The final publication is available at ACM via http://dx.doi.org/10.1145/3341105.3373955

# On the Reliability of Hardware Event Monitors in MPSoCs for Critical Domains

Javier Barrera[†*], Leonidas Kosmidis[†], Hamid Tabani[†], Enrico Mezzetti[†], Jaume Abella[†],
Mikel Fernandez[†], Guillem Bernat[§] and Francisco J. Cazorla[†]
[†]Barcelona Supercomputing Center, Spain
[*]Universitat Politecnica de Catalunya, Spain
[§]Rapita Systems Ltd., UK

***Abstract*—Performance Monitoring Units (PMUs) are at the heart of most-advanced timing analysis techniques to control and bound the impact of contention in Commercial Off-The-Shelf (COTS) SoCs with shared resources (e.g. GPUs and multicore CPUs). In this paper, we report discrepancies on the values obtained from the PMU event monitors and the number of events expected based on PMU event description in the processor's official documentation. Discrepancies, which may be either due to actual errors or inaccurate specifications, make PMU readings unreliable. This is particularly problematic in consideration of the critical role played by event monitors for timing analysis in domains such as automotive and avionics. This paper proposes a systematic procedure for event monitor validation. We apply it to validate event monitors in the NVIDIA Xavier and TX2, and the Zynq UltraScale+ MPSoC. We show that, while some event monitors count as expected, this is not the case for others whose discrepancies with expected values we analyze.**

## I. Introduction

Performance-improving features, until recently only used in processors for the high-performance domain, are increasingly used in processors in domains like automotive [14]. Those features include multicores, multi-level caches, complex on-chip networks, and accelerators, among which GPUs have a dominant position [4], [35], [27]. This transition from simple micro-controllers to complex micro-processors is driven by the unprecedented performance requirements of complex critical software to support functionalities like autonomous driving in automotive and more autonomous missions in space [6], [37].

Commercial Off-The-Shelf (COTS) processors in critical domains have limited hardware support for time predictability. This includes automotive processors and SoCs such as the NVIDIA DrivePX (Parker and Xavier SoCs), RENESAS R-Car H3, QUALCOMM SnapDragon 820, and Intel Go. Similar concerns also arise on SoCs such as the Xilinx Zynq UltraScale+, increasingly considered for avionics and railway applications among others [38]. Trying to achieve full isolation by software resorting for example to page (memory) colouring techniques[1] has been shown insufficient since interference still exists in shared queues and buffers [32]. Promising software solutions for multicores build on event quota budgeting, monitoring, and enforcement [29], [40], [34], [11] to establish and enforce budgets on task (core) 'maximum shared resources utilization'. The latter is measured with event monitors, e.g. last-level cache misses are used to capture the task's memory utilization. The system software monitors task's activities via the hardware event monitors offered by processors PMUs and suspends or slows down task's execution when their assigned budget is about to be exhausted.

**Problem Statement**. Existing software approaches and solutions for quota (event) monitoring and enforcement, as well as software debugging processes, build on the naive assumption that event monitors and their documentation are always correct. In fact, the trustworthiness of event monitors in COTS processors has not been questioned yet in the real-time research community, despite their critical role as functional and non-functional verification means. The validity of all quota-based software solutions cannot be sustained without providing evidence of a correct functioning of the event monitors, according to the specification available in the official documentation. The lack of such supportive evidence ultimately jeopardizes the timing arguments and potentially invalidates the evidence gathered to successfully undergo the mandatory timing V&V process, in accordance with safety regulations.

**Contribution**. In this paper we take an initial step towards reconciling PMU verification (often disregarded) with its critical role for timing analysis. Our contributions are as follows:
(1) *Analysis of Event Monitor Correctness*. We analyse several event monitors present i) in the GPU of the NVIDIA AGX Xavier and TX2 development boards, and ii) in the CPU of the Xilinx UltraScale+ SoC, and we assess them against their technical specification. Our goal is not to cover all event monitors supported by those architectures, which comprise several hundreds [15]. We aim, instead, at illustrating that some event monitors might not behave as expected. For specific code snippets, we show that some discrepancies occur between observed event counts and the values that a performance analyst would expect based on the event monitors specification provided in the corresponding product manuals. Such evidence supports our claim that OEMs/TIER/timing analysis companies cannot blindly trust event monitors without a preliminary validation.
(2) *Monitor Validation Process*. We describe the steps in a manual validation process that helps to validate the event mon-

[1]Colouring is a well-known technique to segregate accesses to the different blocks of memory-like resources [17], like banks of the shared last-level on-chip cache, the banks and ranks in a DDR memory system [26], [24], [33], or even combined cache-memory segregation [16].

itors of COTS SoCs. We apply this process to a small subset of monitors in i) the NVIDIA Jetson AGX Xavier and TX2; and ii) the Zynq UltraScale+ MPSoC. Those event monitors, for which discrepancies are detected w.r.t. the expected values, are put under quarantine and investigated. For some of them, and as a result of the application of the process, we show that discrepancies can be explained, hence regaining trust on the correctness of the monitor.

(3) *Assessment of an automatic validation process*. We discuss the difficulties of developing a systematic and automatic process for event monitor validation. In contrast with other verification activities (e.g., unit testing), the PMU validation process cannot be easily automated because event counters are extremely target-specific and their operation may differ depending on the processor vendor and the specific hardware/-software configuration. However, manual procedures are frequent in verification and certification processes. This includes all safety-related software in an automotive system that needs to undergo a manual inspection process to be certified.

The rest of this paper is organized as follows: Section II motivates the need for correct event monitors for timing analysis, while Section III presents a methodological approach to validate event monitors against their specification and discusses the difficulties of making this process fully automated. Sections IV, V, and VI report on the application of the proposed validation process to a selection of event monitors in the NVIDIA Jetson AGX Xavier, the Xilinx Zynq Ultrascale+, and the NVIDIA Jetson TX2 respectively. Section VII covers the main related works and Section VIII concludes the paper presenting the main take away messages.

## II. The Need for Reliable Event Monitoring for Functional Safety

In critical embedded domains, timing requirements are classified as non-functional software safety requirements. Safety standards require to allocate adequate time budgets to each software unit, so as to determine feasible task schedules for the overall system. For road vehicles, ISO-26262 [25] safety standard requires providing evidence of *freedom from interference* or controlled interference. For avionics, support document CAST-32A [20] explicitly calls for the identification and bounding of *interference channels*. Existing approaches propose to meet safety standard requirements by deriving the worst-impact that a resource access can cause on the other tasks' accesses, and subsequently limiting the number of accesses each task can perform without compromising other tasks' schedulability [29].

PMUs are an effective means to derive additional (and complementary) information on the contention delay tasks can suffer [11], [29], [12]. While PMUs may vary depending on the different architecture and family of processors and platforms, they generally offer the capability to track a large number of events, typically in the extent of few hundreds or even thousands, related to multiple aspects of execution: from cache-hierarchy statistics to accesses over the interconnects, as well as instruction counts for the different instruction types. Instruction counts are fundamental to assess that the program has been executed correctly. At type level, memory operations such as loads and stores are needed to derive cache miss rates. Likewise, uncacheable loads and stores allow to assess the memory accesses of the program.

The fine-grained information that can be obtained from hardware event monitors can be used to improve the understanding of the timing behaviour of an application [11], [12], to enforce usage thresholds for shared components [29], and to define a more accurate timing model of contention-prone hardware resources [12]. Ultimately, these aspects concur with the sought-after properties of freedom from interference in ISO26262 (and interference channels identification in CAST-32A) to guarantee timing faults cannot propagate across software elements with different criticality levels.

However, the following question arises: whether the information derived from event monitors in PMUs can be trusted for supporting timing evidence for certification purposes [13]. The critical role of PMU information clashes with their intended purpose, as PMUs were originally devised as a means to support low-level hardware debugging and to provide rough outlines about the average behaviour of the software running on top of it. In fact, PMUs have been traditionally developed at the lowest-integrity levels (if any), under quite relaxed V&V criteria, and are, thus, more error prone than components intended for higher integrity levels. Moreover, PMUs are generally accompanied with scarce and inaccurate documentation. Therefore, PMU information cannot be straightforwardly used as a cornerstone for the provision of solid certification arguments on the timing behaviour. Instead, PMU must undergo a rigorous validation process to guarantee the information they provide can be trusted for timing V&V.

## III. Event Monitor Validation Methodology

A methodological approach for event monitor validation is required to use Performance Monitoring Counters (PMCs) with confidence as part of MPSoC timing verification [29], [40], [34], [11]. The sheer number of available events, their differences in terms of operation and characteristics across processor vendors (and even across models for the same vendor), and the dependence on hardware and system software configurations, makes it difficult to define a generic toolkit for validation. What can be pragmatically done, instead, is defining an overall methodological process that can be tailored, building upon engineers expertise, to the specific event monitor and platform configuration.

In the following we first present the steps that need to be performed in order to validate the event monitors. We then elaborate on why, despite some steps may benefit from tool support, automating the whole process is not feasible in practice.

### A. Event Monitor Validation Process

Validating event monitors is a test (experiment) driven process, in which each monitor is exercised while running specific programs. The value read from the monitor is then

compared to an expected value to assess whether it can be deemed as a trusted monitor (match or gap within acceptable threshold) or not. The proposed process comprises several steps (see Figure 1). An expert analyst is required to perform (and tailored) some of the activities as some informed tailoring is typically necessary.

**Event Selection**. Following the trend of processors in the high-performance domain, the number of event monitors in the latest processors in domains such as automotive is in the order of hundreds. As an example, the Xavier SoC offers 273 monitors accessible from the profiler and the debugger. Hence, an exhaustive validation of all event monitors can be too costly in general. Instead, the analyst can discard those event monitors that do no affect the timing/safety argumentation based on requirements coming from the upper timing V&V, and hence, do not require any validation. Also, in some architectures, the hardware allows multiple configurations (a.k.a. platform usage domain or PUD), which impact the event monitors to validate. For instance, if a given resource is partitioned (segregated) it might not be needed to track per-core/task access counts to it. Note that, strictly speaking, this step, represented as ⓪ in Figure 1, is not part of the monitor validation process, which only focuses on the validation of the events provided as input. We have added this preliminary step to the diagram for completeness.

**Experiment and representative benchmark (rbe) design**. From the description of the events in the processor manuals or programmers' guidelines and the understanding of the processor architecture, the analyst designs one or several baseline representative benchmarks or *rbe* ①. Those *rbe* must have two key characteristics. First, the *rbe* needs to exercise the event monitor. Second, the analyst can derive the expected value of the event monitor for that *rbe*, which means that the *rbe* must be simple enough to allow the analyst to place enough confidence on the expected values. For a certification argument, the completeness of the used *rbe* to exercise the event monitor under validation must be justified.

**Validation campaign**. Empirical evidence is collected on the target. The *rbe* is executed in controlled scenarios ② configured by the analysis on the target platform to reduce as much as possible external sources of variability, e.g. operating system. In each run, the PMU is configured to read the event monitor under validation.

**Acceptance criteria**. Next the analyst compares the expected results and those captured with event monitors ③. In case a discrepancy is detected, this can be due to either an imprecise technical documentation of the event monitor in the users' manual, or an actual misbehavior in the counter logic. Either the case, the counter cannot be used *as-is* for timing V&V purposes and further investigation is required to understand, and possibly resolve, the cause of the inconsistency. If no discrepancy is detected in the tests carried out, the counter is deemed as trustable ④ based on the tests performed.

**Formulate hypotheses**. For those counters whose measured values do not match expected ones, the analyst formulates hypotheses ⑤ on the causes for the observed misbehaviour. This

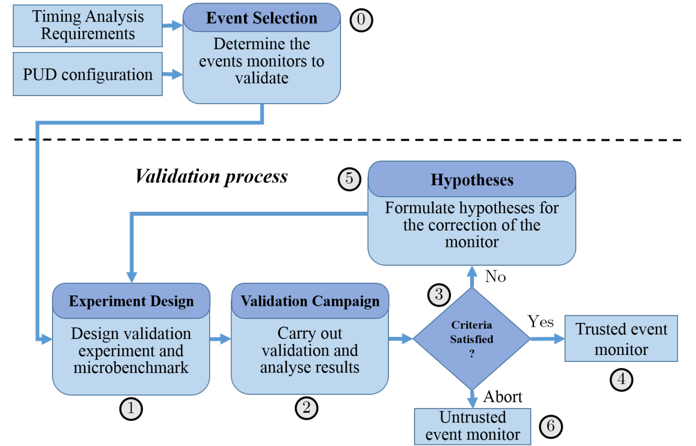


Fig. 1: Proposed validation process.

relates to understand the experiment, the architecture and the expected results. For instance, by determining the magnitude of the discrepancy and the expected values for other related events, the analyst can formulate further hypotheses to be verified. The process continues, going back to step 1, in which the same or new rbe are used to accept/reject the hypotheses. In case it is accepted, then the discrepancy between the observed and the expected values is understood and can be corrected. Instead, if it is rejected, time/effort allowing, new hypotheses are formulated and the whole process starts over. If no further hypotheses can be formulated and/or tested, the event monitor is regarded as untrusted ⑥.

### *B. Systematic and Automated Validation*

The apparently simple assessment process is inherently platform specific and requires deep technical knowledge on both the nominal behavior of the target hardware components and the manifold platform and PMU configurations. Hardware and software development have benefited from some form of automated functional verification based on relatively high-level models of both hardware and software. However, no abstraction model is available for the verification of PMUs. PMUs touch the lowest levels of hardware design and their black-box verification can only be performed building on the understanding and expertise of a hardware expert. In particular, expertise is required in order to select the subset of relevant event monitors to be empirically validated. Further it is not possible to automatically generate the platform configuration and verification snippets necessary to validate a given monitor because both vary across ISA, platforms, models, and versions.

Having an expert supervising a verification or certification process, however, is consolidated practice. Several aspects in testing are delegated to the expertise of testing engineers, especially for the verification of system-wide properties. Several objectives in CAST-32A rely on the guidance of an external assessment as, for example, the identification of interference channels, the verification of inter-core data and control coupling, or the implementation (and coverage) of the safety net [20]. In some of these cases, there is not even a

metric or criteria (such as MCDC or branch for structural coverage) to determine when testing can be deemed sufficient.

For increasingly complex hardware it is infeasible to test all possible scenarios of whether implementation adheres to specifications (pre-silicon verification) and whether manufactured chips carry any type of fault (post-silicon validation) [28]. As a result, both tests and the expected outputs are generally produced by the testing engineers, which need to verify them. Analogously, safety-related software needs to undergo a manual inspection process to verify its correcteness, with no automation possible. Instead, well-detailed procedures are stablished within each company to perform such tedious but critical task. For instance, in the case of the automotive domain, such process is imposed for all criticality levels (from ASIL-A, the lowest, to ASIL-D, the highest) in ISO26262 [25], where the integrity level determines the structure of the group of experts in charge of the process, i.e. how many people are needed and what degree of independence is needed among them to guarantee a reliable code inspection.

While automation is not possible, it is important to establish well-defined procedures that allow performing the verification processes exhaustively and reviewing them easily and avoiding ambiguities and misunderstandings. In the particular case of the reliability of event monitors, the focus of this work, we propose a specific procedure that we apply in specific events and platform examples. This procedure and its results, in the form of evidence verifying what each event monitor counts in practice, is the basis upon which OEMs/TIER/tool vendors can build timing analysis methods and tools for complex SoCs where timing guarantees build upon event quota budgeting, monitoring, and enforcement [29], [40], [34], [11].

### C. Automation Opportunities

Following the discussion in Section III-B, a question that arises is whether some of the steps in the proposed methodological approach can benefit from some form of automation.

Regarding step ①, on experiment and rbe design, while specific procedures can be set, we are not aware of any technology that from the *technical reference manual* of a processor and the event monitor to validate, can systematically and automatically define a (set of) *rbe*(s) to validate it. Instead, this task is to be done manually by a performance analyst, i.e. following predefined procedures, as for design inspection and walkthrough in functional safety verification processes. Once rbe(s) are defined, tool support can be used to derive the expected value for some of the event monitors for that rbe. The analyst could also exploit a database of rbe(s) with precomputed event monitor values, which can be obtained through state-of-the-art simulators or assembly-code analyzers, This of course implies that the used tools shall be *qualified* to the appropriate criticality level according to the applicable safety standards.

Step ② is mostly procedural and can be in large part automated building on an automated test framework.

In terms of acceptance criteria ③, similarly to step ①, there is no systematic approach to determine which acceptance criterion is correct to apply on each case. In fact, such criterion is to be assessed by the expert analyst and properly described and sustained in front of the certification authorities, building upon repeatable protocols.

Likewise, in step ⑤ and once a deviation is detected in the event counter, we are not aware of any solution to automatically formulate hypotheses to explain the observed behavior and design new experiments (and likely a rbe) to assess them. Hence, it requires human intervention.

## IV. NVIDIA JETSON AGX XAVIER

We assess our validation approach on a selection of event monitors in the NVIDIA Jetson AGX Xavier. In particular we focus on type-based instruction counts, a basic information element used for several aspects of timing analysis. This includes the following:

1) For quota monitoring, store counts is important when first level data caches are write-through as each store causes a transfer to the inter-core shared interconnection or the next (second level) shared cache level.
2) Instruction counts for uncacheable loads and stores determine how many times specific devices, subject to contention, are used.
3) Instruction counts are also used for timing validation as they allow assessing whether programs experience preemption by comparing instruction counts between runs on bare metal and on top of the analysed RTOS.

The first column in Table I describes the particular event monitors to validate, while the second column provides the description in the official GPU provider documentation. To obtain this information we used the *NVPROF tool* [2] from CUDA 10.0 version toolkit as follows: `nvprof --query-events --query-metrics`. As it can be seen, each event monitor counts certain instruction types. The particular operation codes under each instruction type are provided in a different document [30]. Column three lists the subset of opcodes under each instruction type on which we focus (extending this to other opcodes is an engineering work following the same EVMP approach). For instance, `inst_integer` captures the following opcodes: `BMSK`, `BREV`, `FLO`, `IABS`, `IADD`, **`IADD3`**, `IADD32I`, `IDP`, `IDP4A`, **`IMAD`**, `IMMA`, `IMNMX`, `IMUL`, `IMUL32I`, `ISCADD`, `ISCADD32I`, **`ISETP`**, `LEA`, `LOP`, **`LOP3`**, `LOP32I`, `POPC`, **`SHF`**, `SHL`, `SHR`, `VABSDIFF`, `VABSDIFF4`. From those we focus on those boldfaced as they are the only ones that appear in our tests. Interestingly, there is not event counter to track `MOV` and `SHFL` instructions.

### A. Experiment and rbe Design

We build on a matrix copy program on which we can derive the number of instructions expected of each type. Figure 2 (top) shows the C code with CUDA calls of the program, and the corresponding GPU assembly (SASS) code produced for this specific GPU, by using `cuobjdump` (bottom).

Instructions 1 and 2 in the SASS code comprise the kernel's prologue, performing the kernel initialization. Instructions 3 to

TABLE I: Instruction types used in this analysis for the NVIDIA Jetson AGX Xavier GPU.

| Event [2] | Official Description [2] | Opcodes counted [30] |
|---|---|---|
| inst_integer | Number of integer instructions executed by non-predicated threads | IMAD, IADD3, SHF, LOP3, ISETP |
| inst_fp_32 | No. of single-precision fp instructions executed by non-predicated threads (arithmetic, compare, etc.) | FSETP, FMUL, FADD, FSEL |
| inst_compute_ld_st | Number of compute load/store instructions executed by non-predicated threads | LDS, LDG, STS, STG |
| inst_control | Number of control-flow instructions executed by non-predicated threads (jump, branch, etc.) | BRA, EXIT |
| inst_bit_convert | Number of bit-conversion instructions executed by non-predicated threads | I2F |
| ***no event*** | Instructions that move data across registers | MOV, SHFL |
| inst_misc | Number of miscellaneous instructions executed by non-predicated threads | NOP, S2R, BAR |
| not_pred_off_thread_inst_exec | Number of thread instructions executed that are not predicated off | Total |

```
1  #include <stdio.h>
2  __global__
3  void copy(int N, float *d_x, float *d_y){
4    if(d_x[0]<0){
5      int x = blockDim.x*blockIdx.x + threadIdx.x;
6      int y = blockDim.y*blockIdx.y + threadIdx.y;
7      d_y[N*y + x]=d_x[N*y + x];
8  } }
9
10 int main(void){
11   int N = 1024;
12   float *x, *y, *d_x, *d_y;
13   x = (float*)malloc(N*N*sizeof(float));
14   y = (float*)malloc(N*N*sizeof(float));
15   dim3 grid(32,32);
16   dim3 block(N/32,N/32);
17   cudaMalloc(&d_x, N*N*sizeof(float));
18   cudaMalloc(&d_y, N*N*sizeof(float));
19   for(int i=0; i<N*N; i++){
20     x[i]=42.0f;
21   }
22   cudaMemcpy(d_x,x,N*N*sizeof(float),cudaMemcpyHostToDevice)
       ;
23   copy<<<grid,block>>>(N,d_x,d_y);
24   cudaMemcpy(y,d_y,N*N*sizeof(float),cudaMemcpyDeviceToHost)
       ;
25   cudaFree(d_x);
26   cudaFree(d_y);
27   free(x);
28   free(y);
29 }
```

```
1  /*0000*/ MOV R1, c[0x0][0x28];
2  /*0010*/ @!PT SHFL.IDX PT, RZ, RZ, RZ, RZ;
3  /*0020*/ S2R R0, SR_CTAID.X;
4  /*0030*/ S2R R2, SR_TID.X;
5  /*0040*/ S2R R3, SR_CTAID.Y;
6  /*0050*/ S2R R4, SR_TID.Y;
7  /*0060*/ MOV R5, 0x4;
8  /*0070*/ IMAD R0, R0, c[0x0][0x0], R2;
9  /*0080*/ IMAD R2, R3, c[0x0][0x4], R4;
10 /*0090*/ IMAD R0, R2, c[0x0][0x160], R0;
11 /*00a0*/ IMAD.WIDE R2, R0,R5,c[0x0][0x168];
12 /*00b0*/ LDG.E.SYS R2, [R2];
13 /*00c0*/ IMAD.WIDE R4, R0,R5,c[0x0][0x170];
14 /*00d0*/ STG.E.SYS [R4], R2;
15 /*00e0*/ EXIT;
16 /*00f0*/ BRA 0xf0;
```

Fig. 2: CUDA/SASS code of matrix copy.

6 load to registers the thread and block identifiers which are used in the right hand side of the CUDA source code in lines 5 and 6. Instructions 7 to 9 in the SASS code compute the thread access positions stored in the variables in the left hand side of source code lines 5 and 6. Instruction 10 calculates the index within the brackets of source code line 7. Instructions 11 and 13 calculate the memory address for arrays $d_x$ and $d_y$ respectively. Instruction 12 performs the load access from $d_x$ while instruction 14 carries out the store access to $d_y$. Finally, instruction 15 terminates the kernel.

TABLE II: Measured/Expected values for matrix copy

| | Event | Expected | Measured | Discrepancy |
|---|---|---|---|---|
| (1) | 'DMOV' | 3,145,728 | 0 | -3,145,728 |
| (2) | inst_misc | 4,194,304 | 6,291,456 | 2,097,152 |
| (3) | inst_integer | 5,242,880 | 5,242,880 | 0 |
| (4) | inst_compute_ld_st | 2,097,152 | 2,097,152 | 0 |
| (5) | inst_control | 2,097,152 | 1,048,576 | -1,048,576 |
| (6) | Total | 16,777,216 | 14,680,064 | -2,097,152 |

As shown in the kernel invocation in line 23 of the source code, the kernel is launched with 1024x1024 threads. Each instruction is executed by all threads, which allows us to compute the number of expected instructions for each type of instruction, in order to validate it with the measurements of those instructions obtained with performance counters in the next step. Therefore, we expect the SASS code on the right to be executed 1,048,576 times, thus leading to 16,777,216 ($16 \cdot 2^{20}$) instructions. Those instructions are broken down into $3 \cdot 2^{20}$ data movement (`MOV` and `SHFL`), $4 \cdot 2^{20}$ miscellaneous (`S2R`), $5 \cdot 2^{20}$ integer (`IMAD`), $2 \cdot 2^{20}$ load/store (LDG and STG), and $1 \cdot 2^{20}$ control flow (`EXIT` and `BRA`). Note that `EXIT` acts as a safeguard following the kernel termination.

### B. First Validation Step

From the collected values we have detected several discrepancies in comparison to the expected values, as shown in Table II. For each instruction type we report the number of instructions expected based on our analysis of the SASS code, those counted with the event monitors, and the discrepancies. Note that we exclude those types for which we both expect and count zero instructions. We extract the following conclusions:
(1) Data movement instructions, as expected, are not counted at all since there is no specific event to count them.
(2) Surprisingly, the number of miscellaneous instructions measured is higher than that in the SASS code. In particular, there are 4 `S2R` in the SASS code executed $\approx 1$ million times each (1,048,576 threads), so we would expect $\approx$ 4 million `MISC` instructions counted. However, `inst_misc` reports $\approx$ 6 million `MISC` instructions, as if there were 2 additional `MISC` instructions per thread in the SASS code.
(3), (4) Integer and loads/stores are counted properly.
(5) The total number of instructions measured matches the addition of the individual types counted. However, this number is different from the total number of expected instructions.

Pre-loop, loop prolog, Loop body, Post loop

```
 1 /*0000*/       MOV R1, c[0x0][0x28];
 2 /*0010*/ @!PT SHFL.IDX PT, RZ, RZ, RZ, RZ;
 3 /*0020*/       FSETP.LT.AND P0, PT, RZ, c[0x0][0x160], PT;
 4 /*0030*/       MOV R0, RZ;
 5 /*0040*/ @!P0 BRA 0x130;
 6 /*0050*/       IMAD.MOV.U32 R2, RZ, RZ, c[0x0][0x160];
 7 /*0060*/       MOV R0, RZ;
 8 /*0070*/       IMAD.MOV.U32 R3, RZ, RZ, RZ;
 9 /*0080*/       FMUL R2, R2, c[0x0][0x160];
10 /*0090*/ @!PT SHFL.IDX PT, RZ, RZ, RZ, RZ;
11 /*00a0*/       IADD3 R4, R3.reuse, 0x1, RZ;
12 /*00b0*/       FADD R0, R0, c[0x0][0x160];
13 /*00c0*/       LOP3.LUT R3, R3, 0x1, RZ, 0xc0, !PT;
14 /*00d0*/       I2F R5, R4;
15 /*00e0*/       ISETP.NE.U32.AND P0, PT, R3, 0x1, PT;
16 /*00f0*/       MOV R3, R4;
17 /*0100*/       FSEL R0, R0, R2, P0;
18 /*0110*/       FSETP.GEU.AND P1, PT, R5, c[0x0][0x160], PT ;
19 /*0120*/ @!P1 BRA 0x90;
20 /*0130*/       S2R R6, SR_TID.X;
21 /*0140*/       S2R R3, SR_CTAID.X;
22 /*0150*/       MOV R7, 0x4;
23 /*0160*/       IMAD R6, R3, c[0x0][0x0], R6;
24 /*0170*/       IMAD.WIDE R4, R6.reuse, R7.reuse, c[0x0][0x170];
25 /*0180*/       IMAD.WIDE R2, R6, R7, c[0x0][0x168];
26 /*0190*/       LDG.E.SYS R4, [R4];
27 /*01a0*/       LDG.E.SYS R2, [R2];
28 /*01b0*/       IMAD.WIDE R6, R6, R7, c[0x0][0x178];
29 /*01c0*/       FADD R9, R4, R2;
30 /*01d0*/       FADD R0, R0, R9;
31 /*01e0*/       STG.E.SYS [R6], R0;
32 /*01f0*/       EXIT;
33 /*0200*/       BRA 0x200;
34 /*0210*/       NOP;
35 /*0220*/       NOP;
36 /*0230*/       NOP;
37 /*0240*/       NOP;
38 /*0250*/       NOP;
39 /*0260*/       NOP;
40 /*0270*/       NOP;
```

Fig. 3: SASS code of the combined example.

Hence, we need to further analyse the event counters `inst_misc`, ‘DMOV’, and `Total`. On the contrary, for `inst_integer`, `inst_control` and `inst_compute_ld_st`, since the counts we observe for both experiments in Figure 2 and Figures 3 – explained later – are precise, we consider them reliable.

**First set of Hypotheses**. From these results, we formulate the following hypotheses. The `inst_misc` monitor counts two instructions beyond those appearing in the SASS code and regarded as `MISC` according to NVIDIA's documentation [30]. We hypothesize that other instructions are counted as `MISC`:

- *Hypothesis 1a*. Either those other instructions correspond to a different category, but are counted as `MISC`.
- *Hypothesis 1b*. Or they are instructions not shown in the SASS code. After reviewing the semantics of the program in the SASS code, we verify that addresses are properly computed, data read from the source matrix and written in the destination matrix. Thus, we cannot attribute any specific operation to the potentially hidden instructions (e.g. they could be `NOP` instructions).

### C. Second Validation Step

In order to test the hypotheses above, we have performed a number of individual experiments. Each of them aims at varying the instruction counts for the different instruction types whose counters report discrepancies w.r.t. the expected values. By doing so and comparing the expected number of instructions for those instruction types against actual event counts, we expect to discern which of the formulated hypotheses is the right one in each case and, if all of them are rejected, obtain additional information to raise new informed hypotheses. For simplicity and due to space constraints, we have merged all experiments into a single one. The combined experiment contains a loop within which we can vary the number of iterations and

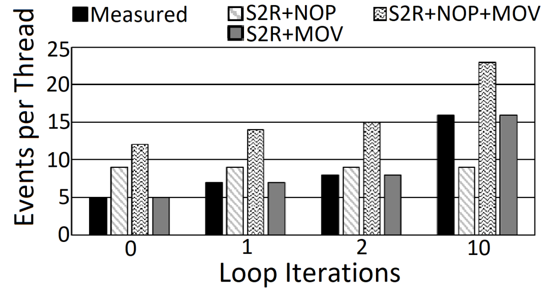


Fig. 4: `MISC` inst. counted and expected (example in Fig. 3).

hence, the number of executed instructions of each type. The SASS code of this example is shown in Figure 3. Hexadecimal numbers on the left show the instruction address. Arrows indicate the direction of the conditional branches, which in fact are predicated unconditional branches. Predicates are shown as `@!PT`, `@!P0` and `@!P1`. The program starts by executing instructions 10h-30h. When the loop is executed at least once, `BRA` at 40h is not taken and the rest of the execution continues until the `BRA` in 120h. That branch is taken for each additional iteration, thus looping in instructions 90h-120h. Whenever it is not taken, instructions from 130h until the end of the program are executed. Therefore, instructions 10h-40h and 130h-1F0h are executed exactly once. Instructions 50h-80h are executed exactly once as long as the loop iterates at least once. Instructions 90h-120h are executed as many times as the loop is intended to execute. Note that, in theory, instructions 200h-270h should not be executed since the `EXIT` instruction at address 1F0h should terminate the kernel execution. Why those instructions are part of the SASS code is not documented by NVIDIA and, in any case, they should not have any functional effect.

TABLE III: Instruction types in Figure 3.

| Event | Exp. 0 iter | Meas. 0 iter | Exp. 1 iter | Meas. 1 iter | Exp. 10 iter | Meas. 10 iter |
|---|---|---|---|---|---|---|
| ‘DMOV’ | 4,096 | 0 | 6,144 | 0 | 15,360 | 0 |
| inst_misc | 9,216 | 5,120 | 9,216 | 7,168 | 9,216 | 16,384 |
| inst_integer | 4,096 | 4,096 | 9,216 | 9,216 | 36,864 | 36,864 |
| inst_fp_32 | 3,072 | 3,072 | 7,168 | 7,168 | 34,816 | 34,816 |
| inst_compute_ld_st | 3,072 | 3,072 | 3,072 | 3,072 | 3,072 | 3,072 |
| inst_control | 3,072 | 2,048 | 2,048 | 1,024 | 11,264 | 10,240 |
| inst_bit_convert | 0 | 0 | 1,024 | 1,024 | 10,240 | 10,240 |
| Total | 26,624 | 17,408 | 37,888 | 28,672 | 120,832 | 111,616 |

For runs with 0, 1, 2 and 10 iterations, table III shows that `inst_fp_32`, `inst_control`, `inst_compute_ld_st`, `inst_integer` and `inst_bit_convert` event counters match exactly the number of instructions executed. For instance, in the case of 1 loop iteration, where all instructions in the SASS code are executed exactly once, one would expect 9 `INT`, 7 `FP32`, 3 `LDST` and 1 `CONV` instructions (see Table I) for each of the 1,024 threads, which matches exactly the corresponding event counters. Also for `inst_control`, when the number of iterations is 0, the `BRA` at 40h is taken, and then only the `EXIT` instruction at the end is executed (the `BRA` at 200h is

never executed). When the number of iterations is $N$, $N > 0$, then the BRA at 40h is not taken, the BRA at 120h is taken $N-1$ times and not-taken once, the EXIT and the BRA at the end are executed also once. Overall, we expect $N+1$ (BRA+EXIT) instructions.

**Assessing hypotheses 1a and 1b**. In order to determine the source of the unexpected MISC instructions, we build upon the example in Figure 3. In particular, consider the case with 1 loop iteration for simplicity, so that all instructions are executed exactly once. The event counter indicates that there are 7 MISC instructions per thread. In the SASS code, we can identify 2 S2R and 7 NOP instructions. Thus, differently to the previous example, where we were expecting more events than the ones provided by the event monitor, this time the monitor is undercounting. In order to have additional information, we also include the result of executing the loop 10 times, where we still would expect 9 MISC instructions per thread, since S2R and NOP instructions are outside the loop, but the MISC counter then counts 16 instructions. Thus, by having 9 additional iterations, the counter increases by 9. This behaviour also holds for other numbers of loop iterations. We conclude that:

(1) Exactly one instruction in the loop (90h-120h) is counted as MISC. If we discard all INT, FP32 and CONV instructions in the loop, which we regarded as precisely counted by their corresponding event counters, we get only a MOV instruction. Thus, we consider that MOV instructions are counted as MISC.

(2) We revisit the example of the matrix copy in Figure 2, where we have exactly 4 S2R and 2 MOV instructions per thread. If we analyse the MISC counter in that case, which overcounted 2 instructions per thread, we realize it is fully precise if we include MOV instructions. Therefore, we conclude that both, S2R and MOV instructions are counted as MISC.

(3) We compare MISC measured against theoretical MISC (S2R+NOP), MISC and MOV (S2R+NOP+MOV), and only S2R+MOV, see Figure 4.

Overall, we conclude that although MOV instructions are classified as data movement instructions in [30], they are effectively counted as MISC instructions. Instead, NOP instructions, classified as MISC, are not counted. However, all those NOP instructions are exactly after the last BRA instruction, thus not executed in practice. Hence, it remains unknown whether MISC counts executed NOP instructions or it never counts them.

**Observation 1**: inst_misc counts S2R and MOV and it remains unknown whether it counts executed NOP instructions.

**Second set of Hypotheses**. We formulate two hypotheses, as consequence of the investigation of hypotheses 1a and 1b;

- *Hypothesis 2a*. MISC does not count NOPs, which matches the fact that, so far, those NOPs found in the SASS code have not been counted in any experiment.
- *Hypothesis 2b*. MISC counts NOP instructions only if effectively executed, which would be in line with NVIDIA documentation [30] for executed NOPs.

```
1  /*0000*/ MOV R1,
2           c[0x0][0x28];
3  /*0010*/ @!PT SHFL.IDX PT,
4           RZ, RZ, RZ, RZ;
5  /*0020*/ S2R R4,SR_CTAID.X;
6  /*0030*/ S2R R2,SR_TID.X;
7  /*0040*/ MOV R5, 0x4;
8  /*0050*/ NOP;
9  /*0060*/ IMAD R4, R4,
10          c[0x0][0x0],R2;
11 /*0070*/ IMAD.WIDE R2, R4,
12          R5, c[0x0][0x168];
13 /*0080*/ LDG.E.SYS R2,
14          [R2];
15 /*0090*/ IMAD.WIDE R4, R4,
16          R5, c[0x0][0x170];
17 /*00a0*/ STG.E.SYS [R4],R2;
18 /*00b0*/ EXIT;
19 /*00c0*/ BRA 0xc0;
20 /*00d0*/ NOP;
21 /*00e0*/ NOP;
22 /*00f0*/ NOP;
```

```
1  /*0000*/ MOV R1,
2           c[0x0][0x28];
3  /*0010*/ @!PT SHFL.IDX PT,
4           RZ, RZ, RZ,RZ;
5  /*0020*/ S2R R4,SR_CTAID.X;
6  /*0030*/ S2R R2,SR_TID.X;
7  /*0040*/ MOV R5, 0x4;
8  /*0050*/ NOP;
9  /*0060*/ NOP;
10 /*0070*/ NOP;
11 /*0080*/ IMAD R4, R4,
12          c[0x0][0x0],R2;
13 /*0090*/ IMAD.WIDE R2, R4,
14          R5, c[0x0][0x168];
15 /*00a0*/ LDG.E.SYS R2,
16          [R2];
17 /*00b0*/ IMAD.WIDE R4, R4,
18          R5, c[0x0][0x170];
19 /*00c0*/ STG.E.SYS [R4],R2;
20 /*00d0*/ EXIT;
21 /*00e0*/ BRA 0xe0;
22 /*00f0*/ NOP;
```

Fig. 5: SASS code of two examples with NOP instructions.

```
1  /*0000*/       MOV R1, c[0x0][0x28];
2  /*0010*/ @!PT  SHFL.IDX PT, RZ, RZ, RZ, RZ;
3  /*0020*/       S2R R6, SR_CTAID.X;
4  /*0030*/       S2R R0, SR_TID.X;
5  /*0040*/       MOV R7, 0x4;
6  /*0050*/       IMAD R6, R6, c[0x0][0x0], R0;
7  /*0060*/       IMAD.WIDE R2,R6.reuse,R7.reuse,c[0x0][0x168];
8  /*0070*/       IMAD.WIDE R4, R6, R7, c[0x0][0x170];
9  /*0080*/       LDG.E.SYS R2, [R2];
10 /*0090*/       LDG.E.SYS R4, [R4];
11 /*00a0*/       IMAD.WIDE R6, R6, R7, c[0x0][0x178];
12 /*00b0*/       FADD R0, R2, R4;
13 /*00c0*/       STG.E.SYS [R6], R0;
14 /*00d0*/       EXIT;
15 /*00e0*/       BRA 0xe0;
16 /*00f0*/       NOP;
```

Fig. 6: SASS code for the vector addition in Global Mem.

### D. Third Validation Step

**Assessing hypotheses 2a and 2b**. To assess whether NOP instructions before the final BRA are counted under inst_misc, we have performed several experiments but, for the sake of space limitations, we present the simplest ones solving the unknown. In particular, we manipulated the source code of the example to enforce the use of NOP instructions, which do not have any functional impact.

As shown in Figure 5, the SASS code of these programs includes NOP instructions before and after the final BRA instruction. According to the observations before, MISC must be at least 4 per thread. In particular, MISC must count the 2 S2R and the 2 MOV instructions, and exclude the NOP(s) after the final BRA. If NOP instructions before the final BRA were not counted, we would obtain in both cases that MISC is exactly 4. However, in the example in the left figure MISC is 5, whereas in the right figure MISC is 7, thus including those 1 and 3 NOPs before the final BRA respectively.

TABLE IV: Event counts for the vector addition benchmarks.

| Event | Gmem | Smem 0 sync | VSmem 0 sync | Smem 1 sync | Smem 2 sync |
|---|---|---|---|---|---|
| inst_misc | 4,096 | 4,096 | 4,096 | 5,120 | 6,144 |
| inst_integer | 4,096 | 5,120 | 5,120 | 5,120 | 5,120 |
| inst_fp_32 | 1,024 | 1,024 | 1,024 | 1,024 | 1,024 |
| inst_compute_ld_st | 3,072 | 6,144 | 9,216 | 8,192 | 9,216 |
| inst_control | 1,024 | 1,024 | 1,024 | 1,024 | 1,024 |
| Total | 13,312 | 17,408 | 20,480 | 20,480 | 22,528 |

**Observation 2**: `inst_misc` also counts `NOP` instructions if executed (thus excluding those after the final `BRA`).

### E. Assessment on Complex Code

In order to further assess our findings, we have evaluated several benchmarks, as well as kernels extracted from the Rodinia benchmark suite [18], [19], a widely used benchmark suite for GPUs. In this paper, we report the results we obtained on benchmarks, which suffices for illustrative purposes. In particular, we analyse a vector addition benchmark whose SASS code has no loops and the only predicated instruction is a `DMOV` instruction (hence not counted). In Figure 6 we analyse the global memory (GMEM) incarnations of that same benchmark, the other variants (shared memory (Smem) with variable synchronization (sync) points) are not listed due to space constraints.

Event counts are shown in Table IV, with each benchmark executing 1,024 threads. Hence, instructions per thread can be matched by dividing by 1,024 the values in the table. For instance, `MISC` instructions count 2 `MOV` and 2 `S2R` instructions in all cases, plus 1 and 2 `BAR` instructions in the two last cases respectively.

In all experiments, the observations we made as part of the application our process hold, hence, event monitor reads match in all cases the (new) expected values:
(i) `inst_misc` includes `MOV` instructions as well as `MISC` instructions (excluding `NOP`s after the final `BRA`);
(ii) `inst_integer`, `inst_control`, `inst_fp_32`, `inst_bit_convert`, and `inst_compute_ld_st`, count their expected instruction types precisely;
(iii) And total instructions match the addition of the other counters.

Overall, the large set of tests conducted for the validation of the event monitors of the Xavier, the most relevant subset of which is presented in this paper, reveals that a methodology like the one we propose is a prerequisite for a reliable use of the even monitors of GPUs in the timing V&V process.

## V. XILINX ZYNQ ULTRASCALE+

We also applied our approach to another architecture from a different vendor, the Xilinx Zynq UltraScale+, and in particular to the CPUs in the Application Processor Unit cluster. The ARM Cortex-A53 CPUs [1] contains more than 63 events from which we focus on a subset, selecting again the number of instructions executed, as well as the number of memory related events. A notable difference between this platform and the Xavier is the lack of event counters breaking down the arithmetic instruction categories. Instead, there are only PMCs about memory operations, branches and total instructions. On the other hand, more events are provided regarding the microarchitectural events taking place. It is important to note that according to the ARM Cortex-A53 CPU technical reference manual [1] and the ARMv8-A architecture reference manual [3], the event values are not expected to be completely accurate, and that the microarchitectural implementation may introduce small absolute variations in the actual number of the events reported due to pipeline effects. For this reason, we perform our validation in rough numbers, reporting only big discrepancies whenever found.

### A. Controlled Experiment

Table V lists the selected PMCs for validation. For the validation experiment, we use a bare-metal configuration in order to guarantee no interference from the operating system, something that was not possible in the Xavier, since the use of the GPU can only be supported by a driver within the operating system. As rbe we selected the same application presented in the previous section and used in the validation of the NVIDIA platform, matrix copy, which we compile with the ARM gcc compiler. For the PMC readings we directly read their values from their memory mapped locations. We disable the hardware prefetcher in to force a more predictable behaviour.

In Figure 7 we show the C code implementing the benchmark, followed by its assembly form. The memory instructions which are of our interest are shown in bold, and load operations are shown in italics to ease load and store identification. In the assembly code we notice again different code sections. The main loop of the benchmark is between the lines 4-24.

Load instructions in lines 5, 11, 16 and 21, and the store instruction in line 19 serve the purpose of loading or updating the loop index, which is located in a fixed memory location so the instructions will cause 5 cache hits (4 load hits, 1 store hit) per iteration.

The load instruction in line 9 loads the data from the source array ($from[]$), which may cause cache misses when crossing cache line boundaries, and the store instruction in line 14 stores the value loaded from the source array into the destination array ($to[]$), which may also cause cache misses when crossing cache line boundaries.

Lines 20-24 perform the out of boundaries check to determine if more iterations are needed or the algorithm has ended. Knowing the number of total iterations and the assembly representation, we are able to tightly estimate the number of expected instructions and events for each of the selected PMCs.

### B. Assessment

Table VI shows the obtained values from the PMCs, together with their expected values. As a first validation step, we validate the accuracy of the instructions that we know. We perform 512K iterations in which we access 5 loads and 2 store instructions. The number of load and stores are as expected (rows C and D). Likewise the number of L1 cache accesses and total memory operations match their expected value (rows B and F). The number of total instructions within the loop is 21, resulting in a total of 10.5M instructions executed (row E).

The total memory footprint of the application is 4MB, as it copies a 2MB array into a new location. Since no data is reused, every read and write accessing a new cache line of data and L2 caches is expected to generate a cache miss, whereas the remaining accesses to those cache lines are expected to

TABLE V: Instruction types used in the analysis for the Xilinx Ultrascale+ ARM Cortex-A53 CPUs [1].

| Event | Official Description | Instruction Types counted [3] |
|---|---|---|
| L1D_CACHE_REFILL | Level 1 data cache refill | Loads and stores missing L1 |
| L1D_CACHE | Level 1 data cache access | Loads and stores |
| LD_RETIRED | Instruction architecturally executed, Condition code check pass, load | Loads |
| ST_RETIRED | Instruction architecturally executed, Condition code check pass, store | Stores |
| INST_RETIRED | Instruction architecturally executed | All instructions |
| MEM_ACCESSES | Data memory access | Loads and stores |
| L2D_CACHE | Level 2 data cache access | Loads, stores and instructions missing L1 caches |
| L2D_CACHE_REFILL | Level 2 data cache refill | Loads, stores and instructions missing L2 |
| BUS_ACCESS | Bus access | Bus acccesses from loads and stores missing the last level cache |

```
1 #define SIZE 2*1024*1024/sizeof(int)    /*512K*/
2 int from[SIZE], to[SIZE];
3
4 for (int i = 0; i < SIZE; i++) {
5     to[i] = from[i];
6 }
```

```
1  /*3358*/   add    x0, x29, #0x400, lsl #12
2  /*335c*/   str    wzr, [x0,#24]
3  /*3360*/   b      33a4 <main+0xd4>
4  /*3364*/   add    x0, x29, #0x400, lsl #12
5  /*3368*/   ldrsw  x0, [x0,#24]
6  /*336c*/   lsl    x0, x0, #2
7  /*3370*/   add    x1, x29, #0x200, lsl #12
8  /*3374*/   add    x1, x1, #0x18
9  /*3378*/   ldr    w2, [x1,x0]
10 /*337c*/   add    x0, x29, #0x400, lsl #12
11 /*3380*/   ldrsw  x0, [x0,#24]
12 /*3384*/   lsl    x0, x0, #2
13 /*3388*/   add    x1, x29, #0x18
14 /*338c*/   str    w2, [x1,x0]
15 /*3390*/   add    x0, x29, #0x400, lsl #12
16 /*3394*/   ldr    w0, [x0,#24]
17 /*3398*/   add    w0, w0, #0x1
18 /*339c*/   add    x1, x29, #0x400, lsl #12
19 /*33a0*/   str    w0, [x1,#24]
20 /*33a4*/   add    x0, x29, #0x400, lsl #12
21 /*33a8*/   ldr    w1, [x0,#24]
22 /*33ac*/   mov    w0, #0x7ffff  //#524287
23 /*33b0*/   cmp    w1, w0
24 /*33b4*/   b.ls   3364 <main+0x94>
25 /*33b8*/   adrp   x0, 1e000 <__exidx_end>
26 /*33bc*/   add    x0, x0, #0x3c0
27 /*33c0*/   ldr    w2, [x0]
28 /*33c4*/   adrp   x0, 14000 <zeroes.5791+0x1c0>
29 /*33c8*/   add    x0, x0, #0x450
30 /*33cc*/   ldr    x0, [x0]
31 /*33d0*/   mov    x1, x0
32 /*33d4*/   mov    w0, w2
```

Fig. 7: C/ARM Assembly code of matrix copy in the Zynq.

TABLE VI: Measured/Expected values for matrix copy

| Event | Expected | Measured | Discrepancy |
|---|---|---|---|
| (A) L1D_CACHE_REFILL | 64K | 65566 | 0 |
| (B) L1D_CACHE | 3.5M | 3670319 | 0 |
| (C) LD_RETIRED | 2.5M | 2621612 | 0 |
| (D) ST_RETIRED | 1M | 1048626 | 0 |
| (E) INST_RETIRED | 10.5M | 11010313 | 0 |
| (F) MEM_ACCESSES | 3.5M | 3670057 | 0 |
| (G) L2D_CACHE | 64K | 130772 | 64K |
| (H) L2D_CACHE_REFILL | 64K | 65559 | 0 |
| (I) BUS_ACCESS | 352K | 360309 | 0 |

hit due to spatial locality. Given that our application has a sequential access pattern, that the cache line size is 64B in both the data and L2 caches, and that the data type used by the application is 4B, we expect that both, source loads and destination store instructions produce 1 miss followed by 15 hits for each cache line. Therefore, we expect 32K read misses and 32K write misses out of a total of 512K accesses of each type. Note that only a load and a store instruction may miss the caches each iteration, as explained in section V-A. Note that the code is small enough to fit in the instruction cache after the first loop iteration. Thus, instructions should cause few (below 10) L2 cache misses, so L2 misses roughly correspond to data misses only. The measured misses in each cache are 64K as expected (rows A and H).

The bus access counter (row I) counts the number of bus transactions issued, which are caused by L2 load misses, L2 store misses, or L2 dirty evictions. A total of 32K load misses and 32K store misses are expected, while only 24K dirty evictions are expected, as 512KB -a fourth- of the data stored will still remain in the L2 when the execution finishes. The total amount of lines issued by the L2 to the bus is 88K, however the bus width is 16B [1], so each line is split in 4 transactions, causing a total of 352K bus accesses. As shown in row I this counter is precise.

Finally, L2 accesses (row G) should be either counting the 32K load misses and 32K store misses (so 64K accesses), or include also the almost 32K dirty evictions (so 96K accesses). However, it counts 128K accesses. According to the ARM Cortex-A53 CPU technical reference manual [1], L1 load misses sent to L2 are served through a 16B bus, whereas write operations use a 32B bus. We have leveraged this information to guess whether they influenced the number of L2 accesses counted, but no reasonable combination led to 128K accesses. In fact, we have conducted additional experiments (e.g. only read misses) and in all cases the number of L2 accesses has doubled the number of L1 data cache misses. However, we could not formulate any reasonable hypothesis to justify this behavior. In fact, our past work on ARM-based platforms already revealed mismatches between event counters obtained and values expected [21].

**Observation 3**: In the absence of any evidence on the existence of additional L2 cache access activity, we regard `L2D_CACHE` as unreliable for timing validation purposes.

TABLE VII: Instruction types for TX2.

| Event | Expected | Measured | Discrepancy |
|---|---|---|---|
| DMOV | 1,048,576 | 0 | -1,048,576 |
| MISC | 4,194,304 | 5,242,880 | 1,048,576 |
| INT | 15,728,640 | 15,728,640 | 0 |
| LDST | 2,097,152 | 2,097,152 | 0 |
| CTRL | 2,097,152 | 1,048,576 | -1,048,576 |
| Total | 25,165,824 | 24,117,248 | -1,048,576 |

## VI. NVIDIA Jetson TX2

For the Jetson TX2 GPU, we build our study incrementally over the one for Xavier using the same control experiments and event counts, since both GPUs are instantiations of the very similar NVIDIA's Pascal and Volta GPU families respectively, which however differ in their SASS representation since they implement different GPU ISAs. Whereas the Pascal family included the `XMAD` integer instruction (integer short multiply and add) and the `IMAD` integer instruction (integer multiply and add), and so TX2 may use these instructions, Volta family, instead, only includes the `IMAD` instruction.

The code of the vector addition with global/shared memory and varying number of synchronization barriers is not shown for the TX2 for space constraints.

In the Pascal architecture used by the TX2, the kernel prologue is only one instruction. Similar to the Volta case, the next four instructions obtain the thread identifiers, but the actual index computation is completely different and longer. Then, the indexed array load performed in the lines 17-19 and store in the 20-22 look very similar. The kernel epilogue also contains an EXIT followed by a safeguard branch as we discovered in the previous section.

Table VII shows the expected counter results without taking into account the findings from our experiments with the Xavier SoC. We can see that there is also a discrepancy between the measurements and the expected values. However, if we apply our observations from Xavier these discrepancies can be explained. In particular, `MOV` instructions are also counted as `MISC` instructions in the TX2, and the final `BRA` instruction following the `EXIT` is not executed.

## VII. Related Work

In the CPU domain, we can find a handful of prior works that build on performance counters for software timing estimation. Paulisch et al. [29] build on performance counter events to create an analysis and runtime monitoring solution for limiting task contention in multicore CPU architectures. In [12] an ILP-based contention model is proposed for the AURIX automotive microcontroller building on the performance counters available on that platform. In [23] authors use performance counters in the CPU of multicore systems for WCET estimation using probabilistic measurement-based timing analysis. [22] also used performance counters for WCET estimation of CPU tasks on multicore systems proposing a method to select the performance counter with highest contribution and a forecast model to predict execution time under unseen configurations. Authors in [41], [36] study the variability caused due to non-deterministic performance counter implementations in CPUs, without analyzing whether values are as expected, which is instead the target of our work. Nevertheless, in these works, authors observe non-null but relatively low variability across measurements.

Several works [5], [31] have focused on automotive platforms featuring GPUs such as NVIDIA's TX1 and TX2. Some of them have discovered undocumented features of those hardware platforms like the scheduling policy [5] or exposed mismatches in the software documentation regarding blocking or asynchronous behaviour of CUDA API calls [39]. However, none of these works studies event monitors, whose behaviour and documentation mismatches we expose in this work. To our knowledge, there is no other work in the real-time literature which considers GPU performance counters.

In addition, [31], [10] present benchmarking and platform characterization studies of automotive platforms. Regarding timing modelling of GPUs, in the literature we can find the seminal works [9], [7], [8]. The first two papers build on a simulated GPU for WCET estimation, while our work uses a real GPU for event monitor validation. On the other hand, [8] relies on end-to-end measurements on a real-GPU platform for WCET estimation and timing analysis, not validation of performance counters.

## VIII. Conclusions

The increasing need for the adoption of high-performance hardware to execute performance-demanding critical real-time software poses stringent V&V constraints on those systems. In particular, a number of activities, such as quota monitoring and software debugging, need to build upon event monitors for efficiency reasons. Trustworthiness of event monitors is a precondition for the reliability of the critical V&V processes built atop. Following the proposed methodology to validate the event monitors via the analysis of GPUs for the automotive domain (NVIDIA Xavier and TX2) and multicores for the railway and avionics domains (Xilinx Zynq UltraScale+), we show that even some of the most basic event counters may fail to match specifications. Our methodology allows, in many cases, discern what they count, which is the basis to build reliable processes for critical real-time systems atop. In particular, by performing specific empirical tests, we are able to accept or reject plausible hypotheses and collect evidence supporting our conclusions. We show how some instructions are misclassified, some others are counted in non-obvious ways, and some events may fully mismatch expectations. However, once this information is obtained and verified empirically, validated event counters in complex hardware can be used for the V&V of critical real-time systems.

## Acknowledgement

This work has been partially supported by the Spanish Ministry of Economy and Competitiveness (MINECO) under grant TIN2015-65316-P, the SELENE European Union's Horizon 2020 (H2020) research and innovation programme under grant agreement No 871467, the SuPerCom European Research Council (ERC) project under the European Union's Horizon 2020 research and innovation programme (grant agree-

ment No. 772773) and the HiPEAC Network of Excellence. MINECO partially supported Jaume Abella under Ramon y Cajal postdoctoral fellowship (RYC-2013-14717), Enrico Mezzetti under Juan de la-Cierva-Incorporacion postdoctoral fellowship (IJCI-2016-27396), and Leonidas Kosmidis under Juan de la Cierva-Formación postdoctoral fellowship (FJCI-2017-34095).

## References


[1] ARM Cortex-A53 MPCore Processor, 2014.

[2] CUDA Binary Utilities, 2018.

[3] ARMv8 Reference Manual v8.5 EAC, 2019.

[4] Sergi Alcaide, Leonidas Kosmidis, Hamid Tabani, Carles Hernandez, Jaume Abella, and Francisco J Cazorla. Safety-related challenges and opportunities for gpus in the automotive domain. *IEEE Micro*, 38(6):46–55, 2018.

[5] T. Amert et al. GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed. In *RTSS*, 2017.

[6] ARM. ARM Expects Vehicle Compute Performance to Increase 100x in Next Decade, 2015. https://www.arm.com/about/newsroom/arm-expects-vehicle-compute-performance-to-increase-100x-in-next-decade.php.

[7] K. Berezovskyi et al. Makespan computation for GPU threads running on a single streaming multiprocessor. In *ECRTS*, 2012.

[8] K. Berezovskyi et al. Measurement-based probabilistic timing analysis for graphics processor units. In *ARCS*, 2016.

[9] A. Betts and A.F. Donaldson. Estimating the WCET of GPU-Accelerated Applications Using Hybrid Analysis. In *ECRTS*, 2013.

[10] N. Capodieci et al. Deadline-based scheduling for GPU with preemption support. In *RTSS*, 2018.

[11] D. Dasari et al. Response time analysis of cots-based multicores considering the contention on the shared memory bus. In *IEEE TrustCom*, 2011.

[12] E. Díaz et al. Modelling multicore contention on the aurix tc27x. In *DAC*, 2018.

[13] Enrico Mezzetti et al. High-integrity performance monitoring units in automotive chips for reliable timing v&v. *IEEE Micro*, 38(1):56–65, 2018.

[14] F. J. Cazorla et al. Reconciling time predictability and performance in future computing systems. *IEEE Design & Test*, 35(2):48–56, 2018.

[15] J. Jalle et al. Contention-aware performance monitoring counter support for real-time mpsocs. In *SIES*, 2016.

[16] Noriaki Suzuki et al. Coordinated bank and cache coloring for temporal protection of memory accesses. In *CSE*. IEEE Computer Society, 2013.

[17] Renato Mancuso et al. Real-time cache management framework for multi-core architectures. In *RTAS*. IEEE Computer Society, 2013.

[18] S. Che et al. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009.

[19] S. Che et al. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. *IISWC*, 2010.

[20] Federal Aviation Administration, Certification Authorities Software Team (CAST). *CAST-32A Multi-core Processors*, 2016.

[21] G. Fernandez et al. Consumer electronics processors for critical real-time systems: a (failed) practical experience. In *ERTS*$^2$, 2018.

[22] D. Griffin et al. Forecast-based interference: Modelling multicore interference from observable factors. In *RTNS*, 2017.

[23] F. Guet et al. Probabilistic analysis of cache memories and cache memories impacts on multi-core embedded systems. In *SIES*, 2016.

[24] H. Yun et al. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *RTAS*, pages 155–166. IEEE Computer Society, 2014.

[25] International Organization for Standardization. *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.

[26] L. Liu et al. A software memory partition approach for eliminating bank-level interference in multicore systems. In *PACT*, pages 367–376. ACM, 2012.

[27] F. Mazzocchetti et al. Performance analysis and optimization of automotive gpus. In *SBAC-PAD*, 2019.

[28] H. T. Nagle et al. Microprocessor testability. *IEEE Transactions on Industrial Electronics*, 36(2):151–163, May 1989.

[29] J. Nowotsch et al. Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement. In *ECRTS*, 2014.

[30] NVIDIA. CUDA toolkit documentation. volta instruction set, 2019.

[31] N. Otterness et al. An evaluation of the nvidia tx1 for supporting real-time computer-vision workloads. In *RTAS*, 2017.

[32] P. K. Valsan et al. Taming non-blocking caches to improve isolation in multicore real-time systems. In *RTAS*, pages 161–172, 2016.

[33] Xing Pan and Frank Mueller. Controller-aware memory coloring for multicore real-time systems. In *SAC*, pages 584–592. ACM, 2018.

[34] R. Pellizzoni et al. Worst case delay analysis for memory interference in multicore systems. In *DATE*, 2010.

[35] H. Tabani et al. Assessing the adherence of an industrial autonomous driving framework to iso 26262 software guidelines. In *DAC*, page 9, 2019.

[36] V. Weaver et al. Non-determinism and overcount on modern hardware performance counter implementations. In *ISPASS*, pages 215–224, 2013.

[37] Adam West. NASA Study on Flight Software Complexity. Final Report. Technical report, NASA, 2009.

[38] Xilinx. Rockwell Collins Uses Zynq UltraScale+ RFSoC Devices in Revolutionizing How Arrays are Produced and Fielded: Powered by Xilinx, 2019.

[39] Ming Yang et al. Avoiding Pitfalls when Using NVIDIA GPUs for Real-Time Tasks in Autonomous Systems. In *ECRTS*, 2018.

[40] H. Yun et al. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *RTAS*, 2013.

[41] D. Zaparanuks et al. Accuracy of performance counter measurements. In *ISPASS*, pages 23–32, 2009.