In 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2021), September 07–10, 2021, Västerås, Sweden.

Automatic Platform-Independent Monitoring and Ranking of Hardware Resource Utilization

Shamoona Imtiaz[1], Jakob Danielsson[1], Moris Behnam[1], Gabriele Capannini[1], Jan Carlson[1], Marcus Jägemar[2]
[1] Mälardalen University, Västerås, Sweden, [2] Ericsson AB, Stockholm, Sweden
[1]first.last@mdh.se, [2]first.last@ericsson.com

***Abstract*—In this paper, we discuss a method for automatic monitoring of hardware and software events using performance monitoring counters. Computer applications are complex and utilize a broad spectra of the available hardware resources, where multiple performance counters can be of significant interest to understand. The number of performance counters that can be captured simultaneously is, however, small due to hardware limitations in most modern computers. We suggest a platform independent solution to automatically retrieve hardware events from an underlying architecture. Moreover, to mitigate the hardware limitations we propose a mechanism that pinpoints the most relevant performance counters for an application's performance. In our proposal, we utilize the Pearson's correlation coefficient to rank the most relevant performance counters and filter out those that are most relevant and ignore the rest.**

I. Introduction

Due to modern trends towards real-time data acquisition, inter-connectivity, data exchange and automation, *Industry 4.0* has revolutionised the industrial technology into cyber physical systems (CPS), Internet of things (IoT) and cloud computing. While bringing improved functioning, enhanced communication capabilities and shared services, this digital transformation has also put an increased pressure on engineers and system administrators. For them to keep such infrastructure functional, efficient, reliable and secure, it is more than ever required to conduct systematic health checks of computer systems and apply performance monitoring routines. A good knowledge of hardware resource demand and utilization by the hosted applications would facilitate the engineers, system administrators and auditors to ensure the Quality of Service (QoS) and security of IT infrastructure from undesired use.

The hardware resources required by an executing application may differ over the time. The demand could be for dedicated and/or shared resource(s) which is on discrete disposal, time scheduled [1] or managed through isolation techniques [2]. Observing the resources utilization can reveal a distinctive behaviour of the application and can be used to tune the quality assurance process. Furthermore, in-depth analysis of performance monitoring data can ensure that system is performing as it is expected and can capture the execution profile of an application.

Monitoring of system performance can be categorised into processor utilization, disk activity, memory usage and network usage. Modern computers have performance monitoring units, responsible for monitoring the micro-architectural events. There are on-chip hard-wired special sets of registers known as performance monitoring counters (PMCs). The type and number of micro-architectural events are absolutely dependent on the underlying architecture and so is the number of PMCs. Regardless of the architectural variations from platform to platform there are events which are consistently available between many models, but this number is quite low and the terminology of event names are not identical across platforms. In these cases, the operator is bound to rely on information coming directly from vendor. Based on specific architectural knowledge, the PMCs can be configured to record hardware event metrics, but the limited number of physical counters bound the number of events that can be monitored simultaneously.

978-1-7281-2989-1/2$31.00

Performance monitoring units (PMUs) are not only available for CPUs, but also for other components of the computer such as GPUs, network interface cards (NICs), network switches etc [3]. By using these PMCs, micro-architectural events can be monitored for resources in the processor pipeline, such as the branch predictor unit (BPU), internal memory events, off-core events, network resource utilization, network problem etc even for the different components in parallel. The current state of modern computers enables us to precisely trace an applications' resource-usage at run-time. In this paper, we attempt to tackle the two following problems:

Application execution. An application typically displays an exceptionally complex execution trace and will utilize several resources simultaneously. Due to the complex execution trace, it is difficult to assess what resources are most relevant to the application's performance. Some applications display a performance that is closely tied to specific resources. We call occurrence *resource-dependency* and can be critical to understand when designing a system.

Performance monitoring counters. The number of available PMCs is limited compared to the number of PMU events that can be observed so that it is difficult to assess which events are important to monitor for a given application.

Other researchers have employed performance counters for various purposes such as monitoring hardware capacity, application performance, system health-check, and for detection purposes. Many of these studies are limited to a static and pre-selected set of hardware events. Jägemar et al. [4] proposed a service associated with CPU scheduler for an improved QoS through performance monitoring counters' measurements. Danielsson et al. [2] used performance monitoring counters to identify resource dependence of application in a multi-core system.

In this paper, we continue the topic of automatic detection

of an application's resource dependency, i.e., how much an application's performance depends on a specific resource. Our paper presents a new approach that monitors all available PMU events (both software and hardware) and builds resource-dependency profiles for the system applications. Our approach presents a holistic approach to measure and rank what PMU events are most closely tied to an application's performance. We present our contributions as follows:

- A cross-platform method to monitor hardware resource usage by utilizing the Performance Monitoring Unit (PMU) for large sets of performance events.
- A measurement approach to distinguish what hardware- and software-resources have the highest impact on an application's performance for the large-scale performance counter event sets.

We have structured the paper as follows. We start by providing a technical background to easily understand the technical scope and contribution made through this work in Section II. Next, Section III presents our approach to achieve the goal of the study as well as giving a theoretical definition of our work. Our implementation details are described in Section IV and the experimental setup in Section V. We discuss our results in Section VI and then present our conclusions in Section VII. Finally, we relate our work to the state-of-the-art in Section VIII and the anticipated future work concludes the paper in Section IX.

II. Background

In this section we describe the PMU and discuss the differences between counters and events. We also discuss application performance in a typical computer and how an application's performance is related to certain PMU events.

A. Performance Monitoring

The concept of performance varies for different applications depending on their primary objective. For example, in network applications, performance is usually measured in number of packets sent per second whereas image processing applications use the frames per second metric. These performance metrics can in-turn depend on other hardware-related metrics such as *utilization and saturation* for memory and CPU, *Operation rate and operation latency* for file systems, *disk utilization and response time* for disks, *throughput, connections, error, TCP re-transmits and TCP out-of-order packets* for networks [5].

Computers perform tasks on the basis of a sequence of instructions. In a classic Reduced Instruction Set Computer (RISC) pipeline one *Instruction* is processed in one cycle, as shown in Figure 1. In here the instruction goes through stages from *Instruction Fetch* to *Write Back*. Modern computers

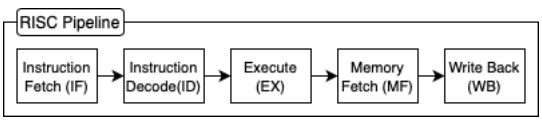

Fig. 1. RISC Pipeline for a Single Instruction

implement this RISC pipeline for instruction-level parallelism to increase processor throughput. The classic RISC pipeline splits the execution of an instruction into five stages that are ideally able to work in parallel on different instructions. An instruction begins with entering the first stage *Instruction Fetch (IF)*. The instruction will then move to the second stage *Instruction Decode (ID)* once completing the IF stage, and another instruction will enter the IF stage. The instruction is completed once it has passed the write-back stage and is then marked as retired. It is, therefore, often preferable to have a high number of instructions retired for a given application since it indicates that the application is performing a lot of work. However, in case of application(s) executing *busy-wait loops*, *Instruction Retired* as a performance metric is not appropriate [6]. For example, in applications using sensors and actuators, actuators usually check the state of sensors which perform a busy-wait loop: this results in a large number of *Instructions Retired* even if the perceived performance still is low. Therefore, It is important to define the metric for performance based on use case.

B. Performance Monitoring Unit

Modern computers have special built-in hardware in the form of registers for performance monitoring. PMUs contain model specific registers (MSRs) those can be configured to monitor events. These hard-wired registers are also called performance monitoring counters (PMCs) [5] [7] [8]. Each core has its own set of counters. These counters count the number of occurrences of a certain event during a specific time-interval.

PMC's are grouped into fixed-function counters and flexible-function counters, where fixed-function counters are hard coded and flexible-function counters can be programmed to monitor any type of event. The number of available performance counters varies depending on the hardware architecture, for instance a typical Intel processors contain 3 fixed-function and 4 flexible-function counters per PMU [7]. The event is an observable activity, state or signal whose occurrence can be from different sources such as hardware, software, kernel etc [5]. One advantage of using PMCs is negligible overhead of data extraction [9] for micro architectural events like branch instructions retired, mis-predicted branches, cache hits/misses or floating point operations. Usually the PMCs are implemented through processor specific codes. These codes along with other attributes of the events are provided by vendor(s) in JSON files which is arch event definition file.

When an event occurs it generates data that can further be utilized for statistical analysis as a metric or to generate an alert. These metrics are result of evaluation or monitoring processes and can be used by technicians for system tuning and detection of faults. Events such as execution-time, application memory-footprint size, memory-latency, and error status can also present important insights. Events those are present over majority of platforms are called architectural and events those are model-specific are called non-architectural events.

C. Perf and PAPI

Perf is a performance analysis tool and the official Linux profiler for both kernelspace and userspace. Perf was originally

developed for the monitoring of PMCs but evolved into a tool capable of tracing kernel activities too [5]. Perf uses processor specific raw hardware descriptors for the PMC events. These codes can be translated into aliases (human readable event names) by using an event mapping table [10]. The hardware vendors provides these hardware-details in the form of JSON files (arch event definition files), as shown in Figure 2. In Linux, these JSON files can be located at *tools/perf/pmu-events/arch/<arch>*. The information is then used by PAPI which aims to provide consistent and OS independent access to PMCs.

Performance Application Programming Interface (PAPI) was introduced as an abstraction layer to access PMCs using the Perf interface. Over the time PAPI has evolved into component-based architecture, which can monitor data from multiple components like CPU, thermal sensors, Network etc [11] [12]. PAPI extracts perf events and maps them into human readable names based on the underlying platform to save users from low level architectural details.

These events are divided into two categories named *presets* and *native*. Presets are events which are common and consistent among majority of platforms (also called *architectural*). However native events are specific to a given platform on which they are running (also called *non-architectural*).

Due to rapid advancements in technology and version changes static solutions require frequent checks and updates which can directly influence the QoS in case of any delays and negligence. So the study is aimed to extract event list directly from underlying hardware such that the results are not dependent to out of date/static list of events at any point in time, as shown in Figure 2.

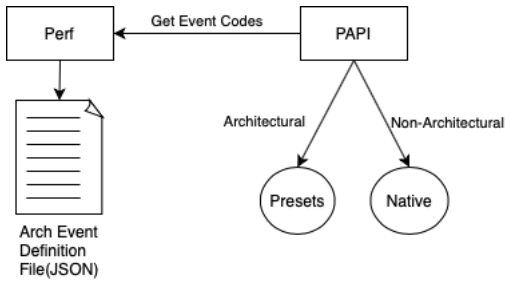

Fig. 2. Illustration of Perf and PAPI in Linux Architecture

D. Multiplexing

Modern computers contain a vast number n of performance counter events. It is, however, not possible to simultaneously monitor n events due to high architectural and operational cost.

Multiplexing is a technique that can be used to monitor all events even if there are more events than counters. When number of events exceeds the core-internal PMCs then the technique is to configure core-external PMCs, if it is allowed. But if core-external PMCs cannot be utilized then time division multiplexing with core-internal PMCs is performed until full event coverage is done.

Perf automatically performs multiplexing by giving a fraction of time t_a to each event, in a round robin fashion [8]. This is done by switching frequency in Perf [13], and metrics are calculated usually at the rate of 100 to 1000 hz using formula:

$$C_T = \frac{C_R \times t_a}{t_e} \tag{1}$$

where C_R is counter value when an event got its turn to be monitored, t_e is the total time to monitor all the events and t_a is fraction of t_e when a particular event availed its turn to run. Here, C_T is an estimated value because it is not the count of an event throughout the execution period of an application.

III. Methodology

We summarize our ranking approach into three steps, listed as follows:

1) Event fetch – In this step, we execute an automatic traversal of n (all) available PMU events in the hardware architecture. Our fetch traversal step fetches the available events directly from the underlying platform.
2) Application characterization using Multiplexing– Here, we characterize an unknown application/process p using rerun multiplexing for n PMU events. As a result of multiplexing time-ordered series m_i of n PMU events are sampled and Pearson's correlation coefficient for each PMU event's time series is calculated as r_i.
3) Rank events – Finally, we sort the Pearson correlation values and highlight the R most important PMU events for application p.

A. Event fetch

Our method is focused on *native* events which is a main distinction from other studies in which proposed static solutions are dependant to a pre-compiled list of known events. We initialize the PAPI engine to traverse through all the available *components* (such as regular perf events and uncore perf events) on the current hardware, as shown in the Algorithm 1. We use the native event mask for event code generation which is the address of physical register where event details are stored.

With *Enumeration* flag set to 0, we traverse through each object in event description file. Function $getEventInfo()$ returns the information of next event available. This event information is then stored into a list. The event list is created per component so that we can distinguish that which event is configured on which component. When there are no more events, $getEventInfo()$ function returns 0 and loop exits. If there are more components available which are active then it moves to fetch events from that component. The process to get events is repeated in the same way for the next component. So in this way we iterate through the event list component by component and fetch n events details from each component.

B. Application Characterization using Multiplexing

Monitoring n events enables us to visualize the complete resource utilization profile of an application p. The obvious solution is multiplexing such as temporal multiplexing described in Section II-D.

Temporal multiplexing is prone to blind spots. These blind spots are points in time when the event was not monitored

Algorithm 1: Get the PMU native event list

```
initialize_PAPI();
setNativeEventMask();
num_components = getNumComponents();
component = 0;
while component ≤ num_components do
    cmpinfo = getComponentInfo();
    ENUM_flag = 0;
    \* when ENUM_flag is set to 0 it iterates through
     all entries in descriptor file till the end of file *\
    while
     (event_info = getEventInfo()) == TRUE
     do
        \* Create Component wise event list *\
        addEventsToCompEventList(event_info);
        \* Move to next event *\
    end
    \* Move to next component *\
    increment(component);
end
\* Create detailed list of native events for
 characterization of application *\
```

and those times could be critical for an event evaluation. So we propose to run the application and monitor first subset sb of size no_PMCs events where $sb \subseteq n$ and re-run the application with next subset sb of size no_PMCs events and so on. In this way we can run the application for T_r times where T_r is total runs:

$$T_r = \lceil \frac{n}{no\_PMCs} \rceil \tag{2}$$

Algorithm 2: Re-run Multiplexing and Sampling of n events

```
sb = no_PMCs;
Quo = n / no_PMCs;
Rem = n % no_PMCs;
while Quo ≥ 0 do
    sb = get next no_PMCs from events(n);
    characterizeApp(p, sb);
    \* Store metrics and calculate Pearsonś Correlation
     coefficient*\
    Quo = Quo - 1;
end
if Rem ≠ 0 then
    sb = get next (Rem) from events(n);
    characterizeApp(p, sb);
    \* Store metrics and calculate Pearsonś Correlation
     coefficient *\
```

So, in our method of rerun multiplexing for complete coverage of events, we rerun the application quotient Quo times for no_PMCs events and then we run the application one last time to monitor remainder Rem events, also shown in Algorithm 2. Here Quo is $\frac{n}{no\_PMCs}$ and remainder Rem is $n \ mod \ no\_PMCs$.

Figure 3 shows the rerun multiplexing for core-internal PMCs in multiples of no_PMCs. If the total number of events n is not a multiple of no_PMCs then the difference is only for last iteration where Rem events are monitored. In each iteration, application is characterized by using the program designed by Danielsson et al. [2]. Characterization is performed with a sampling frequency $freq$ for samples s over the total execution_time t_p of application as

$$freq = \frac{t_p}{s} \tag{3}$$

At the end of characterization each PMU event is sampled as time-ordered series, m_i. All series are then collected in the set $M_{(p)} = \{m_i : 0 \leq i \leq n\}$ and, for each one of them, we calculate Pearson's correlation coefficient, r_i, between m_i and the measured performance of p.

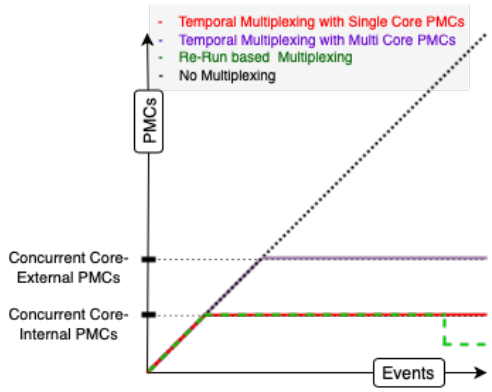

Fig. 3. Illustration of rerun multiplexing in comparison to temporal multiplexing of Hardware Events for PMCs

C. Ranking Events

We determine the most relevant events automatically by sorting them according to the correlation coefficient. The correlation between a specific event count and the number of instructions retired shows the application's resource dependence. In a way, the relation can simply be drawn by taking the difference of *total instructions retired* and *total count of ith PMU event* but Pearson's correlation coefficient can show the linear relationship between two variables.

Pearson's correlation coefficient is sensitive to outliers but in our case it is assumed to be natural even if the event data is not distributed evenly across the timeline. Because it still shows there was any one or more points of times when this event has significant resource demand. However further research is required to know the exact points in time to profile the behaviour.

IV. Implementation

We implement the proposed solution using Linux running *Ubuntu 4.13.0-21-generic* and *g++ 7.2.0*. PAPI library version 5.7.0.0 was used to iterate through all event codes.

As first step, we need to decide the sampling frequency $freq$, see Equation 3. Instead of applying fixed sampling

frequency to every observed process/application, we calculate the sampling frequency based on the process's execution time t_p of p. The process is time stamped before and after the execution and difference between the two gives execution time t_p of p. For symmetric samples, we have opted to use a sampling size of $s = 1ms$ to calculate the $freq$ by using Equation 3. Since the t_p is calculated in micro seconds (μs), this sampling rate was experienced well to get enough number of samples as well as enough time to monitor the probe effect of an event.

To characterize an application for any number of events, a modification was made to the solution provided by Danielsson et al. [2] so that we can dynamically populate the event sets and feed those to PAPI engine and monitor n PMU events. In each iteration, a subset sb of events is monitored leaving the ones those were not able to attach. The reason a event cannot be attached is that the event is specified in the JSON event file, but not implemented on the actual hardware.

Once the characterization is done, Pearson's correlation coefficient of each PMU event's metrics are calculated. Their coefficients r_i are then sorted to rank the events such that higher the coefficient the higher the rank, with 0 being lowest and 1 being highest.

V. Experiments

We list some of the basic internal memory properties of our test computer in Table I. Algorithm running on our experiment platform returns a total of 175 native events. We exemplify some of the events in Table II.

TABLE I
HARDWARE SPECIFICATIONS INTEL® CORE™ I5 8250U

Feature	Hardware Component
Core	4xIntel® Core™ i5-8250U CPU (Kaby Lake) 1.6GHz
L_1-cache	32 KB 8-way set assoc. I-cache/core 32 KB 8-way set assoc. D-cache/core
L_2-cache	256 KB 4-way set assoc. cache/core
L_3-cache	6 MB 12-way set assoc. Inter-core shared cache

TABLE II
SOME EVENT FROM NATIVE EVENT LIST

Event Code	Event Name	Description
0x4000006e	perf::LLC-STORES	Last level cache store accesses
0x40000073	perf::DTLB-LOADS	Data TLB load accesses
0x4000007d	perf::BRANCH-LOADS	Branch load accesses
0x40000089	INSTRUCTION_RETIRED	Number of instructions at retirement
0x400000ca	DSB2MITE_SWITCHES	Number of DSB to MITE switches
0x400000cc	FP_ARITH	Floatingpoint instructions retireds
0x400000d3	SW_PREFETCH	Software prefetches

Then we continue the experiment by choosing a test application, in our case it was a malware for side channel attacks known as *Meltdown* [14]. The reason to choose malware was they are naturally designed with a distinctive behaviour to achieve their purpose as compared to other general purpose applications. It is quite normal for a malware to stay unnoticed for a long time and trigger the hardware events suddenly in a specific time or environment. Due to their unexpected behaviour it is more promising to catch any unusual activity in the event behaviour to visualize it as outlier or anomaly.

Running the community version of *Meltdown* variant on 4xIntel® Core™ i5-8250U CPU (Kaby Lake) 1.6 GHz a total of 175 native events from 2 components (perf event and perf event uncore) using PAPI. These non-architectural events are combination of available hardware and software events. For instance *ix86arch::BRANCH_INSTRUCTIONS_RETIRED* is a hardware event whereas *perf::PAGE-FAULTS* is a software event.

In Figure 4, we exemplify our characterization approach by utilizing the famous meltdown exploit as test application where $InstructionsRetired$ is used as performance metric. It is good to mention that these micro-architectural events are sensitive to the nature of application. Moreover, the selection of sampling frequency may significantly affect the results received.

We run the application 50 times and calculate the median of the Pearson's correlation coefficients. We list the events that displays the highest correlation coefficients in Table III. The micro-architectural events occur at very low level and fast enough that any slight change in sample size affects the counter value significantly. Here, each run presents a high probability of counter value fluctuations, therefore, we rely on median of coefficients to present sound results.

TABLE III
LIST OF MOST RELEVANT PMU EVENTS

Rank	Event Name	Coefficient
01	BR_INST_RETIRED	0.84
02	ix86arch::BRANCH_INSTRUCTIONS _RETIRED	0.79
03	perf::BRANCH-LOADS	0.52
04	perf::DTLB-LOADS	0.51
05	INSTRUCTION_RETIRED	0.51
06	perf::L1-DCACHE-LOADS	0.36
07	perf::BRANCHES	0.31
08	perf::BRANCH-INSTRUCTIONS	0.31
09	perf::PERF_COUNT_HW_BRANCH _INSTRUCTIONS	0.30
10	TLB_FLUSH	0.28
11	BR_MISP_RETIRED	0.25
12	BACLEARS	0.25
13	IDQ_UOPS_NOT_DELIVERED	0.22
14	ix86arch::MISPREDICTED_BRANCH _RETIRED	0.20
15	MOVE_ELIMINATION	0.19
16	perf::INSTRUCTIONS	0.18
17	perf::PERF_COUNT_HW_INSTRUCTIONS	0.18

VI. Discussion

We perform measurements for all available hardware events on a computer and rank their relationships towards an application's performance using Pearson correlation coefficient. For the sake of this study we

Fig. 4. Characterization of Application based on PMU events

measure all hardware events regardless of their nature and redundancy. During event gathering, redundant event names were also observed whose one reason could be the presence of aliases such as $BR\_MISP\_RETIRED$ and $ix86arch :: MISPREDICTED\_BRANCH\_RETIRED$. They seem to be same as per available information but were listed under different event codes.

Though, temporal multiplexing can give a reasonable coverage of events but it is prone to blind spots. Not only the blind spots, counter value is also important to understand, which is an estimation based on the fraction of time it receives in round robin fashion. So for these two reasons there is high probability to miss the information as well as a chance of failure to observe the cascading effect of resource utilization at all. It is good to mention that cascading effects may only be observed through start-to-end or extended timeline processing. In contrast to temporal multiplexing our suggested rerun based multiplexing gives the complete picture of event behaviour for entire event range by utilizing all available PMCs.

As we do not parse the event description, it is required for once by the engineers to know the platform specific $InstructionsRetired$ event name from the acquired list of PMU events. This event name is then used as a benchmark to measure other PMU events during the characterization of application. It was also an option to take execution time as predictor but execution time may not be consistent all the times due to variable number of context switches in general purpose operating systems. And if the multiplexing is based on rerunning the application then it might not give us the same execution-time every time. So it is more reliable to take $InstructionsRetired$ as predictor for performance.

During each iteration, each event set tried to attach a subset of events but for some it was not possible such as for TLB prefetch misses, stalled cycles and blocking loads. The sampling for these events was not successful for $Meltdown$ as a test application but there is a good chance that those events can be monitored for some other application. Another reason could be that those events were listed is arch event definition file (JSON) by vendors but were not available on actual hardware. Moreover, we left the costumed assignment and distribution of events to default between different cores.

Sampling frequency for the events that can be monitored was set to 1 ms which gave us around 50 samples each of test application as shown in Figure 4. Top most relevant events in Figure 4 shows that at the start there was high $InstructionsRetired$ rate and then a sudden drop. The number of $InstructionsRetired$ was quite consistent until just before the end of application where an exponential increase was observed for all relevant events. At first glance, it looked like an outlier but with a careful code analysis of test application the pattern of timeline was logical. In the start, higher activity was observed due to the exploit happening trying to crack down into kernel module from user space. Afterwards the utilization was smooth until the time just before the end of application's execution while reading the pre-fetch memory. The pattern did not show any distinction until it reached the point which according to the code is when the test malware application tried to remove its backtracks by calling a cleanup function. Due to this massive activity a spike is seen for high resource utilization.

Pearson's Correlation is normalized measurement of covariance to reflect the linear relationship between two variables. It is sensitive to outliers but in our case we assume outliers as legitimate points for evaluation. Even if the event data is not normally distributed across the timeline, it shows there was any one (or more) point of time when this event had significant resource demand. However further research can be done to know the exact points in time where the event leaves it marks and profile their behaviour.

Table III shows $BR\_INST\_RETIRED$ and $ix86arch :: BRANCH\_INSTRUCTIONS\_RETIRED$ as most relevant events which means that application was taking many branches. There was also high relevance to $perf :: DTLB - LOADS$ which is a count of event when it reads from TLB. This observation actually brings the most interesting insight about the test application. A TLB load is lookup for actual physical memory address while using virtual memory. During this lookup, access privileges are also checked and if there is any permission violation it throws an exception. So the high relevance to this event indicates a distinctive behaviour of test application which we already know that it tries to access kernel memory from user space and in that case there should be high exception rate. Such knowledge can further be used for categorisation and profiling of applications. Moreover, results showed that application is L1-Dcache bound too. So based on these event ranks engineers can automatically find out resource dependence during the execution of any application. Otherwise, it is based on operators' skills, experience and knowledge base only. The knowledge which comes from experience is valuable but it is good to keep in mind that human-driven approach is prone to mistakes, errors and insufficient skill set.

VII. Conclusion

The study has successfully presented a solution to characterize any application p by sampling n base PMU events. The rerun based multiplexing enabled us to see the start-to-end event behaviour of event. Each sampled PMU event provided a time-ordered series, on which Pearson's correlation coefficients, r_i was calculated. Based on these correlations, ranking of events was performed to shortlist the most relevant PMU events for an application from the performance perspective. For experimental purposes a malware was tested for which our proposed service successfully listed the most relevant events. This knowledge can be further used for QoS, tuning and detection purposes. For instance, the results showed that $Meltdown$ was taking many branches and it was reading highly from TLB and L1_DCache. Such kind of ranking of relevant events is indeed a useful tool for engineers to get better insights of health, performance and resource dependence of an application.

VIII. Related Work

The study is in continuation to the work done by Danielsson et al. [2]. The researchers determined the resource dependence of an application based on architectural events called

PAPI *presets* which are common across many platforms. One of the limitations of their study was to explicitly feed the list of event names for the characterization of program with an eventual focus for last-level caches only. Whereas our study is focused for all native events to automatically extract from the underlying platform.

Rodrigues et al. [15] have used PMCs to dynamically estimate the power consumption by finding a minimal set of hardware events as a predictor. This study is restricted to a very small set of pre-selected hardware events based on human intelligence only. Also it lacks the statistical endorsement of selection of baseline events set. Moreover, the study used simulators instead of bare-metal environment which may jeopardize the accuracy of collected data. In contrast, our approach is aimed for bare-metal environment to capture as many as possible events by direct extraction from underlying platform.

There are other studies who have used PMCs to estimate the power and bandwidth consumption [16] [9] [17] and to check the performance of application in terms of CPU load [18]. Another study has used performance counters for safety and security of the systems by proposing an attack mitigation model [19]. But as per our knowledge, other studies did not automatically monitor all events regardless of which platform they are coming. Moreover, another interesting study was performed by [20] on *Blue Gene/P™* super computer to monitor massive number of PMU events (256 concurrent 64b counters). Although the capability to monitor performance was increased but it is not very commonly available architecture across many SMEs (Small and Medium Enterprises).

IX. Future Work

The study can be extended in many ways such as detection of faults, failure and malicious activity. Based on the hardware dependence a behavioural analysis of metrics can finger print any process. One of the biggest challenges is not only the low number of counters, but is to measure the events based on their nature such as configure the sampling frequency based on the nature of event to be monitored.

Occurrence of some events is not as frequent as others and for some the measurement cost at low frequency is too high. So a model built on top of event nature would improve the reliability of solution. Also, it would be interesting to test the measurement with other AI or statistical methods when the data distribution in non-linear.

References

[1] Marcus Jägemar, Andreas Ermedahl, Sigrid Eldh, and Moris Behnam. A scheduling architecture for enforcing quality of service in multi-process systems. In *Emerging Technologies and Factory Automation (ETFA), 2017 22nd IEEE International Conference on*, pages 1–8. IEEE, 2017.

[2] Jakob Danielsson, Tiberiu Seceleanu, Marcus Jägemar, Moris Behnam, and Mikael Sjödin. Resource depedency analysis in multi-core systems. In *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 87–94. IEEE, 2020.

[3] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. *Collecting Performance Data with PAPI-C*. Springer, Berlin, Heidelberg, 2010.

[4] Marcus Jägemar, Andreas Ermedahl, Sigrid Eldh, and Moris Behnam. A scheduling architecture for enforcing quality of service in multi-process systems. *22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, 2017.

[5] Brendan Gregg. *Systems Performance : Enterprise and the Cloud Second Edition*. Pearson, 2nd edition, 2020§.

[6] Stijn Eyerman and Lieven Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3):42–53, 2008.

[7] Intel. Intel® 64 and ia-32 architectures software developer's manual. Technical report, Intel, 2016.

[8] Andrzej Nowak and Georgios Bitzes. The overhead of profiling using PMU hardware counters. Technical Report CERN Openlab Report, CERN, 2014.

[9] Stéphane Eranian. What can performance counters do for memory subsystem analysis? In *ACM SIGPLAN workshop on Memory systems performance and correctness: held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*, pages 26–30. ACM, 2008.

[10] Linux Foundation. pmu-events, 2021.

[11] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with PAPI-C. In *Tools for High Performance Computing 2009*, pages 157–173. Springer, Berlin, Heidelberg, 2010.

[12] Matthew Johnson, McCraw Heike, Shirley Moore, Phil Mucci, John Nelson, Dan Terpstra, Vince Weaver, and Tushar Mohan. PAPI-V: Performance Monitoring for Virtual Machines. *41st International Conference on Parallel Processing Workshops*, 194-199:189–204, 2012.

[13] Stephane Eranian, Eric Gouriou, Tipp Moseley, and Willem Bruijn. Linux kernel profiling with perf. Technical report, Perf, 2015.

[14] Lipp Moritz, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. pages 973–990, 2018.

[15] Rance Rodrigues, Israel Koren, Annamalai Gracioli, and Sandip Kundu. A study on the use of performance counters to estimate power in microprocessors. *IEEE Transactions on Circuits and Systems II: Express Briefs*, pages 882–886, 2013.

[16] Rafia Inam, Mikael Sjödin, and Marcus Jägemar. Bandwidth measurement using performance counters for predictable multicore software. *IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*, pages 1–4, 2012.

[17] Rance Rodrigues, Annamalai Arunachalam, Koren Israel, and Kundu Sandip. A study on the use of performance counters to estimate power in microprocessors. *IEEE Transactions on Circuits and Systems II: Express Briefs 60*, pages 882–886, 2013.

[18] Marcus Jägemar, Sigrid Eldh, Andreas Ermedahl, and Björn Lisper. Towards feedback-based generation of hardware characteristics. *7th International Workshop on Feedback Computing*, 2012.

[19] Alberto Carelli, Alessandro Vallero, and Stefano Di Carlo. Performance Monitor Counters: interplay between safety and security in complex Cyber-Physical Systems. *IEEE Transactions on Device and Materials Reliability 19*, pages 73–83, 2012.

[20] Valentina Salapura, Karthik Ganesan, Alan Gara, Sexton Gschwind, John James C., and Robert E. Walkup. Next-generation performance counters: Towards monitoring over thousand concurrent events. *ISPASS 2008-IEEE International Symposium on Performance Analysis of Systems and software*, 139-146:189–204, 2008.