PMUV3 Plugin for Performance Analysis

OVERVIEW

There is a way to measure profiling of the target application as a whole box, but sometimes we need to add the instrumentation into the code itself to get more fine-grained and precise measurement of the functions handling a specific task we are interested to know its performance. For that intention, we developed the PMUv3 Plugin to allow the users to do so. This performance monitoring plugin helps to do performance analysis based on the Hardware events available in PMUV3.

To access the user space registers directly, we employ the mmap() system call on the perf event file descriptor. This action prompts the kernel to enable user space access and furnish us with a handle to read the raw counter registers. We have supported the simplest way of measuring CPU Cycle counts as well as measuring many different bundles of events in one shot (like Cache misses et al along with CPU cycles)

The PMUv3_plugin not only records values of raw counter registers but also provides support to visualize the results in a CSV format by providing a post processing code. The PMUv3 source file is written in C language. Hence one can call the APIs within a C codebase by including the header file and in the case of a C++ codebase, one can include the headers within extern. This documentation explains the usage information in detail.

FEATURES OF THE PLUGIN

This section briefs the **PMU events in every bundle and KPIs** that can be derived out of every bundle along with raw event values.

Descriptions of every event can be found in N1 ARM Neoverse guide. [Arm® Neoverse™ N1 PMU Guide]

BUNDLE 0 - CPU_CYCLES, L1&L2 TLB, REFILL, DTLB_WALK

Monitors 6 events including CPU_CYCLES.

Events - CPU_CYCLES, L1D_TLB_REFILL, L1D_TLB, L2D_TLB_REFILL, L2D_TLB, DTLB_WALK

KPIs derived

L2_TLB_miss_rate = L2D_TLB_REFILL / L2D_TLB
L1_data_TLB_miss_rate = L1D_TLB_REFILL / L1D_TLB

Event Code & Description

CPU_CYCLES - 0x11

This event counts CPU clock cycles (not timer cycles). The clock measured by this event is defined as the physical clock driving the CPU logic.

L1D_TLB_REFILL - 0x05

This event counts L1 D-side TLB refills from any D-side memory access. If there are multiple misses in the TLB that are resolved by the refill, then this event will only count once. This event counts for refills caused by preload instructions or hardware prefetch accesses. This event will count regardless of whether the miss hits in L2 or results in a page table walk. This event will not count if the page table walk results in a fault (such as a translation or access fault), since there is no new translation created for the TLB. This event will not count with an access from an AT (address translation) instruction. This event is the sum of the L1D_TLB_REFILL_RD and L1D_TLB_REFILL_WR events.

L1D_TLB - 0x25

This event counts any L1 D-side TLB access caused by any memory load or store operation. Note that load or store instructions can be broken up into multiple memory operations.

L2D_TLB_REFILL - 0x2D

This event counts any allocation into the L2 TLB from either an I-side or D-side access. This event is the sum of the L2D_TLB_REFILL_RD and L2D_TLB_REFILL_WR events.

L2D_TLB - 0x2F

This event counts any access into the L2 TLB except those caused by TLB maintenance operations. This event is the sum of the L2D_TLB_RD and L2D_TLB_WR events.

DTLB_WALK - 0x34
This event counts any page table walk (caused by a miss in the L1 D-side and L2 TLB) driven by a D-side memory access. Note that partial translations that also cause a page walk are counted.

BUNDLE 1 - TLB RD, WR, TLB_REFILL RD, WR

Monitors 5 events including CPU_CYCLES.

Events - CPU_CYCLES, L2D_TLB_REFILL_RD, L2D_TLB_REFILL_WR, L2D_TLB_RD, L2D_TLB_WR

KPIS derived

L2_TLB_write_miss_rate = L2D_TLB_REFILL_WR / L2D_TLB_WR
L2_TLB_read_miss_rate = L2D_TLB_REFILL_RD / L2D_TLB_RD

Event Code

L2D_TLB_REFILL_RD - 0x5C
L2D_TLB_REFILL_WR - 0x5D
L2D_TLB_RD - 0x5E
L2D_TLB_WR - 0x5F

BUNDLE 2 - MEM_ACCESS, BUS_ACCESS, MEM_ERROR

Monitors 4 events including CPU_CYCLES.

Events - CPU_CYCLES, MEM_ACCESS, BUS_ACCESS, MEMORY_ERROR

Event Code

MEM_ACCESS - 0x13
BUS_ACCESS - 0x19
MEMORY_ERROR - 0x1A

BUNDLE 3 - BRANCH EVENTS

Monitors 7 events including CPU_CYCLES.

Events - CPU_CYCLES, BR_MIS_PRED, BR_PRED, BR_RETIRED, BR_MIS_PRED_RETIRED, BR_IMMED_SPEC, BR_INDIRECT_SPEC

Event Code

BR_MIS_PRED - 0x10
BR_PRED - 0x12
BR_RETIRED - 0x21
BR_MIS_PRED_RETIRED - 0x22
BR_IMMED_SPEC - 0x78
BR_INDIRECT_SPEC - 0x7A

BUNDLE 4 - FRONTEND, BACKEND STALLS

Monitors 3 events including CPU_CYCLES.

Events - CPU_CYCLES, STALL_FRONTEND, STALL_BACKEND

KPIS derived

Front_end_stall_rate = STALL_FRONTEND / CPU_CYCLES
Back_end_stall_rate = STALL_BACKEND / CPU_CYCLES

Event Code

STALL_FRONTEND - 0x23
STALL_BACKEND - 0x24

BUNDLE 5 - L1I CACHE & REFILLS

Monitors 3 events including CPU_CYCLES.

Events - CPU_CYCLES, L1I_CACHE_REFILL, L1I_CACHE

KPIS derived

L1_I-cache_miss_rate = L1I_CACHE_REFILL / L1I_CACHE

Event Code

L1I_CACHE_REFILL - 0x01
L1I_CACHE - 0x14

BUNDLE 6 - L1D, L2D, L3D CACHE & REFILLS

Monitors 7 events including CPU_CYCLES.

Events - CPU_CYCLES, L1D_CACHE_REFILL, L1D_CACHE, L2D_CACHE, L2D_CACHE_REFILL, L3D_CACHE_REFILL,L3D_CACHE

KPIS derived
L1_D-cache_miss_rate = L1D_CACHE_REFILL / L1D_CACHE
L2_cache_miss_rate = L2D_CACHE_REFILL / L2D_CACHE

Event Code

L1D_CACHE_REFILL - 0x03
L1D_CACHE - 0x04
L2D_CACHE - 0x16
L2D_CACHE_REFILL - 0x17
L3D_CACHE_REFILL - 0x2A
L3D_CACHE - 0x2B

BUNDLE 7 - L1I TLB, REFILL, ITLB_WALK

Monitors 4 events including CPU_CYCLES.

Events - CPU_CYCLES, L1I_TLB_REFILL, L1I_TLB, ITLB_WALK

KPIS derived

L1_instruction_TLB_miss_rate = L1I_TLB_REFILL / L1I_TLB

Event Code

L1I_TLB_REFILL - 0x02
L1I_TLB - 0x26
ITLB_WALK - 0x35

BUNDLE 8 - IPC & Rate per instructions (INST_RETIRED, PC_WRITE, ASE, ST, INST_SPEC), EXC_TAKEN

Monitors 7 events including CPU_CYCLES.

Events - CPU_CYCLES, INST_RETIRED, PC_WRITE_SPEC, ASE_SPEC, ST_SPEC, INST_SPEC, EXC_TAKEN

KPIS derived
Exception_rate_per_instructions = EXC_TAKEN / INST_RETIRED
Speculatively_executed_IPC = INST_SPEC / CPU_CYCLES
Architecturally_executed_IPC = INST_RETIRED / CPU_CYCLES
PC_WRITE_instruction_rate_per_instructions = PC_WRITE_SPEC / INST_SPEC
SIMD_instruction_rate_per_instructions = ASE_SPEC / INST_SPEC
ST_instruction_rate_per_instructions = ST_SPEC / INST_SPEC

Event Code

INST_RETIRED - 0x08
INST_SPEC - 0x1B
EXC_TAKEN - 0x09
ST_SPEC - 0x71
ASE_SPEC - 0x74
PC_WRITE_SPEC - 0x76

BUNDLE 9 - Rate per Instructions - Branch, Load/Store (LD_SPEC, DSB_SPEC)

Monitors 7 events including CPU_CYCLES.

Events - **CPU_CYCLES, BR_RETURN_SPEC, BR_IMMED_SPEC, BR_INDIRECT_SPEC, INST_SPEC, LD_SPEC, DSB_SPEC**

KPIS derived

BR_IMMED_instruction_rate_per_instructions = BR_IMMED_SPEC / INST_SPEC
ST_instruction_rate_per_instructions = ST_SPEC / INST_SPEC
BR_RETURN_instruction_rate_per_instructions = BR_RETURN_SPEC / INST_SPEC
DSB_rate_per_instructions = DSB_SPEC / INST_SPEC
LD_instruction_rate_per_instructions = LD_SPEC / INST_SPEC'
BR_INDIRECT_instruction_rate_per_instructions = BR_INDIRECT_SPEC / INST_SPEC

Event Code

BR_RETURN_SPEC - 0x79
BR_IMMED_SPEC - 0x78
BR_INDIRECT_SPEC - 0x7A
INST_SPEC - 0x1B
LD_SPEC - 0x70
DSB_SPEC - 0x7D

BUNDLE 10 - L1D_TLB RD/WR, REFILL RD/WR (Miss rate)

Monitors 5 events including CPU_CYCLES.

Events - CPU_CYCLES, L1D_TLB_REFILL_RD, L1D_TLB_REFILL_WR, L1D_TLB_RD, L1D_TLB_WR

KPIS derived
L1_data_TLB_write_miss_rate = L1D_TLB_REFILL_WR / L1D_TLB_WR
L1_data_TLB_read_miss_rate = L1D_TLB_REFILL_RD / L1D_TLB_RD

Event Code

L1D_TLB_REFILL_RD - 0x4C
L1D_TLB_REFILL_WR - 0x4D
L1D_TLB_RD - 0x4E
L1D_TLB_WR - 0x4F

BUNDLE 11 - MPKI (INST_RETIRED, LL_CACHE_MISS_RD, L1D & L1I CACHE_REFILL, ITLB_WALK)

Monitors 6 events including CPU_CYCLES.

Events - **CPU_CYCLES, INST_RETIRED, LL_CACHE_MISS_RD, L1D_CACHE_REFILL, ITLB_WALK, L1I_CACHE_REFILL**

KPIS derived
L1_I-cache_MPKI = L1I_CACHE_REFILL / INST_RETIRED
I-side_page_table_MPKI = ITLB_WALK / INST_RETIRED
L1_D-cache_MPKI = **L**1D_CACHE_REFILL/ INST_RETIRED
LLC_cache_MPKI = LL_CACHE_MISS_RD / INST_RETIRED

Event Code

INST_RETIRED - 0x08
LL_CACHE_MISS_RD - 0x37
L1D_CACHE_REFILL - 0x03
ITLB_WALK - 0x35
L1I_CACHE_REFILL - 0x01

BUNDLE 12 - MPKI (INST_RETIRED , L2D_CACHE_REFILL, DTLB_WALK BR_MIS_PRED_RETIRED)

Monitors 5 events including CPU_CYCLES.

Events - CPU_CYCLES, INST_RETIRED, L2D_CACHE_REFILL, DTLB_WALK BR_MIS_PRED_RETIRED

KPIS derived
L2_cache_MPKI = L2D_CACHE_REFILL / INST_RETIRED
Branch_MPKI = BR_MIS_PRED_RETIRED / INST_RETIRED
D-side_page_table_MPKI = DTLB_WALK' / INST_RETIRED

Description

INST_RETIRED - 0x08
DTLB_WALK - 0x34
BR_MIS_PRED_RETIRED - 0x22
L2D_CACHE_REFILL - 0x17

BUNDLE 13 - L1D_CACHE RD/WR, REFILL RD/WR (Miss rate)

Monitors 7 events including CPU_CYCLES.

Events - CPU_CYCLES, L1D_CACHE_REFILL_OUTER, L1D_CACHE_REFILL, L1D_CACHE_RD, L1D_CACHE_REFILL_WR, L1D_CACHE_REFILL_RD, L1D_CACHE_WR

KPIS derived

L1_D-cache_read_miss_rate = L1D_CACHE_REFILL_RD / L1D_CACHE_RD
L1_D-cache_write_miss_rate = L1D_CACHE_REFILL_WR / L1D_CACHE_WR
L1_D-cache_rate_of_cache_misses_in_L1_and_L2 = L1D_CACHE_REFILL_OUTER / L1D_CACHE_REFILL

Event Code

L1D_CACHE_REFILL_OUTER - 0x45
L1D_CACHE_REFILL - 0x03
L1D_CACHE_REFILL_RD - 0x42
L1D_CACHE_RD - 0x40
L1D_CACHE_REFILL_WR - 0x43
L1D_CACHE_WR - 0x41

BUNDLE 14 - CRYPTO, ISB, DP, DMB, VFP, INST Speculatively executed

Monitors 7 events including CPU_CYCLES.

Events - **CPU_CYCLES, CRYPTO_SPEC, ISB_ SPEC, DP_ SPEC, DMB_ SPEC, VFP_ SPEC, INST_ SPEC**

KPIS derived

VFP_instruction_rate_per_instructions = VFP_SPEC / INST_SPEC
DMB_rate_per_instructions = DMB_SPEC / INST_SPEC
DP_instruction_rate_per_instructions = DP_SPEC / INST_SPEC
ISB_rate_per_instructions = ISB_SPEC /
INST_SPECCRYPTO_instruction_rate_per_instructions = CRYPTO_SPEC / INST_SPEC

Event Code

CRYPTO_SPEC - 0x77
ISB_SPEC - 0x7C
DP_SPEC - 0x73
DMB_SPEC - 0x7E
VFP_SPEC - 0x75
INST_SPEC - 0x1B

REQUIREMENTS

STEP 0: Directory Tree

We recommend the directory structure to be as follows. Kindly place your codebase/project within a directory called /YOUR_HOME_DIR/ut_integration.

```
YOUR_HOME_DIR
    /ut_integration
        /linux (git clone from Linux kernel)
        /PMUv3_plugin
        /YourApplication
```

STEP 1: Clone Linux and PMUv3_plugin

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git

git clone https://github.com/GayathriNarayana19/PMUv3_plugin.git
```

(Below step OPTIONAL, good to enable once.)

Enable userspace access : To ascertain if userspace access to the PMU counters is allowed, check the `perf_user_access` file.

```
cat /proc/sys/kernel/perf_user_access
0

sudo sysctl kernel/perf_user_access=1
kernel/perf_user_access=1

cat /proc/sys/kernel/perf_user_access
1
```

STEP 2: Run ./build.sh

To do the static library compilation, run **`./build.sh`** from `/home/ubuntu/ut_integration/PMUv3_plugin/directory`.

Run `./build.sh` if you are going to instrument around a C++ codebase. If it is a C codebase, then **comment line 19** of `build.sh` and **uncomment line 20** and run `./build.sh`

NOTE: For more information on what the `build.sh` file does, see **Appendix A: Explanation for build.sh**

`build.sh`

```
cd /home/ubuntu/ut_integration
pushd linux/tools/lib/perf
make
popd; pushd linux/tools/lib/api
make
popd

cd /home/ubuntu/ut_integration/PMUv3_plugin

cp /home/ubuntu/ut_integration/linux/tools/lib/perf/libperf.a .
cp /home/ubuntu/ut_integration/linux/tools/lib/api/libapi.a .

gcc -c pmuv3_plugin.c -I/home/ubuntu/ut_integration/linux/tools/lib/perf/include -o pmuv3_plugin.o
gcc -c pmuv3_plugin_bundle.c -I/home/ubuntu/ut_integration/linux/tools/lib/perf/include -o pmuv3_plugin_bundle.o
gcc -c processing.cpp -I/home/ubuntu/ut_integration/linux/tools/lib/perf/include -o processing.o
gcc -c processing.c -I/home/ubuntu/ut_integration/linux/tools/lib/perf/include -o processing_c.o

ar rcs libpmuv3_plugin.a pmuv3_plugin.o
ar rcs libpmuv3_plugin_bundle.a pmuv3_plugin_bundle.o processing.o
#ar rcs libpmuv3_plugin_bundle.a pmuv3_plugin_bundle.o  processing_c.o
```

STEP 3: Include the above static library in Makefile/CMakelists or relevant files

In your application's `Makefile` or `CMakeLists` files, make sure to add or include this `-lpmuv3_plugin_bundle.a` static library and `-L` to point its location.

NOTE: For examples, see **Appendix B: Examples for linking static library.**

INSTRUMENTATION USING PMUV3_PLUGIN

There are 3 scenarios listed below. Ideally, pick the scenario you are looking for and refer to that usage.

Scenario I – Instrumentation In Different Code Blocks in C++ codebase

For Pmuv3_Bundles Instrumentation IN DIFFERENT CHUNK OF CODES in C++ codebase (e.g.: Multiple chunks of code in same testcase, Multiple functions or Nested functions), follow the below steps.

1. In your application source code where the PMUv3 instrumentation will be embedded, you need to include header this way.

```
#include "processing.hpp"

#ifdef PMUV3_CPU_BUNDLES
extern "C" {
    #ifdef PARENT_DIR
        #include "pmuv3_plugin_bundle.h"
    #endif
}
#endif
```

2. Initialize the PMUv3 Event Bundle - void pmuv3_bundle_init(int)

In testcases, in main function, we need to pass the argument for which bundle to choose:

```
int main(int argc, char** argv)
{
    if (argc != 2) {
        printf("Usage: %s <arg>\n", argv[0]);
        exit(1);
    }
     int cur_bundles_no = atoi(argv[1]);
    // then call initialization once:
    pmuv3_bundle_init(cur_bundles_no);
}
```

3. local_index is a unique variable specific to every piece of instrumentation. It will be used to map the end_count to corresponding start_count and helps in post processing to calculate the cycle difference.

The get_next_index() API will help to increment the local_index by 1 at every call.

```
uint64_t local_index = get_next_index();
```

NOTE: This local_index variable that you define should be unique everytime. You call this before calling the get_start_count() API and every single time give unique variable name like local1, local2, local3 etc instead of using local_index everytime. This uniqueness will be useful when there are multiple functions of the same level within a function. Eg: When f2(), f3() are present within f1() and f2(), f3() are of same level, not nested.

4. Start Event Bundle - uint64_t get_start_count(struct PerfData **perf_data, struct CountData *count_data, const char** context, uint64_t index);

For example:

```
get_start_count(&count_data, "DU_HIGH1", local_index);
```

NOTE: The third variable is a context. NOTE: Whatever context (3rd parameter) and index (4th parameter) one passes in get_start_count() should be passed to corresponding get_end_count()

5. End Event Bundle - uint64_t get_end_count(struct PerfData **perf_data, struct CountData *count_data, const char** context, uint64_t index);

```
get_end_count(&count_data, "DU_HIGH1", local_index);
```

6. Define this in a place after all instrumentation is done.

```
process_data(cur_bundle_no);
```

7. Shutdown and release resource for Event Bundle Instrument - int shutdown_resources(struct PerfData **perf_data);*

```
shutdown_resources();
```

Example Instrumentation For Reference

//Just once in main()

```
int main(int argc, char** argv)
{
    if (argc != 2) {
        printf("Usage: %s <arg>\n", argv[0]);
        exit(1);
    }
     int cur_bundles_no = atoi(argv[1]);
    // then call initialization once:
    pmuv3_bundle_init(cur_bundles_no);
```

```
}
```

//In places of instrumentation, do like below. Remember that every get_start_count will have a separate get_end_count API.

```
uint64_t local_1 = get_next_index();

get_start_count(&count_data,"CONTEXT_1", local_1);

/********************************1ST CODE
CHUNK***********************************/

get_end_count(&count_data, "CONTEXT_1", local_1);


.
.
.
.

uint64_t local_2 = get_next_index();

get_start_count(&count_data,"CONTEXT_2", local_2);

/********************************2ND CODE
CHUNK***********************************/


get_end_count(&count_data,"CONTEXT_2",local_2);
```

// Below APIs will be invoked only once per testcase after instrumenting in several places.

```
shutdown_resources();

process_data(cur_bundle_no);
```

Scenario II – Instrumentation Around Single Code block in C++ codebase

For Pmuv3_Bundles Instrumentation "SINGLE CHUNK OF CODE" in C++ CODEBASE, refer the below steps.

1.In your application source code where the PMUv3 instrument will be embedded, you need to include header this way.

```
#include "processing.hpp"

#ifdef PMUV3_CPU_BUNDLES

extern "C" {
    #ifdef PARENT_DIR
        #include "pmuv3_plugin_bundle.h"
    #endif
}
#endif
```

2. Initialize the PMUv3 Event Bundle - void pmuv3_bundle_init(int)

In testcases, in main function, we need to pass the argument for which bundle to choose:

```
int main(int argc, char** argv)
{
    if (argc != 2) {
        printf("Usage: %s <arg>\n", argv[0]);
        exit(1);
    }
    int cur_bundles_no = atoi(argv[1]);

    // then call initialization once:

    pmuv3_bundle_init(cur_bundles_no);
}
```

3. Instrument around single chunk of code

```
process_start_count(&count_data);

///////////CODE CHUNK TO BE INSTRUMENTED/////////////

process_end_count(&count_data);
```

4. Define this in a place after all instrumentation is done.

```
process_single_chunk(cur_bundle_no);
```

5. Shutdown and release resource for Event Bundle Instrument

```
shutdown_resources();
```

This populates bundle0.csv, bundle1.csv etc. as requested by user in the directory where you ran the testcase.

Scenario III– Instrumentation around different code blocks in C codebase

For Pmuv3_Bundles Instrumentation in a C Codebase around different Chunks Of Code, refer below steps.

Follow the same procedure described above with small changes.

Reminder: Before you run ./build.sh, vim build.sh and uncomment line 20 and comment line 19. This was already mentioned in Requirements section.

1. No need for extern in C code base so we include directly.

```
#include <processing.h>

#include "pmuv3_plugin_bundle.h"
```

2. Initialization and instrumentation APIs are the same as mentioned in C++ sections for DIFFERENT CHUNK OF CODES (Section I) scenario.

- **DIFFERENT CHUNK OF CODES**

```
uint64_t local_1 = get_next_index();

get_start_count(&count_data, "CONTEXT_1" ,local_1);

/*********************************1ST CODE
CHUNK***********************************/

get_end_count(&count_data, "CONTEXT_1", local_1);


uint64_t local_2 = get_next_index();

get_start_count(&count_data, "CONTEXT_2", local_2);

/*********************************2ND CODE
CHUNK***********************************/

get_end_count(&count_data, "CONTEXT_2", local_2);
```

3. In post processing,

```
process_data(cur_bundle_no);
```

4. Shutdown resources

```
shutdown_resources();
```

Scenario IV – Instrumentation around single code blocks in C codebase

For Pmuv3_Bundles Instrumentation in a C Codebase around a Single Chunk Of Code, refer below steps.

Follow the same procedure described above with small changes.

Reminder: Before you run ./build.sh, vim build.sh and uncomment line 20 and comment line 19. This was already mentioned in Requirements section.

1. No need for extern in C code base so we include directly.

```
#include <processing.h>

#include "pmuv3_plugin_bundle.h"
```

2. Initialization and instrumentation APIs are the same as mentioned in C++ sections of SINGLE CHUNK OF CODE (Section II) scenario.

Scenario 1 - SINGLE CHUNK OF CODES

```
process_start_count(&count_data);

///////////CODE CHUNK TO BE INSTRUMENTED/////////////

process_end_count(&count_data);
```

3. In post processing,

```
post_process(bundle_num);
```

4. Shutdown resources

```
shutdown_resources();
```

BUILD YOUR PROJECT AGAIN!

DON'T FORGET TO COMPILE AFTER THE INSTRUMENTATION!

After making the above changes, from build directory of your codebase or project,

cmake ../

make

(or)

run the "make" command specific to your codebase.

PMUV3 PLUGIN FUNCTIONALITY TEST

Note: This test is optional and should be tested only on **N1 ampere**. If it gives the expected result, it validates the working nature and granularity of PMUv3_plugin and hence will work fine on all generations of ARM processors.

For users to test the PMUv3 plugin functionality, a Makefile has been provided. Within /PARENT_DIR/ut_integration/PMUv3_plugin/build/

Run the below command

To make clean,

```
make -f ../Makefile clean
```

To make,

```
make -f ../Makefile
```

To run,

```
./test 7
```

Expected result on N1 is a value around the below range.

```
- running pmuv3_plugin_bundle.c...OK
End is 3007936805, Start is 81760, CPU_CYCLES is 3007855045
```

APPENDICES

Appendix A: Explanation for build.sh

We require `libperf.a` and `libapi.a` from Linux standard kernel

Recommendation: Please compile and replace it with your compiled liperf.a and libapi.a as it is platform dependent. How `libperf.a` and `libapi.a` were compiled?

STEP 1: Go to /linux/tools/lib/perf

make

STEP 2: Go to /linux/perf/tools/lib/api

make

STEP 3: Copy libperf.a and libapi.a to the pmuv3_plugin directory

These steps have been automated in `build.sh`. One can execute the above steps manually as well. build.sh will also compile the PMUv3 plugin source files like pmuv3_plugin_bundle.c, post processing files like processing.cpp and generate static libraries (libpmuv3_plugin_bundle.a) as you see in lines 13 to 20.

Appendix B: Examples for linking static library.

Example 1: makefile of a 5G codebase

In line 55, ARM_L2_LIBS variable is augmented with the static libraries we generate in `build.sh` - `libperf.a` , `libapi.a` and `libpmuv3_plugin_bundle.a`.

In line 85, the ARM_L2_LIBS variable has been added to the du_app target to be linked.

In line 82, relative path is mentioned to include the path of PMUv3_plugin. But one can define a parent dir and use it as well which is more appropriate. In line 84, `-L` is used to point to the directory path.

```
50
51 NGP_LIBS=-lngpcomm -lngpexcp -lngplogging -lngpmem -lngpqueue -lngpsys         \
52          -lngpthread -lngpbuffer -lngptimer -lversion
53
54 ifneq (,$(filter -DARM_CPU_CYCLE_COUNT, $(CFLAGS)))
55 ARM_L2_LIBS+= -lpmuv3_plugin_bundle -lperf -lapi
56 #ARM_L2_LIBS+= -lpmuv3_plugin -lperf -lapi
57 endif
58
59 ifneq (,$(filter -DWEB_SOCKET_ENABLED, $(CFLAGS)))
60 NGP_LIBS+= -lngpwebsocket_client
61 endif
62 #L2_LIBS= -lcm -lmt -lcmn -lkw -lrg -ltf -lpmstreamcodec -lducodeccommon -L ../../../../../perf_cycle_codes -Wl,-Bstatic -lperf -L ../../../../../perf_cycle_codes
    -Wl,-Bstatic -lapi -L ../../../../../perf_cycle_codes -Wl,-Bstatic -lpmuv3_plugin
63
64 L2_LIBS= -lcm -lmt -lcmn -lkw -lrg -ltf -lpmstreamcodec -lducodeccommon
65 GTEST_LIB_PATH=../../src/gt_ut/gtest-1.8.0/lib
66
67 ifneq (,$(filter -DFEATURE_IAB=1, $(FEATURE_FLAGS)))
68 APP_LIBS+=-lbap
69 endif
70 ifneq (,$(filter -DFEATURE_IAB_DONOR=1, $(FEATURE_FLAGS)))
71 NGP_LIBS+=-lngptun
72 endif
73 ifneq (,$(filter -DFEATURE_NTN=1, $(DU_FLAGS)))
74 APP_LIBS+=-lncf_handler
75 endif
76
77 $(info ************  FILE_ID GENERATION *****************)
78 #$(shell /bin/bash ../common/check_file_id.sh  >/dev/null)
79 $(info ************  FILE_ID GENERATION DONE ************)
80 #include if any obj file needs to be included
81 OBJS=./obj/*.o
82 PMU_PLUGIN_DIR := ../../../../../../PMUv3_plugin
83 du_app: prepare_dirs du_log_post_process libs
84         $(CC) $(OBJS) -L. $(NGP_LIBS_PATH) $(SAS_THIRD_PARTY_LIB_PATH) $(L2_LIBS_PATH) -L$(PMU_PLUGIN_DIR) $(GTEST_LIB_PATH)/libgtest.a $(PHY_LIBS_PATH) -o $(BIN_
   TARGET)          \
85     $(NGP_LIBS) $(RTE_LIBS) $(NGP_BASE_LIBS) $(PHY_LIBS) $(NS_LIBS) $(LIB_OAM) $(ARM_L2_LIBS)
86
```

flags.mk file of the same codebase is a file where we include relevant directory paths required for the PMUv3_plugin to work.

Line 1 sets the TOP_DIR.

Line 7 include the linux path that contains libraries used by PMUv3_plugin.

Line 8 is the path to the PMUv3_plugin itself.

In line 1164, 1165, -I is used to include these directories to a variable called I_OPTS which was figured to be used in the 5G codebase to include similar paths/directories. So, we followed the same style.

In line 1174, 1171 and 1178, the I_OPTS is added by default.

```
 1 TOP_DIR=/home/ubuntu/ut_integration
 2 CU_BASE_DIR=$(TOP_DIR)/radisys_UT/DU_UT_PM3_CPU_Cycle/5gran/cu
 3 DU_BASE_DIR=$(TOP_DIR)/radisys_UT/DU_UT_PM3_CPU_Cycle/5gran/du
 4 NGP_BASE_DIR=$(TOP_DIR)/radisys_UT/DU_UT_PM3_CPU_Cycle/ngp/
 5 NGP_INCLUDE_DIR=$(NGP_BASE_DIR)/include/
 6 OS_VERSION=$(shell cat /etc/os-release | grep -w "VERSION_ID" | sed -n 1p | cut -d= -f 2)
 7 ARM_LINUX_INCLUDE_DIR=$(TOP_DIR)/linux/tools/lib/perf/include/
 8 ARM_PMUV3_INCLUDE_DIR=$(TOP_DIR)/PMUv3_plugin/
 9 ifeq ($(TARGET),arm)
10 ifeq ($(CROSS_COMPILE),aarch64-poky-linux-)
```

```
endif
I_OPTS+=-I$(ARM_LINUX_INCLUDE_DIR)
I_OPTS+=-I$(ARM_PMUV3_INCLUDE_DIR)
I_OPTS+=-I$(ROOT_DIR)/src/codec/include/
ALL_FLAGS=$(SS_FLAGS) $(ENV_FLAGS) $(DU_FLAGS) $(RTE_FLAGS) $(LNXENV)

ifeq ($(TARGET),arm)
ifeq ($(AIO_GNB),YES)
CFLAGS += -g -O3 -ffast-math -std=c++11 -Wall -Werror -Wno-write-strings \
          -fno-defer-pop -fsigned-char -pipe $(I_OPTS) -DLOGGING_FILE_NAME_LINE_NUM_ENABLED -DCONST_MAX_CELLS_SUPPORTED=1
else
CFLAGS += -g -O3 -ffast-math -std=c++11 -Wall -Werror -Wno-write-strings \
                  -fno-defer-pop -fsigned-char -pipe $(I_OPTS) -DLOGGING_FILE_NAME_LINE_NUM_ENABLED -DCONST_MAX_CELLS_SUPPORTED=6
endif
else
CFLAGS += -g -Ofast -std=c++11 -Wall -Werror -Wno-write-strings \
                  -fno-defer-pop -fsigned-char -pipe $(I_OPTS) -DLOGGING_FILE_NAME_LINE_NUM_ENABLED -DCONST_MAX_CELLS_SUPPORTED=18
endif
```

Example 2: Codebase that has CMakelists

```
26
27 set_directory_properties(PROPERTIES LABELS "du_high|tsan")
28
29 include_directories(../../..)
30
31 add_executable(du_high_benchmark du_high_benchmark.cpp)
32 #add_executable(du_high_benchmark du_high_benchmark.cpp ${PARENT_DIR}/PMUv3_plugin/pmuv3_plugin_bundle.c)
33 # Add the USE_READ_CYCLE_COUNT option
34 option(USE_READ_CYCLE_COUNT "Use readCycleCount for start and end cycles" OFF)
35
36 # Add the USE_READ_CYCLE_COUNT option to compile definitions
37 target_compile_definitions(du_high_benchmark PRIVATE $<$<BOOL:${USE_READ_CYCLE_COUNT}>:USE_READ_CYCLE_COUNT>)
38 target_compile_definitions(du_high_benchmark PRIVATE PARENT_DIR="${PARENT_DIR}")
39
40 target_include_directories(du_high_benchmark PRIVATE ${PARENT_DIR}/linux/tools/lib/perf/include)
41 target_include_directories(du_high_benchmark PRIVATE ${PARENT_DIR}/PMUv3_plugin)
42
43 #target_include_directories(ldpc_encoder_benchmark PRIVATE /home/ubuntu/linux/tools/lib/perf/include)
44 #target_link_libraries(du_high_benchmark srsran_du_high f1ap_du_test_helpers srsran_pcap gtest ${PARENT_DIR}/PMUv3_plugin/libpmuv3_plugin.a ${PARENT_DIR}/PMUv3_plugin/libperf.a ${PARENT_DIR}/PMUv3_plugin/libapi.a)
45
46 target_link_libraries(du_high_benchmark srsran_du_high f1ap_du_test_helpers srsran_pcap gtest ${PARENT_DIR}/PMUv3_plugin/libpmuv3_plugin_bundle.a ${PARENT_DIR}/PMUv3_plugin/libperf.a ${PARENT_DIR}/PMUv3_plugin/libapi.a)
47 #target_link_libraries(du_high_benchmark srsran_du_high f1ap_du_test_helpers srsran_pcap gtest ${PARENT_DIR}/PMUv3_plugin/libperf.a ${PARENT_DIR}/PMUv3_plugin/libapi.a)
48 add_test(du_high_benchmark du_high_benchmark)
```

```
20 set(PARENT_DIR /home/ubuntu/ut_integration)
```

In line 40 and 41, paths to linux libraries and PMUv3_plugin are included.

In line 46, static libraries are included which will be linked in compilation.

I demonstrated how to include the static libraries and paths by showing examples from two different codebases. Similarly, for any codebase, one has to figure out an appropriate way to include.

Appendix C: Significance of mmap()

For reading user space registers using the `perf_event_open` system call, `mmap()` is employed to enable user space access to performance monitoring counters.

The `perf_event_open()` system call initializes a performance monitoring event specified by `struct perf_event_attr`. Once the event is configured, it returns a file descriptor `perf_fd` associated with the event. To access the raw counter registers related to this event in user space, `mmap()` is used on the `perf_fd`.

By mapping this file descriptor using `mmap()`, you create a memory-mapped area that directly corresponds to the counters and related data in the kernel. This mapping allows you to access these counters as if they were in the memory space of your user program, enabling efficient and direct access to performance monitoring data without requiring additional system calls for every read or write operation.

This approach permits more efficient access to performance counters and facilitates their utilization for various monitoring and profiling purposes within user space applications.

REFERENCES

- https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/lib/perf/tests/test-evsel.c#n127
- https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/lib/perf/mmap.c#n400
- Test and Run PMUv3 Counters
- perfmon: Node level stats from PMUv3 + required kernel patch