PERF for Tegra

Version 1.0

Contents

INTRODUCTION

PERF

PERF tool is a part of the Linux kernel project. It was developed as a front-end to the specific Performance Events subsystem of the Linux kernel. This subsystem was designed to provide performance analysts with a way to collect required performance-related statistics with a very small profiling overhead.

PERF uses statistical sampling method for profiling. It periodically pokes into your programs and figures out what code is currently being called. Just after that PERF increments a specific counter for the corresponding function of your application. When your program is finished PERF 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.

Keep in mind that another profiling tool OProfile also uses Performance Events subsystem as a back-end. It means that the only difference between OProfile and PERF is a front-end i.e. how performance analyst interacts with performance analysis tools.

OProfile is a host-target system that requires collected data transferring from device to your host machine for a performance report preparation. This makes profiling tricky and longer.

PERF instead is a small tool that could be executed directly on your device. It does all required operations (data collection, data processing and final report preparation) on the device. It gives you some benefits. First of all it is simpler and faster. Second, PERF tool can automatically get some additional runtime information from the kernel (e.g. kernel symbols addresses via */proc/kallsyms*) that removes a requirement to pass additional data to the profiler by the analyst (for example OProfile required to pass *vmlinux* file to get kernel symbols addresse). And the third benefit is that you can't get any problems with host-side components (they potentially could conflict with your host side machine or operating system) because there are no any host-side components.

We recommend you to give PERF a try. You will not be disappointed.

PREPARING YOUR DEVICE

By default all Android images provided by NVIDIA contain everything needed for PERF operation. If you obtained your image from the NVIDIA Developer Zone website or directly from our support team you may ignore this section.

Keep in mind that only 2.6.36 (K36) kernel is supported. PERF is a pretty modern tool and requires some kernel components that are unavailable in 2.6.32 (K32) kernel. If you're using 2.6.32 (K32) kernel you could try to use OProfile instead.

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

2.6.36 KERNEL (K36)

```
CONFIG_PROFILING=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 two special kernel patches (K36-PERF-BT.patch and K36-PERF-L2) to your Linux kernel tree. First patch fixes one Android-specific issue and allows you to get correct function call backtraces (callgraphs). Second one allows you to collect L2 cache statistics that could be very useful to find out where your program generates a lot of cache misses.

You may apply these patches via the following commands:

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

After making the following changes please recompile your Android/Linux kernel and make sure that you successfully update your device firmware with a new kernel.

Now your device is ready to run PERF profiling tool. Be informed that required kernel configuration options and applied kernel patches do not make any effect on device performance while you are not collecting any profiling data.

PREPARING YOUR APPLICATION

PERF tries to provide you with per-symbol application performance statistics (e.g. telling you that function *func()* takes about 5% of total execution time). To do this PERF needs to get symbolic information from application's executable or library. Symbolic information makes association between addresses inside a target binary and a correct function names from it.

If we are speaking about Android operating system the most commonly used case is when you have a small Java wrapper and a JNI native library which carries out the most part of work. In this case library already contains some symbolic information needed to dynamically link your application. It means that information about symbolic names of all functions *exported* by the library is already in place. In many cases it is enough to have only such information to get all addresses resolved in the final profiling report.

If you see any unresolved addresses in your profiling report you need to carry out some additional procedures to add symbolic information to your library under profiling. Android NDK removes the most part of symbolic information (which is not required for dynamic linking) from libraries just after compilation. This process is called 'stripping'. If you want to save all symbolic information you should get the unstripped version of the Android application library. The term 'unstripped' refers to a binary that contains all symbolic 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
```

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.

PERF TOOL INSTALLATION

At this step you should obtain *perf* binary (you can find it in the PERF package together with this manual), and then copy it to some writable and executable place on the target device. For Android, this is best done with:

```
adb push <path to perf>/perf /data/perf
```

Now, please add executable permission to the PERF binary. This could be done with a *chmod* command of the Android shell:

```
$ adb shell
# chmod 777 /data/perf
```

After this step you're ready to start data collection.

PROFILING

A typical sample collection consists of the following main stages:

- Target application running
- PERF tool customization (which information to collect) and activation
- Typical user experience reproduction in the target application
- PERF tool deactivation

PERF tool can do both *system-wide* and *application-specific performance analysis*. *System-wide analysis* provides you with a report that includes information about all software components

that were running during samples collection. This is a good feature if you have no idea why performance of your device is not okay. But if you want to concentrate on application performance you may find this information redundant because you're interested only in performance statistics for application's procedures. In this case you should run *application-specific analysis*. Good news is that application-specific analysis also makes much smaller influence on the complete system performance than system-wide analysis. So our recommendation is to run *application-specific analysis* in the most part of cases.

Keep in mind that both system-wide analysis and application-specific analysis include kernel-related samples together with user-space application's samples. The only difference is that application-specific analysis includes kernel samples only for a kernel code that was executed in the context of your application. System-wide analysis includes all kernel samples.

You also need to decide whether or not you want to collect 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. 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. Callgraph collection generates additional workload to your device. We recommend you to enable it only if it is really needed.

APPLICATION-SPECIFIC ANALYSIS (RECOMMENDED) SAMPLES WITHOUT CALLGRAPH INFORMATION

First of all you need to run the application that you want to profile. This could be done both using Android graphical user interface and directly from Android shell. Just after starting your application you should get its process ID (PID). This could be done using 'ps' shell command:

```
$ adb shell
# ps
<…>
root      576   2     0      0     c0098ca4 00000000 S kworker/2:1
app_28    577   97    446232 31860 ffffffff 80709888 R com.nvidia.devtech.OpTest
root      592   2     0      0     c00c1904 00000000 S migration/1
<…>
```

Here you can see that process ID of the application 'com.nvidia.devtech.OpTest' equals to 577. Now you could use this process ID to start application-specific analysis using PERF tool. Process ID will be used to identify application to profile:

```
# cd /data/
# ./perf record –p <pid>
```

Now PERF tool is collecting profiling samples. It is time to reproduce typical user experience. Please take into account that you should reproduce typical user experience as fully as possible as this will help PERF 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.

You could stop samples collection using one of the following approaches:

- In Windows please open additional adb shell and run the following command:

```
$ adb shell
# kill -2 <perf PID>
```

  `<perf PID>` - is a process ID of the PERF profiling tool process (do not confuse it with your application's process ID). PERF process ID could also be found using 'ps' utility.
- In Linux or MacOS you could simply press 'Ctrl-C' key combination in the adb shell where you have started PERF before. Or you could open additional adb shell and do the same procedure than in Windows.

Just after PERF deactivation you will get the following output (keep in mind that number of wake ups, *perf.data* file size and the number of samples could be different in your case):

```
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.200 MB perf.data (~8731 samples) ]
```

When samples collection was finished you are ready to start performance report preparation. This could be done by the following way:

```
# export PERF_PAGER=cat
# ./perf report --sort dso,symbol
```

Environment variable *PERF_PAGER* controls which application will be used to show profiling results. In our case we want to get profiling report as-is without any post-processing. That is why we are using *cat* utility that simply prints profiling report to the Android shell. You could try to develop your own post-processing tools that will automate profiling report processing.

Additional *--sort* parameter defines which information we would like to see in final report. Additional information about this option could be found in the PERF tool help subsystem:

```
# export PERF_PAGER=cat
# ./perf report help
```

Application-specific analysis report looks as follows

```
# export PERF_PAGER=cat
# ./perf report –sort dso,symbol

# Overhead  Shared Object       Symbol
# ........  ..................  ................................
#
    47.56%  libOpTest.so        [.] deflate_slow
    20.66%  libOpTest.so        [.] longest_match
     4.74%  libOpTest.so        [.] fill_window
     4.38%  libOpTest.so        [.] adler32
     4.17%  libc.so             [.] __dorand48
     4.05%  libOpTest.so        [.] copy_block
     4.02%  libOpTest.so        [.] pqdownheap
     2.94%  libOpTest.so        [.] randomBuffer(unsigned char*, int)
     1.71%  libc.so             [.] lrand48
     1.38%  libOpTest.so        [.] crc32_little
     0.58%  libOpTest.so        [.] build_tree
     0.53%  libOpTest.so        [.] gen_bitlen
     0.50%  [kernel.kallsyms]   [k] __memzero
     0.38%  libOpTest.so        [.] rand
     0.36%  libc.so             [.] memcpy
     0.17%  libOpTest.so        [.] bi_reverse
     0.14%  libOpTest.so        [.] scan_tree
     0.08%  libOpTest.so        [.] gen_codes
     0.08%  [kernel.kallsyms]   [k] handle_mm_fault
     0.05%  [kernel.kallsyms]   [k] loop
     0.05%  [kernel.kallsyms]   [k] do_page_fault
     0.05%  [kernel.kallsyms]   [k] _raw_spin_unlock_irqrestore
     0.05%  [kernel.kallsyms]   [k] get_page_from_freelist
     0.03%  [kernel.kallsyms]   [k] _raw_spin_unlock
     0.03%  [kernel.kallsyms]   [k] __dabt_usr
     0.02%  [kernel.kallsyms]   [k] bad_range
     0.02%  [kernel.kallsyms]   [k] get_unmapped_area
     0.02%  libOpTest.so        [.] init_block
     0.02%  libGLESv2_tegra.so  [.] glClear
     0.02%  [kernel.kallsyms]   [k] perf_event_mmap_ctx
```

In the profiling report you will see the following information fields:

- *Symbol* – procedure name, where procedures of the target application are marked with a *[.]* flag and kernel procedures are marked with a *[k]* flag
- *Shared Object* – name of the binary that includes the function, it could be executable, shared library or the Android/Linux kernel
- *Overhead* – which percentage of time the procedure was running on the CPU (key value for performance analysis)

In our case *deflate_slow* procedure took about 50% of all execution time. It means that this procedure is a good candidate for optimization. We could say the same in case of *longest_match* procedure too.

APPLICATION-SPECIFIC ANALYSIS
SAMPLES WITH CALLGRAPH INFORMATION

First of all you need to run the application that you want to profile. This could be done both using Android graphical user interface and directly from Android shell. Just after starting your application you should get its process ID (PID). This could be done using 'ps' shell command:

Now you could use this process ID to start application-specific analysis using PERF tool. Process ID will be used to identify application to profile:

```
# cd /data/
# ./perf record –g –p <pid>
```

You could see additional *-g* flag that was passed to the PERF tool. This flag enables specific data collection required to create function backtraces (callgraphs).

Now PERF tool is collecting profiling samples. It is time to reproduce typical user experience. Please take into account that you should reproduce typical user experience as fully as possible as this will help PERF 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.

You could stop samples collection using one of the following approaches:

- In Windows please open additional adb shell and run the following command:

  ```
  $ adb shell
  # kill -2 <perf PID>
  ```

  `<perf PID>` - is a process ID of the PERF profiling tool process (do not confuse it with your application's process ID). PERF process ID could also be found using 'ps' utility.
- In Linux or MacOS you could simply press 'Ctrl-C' key combination in the adb shell where you have started PERF before. Or you could open additional adb shell and do the same procedure than in Windows.

Just after PERF deactivation you will get the following output (keep in mind that number of wake ups, *perf.data* file size and the number of samples could be different in your case):

```
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.200 MB perf.data (~8731 samples) ]
```

When samples collection was finished you are ready to start performance report preparation. This could be done by the following way:

```
# export PERF_PAGER=cat
# ./perf report --sort dso,symbol
```

Environment variable *PERF_PAGER* controls which application will be used to show profiling results. In our case we want to get profiling report as-is without any post-processing. That is why we are using *cat* utility that simply prints profiling report to the Android shell. You could try to develop your own post-processing tools that will automate profiling report processing.

Additional *--sort* parameter defines which information we would like to see in final report. Additional information about this option could be found in the PERF tool help subsystem:

```
# export PERF_PAGER=cat
# ./perf report help
```

Application-specific analysis report with callgraphs information looks as follows:

```
# export PERF_PAGER=cat
# ./perf report --sort dso,symbol
# Events: 4K cycles
#
# Overhead  Shared Object        Symbol
# ........  .................  ....................................
#
    47.47%  libOpTest.so         [.] deflate_slow
            |
            --- deflate
                compress2
                render(_JNIEnv*, _jobject*, float, int, int, unsigned char)
                dvmPlatformInvoke


    19.28%  libOpTest.so         [.] longest_match
            |
            --- deflate
                compress2
                render(_JNIEnv*, _jobject*, float, int, int, unsigned char)
                dvmPlatformInvoke

                                        <…>

     0.46%  libOpTest.so         [.] rand
            |
            --- randomBuffer(unsigned char*, int)
                render(_JNIEnv*, _jobject*, float, int, int, unsigned char)
                dvmPlatformInvoke
```

```
    0.09%  [kernel.kallsyms]    [k] arch_get_unmapped_area
           |
           --- arch_get_unmapped_area
               get_unmapped_area
               do_mmap_pgoff
               sys_mmap_pgoff
               ret_fast_syscall
               render(_JNIEnv*, _jobject*, float, int, int, unsigned char)
               dvmPlatformInvoke
                                          <…>
```

In the profiling report you will see the following information fields:

- *Symbol* – procedure name, where procedures of the target application are marked with a *[.]* flag and kernel procedures are marked with a *[k]* flag
- *Shared Object* – name of the binary that includes the function, it could be executable, shared library or the Android/Linux kernel
- *Overhead* – which percentage of time the procedure was running on the CPU (key value for the performance analysis)

Also in the profiling report you could find function callgraphs. It is pretty straightforward to understand how function callgraphs information is organized in PERF profiling report. Let's look at the *deflate_slow* procedure. According to the profiling report this procedure was called the following way:

```
dvmPlatformInvoke -> render -> heavyTest -> compress2 -> deflate -> deflate_slow
```

Feel free to read callgraph for other procedures using the same methodology.

In our case *deflate_slow* procedure took about 50% of all execution time. It means that this procedure is a good candidate for optimization. We could say the same in case of *longest_match* procedure too.

SYSTEM-WIDE ANALYSIS

There is no need in passing specific application process ID during system-wide performance analysis. You should run PERF tool the following way:

```
# cd /data/
# ./perf record –a
```

We recommend you to collect samples at least 2-3 minutes to get correct performance statistics in the profiling report.

You could stop samples collection using one of the following approaches:

- In Windows please open additional adb shell and run the following command:

```
$ adb shell
# kill -2 <perf PID>
```

  `<perf PID>` - is a process ID of the PERF profiling tool process (do not confuse it with your application's process ID). PERF process ID could also be found using 'ps' utility.
- In Linux or MacOS you could simply press 'Ctrl-C' key combination in the adb shell where you have started PERF before. Or you could open additional adb shell and do the same procedure than in Windows.

Just after PERF deactivation you will get the following output (keep in mind that number of wake ups, *perf.data* file size and the number of samples could be different in your case):

```
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.200 MB perf.data (~8731 samples) ]
```

When samples collection was finished you are ready to start performance report preparation. This could be done by the following way:

```
# export PERF_PAGER=cat
# ./perf report --sort dso,symbol
```

Environment variable *PERF_PAGER* controls which application will be used to show profiling results. In our case we want to get profiling report as-is without any post-processing. That is why we are using *cat* utility that simply prints profiling report to the Android shell. You could try to develop your own post-processing tools that will automate profiling report processing.

Additional *--sort* parameter defines which information we would like to see in final report. Additional information about this option could be found in the PERF tool help subsystem:

```
# export PERF_PAGER=cat
# ./perf report help
```

System-wide analysis report looks as follows:

```
# export PERF_PAGER=cat
# ./perf report –sort dso,symbol

# Overhead  Shared Object       Symbol
# ........  ..................  ................................
#
    45.12%  libOpTest.so        [.] deflate_slow
    21.73%  libOpTest.so        [.] longest_match
    13.48%  libdvm.so           [.] dvmAsmInstructionStart
     4.74%  libOpTest.so        [.] fill_window
     4.40%  libdvm.so           [.] dvmJitToInterpNoChain
     4.20%  libOpTest.so        [.] adler32
     4.07%  libc.so             [.] __dorand48
     4.09%  libOpTest.so        [.] copy_block
     3.58%  libOpTest.so        [.] pqdownheap
     2.85%  libOpTest.so        [.] randomBuffer(unsigned char*, int)
     2.31%  libbinder.so        [.] android::Parcel::scanForFds() const
     1.56%  libc.so             [.] lrand48
     1.43%  libskia.so          [.] SkBlitRow::Color32()
     1.27%  libOpTest.so        [.] crc32_little
     0.83%  libnvrm_graphics.so [.] NvRmChannelSubmit
     0.55%  libOpTest.so        [.] build_tree
     0.52%  libOpTest.so        [.] gen_bitlen
     0.50%  [kernel.kallsyms]   [k] __memzero
     0.36%  libOpTest.so        [.] rand
     0.31%  libc.so             [.] memcpy
```

In the profiling report you will see the following information fields:

- *Symbol* – procedure name, where procedures of the target application are marked with a *[.]* flag and kernel procedures are marked with a *[k]* flag
- *Shared Object* – name of the binary that includes the function, it could be executable, shared library or the Android/Linux kernel
- *Overhead* – which percentage of time the procedure was running on the CPU (key value for performance analysis)

You could see that by making system-wide analysis we got some additional procedures in the profiling report. We got samples that came from other subsystems of the Android operating system. Such external procedures came from *libbinder.so*, *libskia.so*, *libnvrm_graphics.so* and *libdvm.so*. Such kind of analysis could be used to get system performance overview.

Our test application (that uses libOpTest.so library) provides a lot of workload to the CPU that is why we have its procedures in the top even in case of the system-wide analysis.

L2 CACHE EVENTS ANALYSIS

System cache significantly decreases time needed to access data stored in the main memory. It does it by storing frequently used data in rapid access memory.

System **c**ache could accelerate both data reading (that looks reasonable) and data writing (that is a little bit less obvious).

DATA READING

In case of reading, data stored in cache could be accessed much faster than data in the main memory.

If CPU tries to access data that was already placed in cache before it gets data much faster.

If there is no required data portion in cache so-called *cache read miss* happens. In this case cache tries to fetch required data from the main memory (that usually takes a lot of time) and CPU is waiting for it (CPU could try to execute following independent instructions though).

Application (i.e. application developer) should try to optimize execution by the way that could help cache controller to keep all required data directly into cache (by prefetching i.e. executing some smart algorithms that could predict which data will be accesses by CPU next time and making memory access before actual request from CPU which can save some time).

Also you should keep in mind that cache line size (fixed data portion that cache could load from memory at once) is much bigger than typical machine word (for Cortex-A9 cache line size equals to 32 bytes). That is why when you read e.g. one *int* variable from the main memory cache will automatically load 7 next *int*'s into cache. You should use this feature as much as possible.

Also keep in mind that instruction that should be executed by CPU should also be loaded from the main memory. ARM uses different caches for instruction fetching and for data fetching. It means that we could divide all *cache read misses* onto two big classes: misses that were produced during data fetching (*data cache read misses*) and misses that were produced during instruction fetching (*instruction cache read misses*).

In many cases *data cache read misses* is much more important to know because you could react on them by simply re-organizing your application's memory access patterns.

In case of instruction cache read miss you could do much less. It is especially difficult if you're using high level languages and generate machine instruction using any compiler. You could try to use different compiler flags (e.g. enable optimization for smaller size) to decrease *instruction*

cache misses. Anyway compiler tries to do its best to decrease number of *instruction cache read misses*.

DATA WRITING

In case of writing, data that should be written could be placed into cache instead of the main memory. This could give us two main benefits: cache controller could try to optimize memory access (write bigger data block at once and so on) and in case of data over-writing only last value will be moved to the main memory (which can significantly decrease number of memory accesses).

When CPU writes some data into memory cache controller tries to find out any slot inside cache memory that already corresponds to the same memory address as requested by CPU. If it is possible to find one cache controller simply overwrite current data value by the new one. If there is no corresponding slot in the cache memory cache controller has to allocate a new slot and write data there or simply write data directly to the main memory. This situation is called *cache write miss*.

Cache write miss is not really critical to the application because in all modern CPUs data writing can be queued and the CPU can continue to execute next instructions. The only potential problem is the queue overflow but this situation is pretty rare.

CACHE HIERARCHY

Modern ARM CPUs include two levels of cache: first level cache (L1 cache) and second level cache (L2 cache). Significant difference between these two caches is that L1 cache block is per-CPU block (each CPU includes its own L1 cache) but L2 cache is global (even in case of 4 cores only one L2 cache block is used). Additional difference is that L1 cache is divided onto two hardware structures: data cache and instruction cache while L2 cache stores both data and instructions together.

Usually L1 cache is pretty small (about 16 KB for data + 16 KB for instructions) while L2 cache is much bigger (about 256 KB).

CPU tries to get required data from the L1 cache. If there is no required data there *L1 cache read miss* happens and CPU send the same request to the L2 cache controller. If there is no data in the L2 cache too *L2 cache read miss* happens and request is sent to the main memory.

It is really important to estimate time needed to access different levels of cache. Access to L1 cache requires about *5-10 ns*. Access to L2 cache requires about *10-30 ns*. And access to the main memory requires tremendous *200 – 500 ns*.

You can see that significant decrease in performance could have place only when both L1 cache and L2 cache do not contain required data. In this case CPU could be stalled for 200 – 500 ns. In case when L1 does not contain required data but L2 cache does performance decrease would be very small. Due to this fact we could omit L1 cache misses from our performance analysis. *We should concentrate on L2 cache misses. Keep in mind that every L2 cache read miss could stall the CPU for about 200 – 500 ns.* Modern ARM CPUs use out-of-order execution technique that could hide some part of this big delay but we do not recommend you to rely on it too often.

WHICH INFORMATION COULD I GET FROM THE PERF TOOL?

Perf tool could provide you with information on which procedures of your application accesses to memory in not optimal manner and generate a lot of cache missises. In many cases optimization of such kind of procedures could significantly improve application performance because memory latency is one of the most important bottlenecks.

Perf can collect the following L2 cache events using Perf tool:

- L2 cache read misses (during instruction fetching)
- L2 cache read misses (during data fetching)
- L2 cache write misses

You could use *perf list* command to show all profiling events available:

```
# export PERF_PAGER=cat
# ./perf list
List of pre-defined events (to be used in -e):
                                                  <…>
l2cache-dr-misses                                 [Software event]
l2cache-dw-misses                                 [Software event]
l2cache-instr-misses                              [Software event]
                                                  <…>
```

Here only L2-related events were showed:

- *l2cache-dr-misses* - L2 data cache read misses
- *l2cache-dw-misses* - L2 data cache write misses
- *l2cache-instr-misses* - L2 instruction cache read misses

You could use each of these events to profile your application. For example, if you are using *l2cache-dr-misses* event it means that in the final report you will get functions that generate a lot of L2 data cache read misses.

L2 CACHE PROFILING DATA COLLECTION AND REPORT PREPARATION

First of all you need to run the application that you want to profile. This could be done both using Android graphical user interface and directly from Android shell. Just after starting your application you should get its process ID (PID). This could be done using 'ps' shell command:

Now you could use this process ID to start application-specific analysis using PERF tool. Process ID will be used to identify application to profile:

```
# cd /data/
# ./perf record –e <L2-cache-event> –p <pid>
```

`<L2-cache-event>` is one of L2 cache events available (see previous section for details).

For example if you need to find out which routines of your application generate a lot of *L2 cache read misses* you should run PERF tool the following way:

```
# cd /data/
# ./perf record –e l2cache-dr-misses –p <pid>
```

We recommend you to collect samples at least 2-3 minutes to get correct performance statistics in the profiling report.

You could stop samples collection using one of the following approaches:

- In Windows please open additional adb shell and run the following command:

  ```
  $ adb shell
  # kill -2 <perf PID>
  ```

  `<perf PID>` - is a process ID of the PERF profiling tool process (do not confuse it with your application's process ID). PERF process ID could also be found using 'ps' utility.
- In Linux or MacOS you could simply press 'Ctrl-C' key combination in the adb shell where you have started PERF before. Or you could open additional adb shell and do the same procedure than in Windows.

Just after PERF deactivation you will get the following output (keep in mind that number of wake ups, *perf.data* file size and the number of samples could be different in your case):

```
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.200 MB perf.data (~8731 samples) ]
```

When samples collection was finished you are ready to start performance report preparation. This could be done by the following way:

```
# export PERF_PAGER=cat
# ./perf report --sort dso,symbol
```

Environment variable *PERF_PAGER* controls which application will be used to show profiling results. In our case we want to get profiling report as-is without any post-processing. That is why we are using *cat* utility that simply prints profiling report to the Android shell. You could try to develop your own post-processing tools that will automate profiling report processing.

Additional *--sort* parameter defines which information we would like to see in final report. Additional information about this option could be found in the PERF tool help subsystem:

```
# export PERF_PAGER=cat
# ./perf report help
```

L2 cache profiling report looks as follows:

```
# export PERF_PAGER=cat
# ./perf report –sort dso,symbol

# Overhead  Shared Object       Symbol
# ........  ..................  ................................

    56.17%  libOpTest.so        [.] adler32
    23.75%  libOpTest.so        [.] fill_window
     9.80%  libOpTest.so        [.] deflate_slow
     5.28%  libOpTest.so        [.] longest_match
     1.56%  libc.so             [.] memcpy
     1.19%  libOpTest.so        [.] crc32_little
     0.81%  libc.so             [.] __dorand48
     0.39%  libOpTest.so        [.] randomBuffer

                                        <…>
```

In the profiling report you will see the following information fields:

- *Symbol* – procedure name, where procedures of the target application are marked with a *[.]* flag and kernel procedures are marked with a *[k]* flag
- *Shared Object* – name of the binary that includes the function, it could be executable, shared library or the Android/Linux kernel
- *Overhead* – how many L2 cache misses this procedure generated during profiling (in percentages from the total number of L2 cache misses happened)

Here you can see that *adler32* routine (which is a checksum calculation procedure that reads big blocks of data) generated the biggest part of L2 cache read misses (data collection was done with *l2cache-dr-misses* flag). We could try to optimize this routine to use cache more effectively (see Data Reading section for additional information).

TIPS & TRICKS

ADDITIONAL FEATURES OF THE PERF TOOL

PERF is a very customizable tool. You can change performance report format using a lot of specific command line options. You could get detailed information on which options are available using the following command:

```
# export PERF_PAGER=cat
# ./perf report help
```

One of the most frequently used configuration option is *-n* flag. It is usable during L2 cache miss report preparation and allows you to see not only percentage value (which percentage of all cache misses was generated by the function) but the real number of cache misses that was generated by the function. This value could be usable for a further investigation.

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
```

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