Tuning Programs with OProfile

by William E. Cohen

The complexity of computer systems makes it difficult to determine what code consumes processor time and why code takes an excessive amount of time on the processor. *OProfile* is a low-overhead, system-wide sampling profiler for Linux which aids developers in finding where and why the processor spends time in particular sections of code. OProfile operates on a variety of architectures including Intel Pentium 4, AMD Athlon, and AMD AMD64. OProfile is provided in Fedora Core, recent Red Hat Linux distributions, and Red Hat Enterprise Linux.

OProfile is very useful for identifying processor performance bottlenecks. OProfile can be configured to take samples periodically to get time-based samples to indicate which sections of code are executed on the computer system. On many architectures OProfile provides access to the performance monitoring counters. The performance monitoring counters allow samples to be collection based on other events such as cache misses, memory references, instructions retired, and processor clock cycles. These performance events allow developers to determine whether specific performance problems exist in the code and revise the code appropriately based on the performance problem observed.

John Levon started OProfile as a Master's thesis project at Victoria University of Manchester, initially modeling it after the DEC Continuous Profiling Infrastructure (DCPI), a system-wide profiling system that ran on Alpha DEC Ultrix. DCPI used the Alpha performance monitoring hardware to trigger sampling and the data was stored in histograms for the individual executables. The first processors supported by OProfile were the Intel Pentium Pro and AMD Athlon processors and it was extended to support the Intel Pentium 4, Intel Itanium, and AMD64 processors. OProfile internals were revised and incorporated into the Linux 2.5 kernel. Once OProfile support was merged into the Linux kernel, support was added for IBM S/390, Compaq Alpha, and

IBM PowerPC64.

Program tuning may be required to improve the performance of code. However, code correctness always take precedence over speed; fast, incorrect code is of little use. Use of appropriate compiler options can improve the speed of the code generated by the compiler while avoiding changing the source code. Most optimizations performed by the compiler provides a constant factor of improvement. Often, selecting the appropriate algorithms for the program can have a greater effect on program performance than the compiler or hand tuning. For example the time required for a bubble sort grows proportional to the square of the number of elements being sorted (n), but there are much more efficient sorting algorithms that are proportional to $n\log_2 n$. For sorting large lists of items this can have huge impact because n grows much faster than $\log_2 n$, making the proportionality constant unimportant.

Processor architecture also has a huge impact on program performance. The article describes some of the critical architecture features of processors and how they affect program performance. This is followed by a walk through of the OProfile system and an example tuning an image processing program that converts a raw camera image to a PorTable PixMap (PPM) image with OProfile.

Processor Architecture

Processor performance has increased orders of magnitude since the initial microprocessors were developed due to the improvement in the processes used to fabricate the processors and to the changes in the processor architecture. However, the changes in the processor architecture can lead to large differences between best-case and worst-case performance. The code can execute very quickly when the program has the behavior expected by the processor designers, and the code can execute very slowly when the program deviates significantly from the assumption made in the processor design.

A number of architectural features have been incorporated into current processors to improve performance. Virtually all modern processors have cache memory to hide the disparity in speed between the processor and memory. Processors execute multiple instructions concurrently either through pipelining or superscalar execution. Processors also make predictions on what will execute rather than waiting idle for a result to be known. Let us take a look at the common processor architecture features and how program characteristics can affect the effectiveness.

Caches

On modern processors there is a large difference in speed between the processor and memory. For example, on the Pentium 4 the processor clock can be less than .3 nanosecond and an access to memory takes tens of nanoseconds. The small, fast cache memories are designed to store memory locations accessed recently by the processor. Cache memories make two assumptions about programs: *spatial* and *temporal locality*.

Spatial locality assumes rather than just accessing a lone memory location, memory accesses are to groups of nearby memory locations. The result of this assumption is the cache line size hold more than one word from memory. On the Pentium 4 each cache line in the L1 data cache is 64 bytes (16 four-byte words). Thus, when a memory access pulls in a cache line it pulls 64 bytes including the data from the desired memory address. This works well if the program is going through each word in memory, for example stepping through each element of array of integers. However, it does not work so well for a program that is just using one word or byte out of the cache line because it is striding through the array. In this case much of the data transferred to the cache is unused, wasting memory bandwidth.

Temporal locality assumes that the accesses to a particular location in memory are grouped together in time; if there is an access to a memory location now there are likely to be accesses to that location in the near future. The processor cannot predict all the future accesses. A memory location may be accessed very rarely (maybe only once), but the processor evicts a much more commonly used memory location from the cache to make room for the rarely used item. Thus, the cache has no benefit for the rarely accessed location and reduces the caches benefit for the commonly accessed location.

Pipelined, Superscalar, and Out-Of-Order Execution

Pipelined, *Superscalar*, and *Out-Of-Order* execution are different approaches to allow concurrent execution of instructions. Processors may use one or more of these methods to improve performance.

Pipelined processors have a series of stages connected linearly as in Figure 1. The instructions are numbered in Figure 1. Each row of Figure 1 is a snapshot of the pipeline at a particular clock cycle; The row at the top is the earliest clock cycle and the row at the bottom is the latest clock cycle. The instructions enter the pipeline on the left and leave the pipeline on the right. Instructions progress through the different stages of the pipeline in sequence. It is possible to have a new instruction enter the pipeline each clock cycle and an instruction completed each clock cycle. Pipelining can provide a significant improvement in throughput compared to sequential execution of instructions. Designers of processors can adjust the complexity of the pipeline stages to reduce the clock period, providing additional performance for the processor.

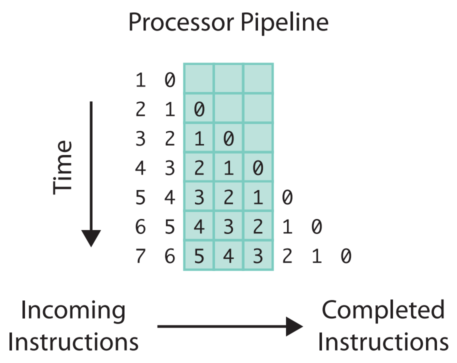

Figure 1. Processor Pipeline

A superscalar processor architecture has multiple execution units, allowing multiple instructions to be executed on each cycle. An example of this is the original Intel Pentium processor, which has two execution units and allows two instructions to be started in the same processor cycle. The execution units maybe pipelined so that a new group of instructions can be started each cycle.

Most programmers expect the instructions to execute in the order specified in the program. Out-Of-Order execution loosens this specification on the processor. The processor may appear to execute the instruction in the order specified in the program, but the hardware may change the order. There are cases where data dependencies in the code may not be known until the code actually executes. Instructions later in the instruction sequence may be executed earlier by the processor because all the required data is available to execute the instruction. Earlier instructions in the sequence may be delayed because some of the data from earlier instruction is not yet available. Thus, with Out-Of-Order execution the processor may be better utilized and reduce the runtime by executing instruction as the required dependencies are satisfied for instructions. The processor maintains the illusion that the instructions are executed in order to the programmer. Thus, if an instruction causes a fault, the processor makes sure the operations from the instructions preceding the fault instruction are completed and all the operations from the instructions following the faulting are nullified.

Pipelined, Superscalar, and Out-Of-Order execution can all reduce the average amount of time required to execute an instruction. These techniques work best if each instruction is independent of the other instruction. Dependencies on data from earlier instructions can stall the processor. An instruction may have to stay at a stage in the pipeline for several cycles until the data becomes available. On a superscalar processor such as the Itanium processors the group of instructions may not be issued until all the dependencies are resolved or a function unit is available. On the Out-Of-Order processors an instruction may be delayed until the data available.

Branches in code can impact performance. Conditional branches affect the instruction sequence. The processor cannot start executing additional instructions until the destination of the branch is fetched. If a condition branch must wait for the immediately preceding instruction to go through the pipeline to produce a result before the destination of the branch can be determined, this can hurt performance. Let us look at an example of the performance impact. Assume that the processor has a 20 stage pipeline and 10% of the instructions are conditional branches. In a case without branches, throughput in the pipeline would be 1 cycle per instruction. With 10% of the instructions branches:

```
(.9*1)+(.1*20) = 2.9 cycles per instruction
```

Branches significantly lowers performance. Processors much avoid delays due to branches.

Branch Prediction and Instruction Speculation

Branches change the stream of instructions executed by the processor. If the processor has to wait for the branch target instructions to be fetched the processor's performance is reduced. Most processors implement branch prediction and instruction speculation to keep the processor busy doing useful work.

Branch prediction hardware predicts the destination of branches before all the information required to execute the branch is known. This prediction is based on the history of the branch. The instruction fetch unit will start supplying instructions to the predicted destination, allowing the processor to start executing instruction before the destination is known for certain. If the prediction is correct, then the processor has done useful work rather than sitting idle. However, if the prediction is incorrect, then the processor has to nullify the instructions from the incorrectly predicted branch.

Branch prediction works well for branches that form loops and for branches with simple repeating patterns of taken/not-taken. However, most branch prediction hardware has difficulties with indirect branches such as the ones that maybe be used in switch case statements, for example the core of an interpreter. Virtual methods in C++ code may also cause difficulty because the same call may have many different destinations because of the different classes.

Processors such as the Pentium Pro may speculatively execute instruction rather than let the processor sit idle. The processor may start to execute both possible paths in the code to keep the processor doing possibly useful work. However, some of the results will be discarded. Speculative execution may not help in cases where the speculative executed instruction further strain a scarce resource such as main memory bandwidth.

The Pentium 4 can also speculative execute instructions before all the result from previous instructions are known, to allow for better utilization of the processor. If it turns out that the some of the data was changed by earlier instructions, the Pentium 4 performs what is known as a *replay* to compute the correct value.

OProfile in Fedora Core 1

OProfile is a low-overhead system-wider profiler for Linux included in Fedora Core 1. The Linux 2.6 kernel supports OProfile for a number of different processor architectures. Fedora Core 1 has a back port of the 2.6 OProfile support in its 2.4 kernel. Fedora Core 1 currently include OProfile 0.7. The major OProfile components are shown in Figure 2. The kernel has a driver which controls the performance monitoring hardware and collects the samples. The driver interfaces to the user space with the *oprofile pseudo file system*. The daemon read data from the oprofile pseudo file system and converts the data into a sample database. The `opcontrol` script manages OProfile's profiling sample system. The analysis programs such as `opreport` and `opannotate` read data from the sample database.

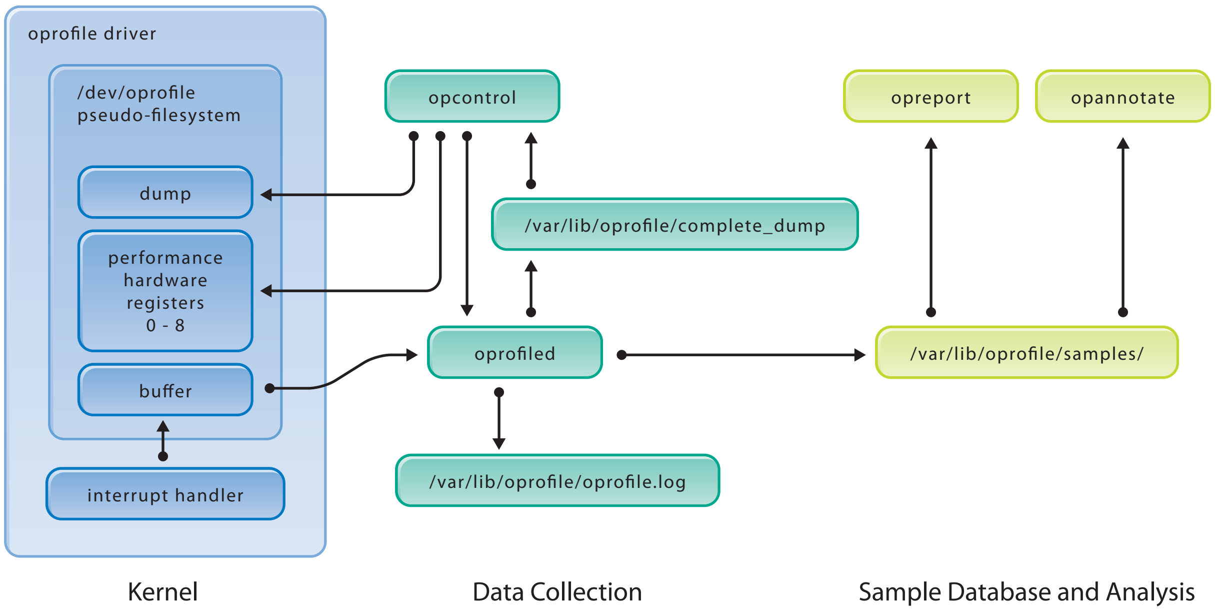
oprofile driver
/dev/oprofile pseudo-filesystem
dump
performance hardware registers 0 - 8
buffer
interrupt handler
opcontrol
/var/lib/oprofile/complete_dump
oprofiled
/var/lib/oprofile/oprofile.log
opreport
opannotate
/var/lib/oprofile/samples/
Kernel
Data Collection
Sample Database and Analysis

Figure 2. OProfile Block Diagram

OProfile can be divided into three sections: the kernel support (left part of Figure 2), the daemon (center of Figure 2), and the sample database with analysis programs (right part of Figure 2). To collect measurements the `opcontrol` script write the events to measure in the oprofile pseudo file system, once the kernel driver has been initialized, `opcontrol` spawns the `oprofiled` daemon which reads the sample data from the OProfile pseudo file system buffer. The daemon processes the data and places the converted data into `/var/lib/oprofile/samples`. Status information is placed into `/var/lib/oprofile/oprofile.log` by the daemon. The `opcontrol` script can also force data to be flushed to the sample database; the script determines when the daemon has finished the operation by referencing the file `/var/lib/oprofile/complete_dump`. The sample database is stored in `/var/lib/oprofile/samples`. The `opreport` and `opannotate` programs extract the profile information from the sample database and display it in human-readable form.

Example Using OProfile

Image processing applications that convert large images from one format to another (such as camera raw format to JPEG format) present an ideal type of application to tune with OProfile. The image processing applications are CPU-intensive with usually input and output limited to reading in the initial image and writing out the converted image. The images files produced by digital cameras are too large to fit in the processor's cache memory. Usually, the conversion process is performed on a batch of dozens of photos. Thus, the time saving due to tuning the conversion program can be very noticeable to the person using it.

The `dcraw` program converts raw image files from digital cameras (if the camera supports raw image files) into standard PorTable PixMap (PPM) format. Most digital cameras sample only one of the three primary colors at each pixel location. The remaining two colors for the pixel must be computed (interpolated) from the values of nearby pixels. Given the number of pixel in the image, such as 6.3 million on the Canon Digital Rebel, this results in a large number of calculations.

A raw image from a Canon Digital Rebel was used as input for the `dcraw` program. The experiments were run on a Dell Dimension 4300. Table 1 describes the system environment.

Hardware	Dell Dimension 4300: • Intel Pentium 4 1500MHz • 8KB L1 data cache (4-way assoc., 64-byte lines) • 256KB L2 Unified cache (8-way assoc., 64-byte lines) • Intel 845 chipset • 640MB PC133 SDRAM • "/" Maxtor 5T060H6, 60GB drive • "/home" Maxtor 6Y120P0, 120GB drive

Software	`kernel-smp-2.4.22-1.2115.nptl`
	`gcc-3.3.2-1`
	`glibc-2.3.2-101.1`
	`oprofile-0.7cvs-0.20030829.6`
	`dcraw-1.159`

Table 1: System Environment

Time-based Profiling

The first step in reducing the amount of time a program takes to execute is to find where the program spends time. Reducing the program hot spots will have the greatest impact on performance. By default OProfile uses the time-based metrics. Table 2 shows the time-based events for various platforms.

Processor	Event
Pentium Pro/PII/PIII	CPU_CLK_UNHALTED
Pentium 4	GLOBAL_POWER_EVENTS unit_mask=1
Athlon/AMD64	CPU_CLK_UNHALTED
Itanium 2	CPU_CYCLES
TIMER_INT	----

Table 2: Time-based Events on Processors

Because the experiments are being performed on a Pentium 4, we will use the `GLOBAL_POWER_EVENTS`.

To allow us to compare results from different programs, each version of `dcraw` will have a suffix to indicate the experiment number. For the initial experiment `dcraw.c` is copied to `dcraw_1.c` and it is compiled with:

```
gcc -g -O3 dcraw_1.c -o dcraw_1 -lm
```

OProfile uses the debugging information provided by the `-g` option, allowing the collected data to be mapped back to the source code. The `-O3` turns on GCC's optimizations. The program requires some math libraries provided by the `-lm`. With the `dcraw` executable and a sample image, we are now ready to find out where `dcraw` spends its time.

The `opcontrol` command performs a number of operations, such as setting up OProfile, and starting and stopping it. The `opcontrol` commands need to be run as root. To clear out previous samples the following command is executed as root:

```
/usr/bin/opcontrol --reset
```

To configure OProfile and start OProfile running the following two commands are executed as root:

```
/usr/bin/opcontrol --setup --no-vmlinux \
--separate=library \
--event=GLOBAL_POWER_EVENTS:750000:0x1:1:1

/usr/bin/opcontrol --start
```

The `--no-vmlinux` command line option indicates that we are not interested in recording samples for the kernel. The `--separate=library` option groups samples for a library with the executable that called the library. Finally, the `--event` option specifies measuring the time-based `GLOBAL_POWER_EVENTS` with a sample recorded for every 750,000 events, a unit mask of 0x1 (required for this event). The two "1"s at the end indicate to count events in kernel space and count events in user space respectively. The event should yield about 2,000 samples per second on the test machine. The `--start` actually starts the OProfile collecting data.

The program is run under the `time` command to give some wallclock time information:

```
/usr/bin/time ../dcraw/dcraw_1 crw_0327.crw
```

Immediately after the program is run OProfile is shutdown with the following command executed as root:

```
/usr/bin/opcontrol --shutdown
```

Analysis can be performed on the data as a normal user with `opreport` and `opannotate`. Listing 1 shows the executable and the associated shared libraries. The `--threshold 10` option was used to exclude any executable with less than 10 percent of the total number of samples. The option `--long-filenames` provides the complete path for each of the binaries. As expected the `dcraw` program is a CPU intensive program. Assuming that there were 2,000 samples per second, 19.48 seconds of user time would produce

38,960 samples. The number of samples actually collected for `dcraw_1` is a slightly lower because of the overhead of the sampling routine, but it is in good agreement with 38,305 samples.

```
/usr/bin/opreport  --long-filenames --threshold 10

CPU: P4 / Xeon, speed 1495.19 MHz (estimated) Counted
GLOBAL_POWER_EVENTS events (time during which
processor is not stopped) with a unit mask of 0x01
(count cycles when processor is active) count 750000

38305  90.7529  /home/wcohen/dcraw/dcraw_1
        32245  84.1796  /home/wcohen/dcraw/dcraw_1
         5339  13.9381  /lib/tls/libm-2.3.2.so
          709   1.8509  /lib/tls/libc-2.3.2.so
           12   0.0313  /lib/ld-2.3.2.so
```

Listing 1. sample counts for dcraw_1

A more detailed per function breakdown of where the program spends its time is shown in Listing 2. The run time is dominated by `vng_interpolate`. This is the function that we want examine more carefully and determine if there are improvements that can be made to this function. The next largest consumer of time is the `__ieee754_pow` function in the math library.

```
/usr/bin/opreport image:/home/wcohen/dcraw/dcraw_1 \
 -l --threshold 1

CPU: P4 / Xeon, speed 1495.19 MHz (estimated) Counted
GLOBAL_POWER_EVENTS events (time during which
processor is not stopped) with a unit mask of 0x01
(count cycles when processor is active) count 750000

vma      samples  %      image name    symbol name
0804d338 25428 66.3830 dcraw_1        vng_interpolate
00c4b8e0 3499   9.1346 libm-2.3.2.so  __ieee754_pow
080514a0 2997   7.8240 dcraw_1         convert_to_rgb
080517ac 1654   4.3180 dcraw_1         write_ppm
08049260 1279   3.3390 dcraw_1         decompress
00c507d0 744    1.9423 libm-2.3.2.so  __isnan
00c4e0d0 698    1.8222 libm-2.3.2.so  __pow
0804cd6c 546    1.4254 dcraw_1        scale_colors
00b6f0c0 494    1.2896 libc-2.3.2.so  getc
00c50800 386    1.0077 libm-2.3.2.so  __GI___finite
```

Listing 2. Per function breakdown of samples for initial program

The samples were mapped back to the source code with:

```
/usr/bin/opannotate \
image:/home/wcohen/dcraw/dcraw_1 --source \
> dcraw_1.annc
```

The mapping is not exact; the event that causes the sample is attributed to a later instruction. However, this view still provides some insight into which loops the program spends time in. The `dcraw_1.annc` file has each line in the source code preceded by the number of samples for that line. Listing 3 shows a segment of code that consumes a significant amount of time. Much of this time is caused by the GCC 3.3 inefficient integer absolute value code.

GCC generated code that tested the value. If it was negative, it would jump to another section of code, negate the value, and jump back. The branch prediction has difficulty accurately predicting whether the branch for the absolute value was taken because it is data dependent.

```
 258  0.6735 :    while ((g = *ip++) != INT_MAX) {
3671  9.5836 :    diff = abs(pix[g] - pix[*ip++]);
2582  6.7406 :    diff <<= *ip++;
1203  3.1406 :    while ((g = *ip++) != -1)
6565 17.1388 :   gval[g] += diff;
               :  }
```

Listing 3. Loop with abs

Our first suggested fix is to change the absolute value function into code without branches. Listing 4 is a C implementation of code suggested by the AMD optimization manual added to `dcraw_2.c`.

```
static inline int abs(int ecx)
{
  int ebx = ecx;
  ecx = ecx >> 31;
  ebx = ebx ^ ecx;
  ebx -= ecx;
  return ebx;
}
```

Listing 4. C Function to compute abs without conditional branches

The revised program was compiled and the data collected on it with the following command:

```
/usr/bin/time ../dcraw/dcraw_2 crw_0327.crw
```

We can see there is some improvement in the run time from 19.48 seconds down to 16.37 seconds, a 16% reduction in the runtime. Listing 5 shows the annotation for lines in Listing 3 and the abs function in Listing 4 with a total of 10,190 samples. The original version had 14,279 samples for the loop and the abs function. Notice that some of the lines in the abs function have no samples even though the instruction must be executed. This is due to the function be inlined and the inexact mapping of samples back to source code.

```
             :static inline int abs(int ecx)
 2417  7.5235 :{
    3  0.0093 :  int ebx = ecx;
  247  0.7688 :  ecx = ecx >> 31;
  219  0.6817 :  ebx = ebx ^ ecx;
              :  ebx -= ecx;
              :  return ebx;
              :}

  252  0.7844 : while ((g = *ip++) != INT_MAX) {
              :    diff = abs(pix[g] - pix[*ip++]);
 1197  3.7260 :    diff <<= *ip++;
 1263  3.9314 :    while ((g = *ip++) != -1)
 4592 14.2937 :      gval[g] += diff;
              :  }
```

Listing 5. Loop with revised abs

The absolute number of samples and the percentage of runtime spent in the `vng_interpolate` function in Listing 6 is reduced when compared to Listing 2. The `__ieee754_pow` and `convert_to_rgb` have a similar number of samples as before, but now are larger percentage of the runtime.

```
/usr/bin/opreport  \
image:/home/wcohen/dcraw/dcraw_2 -l --threshold 1

CPU: P4 / Xeon, speed 1495.19 MHz (estimated)

The counted number of GLOBAL_POWER_EVENTS events
(time during which processor is not stopped) with
a unit mask of 0x01 (count cycles when processor is
active) totals  750000.

vma      samples  %     image name     symbol name
0804d338 19205 59.7802  dcraw_2        vng_interpolate
00c4b8e0 3638  11.3242  libm-2.3.2.so  __ieee754_pow
08051488 2994   9.3196  dcraw_2        convert_to_rgb
08051794 1579   4.9150  dcraw_2        write_ppm
08049260 1326   4.1275  dcraw_2        decompress
00c4e0d0 713    2.2194  libm-2.3.2.so  __pow
00c507d0 699    2.1758  libm-2.3.2.so  __isnan
0804cd6c 552    1.7182  dcraw_2        scale_colors
00b6f0c0 456    1.4194  libc-2.3.2.so  getc
00c50800 405    1.2607  libm-2.3.2.so  __GI___finite
0804962c 325    1.0116  dcraw_2        canon_
compressed_load_raw
```

Listing 6. Per function breakdown of samples for dcraw_ 2 with inline abs

Over 11% of the time is spent in `__ieee754_pow`, the glibc pow function. This function take double precision floating point arguments and returns a double precision floating point number. However, the values being passed in and the variable the value is being assigned to are single precision floats. There is a single precision version of the power function, `powf`. Use of this function may avoid some conversions between float and double. Lising 7 lists the samples per function resulting from using the `powf`:

```
/usr/bin/time ../dcraw/dcraw_3 crw_0327.crw
```

The top three functions change an insignificant amount. However, the change reduces the total number of samples from 32,126 for `dcraw_2` to 31,230 for `dcraw_3`. Comparing Listing 6 and 7 reveals that single precision versions of `__pow`, `__isnan`, and `__GI__finite` have fewer samples.

```
/usr/bin/opreport  \
image:/home/wcohen/dcraw/dcraw_3 -l --threshold 1

CPU: P4 / Xeon, speed 1495.19 MHz (estimated)

Counted GLOBAL_POWER_EVENTS events (time during
which processor is not stopped) with a unit mask of
0x01 (count cycles when processor is active) count
750000

vma      samples  %     image name     symbol name
0804d338 19154 61.3340 dcraw_3         vng_interpolate
00c52730 3598  11.5213 libm-2.3.2.so   __ieee754_powf
08051488 2992   9.5808 dcraw_3         convert_to_rgb
08051794 1607   5.1459 dcraw_3         write_ppm
08049260 1339   4.2877 dcraw_3         decompress
0804cd6c 788    2.5233 dcraw_3         scale_colors
00c54ef0 530    1.6971 libm-2.3.2.so   __powf
00b6f0c0 447    1.4314 libc-2.3.2.so   getc
```

Listing 7. Per function breakdown of samples for dcraw with powf

The `vng_interpolate` function has a section of code that computes minimums and maximums. Listing 8 shows the samples for the loop. The Pentium 4 has conditional move instructions which are ideal for this situation. The GCC compiler has an option to generate Pentium 4 instructions. The main drawback of this option is the generated code will not execute on older x86 processors such as the Intel Pentium III. The code was compiled and run with the following command lines:

```
gcc -g -O3 -march=pentium4 dcraw_4.c -o dcraw_4 -lm

/usr/bin/time ./dcraw_4 crw_0327.crw
```

The annotated code in Listing 9 have much lower counts (682) than the equivalent code in Listing 8 (1273). This does not account for the reduction of 3,603 samples, from 31,230 to 27,717 samples. Much of that reduction was in `vng_interpolate` as shown in Listing 10.

```
 14  0.0448 : gmin = INT_MAX;
189  0.6052 : gmax = 0;
502  1.6075 : for (g=0; g < 8; g++) {
158  0.5059 :  if (gmin > gval[g]) gmin = gval[g];
410  1.3129 :  if (gmax < gval[g]) gmax = gval[g];
            : }
```

Listing 8. Computing minimum and maximum code in dcraw_3

```
  6  0.0216 : gmin = INT_MAX;
212  0.7649 : gmax = 0;
158  0.5700 : for (g=0; g < 8; g++) {
148  0.5340 :  if (gmin > gval[g]) gmin = gval[g];
158  0.5700 :  if (gmax < gval[g]) gmax = gval[g];
            : }
```

Listing 9. Computing minimum and maximum code with Pentium 4 instructions

```
/usr/bin/opreport \
image:/home/wcohen/dcraw/dcraw_4 -l --threshold 1
CPU: P4 / Xeon, speed 1495.19 MHz (estimated)
Counted GLOBAL_POWER_EVENTS events (time during
which processor is not stopped) with a unit mask of
0x01 (count cycles when processor is active) count
750000
vma      samples  %       image name
symbol name
0804d6d4 16543   59.6854 dcraw_4       vng_interpo-
late
00c52730 3292   11.8772  libm-2.3.2.so  __ieee754_powf
08051c28 3163   11.4118  dcraw_4        convert_to_rgb
080492ca 1303    4.7011  dcraw_4        decompress
08051f0f 1151    4.1527  dcraw_4        write_ppm
00c54ef0 516     1.8617  libm-2.3.2.so  __powf
00b6f0c0 498     1.7967  libc-2.3.2.so  getc
0804d120 497     1.7931  dcraw_4        scale_colors
08049695 358     1.2916  dcraw_4        canon_
compressed_load_raw
```

Listing 10. Per function breakdown of samples for dcraw using Pentium 4 instructions

Caching

Effective use of the memory hierarchy in the computer system is very important. There is a huge difference in the latency between accessing data in the first level data cache and main memory. Main memory can have latencies that are two orders of magnitude larger than the first level cache.

Due to limitations in OProfile it cannot directly monitor events on the first level data cache. The Pentium 4 performance monitoring events for the L2 data cache can be off by a factor of two. Thus, sampling on these event on the Pentium 4 give a rough estimate of where to look for L1 cache misses. Tables 3 and 4 lists the appropriate events for various processors.

The miss rates for the L1 cache (L2 accesses) should be relatively rare. Thus, the samples interval should be set much smaller than the one used for the `GLOBAL_POWER_EVENTS`. The follow was used to set the sampling on the Pentium 4 to find where the L2 hits occur:

```
/usr/bin/opcontrol --setup \
--no-vmlinux --separate=library \
--event=BSQ_CACHE_REFERENCE:7500:0x7:1:1
```

The results of the run of `dcraw_4` are in Listing 11. As expected, `vng_interpolate` is at the top of the list. However, the `decompress` function is second. This is run relatively early in the program and creates the initial raw image in memory from the compressed file. Listing 12 shows the L2 cache misses; as expected there are fewer samples for Listing 12. The `dcraw` program goes through working memory in a fairly linear manner contributing to the relatively good cache performance.

```
/usr/bin/opreport  \
image:/home/wcohen/dcraw/dcraw_4  --threshold  5 -l

CPU: P4 / Xeon, speed 1495.19 MHz (estimated)

Counted BSQ_CACHE_REFERENCE events (cache references
seen by the bus unit) with a unit mask of 0x07
(multiple flags) count 7500

vma      samples  %      image name   symbol name
0804d6d4 2002   23.7767  dcraw_4       vng_interpolate
080492ca 1655   19.6556  dcraw_4       decompress
00c52730 1086   12.8979  libm-2.3.2.so __ieee754_powf
08051f0f 989    11.7458  dcraw_4       write_ppm
08051c28 606     7.1971  dcraw_4       convert_to_rgb
00b6f0c0 554     6.5796  libc-2.3.2.so getc
```

Listing 11. L2 Cache Hits

Processor	Event
Pentium Pro/PII/PIII	DATA_MEM_REFS
Pentium 4 (HT and non-HT)	BSQ_CACHE_REFERENCE unit-mask=0x7
Athlon/AMD64	DATA_CACHE_ACCESSES

Table 3. Memory Reference Events

```
/usr/bin/opreport  \
image:/home/wcohen/dcraw/dcraw_4  --threshold  5 -l

CPU: P4 / Xeon, speed 1495.19 MHz (estimated)

Counted BSQ_CACHE_REFERENCE events (cache references
seen by the bus unit) with a unit mask of 0x100
(read 2nd level cache miss) count 7500

vma      samples  %  image name    symbol name
0804d6d4 162   42.9708 dcraw_4       vng_interpolate
08051f0f 59    15.6499 dcraw_4       write_ppm
0804d120 53    14.0584 dcraw_4       scale_colors
08051c28 50    13.2626 dcraw_4       convert_to_rgb
00b81c30 40    10.6101 libc-2.3.2.so __GI_memcpy
```

Listing 12. L2 Cache Misses

Processor	Event
Pentium Pro/PII/PIII	DCU_MISS_OUTSTANDING
Pentium 4 (HT and non-HT)	BSQ_CACHE_REFERENCE unit-mask=0x100
Athlon/AMD64	CPU_CLK_UNHALTED

Table 4. Data Cache Miss Events

Virtual Memory Issues

Although current computers may have many megabytes of RAM only a small portion of that can be directly addressed by the processor at any given time. All the RAM is accessed using physical addresses, but the computer programs use virtual memory address (VMA) to allow for memory protection, paging, swapping, and relocation of code. When accessing RAM the processor must convert the virtual memory address to a physical address. The processor stores recent conversions in Translation Lookaside Buffers (TLBs). If the mapping for a particular VMA is found in the TLBs, then the translation is handled completely in hardware. Usually each entry in the TLB is for one page and there a dozen to a couple hundred entries in the TLB. Linux on i386 processors have 4096 byte pages in user space. The Pentium 4 used for these experiments has 64 TLB entries for data memory, allowing hardware to handle VMA to physical memory mappings for 256 kilobytes of RAM for data; the processor has another 64 TLB entries for instructions, allowing mapping for 256 kilobytes of RAM for instructions. If a VMA is encounter outside the ones handled by the TLB, the kernel must navigate some data structures describing virtual memory, compute a new mapping, select a TLB entry to remove, and place the new mapping. If the TLB misses are frequent, performance will suffer. Given, the small sections of code and the caching performance of dcraw, the TLB performance is not a significant problem. Table 5 lists the TLB miss events for various processors.

Processor	Event
Pentium Pro/PII/PIII	ITLB_MISS (instruction)
Pentium 4 (non-HT)	PAGE_WALK_TYPE (0x01 data miss) PAGE_WALK_TYPE (0x02 instruction)
Pentium 4 (HT and non-HT)	ITLB_REFERENCE (0x02) instruction
Athlon/AMD64	L1_AND_L2_DTLB_MISSES (data) L1_AND_L2_ITLB_MISSES (instruction)

Table 5. TLB Misses

Command	Description
`opcontrol --setup --no-vmlinux`	Configure OProfile for data collection
`opcontrol --start`	Start OProfile data collection
`opcontrol --dump`	Flush data to sample database
`opcontrol --shutdown`	Stop OProfile data collection
`opcontrol --reset`	Clear out the sample database
`opreport --long-filenames`	List out files from most to fewest samples
`opreport image:filename -l`	List out functions in filename from most tofewest samples
`opannotate image:filename --source`	Annotate source code for filename with sample counts
`op_help`	List the available performance events for the processor

Table 6. Common OProfile Commands

Instruction Caching

Because the `dcraw` program spends much of its time in a single function, it is unlikely that the processor has a significant problem with instruction caching. However, we will verify this is the case. The Pentium 4 trace cache is different from traditional L1 instruction caches. The trace cache in the Pentium 4 stores decoded instructions for paths through the code. This avoids some of the scheduling constraints of the Pentium III processors where a 4 micro-ops instruction must be followed by two single micro-op instructions to get the maximum three instructions decoded per cycle. As long as the Pentium 4 is executing out the trace cache, it can issue three micro-ops per cycle. However, the Pentium 4 can only decode one instruction per cycle when building a trace. When building a trace on the Pentium 4, significant delays can be observed with `BPU_FETCH_REQUEST`. To verify the that misses are relatively rare, `dcraw_4` was run with with the `–event=BPU_FETCH_REQUEST:7500:0x1:1:1` option. Despite the relatively small interval between samples, 7500, there are still relatively few samples in Listing 13. Table 7 lists the instruction cache miss events for various processors. Table 8 lists the instruction fetch events.

Processor	Event
Pentium Pro/PII/PIII	IFU_IFETCH_MISS
Pentium 4 (non-HT)	TC_DELIVER_MODE unit-mask=0x38
Pentium 4 (HT/non-HT)	BPU_FETCH_REQUEST unit-mask=0x1
Athlon/AMD64	ICACHE_MISSES
Itanium 2	L1I_FILLS

Table 7. Instruction Fetch Miss

```
/usr/bin/opreport \
image:/home/wcohen/dcraw/dcraw_4  --threshold  1 -l

CPU: P4 / Xeon, speed 1495.19 MHz (estimated)

Counted BPU_FETCH_REQUEST events (instruction fetch
requests from the branch predict unit) with a unit
mask of 0x01 (trace cache lookup miss) count 7500

vma      samples  %    image name    symbol name
00c52730 295   87.5371  libm-2.3.2.so  __ieee754_powf
080540d8 12     3.5608  dcraw_4        _fini
00b81710 10     2.9674  libc-2.3.2.so  __GI_memset
00c54ef0 4      1.1869  libm-2.3.2.so __powf
```

Listing 13. BPU_FETCH_REQUEST events for dcraw_4

Processor	Event
Pentium Pro/PII/PIII	IFU_IFETCH
Pentium 4 (non-HT)	TC_DELIVER_MODE unit-mask=7
Athlon/AMD64	ICACHE_FETCHES
Itanium 2	INST_DISPERSED

Table 8. Instruction Fetch

Conclusion

OProfile is a useful tool for identifing locations in programs and reasons for performance problems. The `dcraw` performance was characterized with OProfile, and the performance was improved by 26%.

William E. Cohen is a performance tools engineer at Red Hat, Inc. Will received his BS in electrical engineering from the University of Kansas. He earned a MSEE and a PhD from Purdue University. In his spare time he bicycles and takes pictures with his digital cameras.

Further Reading

AMD Athlon Processor X86 Code Optimization Guide, February 2002, AMD Publication 22007. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/22007.pdf

Software Optimization Guide for AMD Athlon 64 and AMD Opteron Processors, Sept 2003, AMD Publication 25112. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/25112.PDF

D. Coffin, Raw Digital Photo Decoding in Linux. http://www.cybercom.net/~dcoffin/dcraw/

IA-32 Intel Architecture Optimization Reference Manual, 2003, Intel Order number 248966-09. http://www.intel.com/design/pentium4/manuals/248966.htm

Intel Itanium 2 Processor Reference Manual for Software Development and Optimization, June 2002, Intel Document 251110-002. http://www.intel.com/design/itanium2/manuals/251110.htm

OProfile. http://oprofile.sourceforge.net/news/