OProfile for Tegra

Version 1.0

Contents

INTRODUCTION

OProfile is a system-wide application and kernel profiler for both Android and Linux systems. Its main advantage is its very small profiling overhead. OProfile uses statistical sampling method. It periodically pokes into your programs and figures out what code is currently being called. Just after that OProfile increments a specific counter for the corresponding function of your application. When your program is finished OProfile provides you with information on functions that were executed longer than others. It uses a simple rule here: a large relative value of the function-specific counter means that there were more samples for that function, and this in turn means that the function took more CPU time to execute than others. Such profiling information can help you quickly discover which routines to optimize for the purpose of getting better performance in your application. Internally, OProfile consists of three main components: a kernel driver, a daemon for collecting sample data (both are placed on the device), and post-profiling tools for turning data into information (they are placed on your host machine).

PREPARING HOST-SIDE TOOLS (UBUNTU 10.10)

1. The installation script attempts to fetch and install all required packages for Ubuntu 10.10 platforms; if you are using a different Ubuntu version (or a different Linux distribution) you must install the following packages manually:
   - autoconf
   - automake
   - g++
   - libpopt-dev
   - binutils-dev
   - git
   - unzip
   - graphviz
2. Extract the OProfile installer .zip archive in some temporary directory inside Ubuntu environment (*/tmp/oprofile-installer* for example):

```
mkdir /tmp/oprofile-installer
cd /tmp/oprofile-installer
cp <path-to-archive>/oprofile.zip .
unzip oprofile.zip
```

3. Run installer:

```
sh ./install-oprofile.sh
```

4. You will be presented with information on basic installation settings:

```
### OProfile Installation Path ###
/home/user/tegra-oprofile
### OProfile Temporary Path ###
/tmp/oprofile-tmp
### OProfile sources location ###
OProfile git repository: <currently used git repository>
```

   If you agree to the settings press any key to continue the installation procedure.

   The installation script will attempt to install all required packages automatically using 'apt-get' utility. As such, you may see requests from 'apt-get' to install additional required packages.

   Answer 'Y' to install all required packages.

If you want to change the recommended installation path close the current installation session (Ctrl-C) and define the following environment variables:

```
OPROFILE_INSTALL_PATH -> OProfile installation directory
OPROFILE_TEMP_PATH    -> OProfile temporary path
```

5. You will find your OProfile installation in *$OPROFILE_INSTALL_PATH*

PREPARING HOST-SIDE TOOLS (CYGWIN)

1. Install the following required packages via Cygwin Setup utility:
   - gcc4-core
   - gcc4-g++
   - automake
   - autoconf
   - binutils
   - libtool
   - libiconv
   - popt
   - python
   - wget
   - pkg-config
   - patchutils
   - make
   - unzip
   - git
2. If you want to get callgraphs (function call backtraces) together with profiling data you should also install the following packages:
   - imagemagick
   - ghostscript
   - graphviz (Cygwin Ports repository: http://sourceware.org/cygwinports/)
3. Extract the OProfile installer .zip archive into a temporary directory inside Cygwin environment (*/tmp/oprofile-installer* for example):

```
mkdir /tmp/oprofile-installer
cd /tmp/oprofile-installer
cp <path-to-archive>/oprofile.zip .
unzip oprofile.zip
```

4. Run the installer:

```
sh ./install-oprofile.sh
```

5. You will be presented with information on basic installation settings:

```
### OProfile Installation Path ###
/home/user/tegra-oprofile
### OProfile Temporary Path ###
/tmp/oprofile-tmp
### OProfile sources location ###
OProfile git repository: <currently used git repository>
```

   If you agree to the settings press any key to continue the installation procedure.

   If you want to change the recommended installation path close the current installation session (Ctrl-C) and define the following environment variables:

```
OPROFILE_INSTALL_PATH -> OProfile installation directory
OPROFILE_TEMP_PATH    -> OProfile temporary path
```

6. You will find your OProfile installation in *$OPROFILE_INSTALL_PATH*

PREPARING YOUR DEVICE

1. By default all Android images provided by NVIDIA contain everything needed for OProfile operation. If you obtained your image from the NVIDIA Developer Zone website or directly from our support team you may ignore this step and move directly to step 2.

   If you are building your own Android/Linux kernel please check that the following lines are present in your *.config*:

2.6.32 KERNEL (K32)	2.6.36 KERNEL (K36)
CONFIG_PROFILING=y CONFIG_OPROFILE=y CONFIG_HAVE_OPROFILE=y	CONFIG_PROFILING=y CONFIG_OPROFILE=y CONFIG_PERF_EVENTS=y CONFIG_HW_PERF_EVENTS=y

   You can test if your target's currently running kernel has these already set via:

```
adb pull /proc/config.gz config.gz &&
gunzip -c config.gz | grep '\(CONFIG_O*PROFIL\)\|\(_PERF_EVENTS\)'
```

Additionally you should apply the special kernel patch (K36-OProfile-BT.diff) to your Linux kernel tree. This patch fixes one Android-specific issue and allows you to get correct function call backtraces (callgraphs). You may apply this patch via:

```
cd ${KERNEL SOURCE TREE}
patch -p1 < ${PATH TO PATCH}/<patchname>
```

2. Push the kernel's *vmlinux* file (containing symbolic information required for kernel symbol resolution) from your build tree to your device:

```
$ adb push vmlinux /data/vmlinux
```

This file can be found either in a firmware package provided by NVIDIA or directly in your kernel build directory.

3. Finally, you need to get two address values from your kernel: *_text* and *_etext*.
This can be done via:

```
$ adb shell cat /proc/kallsyms | grep ' _e*text'
```

Store these values somewhere; they will be needed during samples collection.

PREPARING YOUR APPLICATION

If you plan to get per-symbol application performance statistics (e.g. telling you that function *func()* takes about 5% of total execution time) you should get the unstripped version of the Android application library. The term 'unstripped' refers to a binary that contains all symbolic and debug information initially generated by compiler.

In the Android NDK application building folder (that typically contains *libs*, *bin*, *gen*, *jni*, *obj* and *src* subfolders) the unstripped version of your library can be found here:

```
${APPLICATION_BUILDING_FOLDER}/obj/local/${EABI_TYPE}/<libname>.so
```

For example:

```
$ ls
AndroidManifest.xml  default.properties  libs              proguard.cfg
bin                  gen                 local.properties  res
build.xml            jni                 obj               src

$ cd ./obj/local/armeabi/ && ls

libgl2jni.so
```

You should use this unstripped library instead of your original library on the device. To push it to the device via adb:

```
$ adb push libgl2jni.so /data/data/com.android.gl2jni/lib/libgl2jni.so
```

Do not worry if you cannot push the unstripped library to your device. You will experience some slight complications later on (during the analyzing stage) but you will nevertheless get the profiling info required. Still, this is not recommended; you should push the unstripped library to the device if you can.

If you want to get application backtraces (callgraphs) you should build your application with the additional gcc flags: *-fno-omit-frame-pointer*, *-mno-thumb*, and *-O0* (optional). Removing optimization may help you to avoid some potential problems connected with inlining. This could be done by adding these flags to LOCAL_CFLAGS variable in *./jni/Android.mk* file:

```
LOCAL_CFLAGS := <ORIGINAL CONTENT> -fno-omit-frame-pointer –mno-thumb -O0
```

After making this change you will need to rebuild your application, repeat unstripped library preparation and push this library to you device via adb.

COLLECTING SAMPLES

A typical sample collection consists of the following main stages:

- Target application running
- OProfile initialization, customization (which information to collect) and activation
- Typical user experience reproduction (see 'What to do during profiling session' section)
- OProfile deactivation, resultant data processing and profiling report preparation

You should decide whether you want to get profiling information for both kernel functions and your application's functions, or if profiling information for only your application's functions will be enough. In the latter case profiling is simpler. We recommend that you profile only your application if you do not have specific requirements to profile the Android (Linux) kernel.

Additionally you need to decide whether or not you want to collect function call backtraces (callgraphs). Callgraphs can be useful if you want to figure out which specific code path (not a single function) takes so much time to execute. However, callgraph collection requires additional OProfile tuning; for this reason we recommend you to enable it only if it is really needed. To get callgraphs you will need to make specific procedures (see the 'Prepare your device' section for details).

ANDROID 2.2 FROYO (KERNEL 2.6.32 – K32)

RUNNING YOUR APPLICATION

Please run your application.

OPROFILE INITIALIZATION

To initialize OProfile subsystem run the following command on you device (using 'adb shell'):

```
# opcontrol --init
```

APPLICATION PROFILING (RECOMMENDED) SAMPLES WITHOUT CALLGRAPH INFORMATION

Run the following commands on your device (using 'adb shell' environment for example):

```
# opcontrol --setup
# opcontrol --start
```

APPLICATION PROFILING SAMPLES WITH CALLGRAPH INFORMATION

Run the following commands on your device (using 'adb shell' environment for example):

```
# opcontrol --setup
# echo 4 > /dev/oprofile/backtrace_depth
# opcontrol --start
```

KERNEL + APPLICATION PROFILING
SAMPLES WITHOUT CALLGRAPH INFORMATION

Run the following command on your device (replace *${_text}* with the address of *_text* symbol and *${_etext}* with the address of *_etext* symbol – see the 'Prepare your device' section):

```
# opcontrol --setup --vmlinux=/data/vmlinux --kernel-range=${_text},${_etext}
# opcontrol --start
```

For example:

```
# opcontrol --setup --vmlinux=/data/vmlinux --kernel-range=0xC0031000,0xC0518000
# opcontrol --start
```

KERNEL + APPLICATION PROFILING
SAMPLES WITH CALLGRAPH INFORMATION

Run the following command on your device (replace *${_text}* with the address of *_text* symbol and *${_etext}* with the address of *_etext* symbol – see the 'Prepare your device' section):

```
# opcontrol --setup --vmlinux=/data/vmlinux --kernel-range=${_text},${_etext}
# echo 4 > /dev/oprofile/backtrace_depth
# opcontrol --start
```

For example:

```
# opcontrol --setup --vmlinux=/data/vmlinux --kernel-range=0xC0031000,0xC0518000
# echo 4 > /dev/oprofile/backtrace_depth
# opcontrol --start
```

USER EXPERIENCE REPRODUCTION

Please take into account that you should reproduce typical user experience as fully as possible as this will help OProfile to provide you with the most relevant profiling information. Run the first level of your game, start data processing that are typical for your application, and so on.

FINISHING

When you have collected enough samples and want to stop OProfile run the following command on your device (using 'adb shell' environment for example):

```
# opcontrol --stop
```

ANDROID 2.3 GINGERBREAD
ANDROID 3.0 HONEYCOMB
(KERNEL 2.6.36 – K36)

RUNNING YOUR APPLICATION

Please run your application.

OPROFILE INITIALIZATION

OProfile that comes with Android 2.3 and 3.0 doesn't require any additional initialization procedures (like '--init' for OProfile for Android 2.2).

APPLICATION PROFILING (RECOMMENDED)
SAMPLES WITHOUT CALLGRAPH INFORMATION

Run the following command on your device (replace *${_text}* with the address of *_text* symbol and *${_etext}* with the address of *_etext* symbol – see the 'Prepare your device' section):

```
# opcontrol --setup --start --event=CPU_CYCLES:5000
```

APPLICATION PROFILING
SAMPLES WITH CALLGRAPH INFORMATION

Run the following command on your device (replace *${_text}* with the address of *_text* symbol and *${_etext}* with the address of *_etext* symbol – see the 'Prepare your device' section):

```
# opcontrol --setup --callgraph=4 --start --event=CPU_CYCLES:5000
```

KERNEL + APPLICATION PROFILING
SAMPLES WITHOUT CALLGRAPH INFORMATION

Run the following command on your device (replace *${_text}* with the address of *_text* symbol and *${_etext}* with the address of *_etext* symbol – see the 'Prepare your device' section):

```
# opcontrol --setup --start --vmlinux=/data/vmlinux --kernel-range=${_text},${_etext}
--event=CPU_CYCLES:5000
```

For example:

```
# opcontrol --setup --start --vmlinux=/data/vmlinux
--kernel-range=0xC0031000,0xC0518000 --event=CPU_CYCLES:5000
```

KERNEL + APPLICATION PROFILING
SAMPLES WITH CALLGRAPH INFORMATION

Run the following command on your device (replace *${_text}* with the address of *_text* symbol and *${_etext}* with the address of *_etext* symbol – see the 'Prepare your device' section):

```
# opcontrol --setup --callgraph=4 --start --vmlinux=/data/vmlinux
--kernel-range=${_text},${_etext} --event=CPU_CYCLES:20000
```

For example:

```
# opcontrol --setup --callgraph=4 --start --vmlinux=/data/vmlinux
--kernel-range=0xC0031000,0xC0518000 --event=CPU_CYCLES:20000
```

USER EXPERIENCE REPRODUCTION

Please take into account that you should reproduce typical user experience as fully as possible as this will help OProfile to provide you with the most relevant profiling information. Run the first level of your game, start data processing that are typical for your application, and so on.

FINISHING

When you have collected enough samples and want to stop sampling run the command:

```
# opcontrol --stop
```

TIPS & TRICKS

DISABLE DVFS

We recommend that you to disable the DVFS (Dynamic Voltage and Frequency Scaling) subsystem during samples collection. Only with this subsystem disabled will you get correct samples distribution over time (since the sampling is based on CPU_CYCLES having DVFS enabled may cause the number of cycles-per-second to change dynamically):

```
All Tegra boards:

echo performance > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
echo performance > /sys/devices/system/cpu/cpu1/cpufreq/scaling_governor
echo 1 > /sys/devices/system/cpu/cpu1/online

Additional settings only for Tegra3:

echo performance > /sys/devices/system/cpu/cpu2/cpufreq/scaling_governor
echo performance > /sys/devices/system/cpu/cpu3/cpufreq/scaling_governor
echo 1 > /sys/devices/system/cpu/cpu2/online
echo 1 > /sys/devices/system/cpu/cpu3/online
```

NEW OPCONTROL UTILITY

With the official OProfile distribution the 'opcontrol' utility is implemented as a shell script. However, Google did not want to install a full-feature shell in the Android OS, nor did they want to install all the tools that shell-scripts rely upon. Google's answer to this dilemma was to craft their own 'opcontrol' as a compiled program, written in C.

For various reasons the use and command-line syntax of this replacement 'opcontrol' differs from that of the official Oprofile distribution. Due to this fact you should review your OProfile experience a little to update it with new commands and tricks.

ANALYZING THE SAMPLE DATA

Once you have collected the samples on the target device it is time to process them on your host machine. Make sure that your device is connected via USB and switched on.

First, switch to your host-side tools installation directory *$OPROFILE_INSTALL_PATH*. By default this path looks like the following:

```
/home/${USER}/tegra-oprofile
```

Next, figure out which kind of data analysis you want to perform. The types of analysis available depend on the sample collection method you performed (see the 'Collecting Samples' section for details). The following table shows the types of analysis available for the different sample collection methods:

	Application profiling		Kernel + Application profiling	
	Samples without callgraph information	Samples with callgraph information	Samples without callgraph information	Samples with callgraph information
System-wide analysis	✔	✔	✔	✔
Application analysis	✔	✔	✔	✔
Application analysis with callgraphs		✔		✔
Kernel analysis			✔	✔
Kernel analysis with callgraphs				✔

Note that the recommended type of samples collection (Application profiling without callgraph information) allows you to perform only System-wide analysis and Application analysis. The good news is that these two analyses are all you need in most cases.

To help you pick the analysis type(s) you wish to perform see the following detailed descriptions of all available types.

SYSTEM-WIDE ANALYSIS

System-wide analysis provides you with general information about the amount of time that any system-object (kernel, executable or library) takes for executions. You will not get any per-symbol resolution, just a general statistics. This information can be useful for an initial analysis determine which components need to be optimized.

To perform system-wide analysis run *oprofile-analyze.py* script the following way:

```
$ ./oprofile-analyze.py --type=system
```

Here is example of the system-wide analysis report:

```
$ ./oprofile-analyze.py --type=system

CPU: ARM Cortex-A9, speed 0 MHz (estimated)
Counted CPU_CYCLES events (Number of CPU cycles) with a unit mask of 0x00 (No unit
mask) count 5000
  CPU_CYCLES:5000|
  samples|      %|
------------------
12407225 72.1365 vmlinux
  1508662  8.7715 libnvmm_video.so
  1235442  7.1829 oprofiled
   417985  2.4302 libskia.so
   391396  2.2756 libc.so
   292896  1.7029 libdvm.so
   235906  1.3716 bcm4329
   117603  0.6838 libstagefright.so
   115642  0.6724 libwebcore.so
    65499  0.3808 dalvik-jit-code-cache (deleted)
    63783  0.3708 libGLESv2_tegra.so
    47998  0.2791 libcutils.so
    43870  0.2551 libhwui.so
    38656  0.2247 libnvos.so
<...>
```

APPLICATION ANALYSIS

Application analysis provides you with per-symbol statistics of the user application's procedures. Most Android applications consist of a small Java wrapper and a big native library that performs the actual work. This is why you should make OProfile library analysis to get performance information for your application. Also OProfile library analysis could be used to profile some general system libraries (like *libskia.so* for example).

In the 'Prepare Your Application' section you were asked to decide whether to place an unstripped version of your application's library to your device or not. The way to perform this analysis depends on whether or not you placed the unstripped library on your target device.

UNSTRIPPED LIBRARY IS ALREADY PLACED ON THE DEVICE

If you used the unstripped library with symbolic information on the device then use the following command to perform application analysis:

```
./oprofile-analyze.py --type=application <path-to-library-on-your-device>
```

For example:

```
./oprofile-analyze.py --type=application /data/data/com.nv.nv/lib/libnv.so
```

UNSTRIPPED LIBRARY IS AVAILABLE ONLY ON THE HOST MACHINE

If you used the stripped version of the library on the device and the unstripped version of the library is only on the host machine then you first need to place the unstripped binary in a special folder inside the OProfile host installation directory. For example:

```
mkdir -p ./root/data/data/com.nv.nv/lib/
cp libnv.so ./root/data/data/com.nv.nv/lib/libnv.so
```

The folder structure under the *root* directory should mimic the filesystem on the device.

With this in place you must tell OProfile not to pull the library from the device (since it is the stripped version of the library present there) and to prepare the analysis report:

```
./oprofile-analyze.py --not-pull --type=application /data/data/com.nv.nv/lib/libnv.so
```

You should do the same operations for your own library. Do not forget that you should run these commands directly from OProfile installation folder *$OPROFILE_INSTALL_PATH*.

ANALYSIS REPORT

Application analysis provides you with a report like the following:

```
<...>
samples  %        symbol name
339141   22.6908  H264CABACDecodeArithmeticCodeBin_c
136450    9.1294  ProcessResidual_HP
92084     6.1610  DecodeSliceData_HP
76308     5.1055  FindNeighboringMB
42582     2.8490  H264CABACDecodeSignificantMap_c
40795     2.7295  Idct8x8_ColumnLoop
40067     2.6808  SetMotionVectorPredictor
37725     2.5241  PrepareRefPicIdList
33192     2.2208  DecodeMacroBlockLayer_HP
32795     2.1942  MacroBlockPrediction_HP
32692     2.1873  DecodeResidual_HP
```

APPLICATION ANALYSIS WITH CALLGRAPHS

Application analysis with callgraphs provides you with per-symbol statistics of user application's procedures with additional data on function callgraphs. This can be helpful to find out not only the functions that take a lot of time to execute but the whole code path that lead to these functions. This approach is also useful to combine the whole number of functions into 2-3 independent subsystems (game logic, audio and rendering for example) and to estimate total performance of every subsystem independently.

Your application's JNI library should be compiled by *gcc* with *-fno-omit-frame-pointer* and *-mno-thumb* flags for this type of analysis (see the 'Prepare Your Application' section for details) and placed on the device.

In the 'Prepare Your Application' section you were asked to decide whether to place an unstripped version of your application's library to your device or not. The way to perform this analysis depends on whether or not you placed the unstripped library on your target device.

UNSTRIPPED LIBRARY IS ALREADY PLACED ON THE DEVICE

If you used the unstripped library with symbolic information on the device then use the following command to perform the analysis:

```
./oprofile-analyze.py --type=application-backtrace <path-to-library-on-your-device>
```

For example:

```
./oprofile-analyze.py --type=application-backtrace data/data/com.nv.nv/lib/libnv.so
```

UNSTRIPPED LIBRARY IS AVAILABLE ONLY ON THE HOST MACHINE

If you used the stripped version of the library on the device (but compiled with the *-fno-omit-frame-pointer* and *-mno-thumb* flags) and the unstripped version of the library is only on the host machine then you first need to place the unstripped binary in a special folder inside the OProfile host installation directory. For example:

```
mkdir -p ./root/data/data/com.nv.nv/lib/
cp libnv.so ./root/data/data/com.nv.nv/lib/libnv.so
```

The folder structure under the *root* directory should mimic the filesystem on the device.

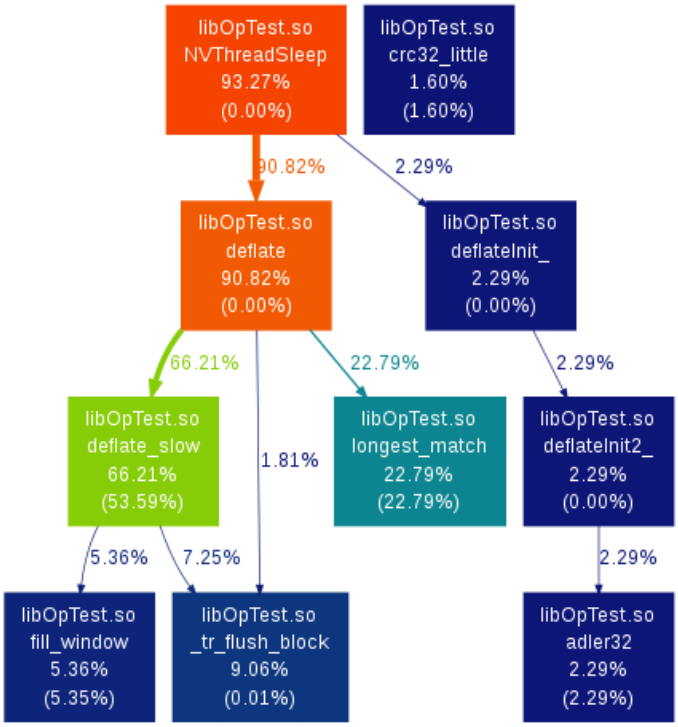
With this in place you must tell OProfile not to pull the library from the device (since it is the stripped version of the library present there) and to prepare the analysis report:

```
./oprofile-analyze.py --not-pull --type=application-backtrace /data/data/com.nv.nv/lib/libnv.so
```

You should do the same operations for your own library. Don't forget that you should run these commands directly from OProfile installation folder *$OPROFILE_INSTALL_PATH*.

ANALYSIS REPORT

Application analysis with callgraphs provides you with a graph represented by an image:



KERNEL ANALYSIS

A kernel analysis provides you with per-symbol statistics of the kernel. With this you can see where (inside the Linux kernel) your application spends a lot of time.

To perform kernel analysis with callgraphs run oprofile-analyze.py script the following way:

```
$ ./oprofile-analyze.py --type=kernel
```

Kernel analysis reports look like this:

```
samples  %        symbol name
946      58.3590  tegra_idle_enter_lp3
80        4.9352  schedule
61        3.7631  ring_buffer_consume
18        1.1104  cpu_idle
15        0.9254  cpuidle_idle_call
15        0.9254  do_select
15        0.9254  tick_nohz_stop_sched_tick
15        0.9254  unix_poll
14        0.8637  fget_light
10        0.6169  unmap_vmas
9         0.5552  do_vfp
9         0.5552  fsl_ep_queue
8         0.4935  fput
8         0.4935  free_poll_entry
8         0.4935  try_to_wake_up
7         0.4318  __do_softirq
7         0.4318  loop
6         0.3701  cpufreq_interactive_idle
6         0.3701  i2c_writel
6         0.3701  op_cpu_buffer_read_entry
6         0.3701  run_timer_softirq
6         0.3701  schedule_timeout
6         0.3701  sock_def_readable
6         0.3701  sync_buffer
```

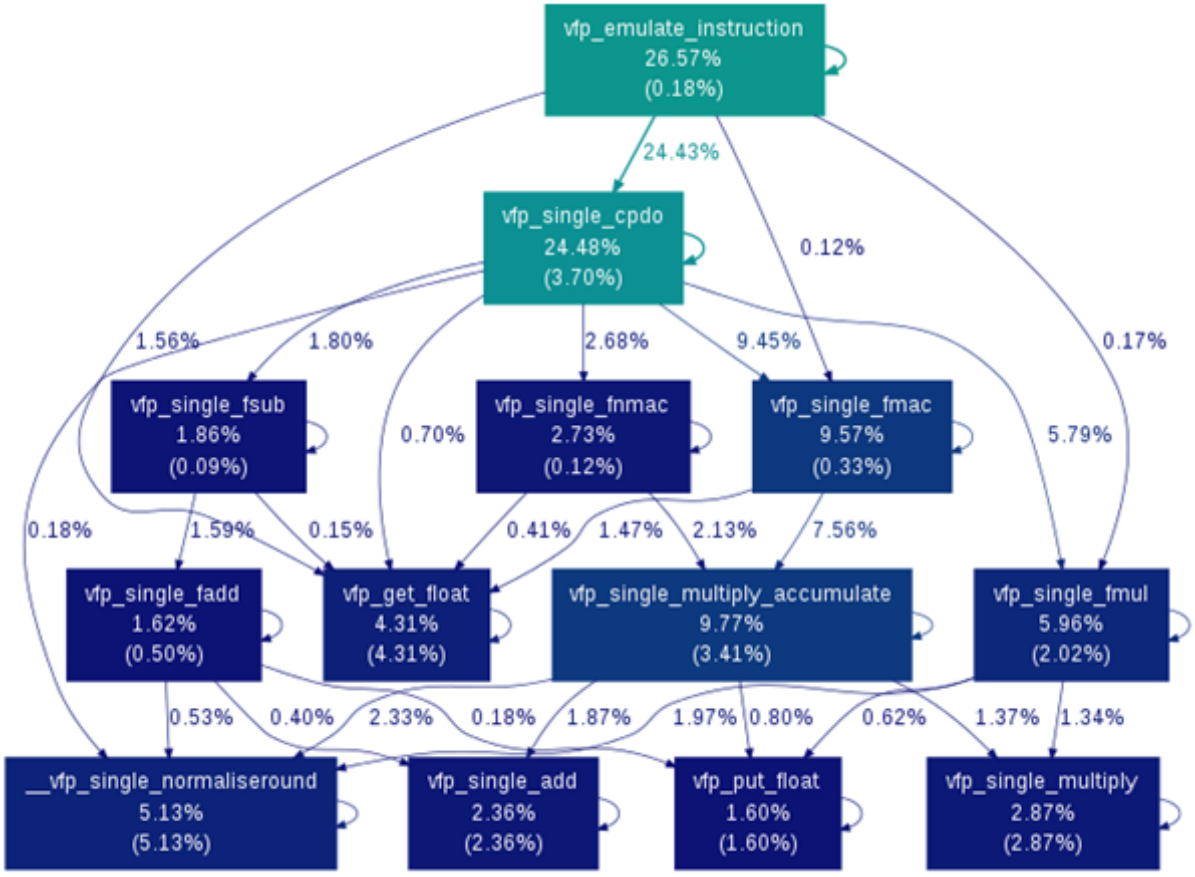
KERNEL ANALYSIS WITH CALLGRAPHS

Kernel analysis with callgraphs provides you with per-symbol statistics of kernel procedures with additional data on function callgraphs. This can be helpful to find out not only the functions that take a lot of time to execute, but the whole code paths associated with these functions. You can see where (inside the Linux kernel) your application spends a lot of time.

To perform kernel analysis with callgraphs run oprofile-analyze.py script the following way:

```
$ ./oprofile-analyze.py --type=kernel-backtrace
```

As a result of kernel analysis with callgraphs you will get a graph represented by an image:



TIPS & TRICKS

HOW TO READ A CALLGRAPH

Let's look at the following callgraph:

What do these numbers, blocks and arrows mean?

Every block represents one function (routine) in your application. Inside every block we see a function name and two numbers. The first number indicates the percentage of the total time the CPU executed some code inside this routine. (For example, *24.48%* for *vfp_single_cpdo*). This value also includes the time for all subroutines that were called. The second number is the time of this function only, excluding all subroutines calls. (For example, *3.70%* for *vfp_single_cpdo*).

Every arrow represents a function call. The function call hierarchy may omit some functions due to statistical nature of the OProfile. When this occurs it simply means that these functions are very lightweight, so much so that OProfile didn't collect any samples for them during the profiling session. The values near the arrow (for example, *2.68%* for the call to *vfp_single_fnmac* routine) indicates how much of the total execution time of the caller procedure this code path represents. To find the hottest code path in your application you should find the function block with the biggest total execution time and follow arrows with the biggest value (weight).

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com