OProfile manual

Chapter 1. Introduction

This manual applies to OProfile version 0.9.3. OProfile is a profiling system for Linux 2.2/2.4/2.6 systems on a number of architectures. It is capable of profiling all parts of a running system, from the kernel (including modules and interrupt handlers) to shared libraries to binaries. It runs transparently in the background collecting information at a low overhead. These features make it ideal for profiling entire systems to determine bottle necks in real-world systems.

Many CPUs provide "performance counters", hardware registers that can count "events"; for example, cache misses, or CPU cycles. OProfile provides profiles of code based on the number of these occurring events: repeatedly, every time a certain (configurable) number of events has occurred, the PC value is recorded. This information is aggregated into profiles for each binary image.

Some hardware setups do not allow OProfile to use performance counters: in these cases, no events are available, and OProfile operates in timer/RTC mode, as described in later chapters.

1. Applications of OProfile

OProfile is useful in a number of situations. You might want to use OProfile when you :

- need low overhead
- cannot use highly intrusive profiling methods
- need to profile interrupt handlers
- need to profile an application and its shared libraries
- need to capture the performance behaviour of entire system
- want to examine hardware effects such as cache misses
- want detailed source annotation
- want instruction-level profiles
- want call-graph profiles

OProfile is not a panacea. OProfile might not be a complete solution when you :

- require call graph profiles on platforms other than 2.6/x86
- don't have root permissions
- require 100% instruction-accurate profiles
- need function call counts or an interstitial profiling API
- cannot tolerate any disturbance to the system whatsoever
- need to profile interpreted or dynamically compiled code such as Java or Python

2. System requirements

Linux kernel 2.2/2.4/2.6

OProfile uses a kernel module that can be compiled for 2.2.11 or later and 2.4. 2.4.10 or above is required if you use the boot-time kernel option nosmp. 2.6 kernels are supported with the in-kernel OProfile driver. Note that only 32-bit x86 and IA64 are supported on 2.2/2.4 kernels.

2.6 kernels are strongly recommended. Under 2.4, OProfile may cause system crashes if power management is used, or the BIOS does not correctly deal with local APICs.

PPC64 processors (Power4/Power5/PPC970, etc.) require a recent (> 2.6.5) kernel with the line #define PV_970 present in include/asm-ppc64/processor.h.

Profiling the Cell Broadband Engine PowerPC Processing Element (PPE) requires a kernel version of 2.6.18 or more recent. Profiling the Cell Broadband Engine Synergistic Processing Element (SPE) requires a kernel version of 2.6.22 or more recent. Additionally, full support of SPE profiling requires a BFD library from binutils code dated January 2007 or later. To ensure the proper BFD support exists, run the configure utility with --with-target=cell-be.

Note

Attempting to profile SPEs with kernel versions older than 2.6.22 may cause the system to crash.

modutils 2.4.6 or above

You should have installed modutils 2.4.6 or higher (in fact earlier versions work well in almost all cases).

Supported architecture

For Intel IA32, a CPU with either a P6 generation or Pentium 4 core is required. In marketing terms this translates to anything between an Intel Pentium Pro (not Pentium Classics) and a Pentium 4 / Xeon, including all Celerons. The AMD Athlon, and Duron CPUs are also supported. Other IA32 CPU types only support the RTC mode of OProfile; please see later in this manual for details. Hyper-threaded Pentium IVs are not supported in 2.4. For 2.4 kernels, the Intel IA-64 CPUs are also supported. For 2.6 kernels, there is additionally support for Alpha processors, MIPS, ARM, x86-64, sparc64, ppc64, and, in timer mode, PA-RISC and s390.

Uniprocessor or SMP

SMP machines are fully supported.

Required libraries

These libraries are required : popt, bfd, liberty (debian users: libiberty is provided in binutils-dev package), dl, plus the standard C++ libraries.

OProfile GUI
The use of the GUI to start the profiler requires the Qt 2 library. Qt 3 should also work.
ELF
Probably not too strenuous a requirement, but older A.OUT binaries/libraries are not supported.
K&R coding style
OK, so it's not really a requirement, but I wish it was...

3. Internet resources

Web page

There is a web page (which you may be reading now) at http://oprofile.sf.net/.

Download

> You can download a source tarball or get anonymous CVS at the sourceforge page, http://sf.net/projects/oprofile/.

Mailing list

> There is a low-traffic OProfile-specific mailing list, details at http://sf.net/mail/?group_id=16191.

Bug tracker

> There is a bug tracker for OProfile at SourceForge, http://sf.net/tracker/?group_id=16191&atid=116191.

IRC channel

> Several OProfile developers and users sometimes hang out on channel #oprofile on the OFTC network.

4. Installation

First you need to build OProfile and install it. ./configure, make, make install is often all you need, but note these arguments to ./configure :

--with-linux

> Use this option to specify the location of the kernel source tree you wish to compile against. The kernel module is built against this source and will only work with a running kernel built from the same source with exact same options, so it is important you specify this option if you need to.

--with-kernel-support

Use this option with 2.6 and above kernels to indicate the kernel provides the OProfile device driver.

--with-qt-dir/includes/libraries

Specify the location of Qt headers and libraries. It defaults to searching in $QTDIR if these are not specified.

--disable-werror

Development versions of OProfile build by default with -Werror. This option turns -Werror off.

--disable-optimization

Disable the -O2 compiler flag (useful if you discover an OProfile bug and want to give a useful back-trace etc.)

You'll need to have a configured kernel source for the current kernel to build the module for 2.4 kernels. Since all distributions provide different kernels it's unlikely the running kernel match the configured source you installed. The safest way is to recompile your own kernel, run it and compile oprofile. It is also recommended that if you have a uniprocessor machine, you enable the local APIC / IO_APIC support for your kernel (this is automatically enabled for SMP kernels). With many BIOS, kernel >= 2.6.9 and UP kernel it's not sufficient to enable the local APIC you must also turn it on explicitely at boot time by providing "lapic" option to the kernel. On machines with power management, such as laptops, the power management must be turned off when using OProfile with 2.4 kernels. The power management software in the BIOS cannot handle the non-maskable interrupts (NMIs) used by OProfile for data collection. If you use the NMI watchdog, be aware that the watchdog is disabled when profiling starts, and not re-enabled until the OProfile module is removed (or, in 2.6, when OProfile is not running). If you compile OProfile for a 2.2 kernel you must be root to compile the module. If you are using 2.6 kernels or higher, you do not need kernel source, as long as the OProfile driver is enabled; additionally, you should not need to disable power management.

Please note that you must save or have available the vmlinux file generated during a kernel compile, as OProfile needs it (you can use --no-vmlinux, but this will prevent kernel profiling).

5. Uninstalling OProfile

You must have the source tree available to uninstall OProfile; a **make uninstall** will remove all installed files except your configuration file in the directory ~/.oprofile.

Chapter 2. Overview

1. Getting started

Before you can use OProfile, you must set it up. The minimum setup required for this is to tell OProfile where the vmlinux file corresponding to the running kernel is, for example :

```
opcontrol --vmlinux=/boot/vmlinux-`uname -r`
```

If you don't want to profile the kernel itself, you can tell OProfile you don't have a vmlinux file :

```
opcontrol --no-vmlinux
```

Now we are ready to start the daemon (oprofiled) which collects the profile data :

```
opcontrol --start
```

When I want to stop profiling, I can do so with :

```
opcontrol --shutdown
```

Note that unlike gprof, no instrumentation (-pg and -a options to gcc) is necessary.

Periodically (or on opcontrol --shutdown or opcontrol --dump) the profile data is written out into the $SESSION_DIR/samples directory (by default at /var/lib/oprofile/samples). These profile files cover shared libraries, applications, the kernel (vmlinux), and kernel modules. You can clear the profile data (at any time) with opcontrol --reset.

To place these sample database files in a specific directory instead of the default location (/var/lib/oprofile) use the --session-dir=dir option. You must also specify the --session-dir to tell the tools to continue using this directory. (In the future, we should allow this to be specified in an environment variable.) :

```
opcontrol --no-vmlinux --session-dir=/home/me/tmpsession
```

```
opcontrol --start --session-dir=/home/me/tmpsession
```

You can get summaries of this data in a number of ways at any time. To get a summary of data across the entire system for all of these profiles, you can do :

```
opreport [--session-dir=dir]
```

Or to get a more detailed summary, for a particular image, you can do something like :

```
opreport -l /boot/vmlinux-`uname -r`
```

There are also a number of other ways of presenting the data, as described later in this manual. Note that OProfile will choose a default profiling setup for you. However, there are a number of options you can pass to opcontrol if you need to change something, also detailed later.

2. Tools summary

This section gives a brief description of the available OProfile utilities and their purpose.

ophelp

> This utility lists the available events and short descriptions.

opcontrol

> Used for controlling the OProfile data collection, discussed in Chapter 3, Controlling the profiler.

opreport

> This is the main tool for retrieving useful profile data, described in Section 2, “Image summaries and symbol summaries (opreport)”.

opannotate

> This utility can be used to produce annotated source, assembly or mixed source/assembly. Source level annotation is available only if the application was compiled with debugging symbols. See Section 3, “Outputting annotated source (opannotate)”.

opgprof

> This utility can output gprof-style data files for a binary, for use with gprof -p. See Section 4, “gprof-compatible output (opgprof)”.

oparchive

> This utility can be used to collect executables, debuginfo, and sample files and copy the files into an archive. The archive is self-contained and can be moved to another machine for further analysis. See Section 5, “Archiving measurements (oparchive)”.

opimport

> This utility converts sample database files from a foreign binary format (abi) to the native format. This is useful only when moving sample files between hosts, for analysis on platforms other than the one used for collection. See Section 6, “Converting sample database files (opimport)”.

Chapter 3. Controlling the profiler

1. Using opcontrol

In this section we describe the configuration and control of the profiling system with opcontrol in more depth. The **opcontrol** script has a default setup, but you can alter this with the options given below. In particular, if your hardware supports performance counters, you can configure them. There are a number of counters (for example, counter 0 and counter 1 on the Pentium III). Each of these counters can be programmed with an event to count, such as cache misses or MMX operations. The event chosen for each counter is reflected in the profile data collected by OProfile: functions and binaries at the top of the profiles reflect that most of the chosen events happened within that code.

Additionally, each counter has a "count" value: this corresponds to how detailed the profile is. The lower the value, the more frequently profile samples are taken. A counter can choose to sample only kernel code, user-space code, or both (both is the default). Finally, some events have a "unit mask" - this is a value that further restricts the types of event that are counted. The event types and unit masks for your CPU are listed by **opcontrol --list-events**.

The **opcontrol** script provides the following actions :

--init

Loads the OProfile module if required and makes the OProfile driver interface available.

--setup

Followed by list arguments for profiling set up. List of arguments saved in /root/.oprofile/daemonrc. Giving this option is not necessary; you can just directly pass one of the setup options, e.g. **opcontrol --no-vmlinux**.

--status

Show configuration information.

--start-daemon

Start the oprofile daemon without starting actual profiling. The profiling can then be started using --start. This is useful for avoiding measuring the cost of daemon startup, as --start is a simple write to a file in oprofilefs. Not available in 2.2/2.4 kernels.

--start

Start data collection with either arguments provided by --setup or information saved in /root/.oprofile/daemonrc. Specifying the addition --verbose makes the daemon generate lots of debug data whilst it is running.

--dump

Force a flush of the collected profiling data to the daemon.

--stop

Stop data collection (this separate step is not possible with 2.2 or 2.4 kernels).

--shutdown

Stop data collection and kill the daemon.

--reset

Clears out data from current session, but leaves saved sessions.

--save=session_name

Save data from current session to session_name.

--deinit

Shuts down daemon. Unload the OProfile module and oprofilefs.

--list-events

List event types and unit masks.

--help

Generate usage messages.

There are a number of possible settings, of which, only --vmlinux (or --no-vmlinux) is required. These settings are stored in ~/.oprofile/daemonrc.

--buffer-size=num

Number of samples in kernel buffer. When using a 2.6 kernel buffer watershed need to be tweaked when changing this value.

--buffer-watershed=num

Set kernel buffer watershed to num samples (2.6 only). When it'll remain only buffer-size - buffer-watershed free entry in the kernel buffer data will be flushed to daemon, most usefull value are in the range [0.25 - 0.5] * buffer-size.

--cpu-buffer-size=num

Number of samples in kernel per-cpu buffer (2.6 only). If you profile at high rate it can help to increase this if the log file show excessive count of sample lost cpu buffer overflow.

--event=[eventspec]

Use the given performance counter event to profile. See Section 1.2, “Specifying performance counter events” below.

--session-dir=dir_path

Create/use sample database out of directory dir_path instead of the default location (/var/lib/oprofile).

--separate=[none,lib,kernel,thread,cpu,all]

By default, every profile is stored in a single file. Thus, for example, samples in the C library are all accredited to the /lib/libc.o profile. However, you choose to create separate sample files by specifying one of the below options.

none	No profile separation (default)
lib	Create per-application profiles for libraries
kernel	Create per-application profiles for the kernel and kernel modules
thread	Create profiles for each thread and each task
cpu	Create profiles for each CPU
all	All of the above options

Note that --separate=kernel also turns on --separate=lib. When using --separate=kernel, samples in hardware interrupts, soft-irqs, or other asynchronous kernel contexts are credited to the task currently running. This means you will see seemingly nonsense profiles such as /bin/bash showing samples for the PPP modules, etc.

On 2.2/2.4 only kernel threads already started when profiling begins are correctly profiled; newly started kernel thread samples are credited to the vmlinux (kernel) profile.

Using --separate=thread creates a lot of sample files if you leave OProfile running for a while; it's most useful when used for short sessions, or when using image filtering.

--callgraph=#depth

Enable call-graph sample collection with a maximum depth. Use 0 to disable callgraph profiling. NOTE: Callgraph support is available on a limited number of platforms at this time; for example:

- x86 with recent 2.6 kernel

- ARM with recent 2.6 kernel
- PowerPC with 2.6.17 kernel

--image=image,[images]|"all"

Image filtering. If you specify one or more absolute paths to binaries, OProfile will only produce profile results for those binary images. This is useful for restricting the sometimes voluminous output you may get otherwise, especially with --separate=thread. Note that if you are using --separate=lib or --separate=kernel, then if you specification an application binary, the shared libraries and kernel code *are* included. Specify the value "all" to profile everything (the default).

--vmlinux=file

vmlinux kernel image.

--no-vmlinux

Use this when you don't have a kernel vmlinux file, and you don't want to profile the kernel. This still counts the total number of kernel samples, but can't give symbol-based results for the kernel or any modules.

1.1. Examples

1.1.1. Intel performance counter setup

Here, we have a Pentium III running at 800MHz, and we want to look at where data memory references are happening most, and also get results for CPU time.

```
# opcontrol --event=CPU_CLK_UNHALTED:400000 --event=DATA_MEM_REFS:10000
# opcontrol --vmlinux=/boot/2.6.0/vmlinux
# opcontrol --start
```

1.1.2. RTC mode

Here, we have an Intel laptop without support for performance counters, running on 2.4 kernels.

```
# ophelp -r
CPU with RTC device
# opcontrol --vmlinux=/boot/2.4.13/vmlinux --event=RTC_INTERRUPTS:1024
# opcontrol --start
```

1.1.3. Starting the daemon separately

If we're running 2.6 kernels, we can use --start-daemon to avoid the profiler startup affecting results.

```
# opcontrol --vmlinux=/boot/2.6.0/vmlinux
# opcontrol --start-daemon
# my_favourite_benchmark --init
# opcontrol --start ; my_favourite_benchmark --run ; opcontrol --stop
```

1.1.4. Separate profiles for libraries and the kernel

Here, we want to see a profile of the OProfile daemon itself, including when it was running inside the kernel driver, and its use of shared libraries.

```
# opcontrol --separate=kernel --vmlinux=/boot/2.6.0/vmlinux
# opcontrol --start
# my_favourite_stress_test --run
# opreport -l -p /lib/modules/2.6.0/kernel /usr/local/bin/oprofiled
```

1.1.5. Profiling sessions

It can often be useful to split up profiling data into several different time periods. For example, you may want to collect data on an application's startup separately from the normal runtime data. You can use the simple command **opcontrol --save** to do this. For example :

```
# opcontrol --save=blah
```

will create a sub-directory in $SESSION_DIR/samples containing the samples up to that point (the current session's sample files are moved into this directory). You can then pass this session name as a parameter to the post-profiling analysis tools, to only get data up to the point you named the session. If you do not want to save a session, you can do **rm -rf $SESSION_DIR/samples/sessionname** or, for the current session, **opcontrol --reset**.

1.2. Specifying performance counter events

The --event option to **opcontrol** takes a specification that indicates how the details of each hardware performance counter should be setup. If you want to revert to OProfile's default setting (--event is strictly optional), use --event=default.

You can pass multiple event specifications. OProfile will allocate hardware counters as necessary. Note that some combinations are not allowed by the CPU; running **opcontrol --list-events** gives the details of each event. The event specification is a colon-separated string of the form *name*:*count*:*unitmask*:*kernel*:*user* as described in this table:

Note

For the PowerPC platforms, all events specified must be in the same group; i.e., the group number appended to the event name (e.g. <*some-event-name*>_GRP9) must be the same.

name	The symbolic event name, e.g. CPU_CLK_UNHALTED
count	The counter reset value, e.g. 100000
unitmask	The unit mask, as given in the events list, e.g. 0x0f
kernel	Whether to profile kernel code
user	Whether to profile userspace code

The last three values are optional, if you omit them (e.g. --event=DATA_MEM_REFS:30000), they will be set to the default values (a unit mask of 0, and profiling both kernel and userspace code). Note that some events require a unit mask.

If OProfile is using RTC mode, and you want to alter the default counter value, you can use something like --event=RTC_INTERRUPTS:2048. Note the last three values here are ignored. If OProfile is using timer-interrupt mode, there is no configuration possible.

The table below lists the events selected by default (--event=default) for the various computer architectures:

Processor	cpu_type	Default event
Alpha EV4	alpha/ev4	CYCLES:100000:0:1:1
Alpha EV5	alpha/ev5	CYCLES:100000:0:1:1
Alpha PCA56	alpha/pca56	CYCLES:100000:0:1:1
Alpha EV6	alpha/ev6	CYCLES:100000:0:1:1
Alpha EV67	alpha/ev67	CYCLES:100000:0:1:1
ARM/XScale PMU1	arm/xscale1	CPU_CYCLES:100000:0:1:1
ARM/XScale PMU2	arm/xscale2	CPU_CYCLES:100000:0:1:1
ARM/MPCore	arm/mpcore	CPU_CYCLES:100000:0:1:1
Athlon	i386/athlon	CPU_CLK_UNHALTED:100000:0:1:1
Pentium Pro	i386/ppro	CPU_CLK_UNHALTED:100000:0:1:1
Pentium II	i386/pii	CPU_CLK_UNHALTED:100000:0:1:1
Pentium III	i386/piii	CPU_CLK_UNHALTED:100000:0:1:1

Pentium M (P6 core)	i386/p6_mobile	CPU_CLK_UNHALTED:100000:0:1:1
Pentium 4 (non-HT)	i386/p4	GLOBAL_POWER_EVENTS:100000:1:1:1
Pentium 4 (HT)	i386/p4-ht	GLOBAL_POWER_EVENTS:100000:1:1:1
Hammer	x86-64/hammer	CPU_CLK_UNHALTED:100000:0:1:1
Itanium	ia64/itanium	CPU_CYCLES:100000:0:1:1
Itanium 2	ia64/itanium2	CPU_CYCLES:100000:0:1:1
TIMER_INT	timer	None selectable
IBM iseries	PowerPC 4/5/970	CYCLES:10000:0:1:1
IBM pseries	PowerPC 4/5/970/Cell	CYCLES:10000:0:1:1
IBM s390	timer	None selectable
IBM s390x	timer	None selectable

2. Using oprof_start

The **oprof_start** application provides a convenient way to start the profiler. Note that **oprof_start** is just a wrapper around the **opcontrol** script, so it does not provide more services than the script itself.

After **oprof_start** is started you can select the event type for each counter; the sampling rate and other related parameters are explained in Section 1, “Using **opcontrol**”. The "Configuration" section allows you to set general parameters such as the buffer size, kernel filename etc. The counter setup interface should be self-explanatory; Section 3.1, “Hardware performance counters” and related links contain information on using unit masks.

A status line shows the current status of the profiler: how long it has been running, and the average number of interrupts received per second and the total, over all processors. Note that quitting **oprof_start** does not stop the profiler.

Your configuration is saved in the same file as **opcontrol** uses; that is, ~/.oprofile/daemonrc.

3. Configuration details

3.1. Hardware performance counters

Note

Your CPU type may not include the requisite support for hardware performance counters, in which case you must use OProfile in RTC mode in 2.4 (see Section 3.2, “OProfile in RTC mode”), or timer mode in 2.6 (see Section 3.3, “OProfile in timer interrupt mode”). You do not really need to read this section unless you are interested in using events other than the default event chosen by OProfile.

The Intel hardware performance counters are detailed in the Intel IA-32 Architecture Manual, Volume 3, available from http://developer.intel.com/. The AMD Athlon/Duron implementation is detailed in http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/22007.pdf. For PowerPC64 processors in IBM iSeries, pSeries, and blade server systems, processor documentation is available at http://www-306.ibm.com/chips/techlib/techlib.nsf/productfamilies/PowerPC. (For example, the specific publication containing information on the performance monitor unit for the PowerPC970 is "IBM PowerPC 970FX RISC Microprocessor User's Manual.") These processors are capable of delivering an interrupt when a counter overflows. This is the basic mechanism on which OProfile is based. The delivery mode is NMI, so blocking interrupts in the kernel does not prevent profiling. When the interrupt handler is called, the current PC value and the current task are recorded into the profiling structure. This allows the overflow event to be attached to a specific assembly instruction in a binary image. The daemon receives this data from the kernel, and writes it to the sample files.

If we use an event such as CPU_CLK_UNHALTED or INST_RETIRED (GLOBAL_POWER_EVENTS or INSTR_RETIRED, respectively, on the Pentium 4), we can use the overflow counts as an estimate of actual time spent in each part of code. Alternatively we can profile interesting data such as the cache behaviour of routines with the other available counters.

However there are several caveats. First, there are those issues listed in the Intel manual. There is a delay between the counter overflow and the interrupt delivery that can skew results on a small scale - this means you cannot rely on the profiles at the instruction level as being perfectly accurate. If you are using an "event-mode" counter such as the cache counters, a count registered against it doesn't mean that it is responsible for that event. However, it implies that the counter overflowed in the dynamic vicinity of that instruction, to within a few instructions. Further details on this problem can be found in Chapter 5, *Interpreting profiling results* and also in the Digital paper "ProfileMe: A Hardware Performance Counter".

Each counter has several configuration parameters. First, there is the unit mask: this simply further specifies what to count. Second, there is the counter value, discussed below. Third, there is a parameter whether to increment counts whilst in kernel or user space. You can configure these separately for each counter.

After each overflow event, the counter will be re-initialized such that another overflow will occur after this many events have been counted. Thus, higher values mean less-detailed profiling, and lower values mean more detail, but higher overhead. Picking a good value for this parameter is, unfortunately, somewhat of a black art. It is of course dependent on the event you have chosen. Specifying too large a value will mean not enough interrupts are generated to give a realistic profile (though this problem can be ameliorated by profiling for *longer*). Specifying too small a value can lead to higher performance overhead.

3.2. OProfile in RTC mode

Note

This section applies to 2.2/2.4 kernels only.

Some CPU types do not provide the needed hardware support to use the hardware performance counters. This includes some laptops, classic Pentiums, and other CPU types not yet supported by OProfile (such as Cyrix). On these machines, OProfile falls back to using the real-time clock interrupt to collect samples. This interrupt is also used by the **rtc** module: you cannot have both the OProfile and rtc modules loaded nor the rtc support compiled in the kernel.

RTC mode is less capable than the hardware counters mode; in particular, it is unable to profile sections of the kernel where interrupts are disabled. There is just one available event, "RTC interrupts", and its value corresponds to the number of interrupts generated per second (that is, a higher number means a better profiling resolution, and higher overhead). The current implementation of the real-time clock supports only power-of-two sampling rates from 2 to 4096 per second. Other values within this range are rounded to the nearest power of two.

You can force use of the RTC interrupt with the force_rtc=1 module parameter.

Setting the value from the GUI should be straightforward. On the command line, you need to specify the event to **opcontrol**, e.g. :

```
opcontrol --event=RTC_INTERRUPTS:256
```

3.3. OProfile in timer interrupt mode

Note

This section applies to 2.6 kernels and above only.

In 2.6 kernels on CPUs without OProfile support for the hardware performance counters, the driver falls back to using the timer interrupt for profiling. Like the RTC mode in 2.4 kernels, this is not able to profile code that has interrupts disabled. Note that there are no configuration parameters for setting this, unlike the RTC and hardware performance counter setup.

You can force use of the timer interrupt by using the timer=1 module parameter (or oprofile.timer=1 on the boot command line if OProfile is built-in).

3.4. Pentium 4 support

The Pentium 4 / Xeon performance counters are organized around 3 types of model specific registers (MSRs): 45 event selection control registers (ESCRs), 18 counter configuration control registers (CCCRs) and 18 counters. ESCRs describe a particular set of events which are to be recorded, and CCCRs bind ESCRs to counters and configure their operation. Unfortunately the relationship between these registers is quite complex; they cannot all be used with one another at any time. There is, however, a subset of 8 counters, 8 ESCRs, and 8 CCCRs which can be used independently of one another, so OProfile only accesses those registers, treating them as a bank of 8 "normal" counters, similar to those in the P6 or Athlon families of CPU.

There is currently no support for Precision Event-Based Sampling (PEBS), nor any advanced uses of the Debug Store (DS). Current support is limited to the conservative extension of OProfile's existing interrupt-based model described above. Performance monitoring hardware on Pentium 4 / Xeon processors with Hyperthreading enabled (multiple logical processors on a single die) is not supported in 2.4 kernels (you can use OProfile if you disable hyper-threading, though).

3.5. Intel Itanium 2 support

The Itanium 2 performance monitoring unit (PMU) organizes the counters as four pairs of performance event monitoring registers. Each pair is composed of a Performance Monitoring Configuration (PMC) register and Performance Monitoring Data (PMD) register. The PMC selects the performance event being monitored and the PMD determines the sampling interval. The IA64 Performance Monitoring Unit (PMU) triggers sampling with maskable interrupts. Thus, samples will not occur in sections of the IA64 kernel where interrupts are disabled.

None of the advance features of the Itanium 2 performance monitoring unit such as opcode matching, address range matching, or precise event sampling are supported by this version of OProfile. The Itanium 2 support only maps OProfile's existing interrupt-based model to the PMU hardware.

3.6. PowerPC64 support

The performance monitoring unit (PMU) for the IBM PowerPC 64-bit processors consists of between 4 and 8 counters (depending on the model), plus three special purpose registers used for programming the counters -- MMCR0, MMCR1, and MMCRA. Advanced features such as instruction matching and thresholding are not supported by this version of OProfile.

Note

Later versions of the IBM POWER5+ processor (beginning with revision 3.0) run the performance monitor unit in POWER6 mode, effectively removing OProfile's access to counters 5 and 6. These two counters are dedicated to counting instructions completed and cycles, respectively. In POWER6 mode, however, the counters do not generate an interrupt on overflow and so are unusable by OProfile. Kernel versions 2.6.23 and higher will recognize this mode and export "ppc64/power5++" as the cpu_type to the oprofilefs pseudo filesystem. OProfile userspace responds to this cpu_type by removing these counters from the list of potential events to count. Without this kernel support, attempts to profile using an event from one of these counters will yield incorrect results -- typically, zero (or near zero) samples in the generated report.

3.7. Cell Broadband Engine support

The Cell Broadband Engine (CBE) processor core consists of a PowerPC Processing Element (PPE) and 8 Synergistic Processing Elements (SPE). PPEs and SPEs each consist of a processing unit (PPU and SPU, respectively) and other hardware components, such as memory controllers.

A PPU has two hardware threads (aka "virtual CPUs"). The performance monitor unit of the CBE collects event information on one hardware thread at a time. Therefore, when profiling PPE events, OProfile collects the profile based on the selected events by time slicing the performance counter hardware between the two threads. The user must ensure the collection interval is long enough so that the time spent collecting data for each PPU is sufficient to obtain a good profile.

To profile an SPU application, the user should specify the SPU_CYCLES event. When starting OProfile with SPU_CYCLES, the opcontrol script enforces certain separation parameters (separate=cpu,lib) to ensure that sufficient information is collected in the sample data in order to generate a complete report. The --merge=cpu option can be used to obtain a more readable report if analyzing the performance of each separate SPU is not necessary.

SPU profile reports have some unique characteristics compared to reports for standard architectures:

- Typically no "app name" column. This is really standard OProfile behavior when the report contains samples for just a single application, which is commonly the case when profiling SPUs.
- "CPU" equates to "SPU"
- Specifying '--long-filenames' on the opreport command does not always result in long filenames. This happens when the SPU application code is embedded in the PPE executable or shared library. The embedded SPU ELF data contains only the short filename (i.e., no path information) for the SPU binary file that was used as the source for embedding. The reason that just the short filename is used is because the original SPU binary file may not exist or be accessible at runtime. The performance analyst must have sufficient knowledge of the application to be able to correlate the SPU binary image names found in the report to the application's source files.

  Note

  Compile the application with -g and generate the OProfile report with -g to facilitate finding the right source file(s) on which to focus.

3.8. Dangerous counter settings

OProfile is a low-level profiler which allow continuous profiling with a low-overhead cost. If too low a count reset value is set for a counter, the system can become overloaded with counter interrupts, and seem as if the system has frozen. Whilst some validation is done, it is not foolproof.

Note

This can happen as follows: When the profiler count reaches zero an NMI handler is called which stores the sample values in an internal buffer, then resets the counter to its original value. If the count is very low, a pending NMI can be sent before the NMI handler has completed. Due to the priority of the NMI, the local APIC delivers the pending interrupt immediately after completion of the previous interrupt handler, and control never returns to other parts of the system. In this way the system seems to be frozen.

If this happens, it will be impossible to bring the system back to a workable state. There is no way to provide real security against this happening, other than making sure to use a reasonable value for the counter reset. For example, setting CPU_CLK_UNHALTED event type with a ridiculously low reset count (e.g. 500) is likely to freeze the system.

In short : **Don't try a foolish sample count value**. Unfortunately the definition of a foolish value is really dependent on the event type - if ever in doubt

Chapter 4. Obtaining results

OK, so the profiler has been running, but it's not much use unless we can get some data out. Fairly often, OProfile does a little *too* good a job of keeping overhead low, and no data reaches the profiler. This can happen on lightly-loaded machines. Remember you can force a dump at any time with :

```
opcontrol --dump
```

Remember to do this before complaining there is no profiling data ! Now that we've got some data, it has to be processed. That's the job of **opreport**, **opannotate**, or **opgprof**.

1. Profile specifications

All of the analysis tools take a *profile specification*. This is a set of definitions that describe which actual profiles should be examined. The simplest profile specification is empty: this will match all the available profile files for the current session (this is what happens when you do **opreport**).

Specification parameters are of the form name:value[,value]. For example, if I wanted to get a combined symbol summary for /bin/myprog and /bin/myprog2, I could do **opreport -l image:/bin/myprog,/bin/myprog2**. As a special case, you don't actually need to specify the image: part here: anything left on the command line is assumed to be an image: name. Similarly, if no session:

is specified, then session:current is assumed ("current" is a special name of the current / last profiling session).

In addition to the comma-separated list shown above, some of the specification parameters can take **glob**-style values. For example, if I want to see image summaries for all binaries profiled in /usr/bin/, I could do **opreport image:/usr/bin/***. Note the necessity to escape the special character from the shell.

For **opreport**, profile specifications can be used to define two profiles, giving differential output. This is done by enclosing each of the two specifications within curly braces, as shown in the examples below. Any specifications outside of curly braces are shared across both.

1.1. Examples

Image summaries for all profiles with DATA_MEM_REFS samples in the saved session called "stresstest" :

```
# opreport session:stresstest event:DATA_MEM_REFS
```

Symbol summary for the application called "test_sym53c8xx,9xx". Note the escaping is necessary as image: takes a comma-separated list.

```
# opreport -l ./test/test_sym53c8xx\,9xx
```

Image summaries for all binaries in the test directory, excepting boring-test :

```
# opreport image:./test/\* image-exclude:./test/boring-test
```

Differential profile of a binary stored in two archives :

```
# opreport -l /bin/bash { archive:./orig } { archive:./new }
```

Differential profile of an archived binary with the current session :

```
# opreport -l /bin/bash { archive:./orig } { }
```

1.2. Profile specification parameters

archive: *archivepath*

> A path to an archive made with **oparchive**. Absence of this tag, unlike others, means "the current system", equivalent to specifying "archive:".

session: *sessionlist*

A comma-separated list of session names to resolve in. Absence of this tag, unlike others, means "the current session", equivalent to specifying "session:current".

session-exclude: *sessionlist*

A comma-separated list of sessions to exclude.

image: *imagelist*

A comma-separated list of image names to resolve. Each entry may be relative path, **glob**-style name, or full path, e.g.

```
opreport 'image:/usr/bin/oprofiled,*op*,./opreport'
```

image-exclude: *imagelist*

Same as image:, but the matching images are excluded.

lib-image: *imagelist*

Same as image:, but only for images that are for a particular primary binary image (namely, an application). This only makes sense to use if you're using --separate. This includes kernel modules and the kernel when using --separate=kernel.

lib-image-exclude: *imagelist*

Same as lib-image:, but the matching images are excluded.

event: *eventlist*

The symbolic event name to match on, e.g. event:DATA_MEM_REFS. You can pass a list of events for side-by-side comparison with **opreport**. When using the timer interrupt, the event is always "TIMER".

count: *eventcountlist*

The event count to match on, e.g. event:DATA_MEM_REFS count:30000. Note that this value refers to the setting used for **opcontrol** only, and has nothing to do with the sample counts in the profile data itself. You can pass a list of events for side-by-side comparison with **opreport**. When using the timer interrupt, the count is always 0 (indicating it cannot be set).

unit-mask: *masklist*

The unit mask value of the event to match on, e.g. unit-mask:1. You can pass a list of events for side-by-side comparison with **opreport**.

cpu: *cpulist*

Only consider profiles for the given numbered CPU (starting from zero). This is only useful when using CPU profile separation.

tgid: *pidlist*

Only consider profiles for the given task groups. Unless some program is using threads, the task group ID of a process is the same as its process ID. This option corresponds to the POSIX notion of a thread group. This is only useful when using per-process profile separation.

tid: *tidlist*

Only consider profiles for the given threads. When using recent thread libraries, all threads in a process share the same task group ID, but have different thread IDs. You can use this option in combination with tgid: to restrict the results to particular threads within a process. This is only useful when using per-process profile separation.

1.3. Locating and managing binary images

Each session's sample files can be found in the $SESSION_DIR/samples/ directory (default: /var/lib/oprofile/samples/). These are used, along with the binary image files, to produce human-readable data. In some circumstances (kernel modules in an initrd, or modules on 2.6 kernels), OProfile will not be able to find the binary images. All the tools have an --image-path option to which you can pass a comma-separated list of alternate paths to search. For example, I can let OProfile find my 2.6 modules by using **--image-path /lib/modules/2.6.0/kernel/**. It is your responsibility to ensure that the correct images are found when using this option.

Note that if a binary image changes after the sample file was created, you won't be able to get useful symbol-based data out. This situation is detected for you. If you replace a binary, you should make sure to save the old binary if you need to do comparative profiles.

1.4. What to do when you don't get any results

When attempting to get output, you may see the error :

```
error: no sample files found: profile specification too strict ?
```

What this is saying is that the profile specification you passed in, when matched against the available sample files, resulted in no matches. There are a number of reasons this might happen:

spelling

You specified a binary name, but spelt it wrongly. Check your spelling !

profiler wasn't running

Make very sure that OProfile was actually up and running when you ran the binary.

binary didn't run long enough

Remember OProfile is a statistical profiler - you're not guaranteed to get samples for short-running programs. You can help this by using a lower count for the performance counter, so there are a lot more samples taken per second.

binary spent most of its time in libraries

Similarly, if the binary spends little time in the main binary image itself, with most of it spent in shared libraries it uses, you might not see any samples for the binary image itself. You can check this by using **opcontrol --separate=lib** before the profiling session, so **opreport** and friends show the library profiles on a per-application basis.

specification was really too strict

For example, you specified something like tgid:3433, but no task with that group ID ever ran the code.

binary didn't generate any events

If you're using a particular event counter, for example counting MMX operations, the code might simply have not generated any events in the first place. Verify the code you're profiling does what you expect it to.

you didn't specify kernel module name correctly

If you're using 2.6 kernels, and trying to get reports for a kernel module, make sure to use the -p option, and specify the module name *with* the .ko extension. Check if the module is one loaded from initrd.

2. Image summaries and symbol summaries (opreport)

The **opreport** utility is the primary utility you will use for getting formatted data out of OProfile. It produces two types of data: image summaries and symbol summaries. An image summary lists the number of samples for individual binary images such as libraries or applications. Symbol summaries provide per-symbol profile data. In the following example, we're getting an image summary for the whole system:

```
$ opreport --long-filenames
CPU: PIII, speed 863.195 MHz (estimated)
Counted CPU_CLK_UNHALTED events (clocks processor is not halted) with a unit mask
```

```
of 0x00 (No unit mask) count 23150
   905898 59.7415 /usr/lib/gcc-lib/i386-redhat-linux/3.2/cc1plus
   214320 14.1338 /boot/2.6.0/vmlinux
   103450  6.8222 /lib/i686/libc-2.3.2.so
    60160  3.9674 /usr/local/bin/madplay
    31769  2.0951 /usr/local/oprofile-pp/bin/oprofiled
    26550  1.7509 /usr/lib/libartsflow.so.1.0.0
    23906  1.5765 /usr/bin/as
    18770  1.2378 /oprofile
    15528  1.0240 /usr/lib/qt-3.0.5/lib/libqt-mt.so.3.0.5
    11979  0.7900 /usr/X11R6/bin/XFree86
    11328  0.7471 /bin/bash
    ...
```

If we had specified --symbols in the previous command, we would have gotten a symbol summary of all the images across the entire system. We can restrict this to only part of the system profile; for example, below is a symbol summary of the OProfile daemon. Note that as we used **opcontrol --separate=kernel**, symbols from images that **oprofiled** has used are also shown.

```
$ opreport -l `which oprofiled` 2>/dev/null | more
CPU: PIII, speed 863.195 MHz (estimated)
Counted CPU_CLK_UNHALTED events (clocks processor is not halted) with a unit mask
of 0x00 (No unit mask) count 23150
vma       samples  %        image name              symbol name
0804be10 14971    28.1993   oprofiled               odb_insert
0804afdc 7144     13.4564   oprofiled               pop_buffer_value
c01daea0 6113     11.5144
$ opreport -l `which oprofiled` 2>/dev/null | more
CPU: PIII, speed 863.195 MHz (estimated)
Counted CPU_CLK_UNHALTED events (clocks processor is not halted) with a unit mask
of 0x00 (No unit mask) count 23150
vma       samples  %        image name              symbol name
0804be10 14971    28.1993   oprofiled               odb_insert
0804afdc 7144     13.4564   oprofiled               pop_buffer_value
c01daea0 6113     11.5144   vmlinux                 __copy_to_user_ll
0804b060 2816      5.3042   oprofiled               opd_put_sample
0804b4a0 2147      4.0441   oprofiled               opd_process_samples
0804acf4 1855      3.4941   oprofiled               opd_put_image_sample
0804ad84 1766      3.3264   oprofiled               opd_find_image
0804a5ec 1084      2.0418   oprofiled               opd_find_module
0804ba5c 741       1.3957   oprofiled               odb_hash_add_node
...
```

These are the two basic ways you are most likely to use regularly, but **opreport** can do a lot more than that, as described below.

2.1. Merging separate profiles

If you have used one of the --separate= options whilst profiling, there can be several separate profiles for a single binary image within a session. Normally the output will keep these images separated (so, for example, the image summary output shows library image summaries on a per-application basis, when using --separate=lib). Sometimes it can be useful to merge these results back together before getting results. The --merge option allows you to do that.

2.2. Side-by-side multiple results

If you have used multiple events when profiling, by default you get side-by-side results of each event's sample values from **opreport**. You can restrict which events to list by appropriate use of the event: profile specifications, etc.

2.3. Callgraph output

When using the opcontrol --callgraph option, you can see what functions are calling other functions in the output. Consider the following program:

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#define SIZE 500000

static int compare(const void *s1, const void *s2)
{
        return strcmp(s1, s2);
}

static void repeat(void)
{
        int i;
        char *strings[SIZE];
        char str[] = "abcdefghijklmnopqrstuvwxyz";

        for (i = 0; i < SIZE; ++i) {
                strings[i] = strdup(str);
                strfry(strings[i]);
        }
        qsort(strings, SIZE, sizeof(char *), compare);
}

int main()
{
        while (1)
                repeat();
}
```

When running with the call-graph option, OProfile will record the function stack every time it takes a sample. **opreport --callgraph** outputs an entry for each function, where each entry looks similar to:

```
samples  %        image name               symbol name
  197       0.1548  cg                       main
  127036   99.8452  cg                       repeat
84590    42.5084  libc-2.3.2.so            strfry
  84590    66.4838  libc-2.3.2.so            strfry [self]
  39169    30.7850  libc-2.3.2.so            random_r
  3475      2.7312  libc-2.3.2.so            __i686.get_pc_thunk.bx
-------------------------------------------------------------------------------
```

Here the non-indented line is the function we're focussing upon (strfry()). This line is the same as you'd get from a normal **opreport** output.

Above the non-indented line we find the functions that called this function (for example, repeat() calls strfry()). The samples and percentage values here refer to the number of times we took a sample where this call was found in the stack; the percentage is relative to all other callers of the function we're focussing on. Note that these values are *not* call counts; they only reflect the call stack every time a sample is taken; that is, if a call is found in the stack at the time of a sample, it is recorded in this count.

Below the line are functions that are called by strfry() (called *callees*). It's clear here that strfry() calls random_r(). We also see a special entry with a "[self]" marker. This records the normal samples for the function, but the percentage becomes relative to all callees. This allows you to compare time spent in the function itself compared to functions it calls. Note that if a function calls itself, then it will appear in the list of callees of itself, but without the "[self]" marker; so recursive calls are still clearly separable.

You may have noticed that the output lists main() as calling strfry(), but it's clear from the source that this doesn't actually happen. See Section 3, “Interpreting call-graph profiles” for an explanation.

2.4. Differential profiles with opreport

Often, we'd like to be able to compare two profiles. For example, when analysing the performance of an application, we'd like to make code changes and examine the effect of the change. This is supported in **opreport** by giving a profile specification that identifies two different profiles. The general form is of:

```
$ opreport <shared-spec> { <first-profile> } { <second-profile> }
```

Note

We lost our Dragon book down the back of the sofa, so you have to be careful to have spaces around those braces, or things will get hopelessly confused. We can only apologise.

For each of the profiles, the shared section is prefixed, and then the specification is analysed. The usual parameters work both within the shared section, and in the sub-specification within the curly braces.

A typical way to use this feature is with archives created with **oparchive**. Let's look at an example:

```
$ ./a
$ oparchive -o orig ./a
$ opcontrol --reset
  # edit and recompile a
$ ./a
  # now compare the current profile of a with the archived profile
$ opreport -xl ./a { archive:./orig } { }
CPU: PIII, speed 863.233 MHz (estimated)
Counted CPU_CLK_UNHALTED events (clocks processor is not halted) with a
unit mask of 0x00 (No unit mask) count 100000
samples  %        diff %    symbol name
92435    48.5366  +0.4999   a
54226    ---      ---       c
49222    25.8459  +++       d
48787    25.6175  -2.2e-01  b
```

Note that we specified an empty second profile in the curly braces, as we wanted to use the current session; alternatively, we could have specified another archive, or a tgid etc. We specified the binary **a** in the shared section, so we matched that in both the profiles we're diffing.

As in the normal output, the results are sorted by the number of samples, and the percentage field represents the relative percentage of the symbol's samples in the second profile.

Notice the new column in the output. This value represents the percentage change of the relative percent between the first and the second profile: roughly, "how much more important this symbol is". Looking at the symbol a(), we can see that it took roughly the same amount of the total profile in both the first and the second profile. The function c() was not in the new profile, so has been marked with ---. Note that the sample value is the number of samples in the first profile; since we're displaying results for the second profile, we don't list a percentage value for it, as it would be meaningless. d() is new in the second profile, and consequently marked with +++.

When comparing profiles between different binaries, it should be clear that functions can change in terms of VMA and size. To avoid this problem, **opreport** considers a symbol to be the same if the symbol name, image name, and owning application name all match; any other factors are ignored. Note that the check for application name means that trying to compare library profiles between two different applications will not work as you might expect: each symbol will be considered different.

2.5. Anonymous executable mappings

Many applications, typically ones involving dynamic compilation into machine code, have executable mappings that are not backed by an ELF file. **opreport** has basic support for showing the samples taken in these regions; for example:

```
$ opreport /usr/jre1.5.0/bin/java
CPU: PIII, speed 863.195 MHz (estimated)
Counted CPU_CLK_UNHALTED events (clocks processor is not halted) with a unit mask
of 0x00 (No unit mask) count 100000
CPU_CLK_UNHALT...|
  samples|      %|
------------------
    27344 100.000 java
        CPU_CLK_UNHALT...|
          samples|      %|
        ------------------
            27236  99.605 anon (tgid:12135 range:0xb2cb8000-0xb2e80000)
              108  0.3949 java
```

Currently, there is no support for getting symbol-based summaries for such regions. Note that, since such mappings are dependent upon individual invocations of a binary, these mappings are always listed as a dependent image, even when using --separate=none. Equally, the results are not affected by the --merge option.

2.6. XML formatted output

The -xml option can be used to generate XML instead of the usual text format. This allows opreport to eliminate some of the constraints dictated by the two dimensional text format. For example, it is possible to separate the sample data across multiple events, cpus and threads. The XML schema implemented by opreport is found in doc/opreport.1.0.xsd. It contains more detailed comments about the structure of the XML generated by opreport.

Since XML is consumed by a client program rather than a user, its structure is fairly static. In particular, the --sort option is incompatible with the --xml option. Percentages are not dislayed in the XML so the options related to percentages will have no effect. Full pathnames are always displayed in the XML so --long-filenames is not necessary. The --details option will cause all of the individual sample data to be included in the XML as well as the instruction byte stream for each symbol (for doing disassembly) and can result in very large XML files.

2.7. Options for opreport

--accumulated / -a

Accumulate sample and percentage counts in the symbol list.

--callgraph / -c

Show callgraph information.

--debug-info / -g

Show source file and line for each symbol.

--demangle / -D none|normal|smart

none: no demangling. normal: use default demangler (default) smart: use pattern-matching to make C++ symbol demangling more readable.

--details / -d

Show per-instruction details for all selected symbols. Note that, for binaries without symbol information, the VMA values shown are raw file offsets for the image binary.

--exclude-dependent / -x

Do not include application-specific images for libraries, kernel modules and the kernel. This option only makes sense if the profile session used --separate.

--exclude-symbols / -e [symbols]

Exclude all the symbols in the given comma-separated list.

--global-percent / -%

Make all percentages relative to the whole profile.

--help / -? / --usage

Show help message.

--image-path / -p [paths]

Comma-separated list of additional paths to search for binaries. This is needed to find modules in kernels 2.6 and upwards.

--include-symbols / -i [symbols]

Only include symbols in the given comma-separated list.

--long-filenames / -f

Output full paths instead of basenames.

--merge / -m [lib,cpu,tid,tgid,unitmask,all]

Merge any profiles separated in a --separate session.

--no-header

Don't output a header detailing profiling parameters.

--output-file / -o [file]

Output to the given file instead of stdout.

--reverse-sort / -r

Reverse the sort from the default.

--session-dir=dir_path

Use sample database out of directory dir_path instead of the default location (/var/lib/oprofile).

--show-address / -w

Show the VMA address of each symbol (off by default).

--sort / -s [vma,sample,symbol,debug,image]

Sort the list of symbols by, respectively, symbol address, number of samples, symbol name, debug filename and line number, binary image filename.

--symbols / -l

List per-symbol information instead of a binary image summary.

--threshold / -t [percentage]

Only output data for symbols that have more than the given percentage of total samples.

--verbose / -V [options]

Give verbose debugging output.

--version / -v

Show version.

--xml / -X

Generate XML output.

3. Outputting annotated source (opannotate)

The **opannotate** utility generates annotated source files or assembly listings, optionally mixed with source. If you want to see the source file, the profiled application needs to have debug information, and the source must be available through this debug information. For GCC, you must use the -g option when you are compiling. If the binary doesn't contain sufficient debug information, you can still use **opannotate --assembly** to get annotated assembly.

Note that for the reason explained in Section 3.1, “Hardware performance counters” the results can be inaccurate. The debug information itself can add other problems; for example, the line number for a symbol can be incorrect. Assembly instructions can be re-ordered and moved by the compiler, and this can lead to crediting source lines with samples not really "owned" by this line. Also see Chapter 5, *Interpreting profiling results*.

You can output the annotation to one single file, containing all the source found using the --source. You can use this in conjunction with --assembly to get combined source/assembly output.

You can also output a directory of annotated source files that maintains the structure of the original sources. Each line in the annotated source is prepended with the samples for that line. Additionally, each symbol is annotated giving details for the symbol as a whole. An example:

```
$ opannotate --source --output-dir=annotated /usr/local/oprofile-pp/bin/oprofiled
$ ls annotated/home/moz/src/oprofile-pp/daemon/
```

```
opd_cookie.h    opd_image.c    opd_kernel.c    opd_sample_files.c    oprofiled.c
```

Line numbers are maintained in the source files, but each file has a footer appended describing the profiling details. The actual annotation looks something like this :

```
...
                 :static uint64_t pop_buffer_value(struct transient * trans)
 11510   1.9661 :{ /* pop_buffer_value total:    89901 15.3566 */
                 :        uint64_t val;
                 :
 10227   1.7469 :        if (!trans->remaining) {
                 :                fprintf(stderr, "BUG: popping empty buffer !\n");
                 :                exit(EXIT_FAILURE);
                 :        }
                 :
                 :        val = get_buffer_value(trans->buffer, 0);
  2281   0.3896 :        trans->remaining--;
  2296   0.3922 :        trans->buffer += kernel_pointer_size;
                 :        return val;
 10454   1.7857 :}
...
```

The first number on each line is the number of samples, whilst the second is the relative percentage of total samples.

3.1. Locating source files

Of course, **opannotate** needs to be able to locate the source files for the binary image(s) in order to produce output. Some binary images have debug information where the given source file paths are relative, not absolute. You can specify search paths to look for these files (similar to **gdb**'s dir command) with the --search-dirs option.

Sometimes you may have a binary image which gives absolute paths for the source files, but you have the actual sources elsewhere (commonly, you've installed an SRPM for a binary on your system and you want annotation from an existing profile). You can use the --base-dirs option to redirect OProfile to look somewhere else for source files. For example, imagine we have a binary generated from a source file that is given in the debug information as /tmp/build/libfoo/foo.c, and you have the source tree matching that binary installed in /home/user/libfoo/. You can redirect OProfile to find foo.c correctly like this :

```
$ opannotate --source --base-dirs=/tmp/build/libfoo/ --search-dirs=/home/user/libfoo/
--output-dir=annotated/ /lib/libfoo.so
```

You can specify multiple (comma-separated) paths to both options.

3.2. Usage of opannotate

--assembly / -a

Output annotated assembly. If this is combined with --source, then mixed source / assembly annotations are output.

--base-dirs / -b [paths]/

Comma-separated list of path prefixes. This can be used to point OProfile to a different location for source files when the debug information specifies an absolute path on your system for the source that does not exist. The prefix is stripped from the debug source file paths, then searched in the search dirs specified by --search-dirs.

--demangle / -D none|normal|smart

none: no demangling. normal: use default demangler (default) smart: use pattern-matching to make C++ symbol demangling more readable.

--exclude-dependent / -x

Do not include application-specific images for libraries, kernel modules and the kernel. This option only makes sense if the profile session used --separate.

--exclude-file [files]

Exclude all files in the given comma-separated list of glob patterns.

--exclude-symbols / -e [symbols]

Exclude all the symbols in the given comma-separated list.

--help / -? / --usage

Show help message.

--image-path / -p [paths]

Comma-separated list of additional paths to search for binaries. This is needed to find modules in kernels 2.6 and upwards.

--include-file [files]

Only include files in the given comma-separated list of glob patterns.

--include-symbols / -i [symbols]

Only include symbols in the given comma-separated list.

--objdump-params [params]

Pass the given parameters as extra values when calling objdump.

--output-dir / -o [dir]

Output directory. This makes opannotate output one annotated file for each source file. This option can't be used in conjunction with --assembly.

--search-dirs / -d [paths]

Comma-separated list of paths to search for source files. This is useful to find source files when the debug information only contains relative paths.

--source / -s

Output annotated source. This requires debugging information to be available for the binaries.

--threshold / -t [percentage]

Only output data for symbols that have more than the given percentage of total samples.

--verbose / -V [options]

Give verbose debugging output.

--version / -v

Show version.

4. gprof-compatible output (opgprof)

If you're familiar with the output produced by **GNU gprof**, you may find **opgprof** useful. It takes a single binary as an argument, and produces a gmon.out file for use with **gprof -p**. If call-graph profiling is enabled, then this is also included.

```
$ opgprof `which oprofiled` # generates gmon.out file
$ gprof -p `which oprofiled` | head
Flat profile:

Each sample counts as 1 samples.
  %   cumulative   self              self     total
 time   samples   samples    calls  T1/call  T1/call  name
 33.13 206237.00 206237.00                             odb_insert
```

```
22.67 347386.00 141149.00                              pop_buffer_value
  9.56 406881.00 59495.00                             opd_put_sample
  7.34 452599.00 45718.00                             opd_find_image
  7.19 497327.00 44728.00                             opd_process_samples
```

4.1. Usage of opgprof

--help / -? / --usage

Show help message.

--image-path / -p [paths]

Comma-separated list of additional paths to search for binaries. This is needed to find modules in kernels 2.6 and upwards.

--output-filename / -o [file]

Output to the given file instead of the default, gmon.out

--threshold / -t [percentage]

Only output data for symbols that have more than the given percentage of total samples.

--verbose / -V [options]

Give verbose debugging output.

--version / -v

Show version.

5. Archiving measurements (oparchive)

The **oparchive** utility generates a directory populated with executable, debug, and oprofile sample files. This directory can be moved to another machine via **tar** and analyzed without further use of the data collection machine.

The following command would collect the sample files, the executables associated with the sample files, and the debuginfo files associated with the executables and copy them into /tmp/current_data:

```
# oparchive -o /tmp/current_data
```

5.1. Usage of oparchive

--help / -? / --usage

Show help message.

--exclude-dependent / -x

Do not include application-specific images for libraries, kernel modules and the kernel. This option only makes sense if the profile session used --separate.

--image-path / -p [paths]

Comma-separated list of additional paths to search for binaries. This is needed to find modules in kernels 2.6 and upwards.

--output-directory / -o [directory]

Output to the given directory. There is no default. This must be specified.

--list-files / -l

Only list the files that would be archived, don't copy them.

--verbose / -V [options]

Give verbose debugging output.

--version / -v

Show version.

6. Converting sample database files (opimport)

This utility converts sample database files from a foreign binary format (abi) to the native format. This is useful only when moving sample files between hosts, for analysis on platforms other than the one used for collection. The abi format of the file to be imported is described in a text file located in $SESSION_DIR/abi.

The following command would convert the input samples files to the output samples files using the given abi file as a binary description of the input file and the curent platform abi as a binary description of the output file.

```
# opimport -a /var/lib/oprofile/abi -o
/tmp/current/.../GLOBAL_POWER_EVENTS.200000.1.all.all.all
```

```
/var/lib/.../mprime/GLOBAL_POWER_EVENTS.200000.1.all.all.all
```

6.1. Usage of opimport

--help / -? / --usage

Show help message.

--abi / -a [filename]

Input abi file description location.

--force / -f

Force conversion even if the input and output abi are identical.

--output / -o [filename]

Specify the output filename. If the output file already exists, the file is not overwritten but data are accumulated in. Sample filename are informative for post profile tools and must be kept identical, in other word the pathname from the first path component containing a '{' must be kept as it in the output filename.

--verbose / -V

Give verbose debugging output.

--version / -v

Show version.

Chapter 5. Interpreting profiling results

The standard caveats of profiling apply in interpreting the results from OProfile: profile realistic situations, profile different scenarios, profile for as long as a time as possible, avoid system-specific artifacts, don't trust the profile data too much. Also bear in mind the comments on the performance counters above - you *cannot* rely on totally accurate instruction-level profiling. However, for almost all circumstances the data can be useful. Ideally a utility such as Intel's VTUNE would be available to allow careful instruction-level analysis; go hassle Intel for this, not me ;)

1. Profiling interrupt latency

This is an example of how the latency of delivery of profiling interrupts can impact the reliability of the profiling data. This is pretty much a worst-case-scenario example: these problems are fairly rare.

```
double fun(double a, double b, double c)
{
  double result = 0;
  for (int i = 0 ; i < 10000; ++i) {
    result += a;
    result *= b;
    result /= c;
  }
  return result;
}
```

Here the last instruction of the loop is very costly, and you would expect the result reflecting that - but (cutting the instructions inside the loop):

```
$ opannotate -a -t 10 ./a.out

     88 15.38% : 8048337:       fadd   %st(3),%st
     48 8.391% : 8048339:       fmul   %st(2),%st
     68 11.88% : 804833b:       fdiv   %st(1),%st
    368 64.33% : 804833d:       inc    %eax
               : 804833e:       cmp    $0x270f,%eax
               : 8048343:       jle    8048337
```

The problem comes from the x86 hardware; when the counter overflows the IRQ is asserted but the hardware has features that can delay the NMI interrupt: x86 hardware is synchronous (i.e. cannot interrupt during an instruction); there is also a latency when the IRQ is asserted, and the multiple execution units and the out-of-order model of modern x86 CPUs also causes problems. This is the same function, with annotation :

```
$ opannotate -s -t 10 ./a.out

               :double fun(double a, double b, double c)
               :{ /* _Z3funddd total:     572 100.0% */
               : double result = 0;
    368 64.33% : for (int i = 0 ; i < 10000; ++i) {
     88 15.38% :    result += a;
     48 8.391% :    result *= b;
     68 11.88% :    result /= c;
               : }
               : return result;
               :}
```

The conclusion: don't trust samples coming at the end of a loop, particularly if the last instruction generated by the compiler is costly. This case can also occur for branches. Always bear in mind that samples can be delayed by a few cycles from its real position. That's a hardware problem and OProfile can do nothing about it.

2. Kernel profiling

2.1. Interrupt masking

OProfile uses non-maskable interrupts (NMI) on the P6 generation, Pentium 4, Athlon and Duron processors. These interrupts can occur even in section of the Linux where interrupts are disabled, allowing collection of samples in virtually all executable code. The RTC, timer interrupt mode, and Itanium 2 collection mechanisms use maskable interrupts. Thus, the RTC and Itanium 2 data collection mechanism have "sample shadows", or blind spots: regions where no samples will be collected. Typically, the samples will be attributed to the code immediately after the interrupts are re-enabled.

2.2. Idle time

Your kernel is likely to support halting the processor when a CPU is idle. As the typical hardware events like CPU_CLK_UNHALTED do not count when the CPU is halted, the kernel profile will not reflect the actual amount of time spent idle. You can change this behaviour by booting with the idle=poll option, which uses a different idle routine. This will appear as poll_idle() in your kernel profile.

2.3. Profiling kernel modules

OProfile profiles kernel modules by default. However, there are a couple of problems you may have when trying to get results. First, you may have booted via an initrd; this means that the actual path for the module binaries cannot be determined automatically. To get around this, you can use the -p option to the profiling tools to specify where to look for the kernel modules.

In 2.6, the information on where kernel module binaries are located has been removed. This means OProfile needs guiding with the -p option to find your modules. Normally, you can just use your standard module top-level directory for this. Note that due to this problem, OProfile cannot check that the modification times match; it is your responsibility to make sure you do not modify a binary after a profile has been created.

If you have run **insmod** or **modprobe** to insert a module in a particular directory, it is important that you specify this directory with the -p option first, so that it over-rides an older module binary that might exist in other directories you've specified with -p. It is up to you to make sure that these values are correct: 2.6 kernels simply do not provide enough information for OProfile to get this information.

3. Interpreting call-graph profiles

Sometimes the results from call-graph profiles may be different to what you expect to see. The first thing to check is whether the target binaries where compiled with frame pointers enabled (if the binary was compiled using **gcc**'s -fomit-frame-pointer option, you will not get meaningful results). Note that as of this writing, the GCC developers plan to disable frame pointers by default. The Linux kernel is built without frame pointers by default; there is a configuration option you can use to turn it on under the "Kernel Hacking" menu.

Often you may see a caller of a function that does not actually directly call the function you're looking at (e.g. if a() calls b(), which in turn calls c(), you may see an entry for a()->c()). What's actually occurring is that we are taking samples at the very start (or the very end) of c(); at these few instructions, we haven't yet created the new function's frame, so it appears as if a() is calling directly into c(). Be careful not to be misled by these entries.

Like the rest of OProfile, call-graph profiling uses a statistical approach; this means that sometimes a backtrace sample is truncated, or even partially wrong. Bear this in mind when examining results.

4. Inaccuracies in annotated source

4.1. Side effects of optimizations

The compiler can introduce some pitfalls in the annotated source output. The optimizer can move pieces of code in such manner that two line of codes are interlaced (instruction scheduling). Also debug info generated by the compiler can show strange behavior. This is especially true for complex expressions e.g. inside an if statement:

```
        if (a && ..
            b && ..
            c &&)
```

here the problem come from the position of line number. The available debug info does not give enough details for the if condition, so all samples are accumulated at the position of the right brace of the expression. Using **opannotate -a** can help to show the real samples at an assembly level.

4.2. Prologues and epilogues

The compiler generally needs to generate "glue" code across function calls, dependent on the particular function call conventions used. Additionally other things need to happen, like stack pointer adjustment for the local variables; this code is known as the function prologue. Similar code is needed at function return, and is known as the function epilogue. This will show up in annotations as samples at the very start and end of a function, where there is no apparent executable code in the source.

4.3. Inlined functions

You may see that a function is credited with a certain number of samples, but the listing does not add up to the correct total. To pick a real example :

```
                  :internal_sk_buff_alloc_security(struct sk_buff *skb)
 353 2.342%       :{ /* internal_sk_buff_alloc_security total: 1882 12.48% */
                  :
                  :        sk_buff_security_t *sksec;
  15 0.0995%      :        int rc = 0;
                  :
  10 0.06633%     :        sksec = skb->lsm_security;
 468 3.104%       :        if (sksec && sksec->magic == DSI_MAGIC) {
                  :                goto out;
                  :        }
                  :
                  :        sksec = (sk_buff_security_t *) get_sk_buff_memory(skb);
   3 0.0199%      :        if (!sksec) {
  38 0.2521%      :                rc = -ENOMEM;
                  :                goto out;
  10 0.06633%     :        }
                  :        memset(sksec, 0, sizeof (sk_buff_security_t));
  44 0.2919%      :        sksec->magic = DSI_MAGIC;
  32 0.2123%      :        sksec->skb = skb;
  45 0.2985%      :        sksec->sid = DSI_SID_NORMAL;
  31 0.2056%      :        skb->lsm_security = sksec;
                  :
                  :      out:
                  :
 146 0.9685%      :        return rc;
                  :
  98 0.6501%      :}
```

Here, the function is credited with 1,882 samples, but the annotations below do not account for this. This is usually because of inline functions - the compiler marks such code with debug entries for the inline function definition, and this is where **opannotate** annotates such samples. In the case above, memset is the most likely candidate for this problem. Examining the mixed source/assembly output can help identify such results.

This problem is more visible when there is no source file available, in the following example it's trivially visible the sums of symbols samples is less than the number of the samples for this file. The difference must be accounted to inline functions.

```
/*
 * Total samples for file : "arch/i386/kernel/process.c"
 *
 *       109    2.4616
 */

 /* default_idle total:          84    1.8970 */
 /* cpu_idle total:                21    0.4743 */
 /* flush_thread total:             1    0.0226 */
 /* prepare_to_copy total:       1    0.0226 */
 /* __switch_to total:             18    0.4065 */
```

The missing samples are not lost, they will be credited to another source location where the inlined function is defined. The inlined function will be credited from multiple call site and merged in one place in the annotated source file so there is no way to see from what call site are coming the samples for an inlined function.

When running **opannotate**, you may get a warning "some functions compiled without debug information may have incorrect source line attributions". In some rare cases, OProfile is not able to verify that the derived source line is correct (when some parts of the binary image are compiled without debugging information). Be wary of results if this warning appears.

Furthermore, for some languages the compiler can implicitly generate functions, such as default copy constructors. Such functions are labelled by the compiler as having a line number of 0, which means the source annotation can be confusing.

4.4. Inaccuracy in line number information

Depending on your compiler you can fall into the following problem:

```
struct big_object { int a[500]; };

int main()
{
		big_object a, b;
		for (int i = 0 ; i != 1000 * 1000; ++i)
			b = a;
		return 0;
}
```

Compiled with **gcc** 3.0.4 the annotated source is clearly inaccurate:

```
                   :int main()
                   :{    /* main total: 7871 100% */
                   :          big_object a, b;
                   :          for (int i = 0 ; i != 1000 * 1000; ++i)
                   :                   b = a;
 7871 100%         :          return 0;
                   :}
```

The problem here is distinct from the IRQ latency problem; the debug line number information is not precise enough; again, looking at output of **opannoatate -as** can help.

```
                   :int main()
                   :{
                   :          big_object a, b;
                   :          for (int i = 0 ; i != 1000 * 1000; ++i)
                   : 80484c0:        push    %ebp
                   : 80484c1:        mov      %esp,%ebp
                   : 80484c3:        sub     $0xfac,%esp
                   : 80484c9:        push    %edi
                   : 80484ca:        push    %esi
                   : 80484cb:        push    %ebx
                   :                   b = a;
                   : 80484cc:        lea     0xfffff060(%ebp),%edx
                   : 80484d2:        lea     0xfffff830(%ebp),%eax
                   : 80484d8:        mov      $0xf423f,%ebx
                   : 80484dd:        lea     0x0(%esi),%esi
                   :          return 0;
     3 0.03811% : 80484e0:        mov     %edx,%edi
                   : 80484e2:        mov      %eax,%esi
     1 0.0127%    : 80484e4:        cld
     8 0.1016%    : 80484e5:        mov      $0x1f4,%ecx
 7850 99.73%      : 80484ea:        repz movsl %ds:(%esi),%es:(%edi)
     9 0.1143%    : 80484ec:        dec     %ebx
                   : 80484ed:        jns     80484e0
                   : 80484ef:        xor     %eax,%eax
                   : 80484f1:        pop     %ebx
                   : 80484f2:        pop     %esi
                   : 80484f3:        pop     %edi
                   : 80484f4:        leave
                   : 80484f5:        ret
```

So here it's clear that copying is correctly credited with of all the samples, but the line number information is misplaced. **objdump -dS** exposes the same problem. Note that maintaining accurate debug information for compilers when optimizing is difficult, so this problem is not suprising. The problem of debug information accuracy is also dependent on the binutils version used; some BFD library versions contain a work-around for known problems of **gcc**, some others do not. This is unfortunate but we must live with that, since profiling is pointless when you disable optimisation (which would give better debugging entries).

5. Assembly functions

Often the assembler cannot generate debug information automatically. This means that you cannot get a source report unless you manually define the neccessary debug information; read your assembler documentation for how you might do that. The only debugging info needed currently by OProfile is the line-number/filename-VMA association. When profiling assembly without debugging info you can always get report for symbols, and optionally for VMA, through **opreport -l** or **opreport -d**, but this works only for symbols with the right attributes. For **gas** you can get this by

```
.globl foo
        .type   foo,@function
```

whilst for **nasm** you must use

```
          GLOBAL foo:function               ; [1]
```

Note that OProfile does not need the global attribute, only the function attribute.

6. Other discrepancies

Another cause of apparent problems is the hidden cost of instructions. A very common example is two memory reads: one from L1 cache and the other from memory: the second memory read is likely to have more samples. There are many other causes of hidden cost of instructions. A non-exhaustive list: mis-predicted branch, TLB cache miss, partial register stall, partial register dependencies, memory mismatch stall, re-executed µops. If you want to write programs at the assembly level, be sure to take a look at the Intel and AMD documentation at http://developer.intel.com/ and http://www.amd.com/products/cpg/athlon/techdocs/.

Chapter 6. Acknowledgments

Thanks to (in no particular order) : Arjan van de Ven, Rik van Riel, Juan Quintela, Philippe Elie, Phillipp Rumpf, Tigran Aivazian, Alex Brown, Alisdair Rawsthorne, Bob Montgomery, Ray Bryant, H.J. Lu, Jeff Esper, Will Cohen, Graydon Hoare, Cliff Woolley, Alex Tsariounov, Al Stone, Jason Yeh, Randolph Chung, Anton Blanchard, Richard Henderson, Andries Brouwer, Bryan Rittmeyer, Maynard P. Johnson, Richard Reich (rreich@rdrtech.com), Zwane Mwaikambo, Dave Jones, Charles Filtness; and finally Pulp, for "Intro".