.num_proc1 = 1;
    }

/* The function below is called by the cloned thread, which just toggles the
 * value of num_proc2 every time in the loop.
 */
int func1(struct share_both *com)
{
    int i, j;
    /* toggles the value of num_proc2 in loop */
    for (j = 0; j < 200; j++)
        for(i = 0; i < 100000; i++) {
            if (com->num_proc1)
                com->num_proc2 = 0;
            else
                com->num_proc2 = 1;
        }
}
```

Smashing performance with OProfile
Branch misprediction can be shown by compiling the above code without optimization.

```
gcc -o branch parent_thread_source_code clone_thread_source_code
```

Now, profile the branch application for the event BR_MISS_PRED_TAKEN_RET with the count set to 500, as shown in Listing 2. Note the samples collected in main and func1.

Also profile the same executable for the event CPU_CLK_UNHALTED and with the count set to 10000, as shown in Listing 2.

You can optimize branch misprediction by using the -O2 compiler option:

```
gcc -O2 -c clone_thread_source_code
#gcc -o branch clone_thread_source_code.o parent_thread_source_code
```

Now, start profiling the application for the BR_MISS_PRED_TAKEN_RET event and CPU_CLK_UNHALTED as in Listing 2; note the performance improvement.

Let's look at another example of branch misprediction in the next section on kernel profiling.

### Examples of kernel profiling

The profile data shown below were collected for the event BR_MISS_PRED_TAKEN_RET by the kernbench benchmark for the 2.5.70 kernel. 23,360 samples were collected for vma_merge and 20,717 for do_mmap_pgoff.

### Listing 6. Kernel profile data

```
# oprofpp -l -i /boot/vmlinux | tail -20
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Count</th>
<th>Time</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>c0143510</td>
<td>4719</td>
<td>1.26446</td>
<td>page_add_rmap</td>
</tr>
<tr>
<td>c0117740</td>
<td>4791</td>
<td>1.28375</td>
<td>schedule</td>
</tr>
<tr>
<td>c0140320</td>
<td>4825</td>
<td>1.29286</td>
<td>find_vma_prepare</td>
</tr>
<tr>
<td>c010f720</td>
<td>4862</td>
<td>1.30278</td>
<td>sys_mmap2</td>
</tr>
<tr>
<td>c0134fc0</td>
<td>5005</td>
<td>1.34109</td>
<td>__alloc_pages</td>
</tr>
<tr>
<td>c0123670</td>
<td>5473</td>
<td>1.46649</td>
<td>run_timer_softirq</td>
</tr>
<tr>
<td>c0134800</td>
<td>5648</td>
<td>1.51339</td>
<td>bad_range</td>
</tr>
<tr>
<td>c0139250</td>
<td>6571</td>
<td>1.7607</td>
<td>mark_page_accessed</td>
</tr>
<tr>
<td>c0143bd0</td>
<td>6919</td>
<td>1.85395</td>
<td>_pte_chain_free</td>
</tr>
<tr>
<td>c013f180</td>
<td>6973</td>
<td>1.86842</td>
<td>do_no_page</td>
</tr>
<tr>
<td>c0140ec0</td>
<td>7393</td>
<td>1.98096</td>
<td>get_unmapped_area</td>
</tr>
<tr>
<td>c01400f0</td>
<td>8020</td>
<td>2.14896</td>
<td>vm_enough_memory</td>
</tr>
<tr>
<td>c0140ff0</td>
<td>9897</td>
<td>2.65191</td>
<td>find_vma</td>
</tr>
<tr>
<td>c01594e0</td>
<td>10939</td>
<td>2.93111</td>
<td>link_path_walk</td>
</tr>
<tr>
<td>c0143e70</td>
<td>11467</td>
<td>3.07259</td>
<td>buffered_rmqueue</td>
</tr>
<tr>
<td>c0117370</td>
<td>11690</td>
<td>3.13234</td>
<td>scheduler_tick</td>
</tr>
<tr>
<td>c013eeb0</td>
<td>17463</td>
<td>4.67922</td>
<td>do_anonymous_page</td>
</tr>
<tr>
<td>c01153e0</td>
<td>20322</td>
<td>5.44529</td>
<td>do_page_fault</td>
</tr>
<tr>
<td>c01408e0</td>
<td>20717</td>
<td>5.55113</td>
<td>do_mmap_pgoff</td>
</tr>
<tr>
<td>c0140600</td>
<td>23360</td>
<td>6.25933</td>
<td>vma_merge</td>
</tr>
</tbody>
</table>

Branch misprediction can be eliminated by removing branches. In the Intel IA32 processor, branches can be eliminated using SETcc instructions or using P6 processor conditional move CMOVcc or FCMOVcc instructions.

The following line of C code shows conditional branching:

```
Branch misprediction can be shown by compiling the above code without optimization.

```
(A > B) ? C1 : C2;

Here are the assembly instructions for the above C code:

**Listing 7. Equivalent assembly instructions**

```assembly
cmp A, B              ; compare
jge L30               ; conditional branch
mov ebx, CONST1
jmp L31               ; unconditional branch
L30:
  mov ebx, CONST2
L31:
```

This code can be optimized to eliminate branches like this:

**Listing 8. Equivalent assembly instructions, branches removed**

```assembly
xor ebx, ebx           ;

cmp A, B
  setge b1            ; if ebx = 0 or 1
  dec ebx
  and ebx, (CONST2-CONST1) ; ebx = CONST1 or CONST2
add ebx, min(CONST2,CONST1) ; ebx = CONST1 or CONST2
```

The optimized code sets register `EBX` to zero, then compares A and B. If A is greater than or equal to B, `EBX` is set to 1. `EBX` is decremented and `AND`ed with the difference between the two constant values. This sets `EBX` to either zero or the difference of the two values. By adding the smaller of the two constant values, the correct value is written to `EBX`.

I hope you have gained some insight into OProfile and the ways you can optimize the kernel code. The 2.6 kernel release is coming up and there will be a lot to profile.
Related topics

- Read more about OProfile at the OProfile home page on SourceForge.
- OProfile's system requirements, installation instructions, and download page are also hosted at SourceForge.
- Myriad options can be found on the OProfile man page.
- OProfile has its own mailing list.
- "Hyper-Threading speeds Linux" analyzes the performance improvements promised by the Hyper-Threading feature of the Xeon processor (developerWorks, January 2003).
- "Efficient, Unified, and Scalable Performance Monitoring for Multiprocessor Operating Systems" is the subject of a paper from IBM Research (in PDF).
- JaViz, a client/server profiling tool for Java, is covered in this article from IBM Systems Journal.
- Dynamic Probes is a pervasive debugging facility for gathering difficult-to-acquire diagnostic information.
- The Linux Kernel Performance project aims to improve Linux performance through benchmark testing and analysis, with an emphasis on SMP scalability in server environments.
- If all attempts at improving application performance fail, it might be time for a hardware upgrade. Get more information about running Linux on IBM systems at the Linux for eServer page.

© Copyright IBM Corporation 2003
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)