Enabling power-awareness for the Xen Hypervisor*

Matteo Ferroni
Politecnico di Milano
matteo.ferroni@polimi.it

Juan A. Colmenares
Samsung Research America
juan.col@samsung.com

Steven Hofmeyr
Lawrence Berkeley National Laboratory
shofmeyr@lbl.gov

John D. Kubiatowicz
University of California Berkeley
kubitron@cs.berkeley.edu

Marco D. Santambrogio
Politecnico di Milano
marco.santambrogio@polimi.it

ABSTRACT

Virtualization allows simultaneous execution of multi-tenant workloads on the same platform, either a server or an embedded system. Unfortunately, it is non-trivial to attribute hardware events to multiple virtual tenants, as some system's metrics relate to the whole system (e.g., RAPL energy counters). Virtualized environments have then a rather incomplete picture of how tenants use the hardware, limiting their optimization capabilities. Thus, we propose XeMPower, a lightweight monitoring solution for Xen that precisely accounts hardware events to guest workloads. It also enables attribution of CPU power consumption to individual tenants. We show that XeMPower introduces negligible overhead in power consumption, aiming to be a reference design for power-aware virtualized environments.

Categories and Subject Descriptors

H.4 [**Software and its engineering, Virtual machines, Performance monitoring**]

1. INTRODUCTION

In the last few years, embedded systems have experienced a shift from microcontrollers to multi-core processors, as these have become cheaper, smaller, and less power-hungry. This shift brings two advantages: 1) multiple embedded applications can be consolidated on the same System-on-Chip (SoC), improving the overall resource utilization, and 2) some applications can exploit concurrency and parallelism to obtain better performance.

In the context of embedded systems, hardware-assisted and software virtualization technologies have been developed to allow colocated applications to share physical resources while having strong security and isolation [24, 26, 22].

Those technologies seek to offer a stable and predictable execution environment to make it easier for embedded applications to meet different Quality of Service (QoS) requirements.

The virtualized runtime can be a full-fledged guest Operating System (OS) or more suitable to embedded systems, a light-weight OS (e.g., [14, 25]), customized for a specific application. Applications executing in such runtimes generally have different performance objectives, such as: hard and soft deadlines, and peak throughput. Moreover, they are often different from one another in terms of workload limitations (i.e., memory-bound, I/O-bound, or CPU-bound) and evolving load patterns (e.g., algorithmic phases).

Unfortunately, this high *heterogeneity* comes at a price: The isolation between simultaneously resident applications, enforced by virtualization, shifts the burden of optimization from developers to the hypervisor itself because only a privileged arbiter can thoroughly observe what happens on the bare metal. Hence, it becomes clear that a smart *online* monitoring strategy is necessary to accurately observe and model applications' behavior to guarantee requirements and optimize physical resources utilization.

Since power consumption is currently a key technological limitation [13], recent works propose approaches to optimizing power [9], while maintaining Service Level Agreements (SLAs) with each hosted guest. Again, these approaches have an essential need for precise and thorough *observation* of both hardware and guests' behavior over time. Lacking appropriate tools, many of these approaches employ custom monitoring solutions or rely on outdated tools that do not provide support for the latest hardware monitoring features. Often, these ad-hoc approaches overlook the impact of measurements on the overall system's behavior.

Seeking to fill the gap, this paper proposes XeMPower, a lightweight hardware and resource monitoring solution for the Xen hypervisor [6]. It is meant to be agnostic to the hosted applications, and results show it incurs negligible power consumption overhead. XeMPower has been released as open source,[1] and it aims to be a reference design for future works in the field of virtualized systems.

To prove its effectiveness, we present a use case in which XeMPower precisely accounts hardware events to virtual guests, enabling real-time attribution of CPU power consumption to each guest or "domain".[2] XeMPower starts with socket-level energy measurements through the Intel Running Average Power Limit (RAPL) interface [23], and then utilizes a performance-counter-driven model to account for the proportional uses of energy by simultaneously resident domains over time. This proportional attribution of power is XeMPower's secret sauce – the contribution is evaluated by measuring a subset of architectural performance counters related to each domain, but regardless of the physical core.

The paper is organized as follows. Section 2 presents an

*EWiLi'16, October 6th, 2016, Pittsburgh, USA.

[1] Available at: https://bitbucket.org/necst/xempower-4.6

[2] Adopting Xen terminology, the remainder of this paper will refer to virtual guests as *domains*.

overview of XeMPower. Section 3 details the implementation of the tool, while Section 4 shows how to attribute power to each domain. Next, Section 5 investigates the performance overhead of XeMPower while its limitations are discussed in Section 6. Finally, Section 7 presents the related work and Section 8 concludes.

2. PROPOSED APPROACH

XeMPower is a lightweight monitoring solution for Xen designed to: 1) provide *precise attribution* of hardware events to virtual tenants, 2) be *agnostic* to the mapping between virtual and physical resources, hosted applications and scheduling policies, and 3) add *negligible overhead*.

Our approach uses hypervisor-level instrumentation to monitor every context switch between domains. More precisely, the monitoring flow proceeds as follows:

A. At each context switch and before the domain chosen by the scheduler starts running on a CPU, we begin counting the hardware events of interest. From that moment the configured Performance Monitoring Counter (PMC) registers in the CPU store the counts associated with the domain that is about to run.

B. At the next context switch, the PMC values are read from the registers and accounted to the domain that was running. The counters are then cleared for the next domain to run.

C. Steps A and B are performed at every context switch on every system's CPU (i.e., physical core or hardware thread). The reason is that each domain may have multiple virtual CPUs (VCPUs). Socket-level energy measurements are also read (via Intel RAPL interface) at each context switch.

D. Finally, the PMC values are aggregated by domain and finally reported or used for other estimations (e.g., power consumption per domain).

Figure 1 illustrates the monitoring flow described above. Steps A and B for domains 1, 2, and 3 are shown at every context switch on the left side of the figure. On the right side, steps C and D are performed by the *XeMPower daemon* and *Command Line Interface (CLI) program*, both in Dom0.

Steps A and B allow us to meet the first requirement (precise event attribution), while the second requirement (being agnostic) is satisfied by steps C and D. Regarding the third requirement of low overhead, we empirically confirm that XeMPower meets this requirement in Section 5, while the technical aspects enabling that are discussed in Section 3.

3. IMPLEMENTATION

XeMPower implementation is inspired by *XenMon* [12], a performance monitoring tool for Xen. Unlike XeMPower and other works discussed in Section 7, XenMon does not collect PMC reads. Nevertheless, since XenMon's authors report a maximum overhead of 1-2%, their implementation approach was an interesting starting point for our work and a reasonable baseline to compare our overhead with.

XeMPower operates at two levels (see Figure 1). At the first level, PMC reads are collected inside the Xen kernel and then aggregated by the *XeMPower daemon* running in

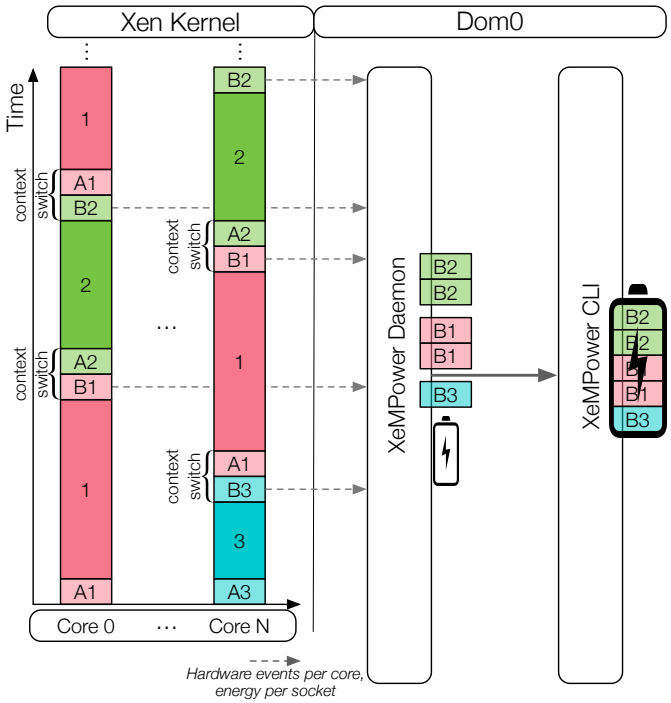

Figure 1: Monitoring flow of XeMPower.

Dom0, while at the second level, a *CLI program* reports aggregated values. In this section, we describe implementation details of the components forming the proposed toolchain.

3.1 Xen Kernel Instrumentation

Xen runs a separate scheduler instance on each CPU, and each scheduler instance has its own queue containing runnable VCPUs of domains [8]. Xen kernel's `schedule()` function[3] preempts the currently running VCPU (scheduler-independent), chooses the VCPU that will run next (scheduler-dependent), and then makes the chosen VCPU run (scheduler-independent). Hence, this function is a suitable place to incorporate the steps A and B presented in Section 2.

Even though there are libraries and APIs (e.g., PAPI [7]) that give developers access to hardware events independently from the underlying architecture, we decided to directly use `RDMSR` and `WRMSR` assembly instructions to set the count of desired hardware events as well as read and clear the CPU's PMC. The reason is that these operations are performed at every context switch and we want the overhead to be as low as possible at the kernel level, in terms of execution time and memory footprint. We then accept the trade off and tie the current implementation to the Intel instruction set; however, other architectures (e.g., ARM and AMD) can be supported by modifying the registers addresses at compile time.

Our current XeMPower implementation only counts *architectural performance monitoring events*. We made that decision because these events have consistent visible behavior across processor implementations [5]. Moreover, previous work shows that they are the most significant metrics to correlate CPU power consumption [28], which is the focus of our motivating use case in Section 4. Since the available

[3]Source code: `xen/common/schedule.c`

Event Mask Mnemonic	Register mapping
Instruction Retired	IA32_FIXED_CTR0
UnHalted Core Cycles	IA32_FIXED_CTR1
UnHalted Reference Cycles	IA32_FIXED_CTR2
LLC Reference	IA32_PMC0
LLC Misses	IA32_PMC1
Branch Instruction Retired	IA32_PMC2
Branch Misses Retired	IA32_PMC3

Table 1: Monitored hardware events [5]

PMCs are limited (e.g., 8 per core and 4 per hardware thread on Intel Sandy Bridge 2nd Gen processors), we map some monitoring events onto 4 PMCs and others are counted using auxiliary *fixed-function* counters. Table 1 summarizes the monitored events and their register mapping.

Regarding power monitoring, Intel RAPL interface provides dedicated read-only registers that can be accessed like standard PMCs. These are available since Sandy Bridge 2nd generation processors and provide CPU power measurements with a time granularity of 1ms approximately. XeMPower currently samples the register `MSR_PKG_ENERGY_STATUS`, which accumulates the actual energy consumption (in Joules) of the whole processor package; the average power consumption is then easily obtained as *energy/time* for the time window considered. For the moment, we decided not to sample the other RAPL power planes (related to on-chip DRAM and "uncore" devices) because their availability varies across different processors.

Finally, we need to expose the collected data to a higher level. For that, we use `xentrace` [8], a lightweight trace capturing facility present in Xen that can record events at arbitrary control points in the hypervisor. We tag every trace record with the ID of the scheduled domain and its current VCPU, as well as a timestamp to be able to later reconstruct the trace flow.

3.2 XeMPower Daemon

The stream of trace records produced by `xentrace` flows from the Xen kernel to the *XeMPower daemon* running in Dom0 (see Figure 1). The daemon, a user-space program written in C, receives the records and performs aggregation operations on them. Note that we do not use the `xentrace` user-space tool, as it can produce a very large amount of data that may potentially cause intense disk writes. Our daemon directly accesses `xentrace` memory buffers, to avoid any additional access to disk.

We defined two bitmasks, `TRC_POWER_PMC` and `TRC_POWER_RAPL`, to differentiate trace records with PMC and RAPL events in the `xentrace` buffers (one per hardware thread). These buffers are constantly monitored by the XeMPower daemon – when a new record arrives, a callback function is invoked to process and store it.

The XeMPower daemon performs aggregations in three stream processing stages. First, records are grouped in tumbling windows with a configurable time interval. Second, in each tumbling window an aggregation is performed per hardware event. In this stage, the deamon also stores the difference between the values of the RAPL energy counter at the beginning and the end of the tumbling window. Finally, in each tumbling window and for each hardware event PMCs are collated per domain. Note that after aggregating the records the notions of physical and virtual CPUs disappear, bringing about a hardware-agnostic data structure.

The XeMPower daemon allocates a shared memory region to store a configurable number of tumbling windows in a circular buffer. Processes other than the deamon can only read from the region. Shared access to the tumbling windows allows multiple front-end applications to read and display different statistics from the same data. The tumbling window time interval, the capacity of the circular buffer of tumbling windows, and other configuration parameters can be specified at compilation time. Currently, the default value for the tumbling window interval is 100 ms and the circular buffer's capacity is 100. These values are used in our experiments reported in Section 5.

3.3 XeMPower Command Line Interface

XeMPower CLI is a basic command line tool written in Python. It periodically scans the tumbling windows produced by the XeMPower deamon (in the shared memory region), and performs aggregations in two time intervals: every second and every 10 seconds. It is also in charge of converting the RAPL counter values into energy consumption values (in Joules). The conversion factor is given by the `MSR_RAPL_POWER_UNIT` register, which is architecture-specific and can be read once when the XeMPower deamon is started. The socket power consumption is then obtained as the ratio of the energy consumption and the considered time interval. XeMPower CLI is designed to show live statistics on console or to log them into a file for a later processing.

4. USE CASE: PER-DOMAIN CPU POWER ATTRIBUTION

As a motivating use case, we describe how XeMPower can perform per-domain attribution of CPU power consumption.

Zhai et al. [28] examined multiple metrics (such as instruction counts, and last-level-cache references and misses) in a wide range of microbenchmarks, including a busy-loop benchmark (high instruction issue rate), a pointer chasing benchmark (high cache miss rate), a CPU and memory intensive benchmark (to mimic virus behavior), and a set of bubble-up benchmarks that incur adjustable amounts of pressure on the memory systems. They concluded that *non-halted cycle* is the best metric to correlate power consumption (linear correlation coefficient above 0.95). Such high correlation suggests that the higher the rate of non-halted cycles for a domain is, the more CPU power the domain consumes.

We then decided to use this result along with the data produced by XeMPower. The approach is simple:

1. For each tumbling window, XeMPower CLI calculates the power consumed by the whole socket, while XeMPower daemon calculates the total number of non-halted cycles (one of the PMC traced).

2. Since we have the number of non-halted cycles per domain, we estimate the percentage of non-halted cycles for each domain over the total number of non-halted cycles. This percentage is adopted as the contribution of each domain to the whole CPU power consumption.

3. Finally, we split the socket power consumption pro-

portionally to the estimated contributions for each domain.

The proposed approach works well even when CPU power states (i.e., C-states and P-states) are enabled. XeMPower is not affected by CPU voltage and frequency scaling, as it continues to measure the actual socket power consumption and to trace and account hardware events consistently.

Note that we do not claim that this trivial use case is highly accurate. Instead, we just present it as an example of how XeMPower enables online attribution of coarse-grained measurements to multiple tenants on a virtualized environment, thanks to per-domain accounting of hardware events.

5. EXPERIMENTAL RESULTS

XeMPower aims to be the tool of choice for any computing system demanding precise and thorough observations of hardware events attributed to domains in Xen. Since the tool is meant to continuously provide statistics at run-time, one of its key requirements is to add negligible overhead to the monitored system. Therefore, in this section we empirically show that XeMPower monitoring components incur very low overhead under different configurations and workload conditions. We defined the *overhead metric* as the difference in the system's power consumption while using XeMPower versus an off-the-shelf Xen 4.6 installation.

5.1 Experimental Setup and Test Cases

Our test platform is a machine equipped with a 2.8-GHz quad-core Intel Xeon E5-1410 processor (4 hardware threads) and 32GB of RAM. We use a Watts up? PRO meter [4] to *independently* monitor the entire machine's power consumption without being influenced by the system configuration in use.

We conduct our experiments under three system configurations: 1) the *baseline* configuration uses off-the-shelf Xen 4.4, 2) the *patched* configuration uses Xen modified as described in Section 3 without running the XeMPower daemon, and 3) the *monitoring* configuration is the same as the patched configuration but with the XeMPower daemon actually running and reporting statistics to an attached console. In all three configurations we assign a single virtual CPU (VCPU) and 4GB of RAM to Dom0, and also dedicate physical core 0 to it. Dedicating core 0 to Dom0, besides adhering to Xen best practices [2], results in that any computational overhead introduced by XeMPower monitoring phase in Dom0 can be measured as an increment in power consumption on core 0 and in the whole system.

We consider four runtime *scenarios*: an *idle* scenario in which the system only runs Dom0, and the *running-n* scenarios, where $n = \{1, 2, 3\}$ indicates the number of guest domains in addition to Dom0. Each guest domain repeatedly runs a multi-threaded compute-bound microbenchmark[4] on three VCPUs and uses a stripped-down Linux 3.14 as the guest OS. The idea in the running-n scenarios is to stress the system with an increasing number of CPU-intensive tenant applications, thus increasing the amount of data traced by the Xen kernel and collected in Dom0.

[4]CoEVP, a simplified proxy material science application from the ExMatEx Center. It is available at https://github.com/exmatex/CoEVP.

Table 2: Mean power consumption (μ), in Watts, for the *pinned-VCPU* test case, scenarios *idle* and *running-{1,2,3}*, and configurations *baseline (b)*, *patched (p)*, and *monitoring (m)*. Mean power values are reported with their 95% confidence interval.

	baseline (μ_b)	patched (μ_p)	monitoring (μ_m)
idle	34.10 ± 0.05	33.72 ± 0.05	33.83 ± 0.05
running-1	56.03 ± 0.09	56.07 ± 0.11	56.19 ± 0.08
running-2	66.14 ± 0.11	66.30 ± 0.06	66.56 ± 0.09
running-3	74.62 ± 0.07	74.60 ± 0.11	74.88 ± 0.29

Table 3: Mean power consumption (μ), in Watts, for the *unpinned-VCPU* test case, scenarios *idle* and *running-{1,2,3}*, and configurations *baseline (b)*, *patched (p)*, and *monitoring (m)*. Mean power values are reported with their 95% confidence interval.

	baseline (μ_b)	patched (μ_p)	monitoring (μ_m)
idle	34.32 ± 0.30	34.14 ± 0.08	34.19 ± 0.05
running-1	70.82 ± 0.10	71.20 ± 0.09	70.78 ± 0.10
running-2	72.99 ± 0.09	73.55 ± 0.12	73.17 ± 0.10
running-3	73.68 ± 1.09	74.67 ± 0.27	74.10 ± 0.09

Finally, we define two test cases for the running-n scenarios. In the *pinned-VCPU case*, each guest domain has each VCPU assigned to a dedicated physical CPU. In the *unpinned-VCPU case*, on the other hand, the guest domains are assigned VCPUs with no physical mapping (i.e., VCPUs can migrate between physical CPUs). The idea is to increase the number of context switches and thereby the amount of traces reported to Dom0.

5.2 Results and Discussion

We compare the power that our test platform consumes for the different scenarios and test cases under the *baseline (b)*, *patched (p)*, and *monitoring (m)* configurations. Under each configuration, we run the idle scenario and the running-1,2,3 scenarios, with and without VCPUs pinned to dedicated physical CPUs (i.e., *pinned-VCPU* and *unpinned-VCPU* test cases). We report the system's mean power consumption (μ) in Watts over a 60-second interval. We performed a set of 40 independent experiments for each [test case, scenario, configuration] combination.

Table 2 and Table 3 present the system's mean power consumption for the *pinned-VCPU* and *unpinned-VCPU* test cases, respectively, across the considered scenarios and configurations. Empirical mean power values are reported with their 95% confidence interval.

At a glance, we can see how measurements are pretty close. However, given the limited accuracy of the power meter, some of them may seem misleading, e.g., the mean power consumption of the baseline case sometimes is higher than the others. This is the reason why we estimate an upper bound ϵ for the maximum overhead by performing the following hypothesis test [18]:

$$T(\mu) := \begin{cases} H_0: & \mu \geq \epsilon + \mu_b \\ H_1: & \mu < \epsilon + \mu_b, \end{cases}$$

where a rejection of the null hypothesis H_0 means that there is strong statistical evidence that the power consumption overhead is lower than ϵ (or equivalently, the mean μ is lower

Table 4: Estimated upper bound ϵ for the power consumption overhead, in Watts, across the considered test cases and scenarios under the *patch (p)* and *monitoring (m)* configurations. Parenthetical values are the percentage overheads with respect to the mean power consumption.

		$\mu = \mu_p$	$\mu = \mu_m$
pinned	idle	<0.01 (<0.02 %)	<0.01 (<0.02 %)
	running-1	0.08 (0.14 %)	0.19 (0.34 %)
	running-2	0.19 (0.28 %)	0.45 (0.67 %)
	running-3	0.01 (0.01 %)	0.34 (0.45 %)
unpinned	idle	<0.01 (<0.02 %)	<0.01 (<0.02 %)
	running-1	0.44 (0.61 %)	0.02 (0.02 %)
	running-2	0.61 (0.83 %)	0.23 (0.31 %)
	running-3	1.18 (1.58 %)	0.60 (0.81 %)

than the baseline mean μ_b increased by ϵ). We compute ϵ for the considered test cases and scenarios, ensuring average values of power consumption (μ) with confidence $\alpha = 5\%$.

Table 4 shows the values of ϵ across the considered test cases and scenarios for the *patched* and *monitoring* configurations. The values in parenthesis represent the percentage overheads relative to the mean power consumption (i.e., μ_p and μ_m, respectively). Our results indicate (with confidence $\alpha = 5\%$) that XeMPower introduces an overhead not greater than 1.18W (1.58%), observed for the [*unpinned-VCPU*, *running-3*, *patched*] case. In all the other cases, the overhead is less than 1W, and less than 1% in relative terms.

This is a satisfactory result when compared to a maximum overhead of 1-2% observed for XenMon [12], which we adopted as a reference point for our XeMPower implementation. We consider this overhead a negligible and reasonable price to pay, given the high-precision information that XeMPower can provide at runtime.

6. LIMITATIONS

We are actively working to bring XeMPower to the next level. It currently offers little flexibility since it monitors a fixed set of PMCs, while we want to be able to configure the set of monitored PMCs at runtime, as well as parametrized tumbling windows for the per-domain attribution of CPU power consumption. Moreover, we want to extend the tool to deal with Non-Uniform Memory Access (NUMA) systems. We plan to evaluate the overhead introduced by such flexibility improvements.

Additional experimental studies will involve different hardware platforms, like ARM-based mobile systems and microservers [1]. Moreover, XeMPower should be evaluated with other than compute-bound workloads.

Finally, the presented approach to power consumption attribution to domains is very simple, as it was a mere example to show the tool's potential. We are currently exploring ways to improve its accuracy, for example with offline characterization of both hardware and guest workloads. As shown in [17, 28, 19], data-driven power models can be exploited at runtime to improve the accuracy of power estimations and to make predictions for the near future.

7. RELATED WORK

Performance monitoring and profiling has always been crucial in every computing system over the last 30 years [11]. The need for a constant monitoring solution has then grown, especially in virtualization environments, where the same hardware is shared between multiple tenants. Unfortunately, every monitoring tool is affected by a tradeoff between *accuracy* and *overhead*; the effective implementation of these systems is then far from trivial. In the literature, this problem has been tackled with two different approaches: code instrumentation and performance counter monitoring.

Code instrumentation solutions, like Valgrind [20] and IgProf [10], inject extra code in the applications at compile time and/or runtime, allowing complex analysis, e.g., on memory and cache accesses. These tools are excellent for an initial analysis of errors and inefficiencies in programs, but are not suitable for performing runtime analysis in production, as the overhead introduced is often high [20].

Performance counter tools, on the other hand, focus on sampling system's events at different granularity (e.g., thread level, process level, set of processors, or the entire systems). These tools provide information on hardware utilization that may not be closely related to the application domain, but their overhead can be tuned accordingly to the actual needs [16]. They differ in functionality, data granularity, level of abstraction, and interfaces they rely on.

Low-level performance counter libraries do not hide architecture-specific event types from the user and lie directly on the hardware. Perf [3] and OProfile [15] are the most popular tools available; they make use of kernel modules to access different categories of events: hardware events, software events (context switches or minor faults), and tracepoint events (disk I/O and TCP events).

Higher-level libraries (e.g., PAPI [7]) hide microarchitecture event types behind a uniform API. They support event multiplexing to compensate for the limited number of performance counter registers that can be monitored at a time: only a subset of the desired event sets is monitored during subsections of a program's execution, then results are scaled to statistically estimate rates for the entire program.

In addition, some works in the literature focus on PMC virtualization [27, 16, 21], providing low-level metrics to virtual tenants. As XeMPower, all these solutions require to patch Xen Hypervisor's kernel to implement operations that require privileged access, such as reprogramming counters or setting up interrupt handlers.

In the context of Xen, the most common solution is Xenoprof [16], a system-wide statistical profiling toolkit based on OProfile and specifically crafted for the hypervisor. It is a valid solution to profile a standard workload running in Dom0 or other domains in *active mode* (i.e., the domain itself collects its own hardware event counters). However, when profiling in *passive mode* (i.e., the domain is treated as a "black box"), the results indicate which domain is running at sample time but do not delve more deeply into what is being executed. Therefore, it does not satisfy the requirement of being agnostic to hosted applications.

Another interesting tool is Perfctr-Xen [21]. It supports performance counter virtualization in Xen for: (1) paravirtualized guest kernels, using hypercalls to communicate performance counter configuration changes to the hypervisor; (2) fully-virtualized guest kernels, using the "save-and-restore" approach for all registers; and (3) a hybrid approach that of-

fers a tradeoff between the first two. Similar to XeMPower, Perfctr-Xen re-programs the Performance Monitoring Unit (PMU) configuration registers (e.g., event selectors) at every context switch. Although this tool is good for workload profiling inside a domain, it is not designed as a centralized runtime monitoring solution.

8. CONCLUSION AND FUTURE WORK

We presented XeMPower, a lightweight monitoring solution for Xen that precisely accounts hardware events to virtual guests. As a motivating use case, we described its use in online attribution of CPU power consumption to individual domains. Our results show that XeMPower can provide continuous statistics with very low overhead compared to an off-the-shelf Xen installation.

As future work, we plan to adopt the tool as a starting point and improve the accuracy of CPU power consumption attribution to domains, considering, for example, other Performance Monitoring Counters (PMCs) in the estimation of domains' contributions. In addition, we plan to explore the complementary use of offline characterization of both hardware and guest workloads in order to predict power consumption before their final deployment.

9. REFERENCES

[1] Intel Xeon processor D product family technical overview. https://software.intel.com/en-us/articles/intel-xeon-processor-d-product-family-technical-overview. Accessed: 2016-07-11.

[2] Tuning Xen for performance. http://wiki.xenproject.org/wiki/Tuning_Xen_for_Performance. Accessed: 2015-11-19.

[3] The unofficial linux perf events web-page. http://web.eece.maine.edu/~vweaver/projects/perf_events/. Accessed: 2015-11-13.

[4] Watts up plug load meters. https://www.wattsupmeters.com/secure/products.php. Accessed: 2015-11-19.

[5] *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume B. 2015. 19-2.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *19th ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[7] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, Aug. 2000.

[8] D. Chisnall. *The Definitive Guide to the Xen Hypervisor*. Prentice Hall Press, Upper Saddle River, NJ, USA, first edition, 2007.

[9] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *18th ACM Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pages 77–88, 2013.

[10] G. Eulisse and L. Tuura. Igprof profiling tool. 2005.

[11] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. 17(6):120–126, 1982.

[12] D. Gupta, R. Gardner, and L. Cherkasova. Xenmon: QoS monitoring and performance profiling tool. *Hewlett-Packard Labs, Tech. Rep. HPL-2005-187*, 2005.

[13] J. Henkel, H. Khdr, S. Pagani, and M. Shafique. New trends in dark silicon. In *52nd ACM/EDAC/IEEE Design Automation Conference (DAC),*, pages 1–6, 2015.

[14] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov. OSv: optimizing the operating system for virtual machines. In *USENIX Annual Technical Conference*, pages 61–72, 2014.

[15] J. Levon and P. Elie. Oprofile: A system profiler for Linux, 2004.

[16] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In *1st ACM/USENIX Int'l Conference on Virtual Execution Environments*, pages 13–23, 2005.

[17] C. Mobius, W. Dargie, and A. Schill. Power consumption estimation models for processors, virtual machines, and servers. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1600–1614, June 2014.

[18] D. C. Montgomery and G. C. Runger. *Applied statistics and probability for engineers*. Wiley. com, 2010.

[19] A. Nacci, F. Trovò, F. Maggi, M. Ferroni, A. Cazzola, D. Sciuto, and M. D. Santambrogio. Adaptive and flexible smartphone power modeling. *Mobile Networks and Applications*, 18(5):600–609, 2013.

[20] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100, 2007.

[21] R. Nikolaev and G. Back. Perfctr-Xen: a framework for performance counter virtualization. 46(7):15–26, 2011.

[22] D. Rossier. EmbeddedXen: A revisited architecture of the Xen hypervisor to support ARM-based embedded virtualization. *White paper, Switzerland*, 2012.

[23] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan. Power-management architecture of the Intel microarchitecture code-named Sandy Bridge. *IEEE Micro*, 32(2):20–27, Mar. 2012.

[24] A. A. Semnanian, J. Pham, B. Englert, and X. Wu. Virtualization technology and its impact on computer hardware architecture. In *Eighth Int'l Conference on Information Technology: New Generations (ITNG)*, pages 719–724. IEEE, 2011.

[25] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter. Cooperation and security isolation of library OSes for multi-process applications. In *9th European Conference on Computer Systems*, 2014.

[26] S. Xi, J. Wilson, C. Lu, and C. Gill. RT-Xen: Towards real-time hypervisor scheduling in Xen. In *2011 IEEE Int'l Conference on Embedded Software*, pages 39–48, 2011.

[27] X. Xie, H. Jiang, H. Jin, W. Cao, P. Yuan, and L. T. Yang. Metis: a profiling toolkit based on the virtualization of hardware performance counters. *Human-centric Computing and Information Sciences*, 2(1):1–15, 2012.

[28] Y. Zhai, X. Zhang, S. Eranian, L. Tang, and J. Mars. Happy: Hyperthread-aware power profiling dynamically. In *USENIX Annual Technical Conference*, pages 211–217, 2014.