Low Overhead Hardware-Assisted Virtual Machine Analysis and Profiling

Suchakrapani Datt Sharma*, Hani Nemati†, Geneviève Bastien‡ and Michel Dagenais§
Department of Computer and Software Engineering
Polytechnique Montreal, Quebec, Canada
Email: {*suchakrapani.sharma; †hani.nemati; ‡genevieve.bastien; §michel.dagenais}@polymtl.ca.

***Abstract*—Cloud infrastructure providers need reliable performance analysis tools for their nodes. Moreover, the analysis of Virtual Machines (VMs) is a major requirement in quantifying cloud performance. However, root cause analysis, in case of unexpected crashes or anomalous behavior in VMs, remains a major challenge. Modern tracing tools such as LTTng allow fine grained analysis - albeit at a minimal execution overhead, and being OS dependent. In this paper, we propose *HAVAna*, a hardware-assisted VM analysis algorithm that gathers and analyzes pure hardware trace data, without any dependence on the underlying OS or performance analysis infrastructure. Our approach is totally non-intrusive and does not require any performance statistics, trace or log gathering from the VM. We used the recently introduced Intel PT ISA extensions on modern Intel Skylake processors to demonstrate its efficiency and observed that, in our experimental scenarios, it leads to a tiny overhead of up to 1%, as compared to 3.6-28.7% for similar VM trace analysis done with software-only schemes such as LTTng. Our proposed VM trace analysis algorithm has also been open-sourced for further enhancements and to the benefit of other developers. Furthermore, we developed interactive Resource and Process Control Flow visualization tools to analyze the hardware trace data and present a real-life usecase in the paper that allowed us to see unexpected resource consumption by VMs.**

***Keywords*—Virtualization; Hardware Tracing; Intel PT; Trace Analysis; VM Analysis**

I. Introduction

The backbone of modern distributed cloud systems are virtualization technologies that enable VMs to provide the necessary infrastructure. Public and private cloud infrastructure providers allow the users to access a pool of resources based on a Pay-as-Use (PaU) model where numerous automated cloud orchestration tools allow seamless control of bring-up and tear-down of VMs. This flexibility in resource scaling leads to imbalanced workload distribution on the underlying hardware on which the VMs run. Users can also intermittently run demanding applications which may need their VMs to be migrated to difference resource groups. Cloud infrastructure administrators therefore need modern tools for performance analysis of such VMs. However, efficient debugging, troubleshooting and analysis of such massive distributed systems is a still a known challenge [1]. For fine-grained post-mortem root cause analysis of problems occurring on VMs, the administrators may need highly detailed information about the characteristics of VMs on their infrastructure - such as profile of processes running on them, virtual CPU (vCPU) consumption, pattern of scheduling of processes on VMs and their interactions with underlying hypervisors. Most of such information can be gathered by proper configuration tools provided by the host OS kernel. Software-only diagnosis of problems on VMs, however, calls for recording all software events such as occurrences of `vm_entry`, `vm_exit`, `sched_switch`. The added overhead from such events can be mitigated by using tracing tools such as LTTng[5]. However, these tools also alter the execution flow of the VMs and require careful configuration (such as adding additional static tracepoints in QEMU and KVM). In addition, proprietary close-sourced operating systems on specialized hardware may not expose tracing tools or APIs and would be opaque to the administrator. In such scenarios, pure hardware tracing can help in diagnosing abnormal executions.

In this paper, we introduce a novel approach that uses hardware trace support provided in modern processors for VM analysis. Special trace data emitted by the trace hardware on the processor can be collected and analyzed offline to gather in-depth information about execution profiles of VMs and hypervisors on the host. Our approach allows for a near-zero tracing overhead and a new technique to visualize such data.

We demonstrate the uniqueness of our approach through the hardware trace support provided in Intel's Skylake series of processors in the form of Intel Processor Trace (PT)[7]. These trace blocks generate huge amounts of hardware trace data, consisting of mostly branch related packets that can be used to reconstruct the program flow. The trace data also contains certain trace packets such as PIP and VMCS which we record, extract and use with our algorithm to generate synthetic trace events that identify important states of processes in VMs such as entry/exit from hypervisor, to or from the VM, and scheduling events between processes in VMs. We generate synthetic events from such hardware trace packets that profile vCPU consumption by processes, without any software and operating system (OS) intervention, thus ensuring a low overhead and minimum interference with the VMs. Since this approach is OS independent, it works on any OS platform without any necessary configuration. Through this, we were able to identify processes inside VMs that would cause undesired vCPU load. To the best of our knowledge, there is no pre-existing efficient technique to gather such high level VM analysis from low level hardware trace packets. Our main contributions in this paper are as follows :

- A novel low overhead hardware-assisted approach to ex-

978-1-5090-2482-7/16/$31.00 ©2016 IEEE

tract, group and analyze hardware trace packets gathered from the processor for VM analysis. The VMs, host hypervisor and host OS are oblivious to our tracing and analysis phase. Therefore, there is no need for internal access within VMs, which may not be allowed in most situations due to security reasons.
- A visualization strategy to display these hardware trace events on a time series graph, and identify hard to diagnose issues such as processes contending for resources in VM. Our graphical views show CPU usage inside the VM along with their interaction with the Virtual Machine Monitor (VMM). We also implemented a graphical view for the execution flow of processes inside the VM.

The rest of the paper is organized as follows: Section II presents related work, comparing the closest approaches to ours. Section III introduces important processor trace packets for VM analysis and explains the different layers of the architecture that we use in our paper. Here, we also present the algorithm used to retrieve information from processor trace packets. We show a use-case of our analysis in Section IV. The added overhead with our approach is compared with existing approaches in Section V. Section VI concludes the paper.

II. Related Work

Program flow tracing based instruction counting, and tracking blocks of code, has been discussed earlier [3]. In order to reduce the bandwidth of such tracing, Merten et al. [9] have earlier proposed the use of a Branch Trace Buffer (BTB) and their hardware table extension for profiling branches. Custom hardware-based path profiling has also been discussed by Vaswani et al. [16]. Linux Kernel tools such as Ftrace and Kprobes allow such code instrumentation and program flow deduction. Modern architectures snoop bus activity at very low level inside the processor and allow recording each and every instruction being executed. This, however, generates a huge amount of data. To mitigate this, the new approach is to only record instructions that cause the program flow to change, such as direct/indirect jumps, calls, exceptions etc. By following these change-of-flow instructions, it is quite easy to generate a complete program flow with the help of additional offline binary disassembly at the decoding phase. Dedicated hardware blocks in the Intel architecture, such as Last Branch Record, Branch Trace Store [15], and more recently Intel Processor Trace (PT) follow this approach. A detailed study of hardware-assisted tracing and profiling with Intel PT has been recently presented in [12].

Hardware-trace based compiler optimization techniques have also been discussed before [11] where results from execution profiles of a software application can be fed back to the compiler to optimize the resulting binary. Jörg et al. in [13] present a data parallel provenance algorithm which uses Intel PT for improving security and dependability in software.

AWS CloudWatch and Openstack Ceilometer are the metering, monitoring and alarming tools for clouds. They provide basic metrics such as physical CPU and number of vCPU used for each VM. The information presented by such tools is not suitable for analyzing VMs. Most existing Linux tools such as `vmstat` gather statistics by reading `procfs` file with significant overhead. Therefore, they are not recommended for implementing a low overhead tool for analyzing and debugging VMs. In [6], the authors proposed a significant multi-layer tracing and analyzing technique to detect anomalies inside VMs. They implemented an execution flow recovery of specific processes by tracing the host and VMs at the same time. In their work, they need internal access to the VMs. Furthermore, their work is limited to the Linux OS since they use LTTng as Linux kernel tracer. PerfCompass[4] uses a VM's kernel trace (from LTTng) and gathers information from *syscall* events to detect anomalies inside the VM. Authors in [2] implemented a vCPU monitoring tool based on *"perf kvm record"*. With their monitoring tool, they are able to gather statistics about CPU usage for processes and the hypervisor. Wang in [17] used Perf to detect over-commitment of pCPUs. From all available CPU metrics, they used LLC which has a direct relationship with pCPUs over-commitment. Our profiling technique uses hardware trace packets that show vCPU usage along with processes execution flow, without any software and OS intervention. Our approach adds less overhead to VMs compared to other techniques and it does not rely on a specific OS or hypervisor. As per our knowledge, no prior work has been done for analyzing VMs at a high level from such low level hardware traces yet. Our work, therefore, is unique and novel in this aspect.

III. Hardware Tracing VMs

Hardware trace generation can be configured and controlled by certain configuration registers such as MSRs in Intel or CoreSight ETM/ETB Configuration registers on ARM. In this paper, we select Intel PT as an experimentation platform for our hardware-asssisted VM analysis approach. Once hardware trace generation is enabled, the tracing blocks from processor cores generate compressed encoded trace packets for eventual decoding. These hardware trace packets can contain information such as paging (changed CR3 value), time stamps, core-to-bus clock ratio, taken-not-taken (tracking conditional branch directions), record the target IP of branches, exceptions and interrupts, and record the source IP for asynchronous events (exceptions, interrupts). Keeping track of all these packets can be quite expensive - especially, if the required analysis is at a high level (such as VMs in our case). Therefore, we isolate only those packets for analysis that are sufficient to reconstruct the flow in the VMs. Some of the important hardware packets and their role in our analysis are as follows :

1) ***PIP****:* The Paging Information Packet (PIP) is generated whenever the `CR3` register value is modified. This includes scenarios such as a task switch, a `MOV CR3` instruction or, more importantly, a VM Entry and VM Exit at the time when VM execution is enabled. This packet allows the decoder to uniquely identity which process was executing on the processor. During VM execution, the packet also contains a Non-Root (NR) bit that can further indicate if the process was executing in a NR context (guest mode) or in the VMM

context. Together with other packets generated for VMXON instructions, we can generate a detailed view of the VM.

*2) **VMCS**:* This packet is generated at a successful VMPTRLD instruction, which indicates interaction between the VMM and the guest OS. The VMCS packet payload consists of the VMCS pointer of the logical processor that will execute the VM guest context. This packet helps us in determining which vCPU was being utilized at what time.

*3) **Timing**:* : Timing information for each event can be deduced from 3 more important packets. The first one is Time Stamp Counter (TSC) which gives the lower 7 bytes of the time-stamp counter - the same as the one returned by the RDTSC instruction. The Mini Time Counter (MTC) packet contains the 8-bit value derived from the Always Running Timer (ART) on Intel processors. Along with a Timing Alignment (TMA) packet, the MTC and TSC values can be used to estimate the precise timing of each event up to nanosecond precision [7].

For our analysis, we record all these packets and analyze them in post VM execution scenarios. We also record the physical CPU (pCPU) associated with the relevant packet. We then create synthetic events with all the packets and the relevant context information attached to them.

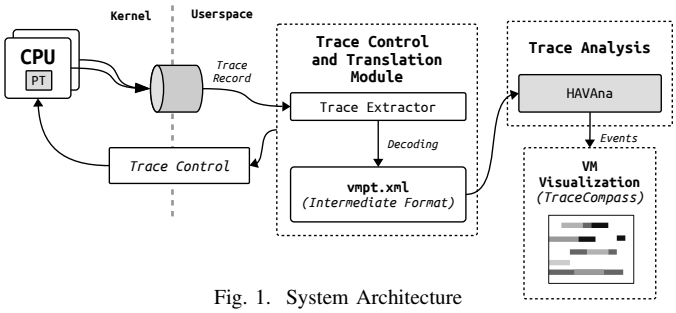

Fig. 1. System Architecture

A. System Architecture

As seen in Figure 1, the trace control module configures the tracing hardware on the processor. Enabling the tracer generates a huge amount of encoded trace data that is stored on disk, with context information for each pCPU attached to it. The translation module filters and extracts the raw packets for VM analysis. The PIP, VMCS, TSC and MTC packets are decoded from the per-CPU trace stream and converted to an XML derived intermediate format (IF). The synthetic events are identified from the decoded trace and stored in this format. For example, <event> tags contain each event with their timestamps along with event specific data. There are two event packets - PIP and VMCS. The main driver for this module is our Hardware-Assisted VM Analysis (HAVAna) Algorithm that is based on the state machine which analyses the packet IF and generates visualizations describing the VM behavior. The XML driven visualizations are consumed by the Trace Compass [14] trace analysis tool for an in-depth interactive view of the VM execution.

B. HAVAna Algorithm

The main feature of our proposed technique is the state machine that classifies hardware packets and generates synthetic events for visualization. The input to the algorithm is the raw XML event description stored in the IF generated during trace translation. Each event packet from the stream is sent to the state machine shown in Figure 2. The occurrence of a VMCS event packet in the IF, succeeding a PIP packet, marks the process for being scheduled on the vCPU and indicates the beginning of a VM execution at a high level. The process enters the **VMM Mode** (Root Mode). A PIP packet with a new CR3 value and Non-Root (NR) bit (extracted from PIP hardware trace packet) as 1 indicates that a VM process is now being executed. This is marked by the **VM Mode** (Non-Root Mode). Successive transitions of PIP packets with NR bit value indicate the execution switching between VMM and VM mode. Along with the timestamps in all the states gathered from the IF, we can start creating a time series graph that shows the process activity in VM and VMM. By associating vCPUs with VMCS base pointers, we can identify the vCPU consumption as well. The output of the state machine are the synthetic events that are then stored and input to the trace visualization tool. The pseudocode for our algorithm to uncover different states for vCPUs and processes inside VMs is shown in Algorithm 1. It receives events as input and updates the State History Tree [14] as output. For each packet, it checks the name. In case of the packet name is VMCS, it saves the VMCS based address and changes the Status of the related vCPU as VMM (Line 4). When our algorithm receives a PIP packet, it checks the NR field. If the NR field is 1, it first queries the current running vCPU base address and modifies the Status of the related vCPU as VM (Line 8) and then it queries the current running VMM and modifies its Status as $IDLE$ (Line 9). It also changes the Status of the current process (identified by the CR3 value) running inside the VM as VM (Line 10). If the NR field is zero, and the current status of the vCPU is VMM, it modifies the Status of the vCPU, process and VMM as $IDLE$, (Line 13-15). In case the NR field is zero and the current status of the vCPU is VM, it sets all the attributes to VMM, (Line 18-20).

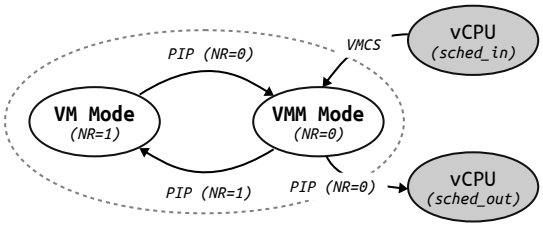

Fig. 2. HAVAna State Machine

C. Trace Visualization

For constructing the Synthetic Events (SE) for visualization, we follow a XML based scheme similar to the one used by Kouame et al. [8]. In our case, however, we define rules

Algorithm 1 HAVAna Algorithm

procedure HAVANA(**Input**: Event Packets ($P_e[i]$) from IF **Output**: Updated SHT)
  $SE[i] = parseXML(P_e[i])$
  if ($SE[i].name == VMCS$) **then**
    Modify Status attribute of SE[i].base as VMM
  else if ($SE[i].name == PIP$) **then**
    if ($SE[i].NR == 1$) **then**
      Query Status attribute of current running base
      Modify Status attribute as VM
      Modify VMM Status as $IDLE$
      Modify Status attribute of $SE[i].cr3$ as VM
    else if (Query Status attribute of $SE[i].base == VMM$) **then**
      Query Status attribute of current running base
      Modify $base$ Status as $IDLE$
      Modify VM Status as $IDLE$
      Modify VMM Status as $IDLE$
    else
      Query Status attribute of current running base
      Modify $base$ Status as VMM
      Modify Status attribute of $SE[i].cr3$ as VMM
      Modify VM Status as $IDLE$
    end if
  end if
end procedure

for state transitions in XML, as described in Algorithm 1, and input them to the TraceCompass[14] tool, an open source tool for analyzing traces and logs. It provides an extensive and well documented interface to build analysis views and graphs. We have also open sourced[1] our hardware-assisted VM analysis scheme and algorithm. We created two analysis views based on the synthetic events. The first one is the *VM Resource View* that shows the vCPU resource usage by VMMs as well as the processes running on the VM. This can be useful for analyzing transitions between the VMM and VM modes and identify abnormal latencies in either VM, process or VMM mode. The second view is the *VM Control Flow View*, which shows each process on the VM and their flow of execution. We describe these views with a usecase in the following section.

IV. Usecases - Resource Contention

To show the efficiency of our approach, we first show our VM Resource View with an example of a 4 threaded application which calculates prime numbers. We configured our test VM with 4 vCPUs pinned to one pCPU, which can represent an ideal low-tier VPS. We ran our test application while recording a hardware trace from Intel PT in a trace buffer. We extracted the trace data, decoded and converted it to the XML IF and applied our HAVAna algorithm. The resulting VM Resource view, as seen in Figure 3, shows an execution window of about 3 seconds with the 4 threads executing on the 4 vCPUs while contending for CPU resources. The red bars show the process execution in the VM while the green bars shows the VMM mode execution. As the visualization is interactive, we can zoom the slightly anomalous looking green bar and observe how much extra time was spent in the VMM mode as compared to the VM to VMM switches adjacent to that execution, as shown in the same figure. Usually such behavior is indicative of VM page faults and VM PAUSE states. However further analysis of each extra time requires detailed software trace from the host kernel.

For our VM Control Flow View, we demonstrate a RabbitMQ based message queuing system that performs MD5 hashing. With the same VM configuration as above, we setup 3 worker threads that do the hashing in a round robin fashion and sent 3 jobs simultaneously to them. Each worker process would execute for some duration and the scheduler on the VM then passes the execution to the other worker processes. We can observe such a pattern for the 3 workers in Figure 4. Each process intermittently does the job and then relinquishes control to the next process in queue and so on. This view can be used to show how the control flow was passed between processes, their relationships with their parents, children and abnormal executions if any.

All of these views have been populated with hardware trace data gathered from PT, without any software trace intervention, thus making our approach agnostic of any OS platform or software infrastructure dependency.

V. Overhead Analysis Experiments

One of the major benefits of our work is that we avoid interacting with software altogether during the trace recording phase - unlike the current software based tracers such as Ftrace, LTTng or SystemTap, that cause some overhead in the target trace execution while trace recording. To quantify the reduction in trace timing overhead with our approach, we used the `sysbench` benchmark to measure the overhead caused when LTTng kernel tracing was enabled. We compared it to the hardware trace overhead incurred while Intel PT was being used. The test machine was an Intel i5-6600K processor clocked at 3.5GHz with 16GB of main memory. We ran our tests and benchmarks on a vanilla Linux kernel v4.5. We used KVM as kernel hypervisor and QEMU as its userspace counterpart. Our results have been summarized in Table I.

TABLE I
PT BASED VM TRACE AND LTTNG TRACE OVERHEAD

Benchmark	*Execution Overhead*(%)	
	PT	LTTng
File I/O	1.00	28.724
Memory	0.00	0.087
CPU	0.00	0.026

We observed that, for the File I/O benchmark, the hardware tracing overhead for the analysis was only 1%, as compared to a similar VM analysis trace overhead with LTTng at 28.7% where tracing was activated on the host kernel[10]. This large overhead was mostly due to the trace storage competing with the benchmark for disk bandwidth. In other cases, the PT overhead was not statistically significant, and hence was neglected, while the LTTng overhead stood at around 3.6%. Even though LTTng's trace analysis could give deeper insights about the host and VM than our pure hardware trace approach, the completely non-intrusive, platform OS independent, approach

[1] `http://step.polymtl.ca/~suchakra/havana.tar.gz`

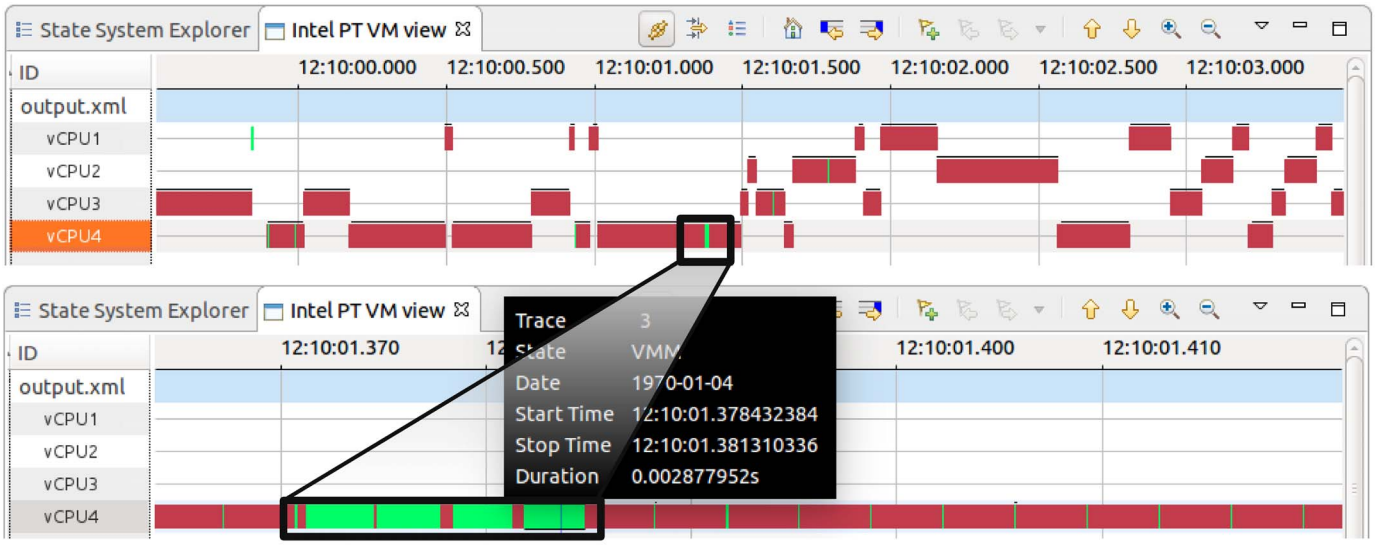

Fig. 3. Resource View showing 4 vCPUs and their execution distribution on single pCPU along with a zoomed view of the VMM mode

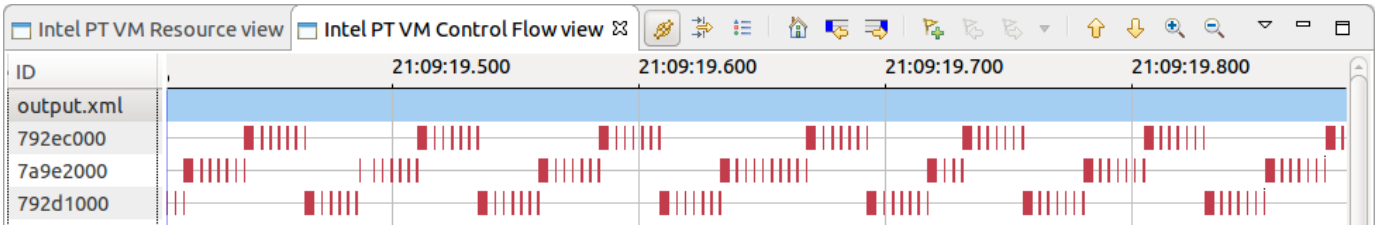

Fig. 4. Control Flow View showing 3 RabbitMQ worker processes contending for existing pCPU

of hardware tracing can yield similar end results at a much lower execution cost on host and VM.

VI. Conclusion

The use of tracing allows cloud infrastructure providers to diagnose issues that may be hard to reproduce otherwise. As virtualization is the base layer for building up cloud services, it is important to tackle issues in VMs. We observed that most of such analyses would require gathering data from the VM, the hypervisor and the host kernel which needs agents running inside client VMs. To overcome this limitation, we propose a new hardware trace analysis based *HAVAna* algorithm that allows detailed diagnosis of CPU resource consumption by processes on VMs, their states and flow of execution, in a completely non-intrusive manner, without the involvement of any OS or VM interface. We demonstrate that our technique allows detection of resource contention in the VMs without querying the guest at all, thereby allowing infrastructure providers to meet their SLAs effectively without any support from the clients' VMs. This can also be beneficial to further analyze malicious executions in the target VMs or move them to different resource groups based on observed workloads.

Even though our approach and algorithm are independent from any VM interaction, the amount of information, such as identifying the faulty process by name or PID, or gathering instruction profile data for individual processes is reduced. Such problems can be tackled eventually by fetching minimal statistics from VMs, such as process maps from the guest kernel. Another obvious addition can be to identify executions in VMs intended to have small lifespans (such as those which mimic containers in their behavior) and compare their successive startup and teardown profiles by comparing instruction executions. This would help in clustering them and moving them to different resource groups as required.

References

[1] Giuseppe Aceto et al. "Cloud monitoring: A survey". In: *Computer Networks* 57.9 (2013), pp. 2093 –2115.

[2] A. Anand et al. "Resource usage monitoring for KVM based virtual machines". In: *Advanced Computing and Communications (ADCOM), 2012 18th Annual International Conference on*. 2012, pp. 66–70.

[3] Thomas Ball and James R. Larus. "Optimally Profiling and Tracing Programs". In: *ACM Trans. Program. Lang. Syst.* 16.4 (July 1994), pp. 1319–1360.

[4] D. Dean et al. "PerfCompass: Online Performance Anomaly Fault Localization and Inference in Infrastructure-as-a-Service Clouds". In: *IEEE Transactions on Parallel and Distributed Systems* PP.99 (2015), pp. 1–1. ISSN: 1045-9219.

[5] Mathieu Desnoyers and Michel R. Dagenais. “The LTTng tracer: A Low Impact Performance and Behavior Monitor for GNU/Linux”. In: *OLS (Ottawa Linux Symposium) 2006*. 2006, pp. 209–224.

[6] Mohamad Gebai, Francis Giraldeau, and Michel R Dagenais. “Fine-grained preemption analysis for latency investigation across virtual machines”. In: *Journal of Cloud Computing*. December 2014, pp. 1–15.

[7] Intel. *Intel Processor Trace*. Accessed: 2016-04-3. Intel Press, 2015, pp. 3578–3644.

[8] K. Kouame, N. Ezzati-Jivan, and M. R. Dagenais. “A Flexible Data-Driven Approach for Execution Trace Filtering”. In: *2015 IEEE International Congress on Big Data*. 2015, pp. 698–703.

[9] Matthew C. Merten et al. “A Hardware Mechanism for Dynamic Extraction and Relayout of Program Hot Spots”. In: *SIGARCH Comput. Archit. News* 28.2 (May 2000), pp. 59–70. ISSN: 0163-5964.

[10] Hani Nemati and Michel Dagenais. “Virtual CPU State Detection and Execution Flow Analysis by Host Tracing”. In: *The 6th IEEE International Conference on Big Data and Cloud Computing (BDCloud 2016)* (2016).

[11] Vinodha Ramasamy et al. “Feedback-Directed Optimizations in GCC with Estimated Edge Profiles from Hardware Event Sampling”. In: *Proceedings of GCC Summit 2008*. 2008, pp. 87–102.

[12] Suchakrapani Sharma and Michel Dagenais. “Hardware-Assisted Instruction Profiling and Latency Detection”. In: *The Journal of Engineering* (2016). DOI: 10.1049/joe.2016.0127. URL: http://digital-library.theiet.org/content/journals/10.1049/joe.2016.0127.

[13] Joerg Thalheim, Pramod Bhatotia, and Christof Fetzer. “Inspector: Data Provenance using Intel Processor Trace (PT)”. In: *proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS)*. 2016.

[14] *Trace Compass*. https://projects.eclipse.org/projects/tools.tracecompass. Accessed: 2016-04-3.

[15] A. Vasudevan, N. Qu, and A. Perrig. “XTRec: Secure Real-Time Execution Trace Recording on Commodity Platforms”. In: *System Sciences (HICSS) 44th Hawaii International Conference on*. 2011, pp. 1–10.

[16] Kapil Vaswani, Matthew J. Thazhuthaveetil, and Y. N. Srikant. “A Programmable Hardware Path Profiler”. In: *Proceedings of the International Symposium on Code Generation and Optimization*. CGO ’05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 217–228. ISBN: 0-7695-2298-X.

[17] S. Wang et al. “VMon: Monitoring and Quantifying Virtual Machine Interference via Hardware Performance Counter”. In: *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*. Vol. 2. 2015, pp. 399–408.