energies

MDPI

Article

Containergy—A Container-Based Energy and Performance Profiling Tool for Next Generation Workloads

Wellington Silva-de-Souza [1,*], Arman Iranfar [2], Anderson Bráulio [3], Marina Zapater [2], Samuel Xavier-de-Souza [1], Katzalin Olcoz [4] and David Atienza [2]

1 Department of Computer Engineering and Automation, Universidade Federal do Rio Grande do Norte, Natal 59078-970, Brazil; samuel@dca.ufrn.br
2 Swiss Federal Institute of Technology Lausanne, 1015 Lausanne, Switzerland; arman.iranfar@epfl.ch (A.I.); marina.zapater@epfl.ch (M.Z.); david.atienza@epfl.ch (D.A.)
3 Instituto Federal da Paraíba, João Pessoa 58015-020, Brazil; anderson.silva@ifpb.edu.br
4 Department of Computer Architecture and Automation, Universidad Complutense de Madrid, 28040 Madrid, Spain; katzalin@ucm.es
* Correspondence: wellingtonsouza@imd.ufrn.br

Received: 18 March 2020; Accepted: 11 April 2020; Published: 1 May 2020

Abstract: Run-time profiling of software applications is key to energy efficiency. Even the most optimized hardware combined to an optimally designed software may become inefficient if operated poorly. Moreover, the diversification of modern computing platforms and broadening of their run-time configuration space make the task of optimally operating software ever more complex. With the growing financial and environmental impact of data center operation and cloud-based applications, optimal software operation becomes increasingly more relevant to existing and next-generation workloads. In order to guide software operation towards energy savings, energy and performance data must be gathered to provide a meaningful assessment of the application behavior under different system configurations, which is not appropriately addressed in existing tools. In this work we present Containergy, a new performance evaluation and profiling tool that uses software containers to perform application run-time assessment, providing energy and performance profiling data with negligible overhead (below 2%). It is focused on energy efficiency for next generation workloads. Practical experiments with emerging workloads, such as video transcoding and machine-learning image classification, are presented. The profiling results are analyzed in terms of performance and energy savings under a Quality-of-Service (QoS) perspective. For video transcoding, we verified that wrong choices in the configuration space can lead to an increase above 300% in energy consumption for the same task and operational levels. Considering the image classification case study, the results show that the choice of the machine-learning algorithm and model affect significantly the energy efficiency. Profiling datasets of AlexNet and SqueezeNet, which present similar accuracy, indicate that the latter represents 55.8% in energy saving compared to the former.

Keywords: performance profiling; energy profiling; software containers; performance counters; DVFS

1. Introduction

Data centers are increasing in number, size, complexity and capacity, with a corresponding impact in demand for energy. The electric power spent on data centers reached 0.9% of global energy consumption in 2015 and is expected to reach 4.5% in 2025. [1]. Considering this trend, the development of solutions towards the optimization of energy consumption is critically important. The growing financial and environmental impacts of data centers make optimal software operation a very important

target to enable green and affordable cloud computing. However, the wider configuration space and more heterogeneity of current data center and cloud architectures make software operation more complex.

Ideally, under the hardware perspective, servers should be energy-proportional, i.e., power consumption should be proportional to its utilization [2,3]. In non-energy-proportional servers, power has a close to cubic relationship with throughput. Activating more cores for a computation leads, at first sight, to an increase in power and energy demands. Nonetheless, this can be compensated by the reduced execution time of the parallel computation. Thus, dynamic voltage and frequency scaling (DVFS) combined with multi-core approach can satisfy performance demands while obtaining large energy savings [4,5].

Despite optimal hardware and software design, software operation is critical to energy efficiency. For example, in order to make an application satisfy a given Quality of Service (QoS) requirement, there is no point in using configurations with performance higher than necessary. This would lead to extra energy usage. Thus, a desirable approach would meet application deadline and real-time requirements while guiding software operation towards energy savings.

Energy and performance profiling are fundamental in determining the software operation of parallel applications. However, existing tools focus on performance of individual configurations, i.e., they do not explore software operation fully, especially regarding energy efficiency. In most cases, performance is evaluated as a product of execution time and deadline constraints—application output and associated QoS levels (when applicable) are not considered while evaluating performance, which leads to profiling results of limited applicability. In order to enforce QoS and power consumption requirements, energy and performance data must be gathered and matched chronologically to provide meaningful assessments of the application behavior under different points in the configuration space.

This work focus on presenting a tool that is able to tackle these requirements, using a combination of software containers, control groups, hardware performance counters and DVFS. Containergy generates comprehensive application profiling data by cycling software and hardware parameters on the target system. In particular, our contributions are as follow:

- we propose a profiling tool that uses software containers as a methodology to profile diverse workloads, focusing on energy efficiency;
- the proposed tool uses a combination of control groups, multi-sampling, and performance counters to enable profiling with less interference;
- we present practical profiling results (energy and performance) from case studies of different domains: (a) High Efficient Video Transcoding; (b) Machine Learning/Image Classification.

The paper is organized as follows. Section 2 presents the related works, a comparison of their main characteristics and their differences from Containergy. Section 3 describes the details of the proposed Containergy profiling tool. On Section 4, we present profiling experiments with two case-study applications performed using the proposed tool, whose results are presented and analyzed. Section 5 presents the conclusions emphasizing the evaluated issues and the main contributions of the work.

2. Related Works

cAdvisor [6] is a monitoring and profiling tool specifically designed for containers. It can collect system-wide and per container execution metrics. An important point in cAdvisor is that it provides support for application metrics, which can be incorporated in profiling results. As a drawback, it runs as a daemon inside a concurrent container, which could interfere in the profiling. Also, energy metrics are not available, and there is no support for automatic manipulation of host parameters in order to provide comprehensive results in multiple system configurations.

The Linux Perf Tools [7] is a framework designed to collect performance counter statistics from processes running on a system. It is part of the Linux Kernel Performance Events Subsystem (also known as “perf events”), and includes command-line tools and libraries that can be used to instrument

code. Although it can collect power-oriented performance counters, there is no automatic iteration through system configurations, thus energy profiling support is partial. Also, application metrics are not considered, thus relationship between the latter and performance counters cannot be directly established.

OProfile [8] is an open source profiling framework. It is able to trace application execution through symbolic resolution, evaluating run-time of each part on the compiled code. It is also compatible with perf events and is capable to profile: a single process, through direct command execution or an already running process using its identifier (PID); multiple processes, using system-wide profiling or CPU list. However, it supports only a maximum sampling granularity of 100 ms [9] and there is no support for Precision Event-Based Sampling (PEBS), degrading quality of results [10]. Also, it presents issues (application hang) with PID monitoring and processes that frequently create and destroy threads [11].

With regard to software containers, OProfile is not directly compatible with them, as system-wide and CPU list profiling are not feasible due to interference of concurrent processes and necessity of extra setup on target system (CPU isolation, core affinity). Direct command evaluation is ineffective as well, since container processes are not spawned from container execution commands, but rather from container daemon through IPC. PID monitoring, despite its limitations [11], is the only practicable option, but it requires scripting to collect the real container process identification and detect the end of its execution. Nonetheless, the time lapse between container startup and effective monitoring will incur changes in results.

Another important aspect to mention is that OProfile is not able to provide energy profiling. Neither power-oriented performance counters nor automatic iteration through system configurations are available. It can trace execution and provide insights about code bottlenecks but, as Perf Tools, application output is not considered.

Performance Application Programming Interface (PAPI) [12,13] specifies a application programming interface (API) for accessing hardware performance counters. It provides high and low level programming interfaces for performance assessment, using hardware events in groups called EventSets. PAPI can report power and energy measurement through "RAPL Events", which are based on Intel Running Average Power Limit (RAPL) [14,15]. RAPL events rely on reading the RAPL model specific registers (MSR) directly instead of using perf events. A major drawback of PAPI is the need for code instrumentation, implying making modifications to the application programming and recompiling, which is not always feasible (lack of deep understanding on the source code; rework every new version), or possible (source code not available). Moreover, PAPI does not provide a toolset or command-line utilities for profiling.

Some third-party tools extend PAPI and partially address its issues. Tuning and Analysis Utilities (TAU) [16] provide automatic instrumentation, but only for codes written in Fortran, C, C++, UPC, Java and Python. Open|SpeedShop (O|SS) [17] utilizes dynamic binary instrumentation, not requiring source code modification or recompilation. PapiEx [18] provides a command-line interface to PAPI and allows profiling uninstrumented applications, working similarly to Perf Tools (including its limitations).

Containergy differs from previous tools in key aspects, summarized on Table 1. By design choice, it does not rely on code instrumentation in order to simplify its utilization and make it useful in situations where the interest of analysis lies only on the normal execution of the application or when source code is not available. As a trade-off, it cannot be used to identify inner performance bottlenecks or to assess energy on specific parts of the code. The tool can profile container-packed applications while isolating hardware performance metering by using control groups. Likewise, it extracts application metrics to evaluate performance. Energy measurements are also available. Also, it performs automatic exploration of configuration space, cycling parameters in order to provide a comprehensive profiling resultset.

Table 1. Profiling tools—Comparison.

	cAdvisor	Perf	OProfile	PAPI	TAU	O|SS	PapiEx	Containergy
energy metrics	-	X	-	X	X	X	X	X
application metrics	X	-	-	-	-	-	-	X
container profiling	X	-	-	-	-	-	-	X
code instrumentation	-	X	X	X	X	-	-	-
configuration cycling	-	-	-	-	-	-	-	X

3. Container-Based Energy and Performance Profiling

This work proposes Containergy, a profiling tool focused on collecting energy and performance data from containerized applications. It combines the use of software containers, control groups, hardware performance counters and DVFS to isolate the application and allow users to assess its execution over different hardware and software setups.

In order to accomplish the aforementioned statements, we defined the following requirements. The proposed tool should:

- enable profiling of container-packed applications;
- isolate the profiling environment from other running tasks on the target system;
- collect energy and performance data, matching software and hardware measurements chronologically;
- provide automatic setup cycling throughout the configuration space;
- allow statistics on multiple-run experiments.

These requirements are analysed and addressed in Section 3.1, resulting in the implementation detailed in Section 3.2. Section 3.3 presents a practical application of the data that the proposed tool generates.

3.1. Overview of the Approach

Next, we provide an explanation of how the requirements previously introduced are fulfilled by Containergy. The first two requirements were tackled with "control groups" and "software containers". Control groups (cgroup) is a mechanism to distribute system resources among processes that are organized in hierarchical trees [19]. Assigning a cgroup to each container provides process isolation and control over resource utilization, such as cores, memory and input/output, as well as isolated accounting. Additionally, Performance Counters are also evaluated per cgroup, providing a simpler and more accurate mechanism for obtaining hardware metrics when compared to CPU isolation and processor affinity.

The combination of cgroups with "namespaces" forms the basis of lightweight process virtualization, also known as "software containers" [20]. Container technology aims to provide a self-contained computing unit to run an application in a computational environment [21,22]. In comparison to hypervisor-based virtualization, containers provide similar level of isolation with considerably less overhead and better performance [23–25].

A software container uses the resources of cgroups to separate application execution from other running tasks. It provides the additional feature of packaging the software and complementary files (libraries, configuration files, binaries), reducing dependency and library conflicts (version, file system location) on the target system. In turn, this makes software distribution and utilization easier, specially in cloud computing environments.

Containergy profiles software containers through their associated control groups. Process isolation and accounting provide an ideal environment to extract run-time measurements with low overhead and interference.

The requirement of collecting energy and performance data with the proposed tool is tacked with a combination of Performance Monitoring Counters (PMC), container log parsing and timestamps.

In modern processors, the Performance Monitoring Unit (PMU) enables the extraction of low-level execution information through PMCs. The proposed profiling tool uses Performance Counters to collect raw energy and performance data from the hardware. In parallel, application data is collected using container logs.

Hardware and software data are matched using timestamps within a microseconds resolution. This process is performed by Application and Performance Data Assembler block, detailed on Section 3.2.1.

The last two requirements are addressed by automatic switching of system configurations and by performing multiple executions of the application.

Current architectures provide processors with multiple working frequencies accessible by the DVFS settings. Also, the accounting of hardware-threads can be defined per process. The combination of these features creates a wide configuration space and poses some challenges to the process of assessing application execution since it affects directly energy and performance. Manual setup for profiling is a counterproductive and error-prone process.

Containergy scans and iterates automatically throughout the configuration space, adjusting the frequency and allocating hardware-threads to container execution. This task is coordinated by the Profiling Orchestrator, defined on Section 3.2.1. Also, it has a built-in configurable container starter to control multiple executions of the same application. The Multicore Input Parser, Preprocessor and Statistics Generator, also presented in Section 3.2.1, generates the multiple-runs statistics.

3.2. Detailed Workflow

In this Section we summarize Containergy's implementation, operation and workflow. Figure 1 shows an overview of the tool. Elements in darker colors are inner components of an entity (e.g., MIPPSG → Containergy). Different colors represent distinct entities. DVFS Governor has a gradient color due its hybrid nature (Software ↔ Hardware).

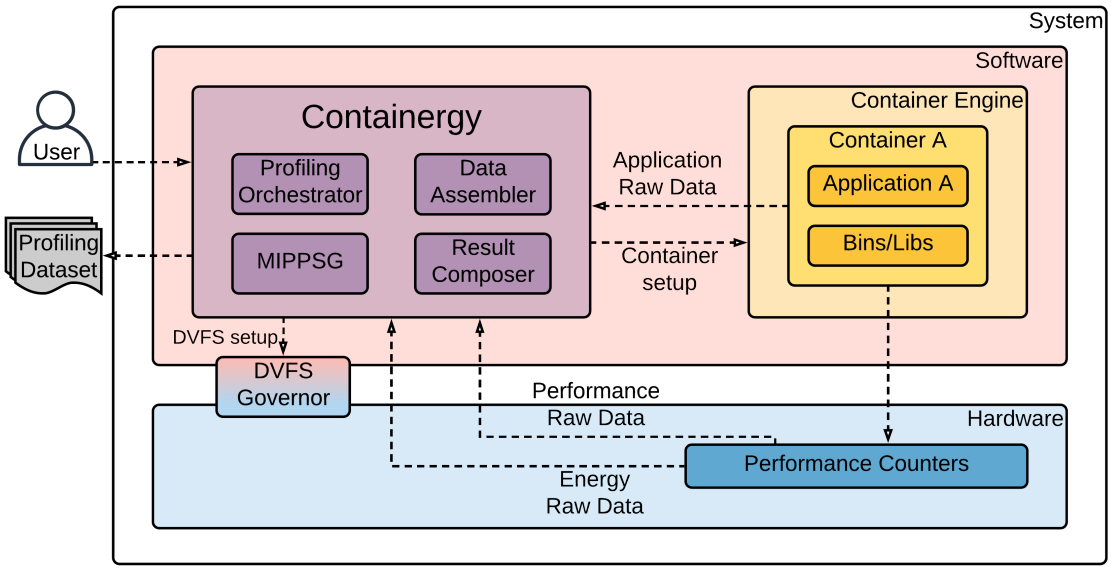


Figure 1. Containergy Overview.

The user interacts with the Containergy by setting the profiling parameters: container image to be profiled, performance counters to evaluate, sampling rate (in milliseconds) and number of executions for multi-sampling (see Section 4.2). The tool collects raw data generated by applications inside software containers. Meanwhile, it adjusts the frequency and number of threads and extracts raw energy and performance data from performance counters and the software output. Then, the tool combines and processes these raw values, resulting in a comprehensive profiling dataset.

All steps are performed automatically throughout the configuration space of the target system, according to the execution sequence presented on Figure 2 and defined as follows:

1. DVFS setup: adjusts hardware execution settings (frequency);
2. container setup: runs application inside a software container with a specific number of threads and proper software parameters;
3. application and performance monitoring: collects raw profiling data in runtime;
4. data assembly: combines asynchronous application data and raw performance data;
5. parsing and statistics generation: parses and processes combined raw data (*per* configuration), using multicore task distribution;
6. result composition: combines *per* configuration profiling data, generating a comprehensive profiling dataset.

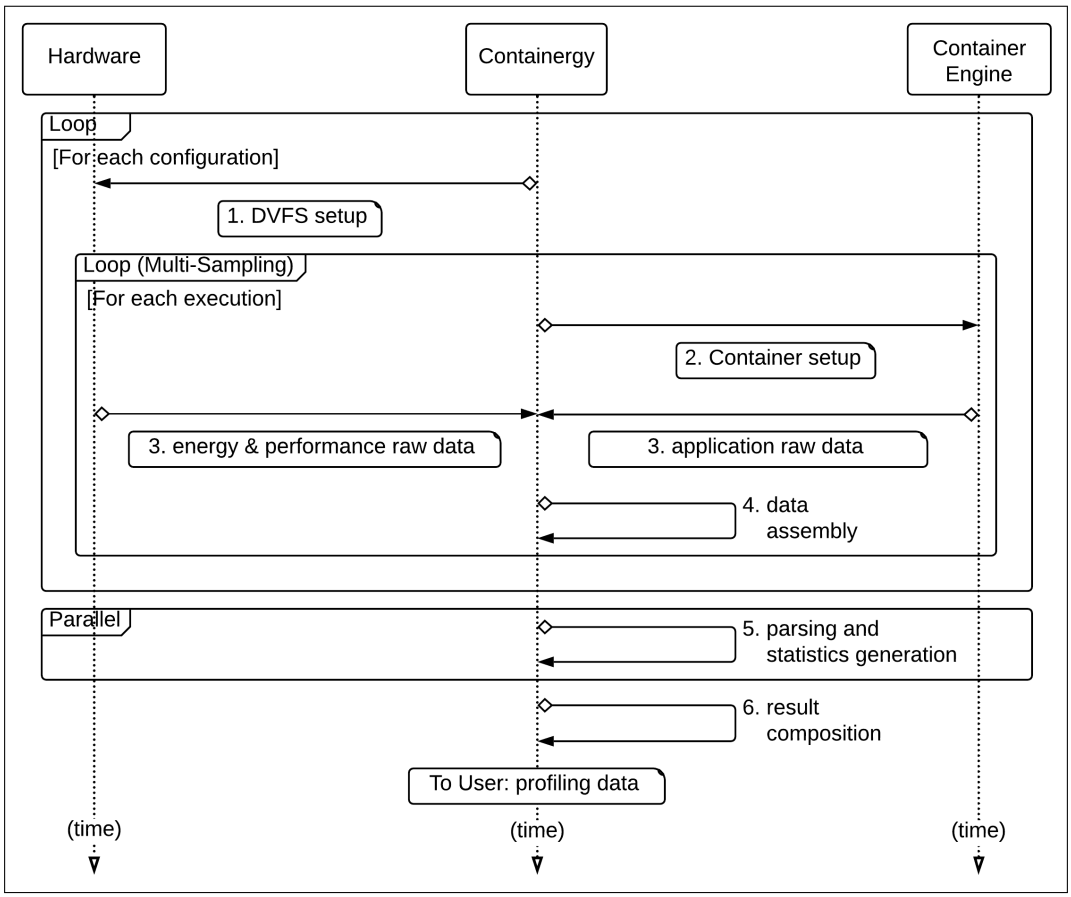


Figure 2. Containergy Sequence Diagram.

The tasks of the execution sequence are executed by Containergy's building blocks, detailed as follows.

3.2.1. Building Blocks

The tool is formed by building blocks, where different records (raw application data, performance counters, configuration, etc.) flow in order to control application execution and profiling. System related activities and file handling were implemented with shell scripting and data manipulation routines were written in Python.

The interaction between blocks is accomplished by using intermediate datastores. Figure 3 presents the building blocks and the data flow across them.

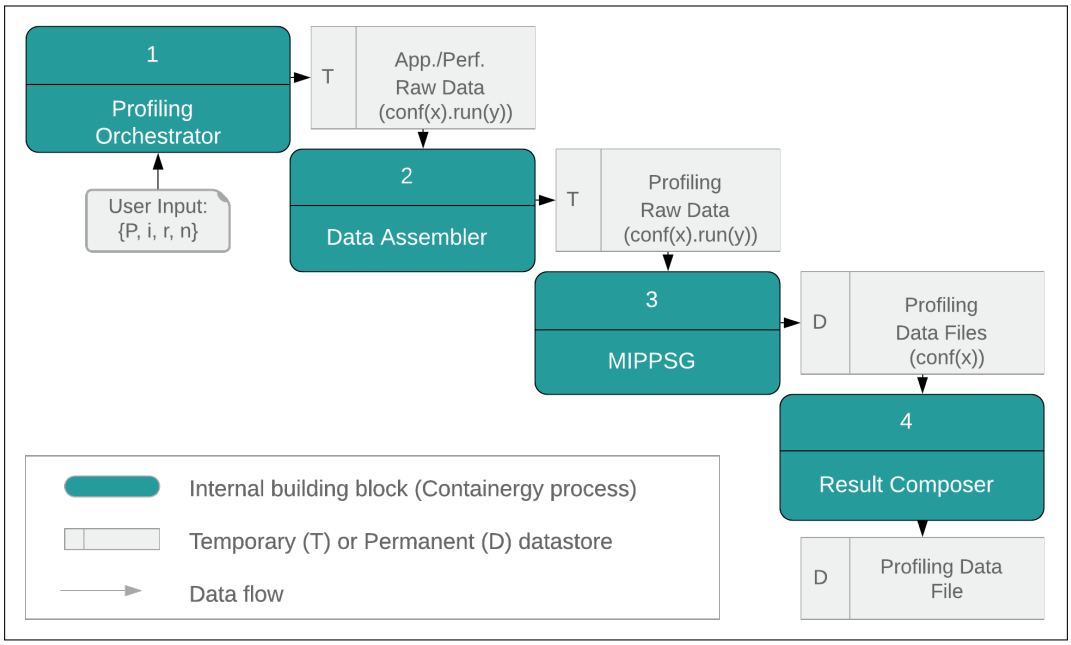


Figure 3. Containergy Data Flow Overview.

The Profiling Orchestrator receives the user input, defined in Section 3.2. As output, it generates application and performance raw data. In comparison with previous works (detailed on Section 2), this building block implements as novelty the use of software containers as workloads, control groups to isolate application execution and metering, and automatic configuration cycling.

Algorithm 1 presents the pseudo code of what is performed by this block. Line 1 executes an outer loop, iterating through the frequency set and adjusting DVFS Governor accordingly (2). An inner loop (3) evaluates each frequency and number of threads (f, t) combination, performing multi-sampling (4) when defined by the user. Line 5 creates a container instance from the base image (specified by user) on top of a predefined cgroup. Raw data are extracted (6) and stored on intermediate datasets (7,8). Finally, the container instance is destroyed and the cgroup flushed (9).

Algorithm 1: Orchestration

Input : C(PMCs P, Image i, Sampling_Rate r, Runs n): user defined,
F_s: set of all frequencies on the target system,
T_s: set of all thread combinations on the target system

Output: $AppRawData_{F_sT_s}$, $PerfRawData_{F_sT_s}$

1 **foreach** $f \in F_s$ **do**
2 set $DVFS\_Governor(f)$;
3 **foreach** $t \in T_s$ **do**
4 **for** $i = 1$ **to** n **do**
5 define cgroup g and an exec container c_t from i on g;
6 extract raw data $d_1, d_2 \leftarrow c_{tr}, P_{gr}$;
7 put d_1 in the set $AppRawData_{F_sT_s}$;
8 put d_2 in the set $PerfRawData_{F_sT_s}$;
9 unset c_t, g;

The Application and Performance Data Assembler block collects raw data asynchronously in separated buffers. In order to provide correct data correlation over time, a microsecond-timestamp-based assembler is used to combine both outputs. Considering that data

sampling is performed on a millisecond basis, microsecond-timestamps provide adequate resolution for data regrouping in case of overlapping.

For each configuration ft ($f \in F_s$, $t \in T_s$), the Multicore Input Parser, Preprocessor and Statistics Generator (MIPPSG) receives as input raw profiling data files from a single or multiple executions of the workload. Application and performance data are parsed and combined in intrasampling or intersampling (see Section 4.2). The output of the block is a profiling data file *per* input configuration. The MIPPSG and Application and Performance Data Assembler blocks enable Containergy to extract the software output information and relate it to the simultaneous hardware performance and energy measurements, representing novelty over previous works (see Section 2).

To speed up processing, MIPPSG tasks are spread throughout available cores. A major advantage on the design of MIPPSG as a separate post-processing stage is that it reduces overhead in the profiling phase. Also, it can be performed on a different host, reducing the overall usage time on the target system. This is especially important in shared-use environments such as HPC and data centers. Furthermore, it speeds up the profiling process on targets with reduced processing power (such as embedded devices) since this step can be performed on a host with larger processing power.

The Result Composer is responsible to consolidate results from all configurations of a PMC group into a single profiling data output file, representing the final output of profiling. It can be used for performance and energy assessment, modeling, and statistical evaluation. Section 4.1 presents case studies that use this output for practical purposes. The absence of automatic configuration cycling in other tools implies the need for a manual consolidation process *per* setup, which is not necessary on Containergy (see Section 2 for more details).

3.3. Application Case: QoS Optimization

As an application case for the results obtained with Containergy, we present a general QoS optimization methodology for energy saving, which is defined in Algorithm 2.

Algorithm 2: QoS Compliant Energy Optimization

Input : *ProfilingDataset*, *QoS_feasiblevalues*
Output: *best_config*, *FeasibleConfigs*

1 **foreach** *configuration* $f, t \in$ *ProfilingDataset* **do**
2 $perf \leftarrow$ performance data from *ProfilingDataset(f,t)*;
3 **if** $perf \in$ *QoS_feasiblevalues* **then**
4 *EnergyDataset(f,t)* $\leftarrow$ energy data from *ProfilingDataset(f,t)*;
5 put f, t in *FeasibleConfigs*;
6 best_config $\leftarrow$ $\underset{f,t}{\textbf{arg min}}$ *EnergyDataset(f,t)*;

As input, the algorithm takes data returned by Result Composer and target QoS thresholds (feasible values). Then it iterates over the profiling dataset (line 1) determining feasible configurations (which performance meets QoS constraints; lines 2–5), and selecting the one that minimizes energy consumption (line 6). As output, it presents the set of setups that comply with the QoS, highlighting the best one in terms of energy saving.

4. Experimental Results

This section presents an experimental evaluation of the proposal. The experiments have the objective of appraising its effectiveness in assessing and comparing energy and performance of:

- **objective A**: an application, varying the input, i.e., single application, distinct input-sets, and single target-system;

- **objective B**: similar applications using the same input, i.e., distinct applications, single input-set, and single target-system;
- **objective C**: a workload on different hardware setups, i.e., single application, single input-set, and distinct target-systems;
- **objective D**: a workload subject to QoS, i.e., single application, single input-set, single target-system with QoS thresholds.

The applications used as case study and hardware setup are presented in Section 4.1; the selected PMCs, the sampling strategy and overhead in Section 4.2; and the results on Sections 4.3 and 4.4.

4.1. Case Study Applications

In order to evaluate the effectiveness of the proposed tool in profiling and retrieving meaningful energy and performance data, we selected two emerging workloads from different domains as a case study: HEVC Video Transcoding and Machine Learning Image Classification. We justify our selection considering that while the latter tends to be Memory/IO-bounded, the former is typically CPU-bounded. Moreover, these applications represent the next generation of workloads that will become increasingly present in HPC and data center environments over the next years.

The experiments were performed on two different setups: A—consumer grade workstation; B—enterprise grade server, according Table 2. These settings have been chosen to provide a more comprehensive evaluation of the tool, by using a variation not only of application and input loads, but also of the execution platform.

Table 2. Software & Hardware Setup.

	A	**B**
setup type	consumer grade workstation	enterprise grade server
operating system OS kernel	Ubuntu 18.04.1 LTS Linux Kernel 4.15.0-22	CentOS 7.3.1611 Linux Kernel 4.17.2
CPU set frequency range	1x Intel Core i7 CPU 800 MHz to 4.2 GHz	2x Intel Xeon E5 v4 1.2 GHz to 3.6 GHz
cache	1 × 8 MB	2 × 20 MB
RAM	16 GB	128 GB
threads × freq.	8 × 16	32 × 16
configuration set	**128**	**512**

4.2. PMC Selection, Sampling Strategy and Overhead

Containergy allows the user to select desired PMCs and to define how the tool deals with multiple samples of the same data. However, depending on the CPU architecture, the list of available PMCs can be considerably large (thousands). Literature [26–28] shows that adequate PMCs for parallel applications fit into these groups: cache-misses, bus-access, retired instructions and micro-operations, CPU utilization, memory utilization, and energy. Considering this, we select a default base set of performance counters to be used with the tool, which is presented on Table 3.

This selection is also a novelty defined by the proposed tool. It makes easier the process of modeling a software operation under the perspective of performance and energy, considering their relationship with good candidates for metering at the hardware level (PMCs with high correlation to application performance).

Table 3. Default Performance Counters Set in Containergy.

	Category
energy-pkg, energy-ram	Energy (J)
bus-cycles	Bus-Access (1)
resource_stalls, branch-misses	CPU Utilization (1)
l2_trans.all_pf, llc_references.code_llc_prefetch	Memory Utilization (1)
instructions, μops_executed, fp_arith_inst_retired.scalar_double	Instructions & μ-Operations (1)
l2_rqsts.l2_pf_miss, mem_load_$u$ops_retired.l1_miss, mem_load_$u$ops_retired.l2_miss, mem_load_$u$ops_retired.l3_miss, llc_misses.mem_read, llc_misses.mem_write, offcore_response.all_code_rd.llc_miss.local_dram	Cache-Misses (1)

When performing multiple executions the samples are combined, resulting in statistical data for analysis. It can be used in two modes of operation:

- Intrasampling: The user can define to retrieve the same performance counter multiple times. If a PMC is selected twice or more on the P set, the multiple values for that counter in each sampling instant will be combined (average) in a sole result. It is performed by MIPPSG in the preprocessing and statistics generation phase. This feature is specially useful to analyse the PMC multiplexing effects (sharing a counter among several event measurements over a time period [29]).
- Intersampling: Containergy can be configured to execute multiple runs of the same workload. In this case, the result of a PMC p will be the average of p on the same timeframe t across each execution. In addition, percentage standard deviation, minimum and maximum values will be reported. This way of execution allows for a reduction of errors due to external interference during application execution, as well as for an evaluation of the variability of results. Its execution is the responsibility of the Profiling Orchestrator and MIPPSG.

Case studies were performed on intersampling mode with five executions for each configuration and workload, and a sampling period of 100 milliseconds. To avoid counter multiplexing, we limit the number of simultaneously collected PMCs during run-time.

To determine the overhead introduced by proposed tool, we measured the complete execution time for all workloads presented in previous Section, comparing the results while profiling by Containergy and without profiling. The average overhead measured over 100 runs was 1.18%.

4.3. HEVC Video Transcoding

High Efficiency Video Coding (HEVC) [30] appears as a promising solution for next generation video encoding standards. In comparison with its predecessor (H264), it needs half bandwidth/storage space to provide same video quality, but at the cost of more processing and hence more energy.

We have taken in consideration some key aspects to choose this workload: relevance to actual Internet scenario, high parallelization opportunities due to its design (good applicability to save energy in multi-core cloud/HPC environments) and QoS constraints (threshold ranges in terms of

quality and performance). Our tests were performed using Kvazaar [31] as HEVC transcoding tool and industry standard video test sequences [32]: RaceHorses_832x480_30.yuv (hereinafter defined as VT1), E_Johnny_1280x720_60.yuv (VT2), and Kimono1_1920x1080_24.yuv (VT3). As QoS constraint, we defined an average range of 25–30 frames per second (FPS).

We start our considerations through a quantitative analysis on max/min values gathered in our tests, which are presented on Table 4. Application performance (FPS) was highest in the most resource consuming configuration, except in VT2@A (#7 threads). The difference between the maximum and minimum performance was 10× in experiments run in server A (where the differences in energy and power where 5× and 13× respectively) and it was 33× on VT3@B, where differences were 11× for energy and 4× for power.

Table 4. HEVC—Energy and Performance.

	Performance (FPS)		**Energy (J)**		**Power (W)**	
	min	**max**	**min**	**max**	**min**	**max**
VT1@A	**9.94**	**108.66**	**52.84**	**274.87**	**3.72**	**48.82**
threads (#)	#1	#8	#8	#1	#1	#8
freq. (GHz)	1.2	4.2	1	4.2	0.8	4.2
VT2@A	**12.66**	**125.26**	**81.29**	**436.35**	**3.65**	**45.98**
threads (#)	#1	#7	#8	#1	#1	#5
freq. (GHz)	1	4.2	1.2	4.2	0.8	4.2
VT3@B	**2.4**	**80**	**255.56**	**2829.9**	**27.37**	**108.81**
threads (#)	#1	#32	#10	#1	#9	#13
freq. (GHz)	1.2	3.6	2.9	1.2	1.2	3.6

Almost all max/min values exceed the defined QoS limits. Thus, we deepen the analysis in search for the configuration that presents better energy efficiency but respecting QoS constraints. This was determined by Algorithm 2.

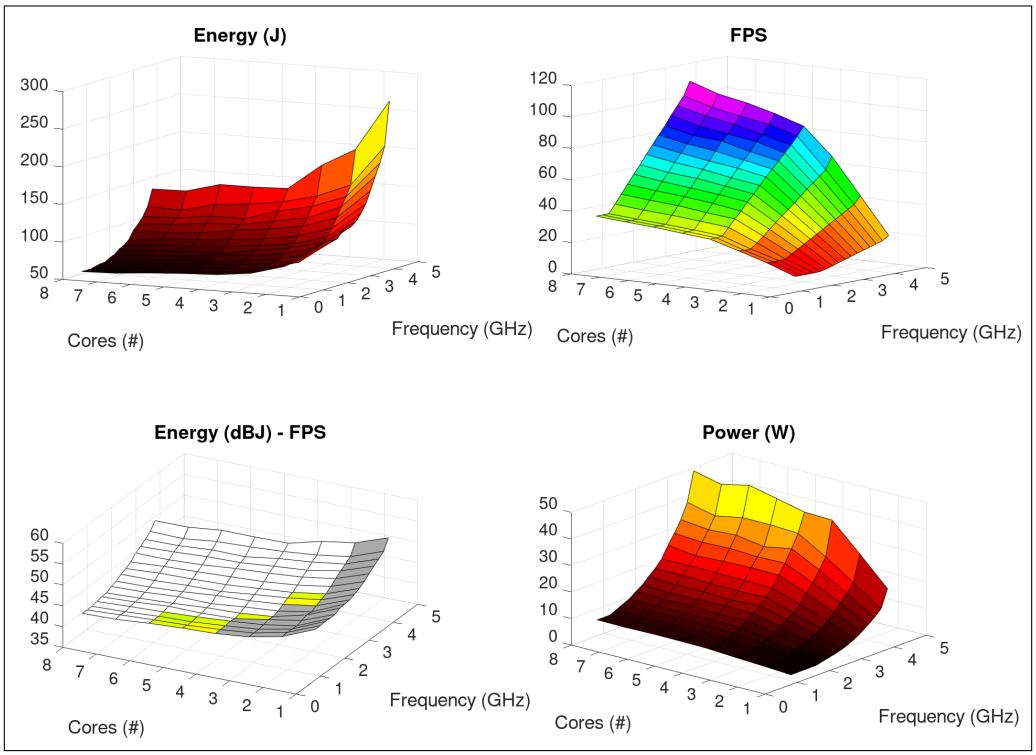
Figure 4a shows Containergy performance/energy profiling results of HEVC application transcoding test sequence VT1 on setup A. In the figure, the sub-graph "Energy (dBJ)—FPS" combines Energy (on a logarithmic scale) and FPS charts, selecting (coloring) feasible configurations—i.e., that meet QoS constraints (**objective D**), according to Algorithm 2. Gray areas represent settings that lead application to perform below target threshold (<25 FPS). White regions group system configurations that exceed QoS levels (>30 FPS).

Slight modifications in configuration imply in QoS violation, since useful configuration regions are non-contiguous. As an example, consider #3 @ 1.6 GHz (the isolated "green" configuration near the center of Energy (dBJ)—FPS chart), which performs 28.53 FPS. Adjusting system to #2 @ 1.6 GHz and #4 @ 1.6 GHz leads performance to 19.25 and 37.47 FPS. Similarly, #3 @ 1.4 GHz and #3 @ 1.8 GHz (surrounding frequencies) result in 24.99 and 32.16 FPS, respectively.

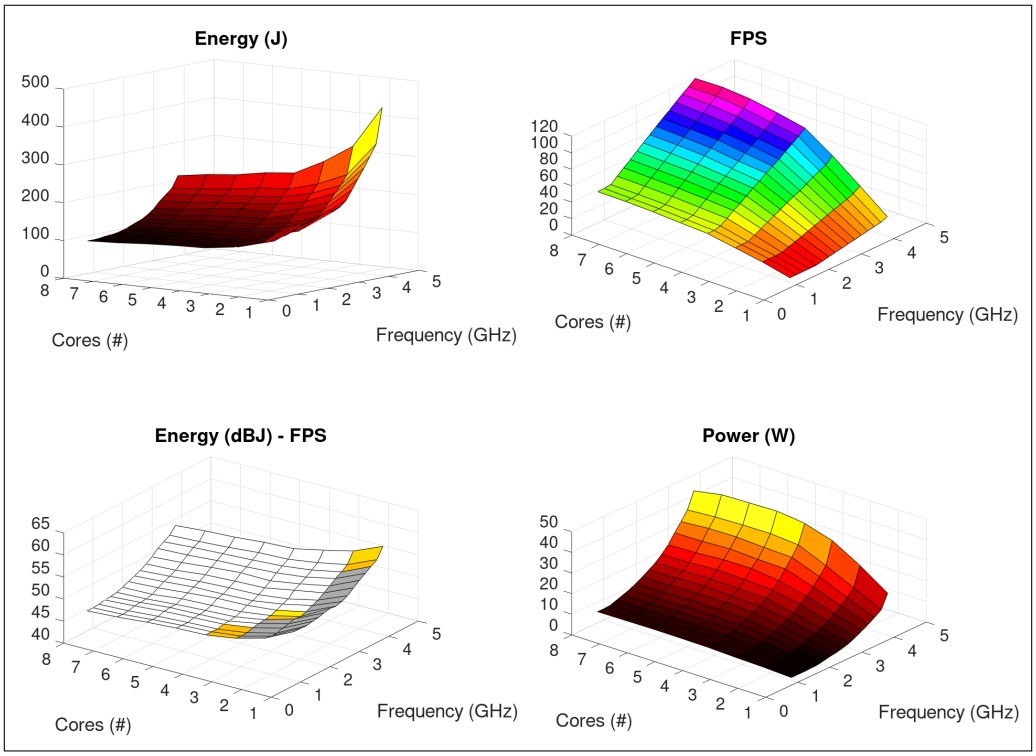
Following **objective A**, we compared VT1@A (Figure 4a) and VT2@A (Figure 4b). It is easy to verify that energy/performance relationship depends on the content of the input video. In the Figure 4b, interest configurations are represented on Energy (dBJ)—FPS chart. The optimal setting, defined by Algorithm 2, is #3 @ 800 Mhz. Conversely, adjusting execution to #1 @ 4.2 GHz leads to the worst energy scenario, with an increase above 300% in comparison with the results obtained with the best configuration (**objective D**).

Figure 5 shows results of VT3@B. Between #10→#11 and #29→#30 threads, the application/system presents a disparity from the other regions, especially in power gradient. While the average variation for each thread added was ±5.47%, for #10→#11 it was +74, 8% and for #29→#30, +25, 8%.

This reflects out changes in application parallel behavior [33] in association with threads placement (core/socket) in the NUMA system, changing C-states in the CPUs. Analyzing these details is beyond the scope of this work. However, profiling results obtained with Containergy can support these studies.



(**a**) RaceHorses_832x480_30.yuv



(**b**) Johnny_1280x720_60.yuv

Figure 4. HEVC—profiling on a Intel Core i7-7700 system. On Energy and Power, lighter colors represent higher values. FPS colors are defined in HSV scale, corresponding to application performance levels. On Energy—FPS chart, gray represents settings with application performance below defined threshold; white regions group system configurations that exceed QoS levels; HSV colors reflect out same application performance levels as FPS.

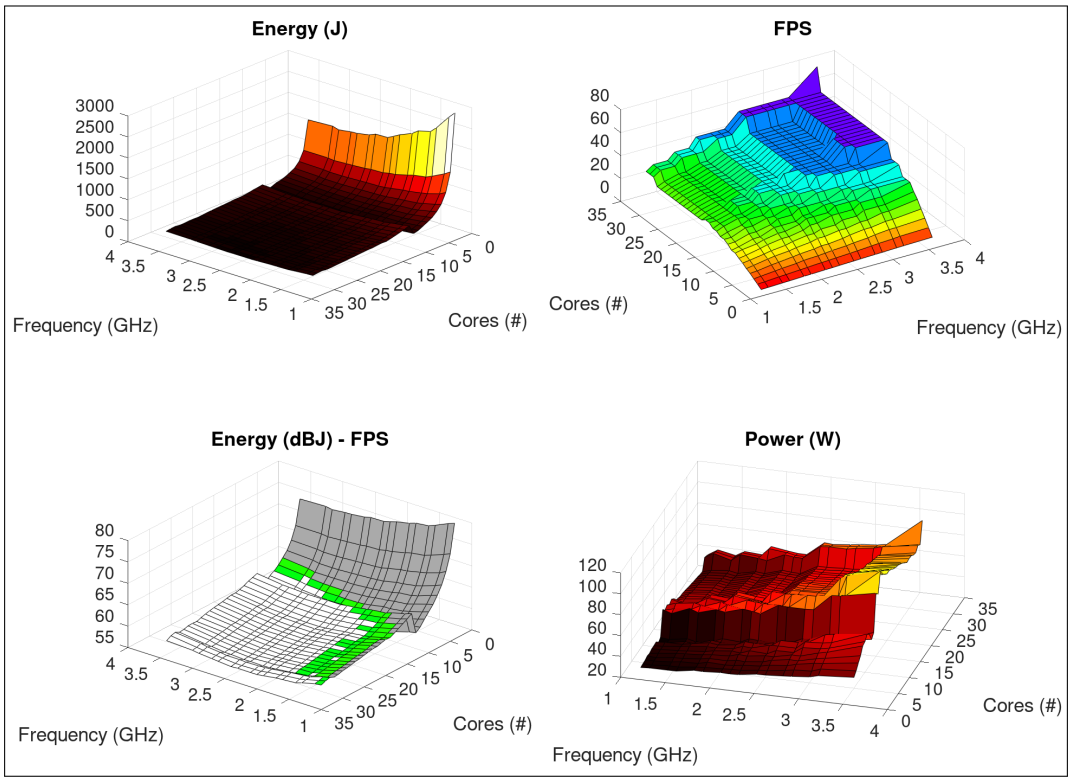


Figure 5. HEVC: Kimono1_1920x1080_24.yuv—profiling on a dual-Intel Xeon E5 v4 system, using same color scheme of Figure 4.

Green region on Energy (dBJ)—FPS chart complies with QoS and includes settings with this peculiar high-gradient characteristic. Depending on HPC/datacenter power restrictions (budget, energy acquisition/generation), tradeoff between energy, power and application performance can determine the policy to be applied, eventually skipping these configurations.

4.4. Machine Learning Image Classification

Considering the reawakening of artificial intelligence usage in recent years, especially as a result of the advent of new techniques as convolutional neural networks (CNN), it is expected that machine learning will play an important role as workload in computing facilities. We evaluated AlexNet [34,35] (ML1) and SqueezeNet [36,37] (ML2) with pre-trained weights on TensorFlow [38], performing image classification (inference) on a subset (500 images, randomly selected) of ImageNet [39] validation dataset.

Albeit this type of application is usually run on a best effort to performance approach, we defined an average range of 25–30 inferences per second (IPS) as QoS constraint in order to verify its consequences in terms of energy savings. This configuration was chosen considering a hypothetical scenario of real-time operation on a live video stream, with one inference per input frame.

We evaluated the effects of QoS applied on the target ML applications (**objective D**), the results of which are presented in Table 5. Comparing to the best effort approach, the best QoS configuration resulted in energy savings/power reduction of 30%/40.53% (ML1@A), 51,37%/73.25% (ML2@A) and 14.34%/44.56% (ML2@B).

Table 6 presents maximum and minimum values for machine-learning workloads. It is possible to verify that the maximum/minimum ratio was lower compared to HEVC transcoding workloads (Table 4).

In previous case we compared the same application with different inputs (VT1@A/VT2@A). Here we compare different applications (but equivalent in results) with the same input (**objective B**). Table 6 shows also that SqueezeNet had better performance and energy efficiency than AlexNet. Max

performance was 55.2% higher. While maintaining similar average instant power, difference in max energy consumption was 73.4%, as a result of contraction in execution time and the consequent increase of IPS. Using the same system configuration (#1 @ 4.2 GHz), AlexNet needed 28s to complete the task, while SqueezeNet, 17s.

Table 5. ML—QoS_E_{min} X Best Effort.

	ML1@A		**ML2@A**		**ML2@B**	
	QoS E_{min}	**Best Effort**	**QoS** E_{min}	**Best Effort**	**QoS** E_{min}	**Best Effort**
Perf. (IPS)	25	29.41	25	45.46	29.41	45.46
exec. time (s)	20	17	20	11	17	11
Energy (J)	339.59	485.34	149.96	308.39	135.1	157.72
Power (W)	16.98	28.55	7.5	28.04	7.95	14.34
threads (#)	#8	#8	#6	#8	#31	#32
freq. (Ghz)	3.2	4.2	2.2	4.2	2.2	3.6

Table 6. ML—Energy and Performance.

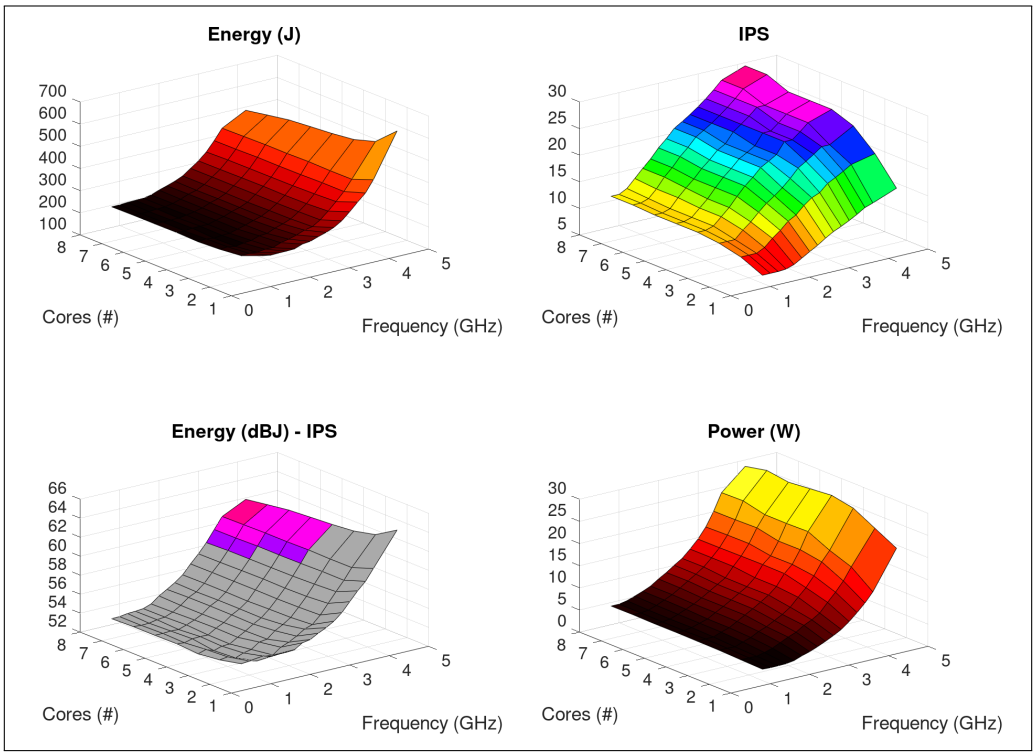
	Performance (IPS)		**Energy (J)**		**Power (W)**	
	min	**max**	**min**	**max**	**min**	**max**
ML1@A	**7.46**	**29.41**	**191.30**	**666.77**	**3.6**	**29.86**
threads (#)	#1	#7	#4	#1	#1	#4
freq. (GHz)	0.8	4.2	1.2	4.2	1	4.2
exec. time (s)	67	17	45	28	67	18
ML2@A	**12.2**	**45.46**	**128.57**	**384.56**	**3.5**	**28.56**
threads (#)	#1	#8	#7	#1	#1	#3
freq. (GHz)	0.8	4.2	1.2	4.2	0.8	4.2
exec. time (s)	41	11	32	17	41	13
ML2@B	**10**	**45.46**	**130.22**	**887.53**	**5.77**	**31.87**
threads (#)	#1	#6	#8	#1	#32	#1
freq. (GHz)	1.2	3.6	2.6	1.2	1.2	3.6
exec. time (s)	50	11	15	50	29	18

Figure 6a,b presents energy and performance profiling of AlexNet and SqueezeNet, respectively, on setup A. Considering the QoS region (**objective D**), SqueezeNet achieved best configuration (minimum energy) with #6 @ 2.2 GHz, with an average of 149.96 J, 7.5 W, and 25 IPS. AlexNet's best setting was found with #8 @ 3.2 GHz, with an average of 339.59 J, 16.98 W and 25 IPS. Thus, complying with QoS constraints, SqueezeNet represents 55.8% in energy saving compared to AlexNet.

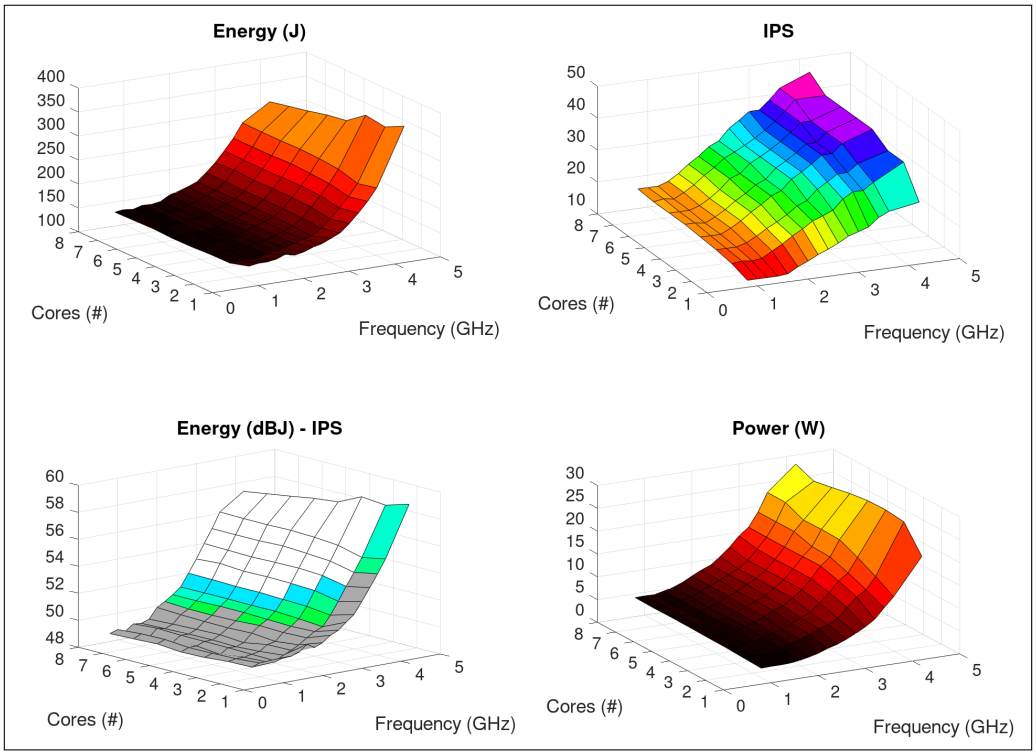
It is possible to justify the difference in results based on the neural network model size. Whilst presenting similar accuracy, AlexNet's model has 233 MB while SqueezeNet's model, on the other hand, occupies 5 MB. Therefore, the number of computations for a single inference is bigger in AlexNet, implying in more processing and energy usage.

Considering (**objective C**), we compared ML2@A with ML2@B. Performance (IPS) on setup A was higher in more robust configurations, as observed on HEVC transcoding. However, on setup B results show these applications present a very distinct behavior with regard to threads .

A parallel saturation point was reached with 6 threads, which can be better observed on Figure 7. It is possible to verify that IPS creates a plateau for each frequency when #threads $\geq$ 6. We can observe QoS region on the Energy (dBJ)—IPS graph. The worst QoS compliant configuration (#1 @ 3.2 GHz) leaded the system to use 588.93 J, 4.36 times higher than the best setting (**objective D**).



(**a**) AlexNet



(**b**) SqueezeNet

Figure 6. ML image inference—profiling on a Intel Core i7-7700 system. On Energy and Power, lighter colors represent higher values. IPS colors are defined in HSV scale, corresponding to application performance levels. On Energy—IPS chart, gray represents settings with application performance below defined threshold; white regions group system configurations that exceed QoS levels; HSV colors reflect out same application performance levels as IPS.

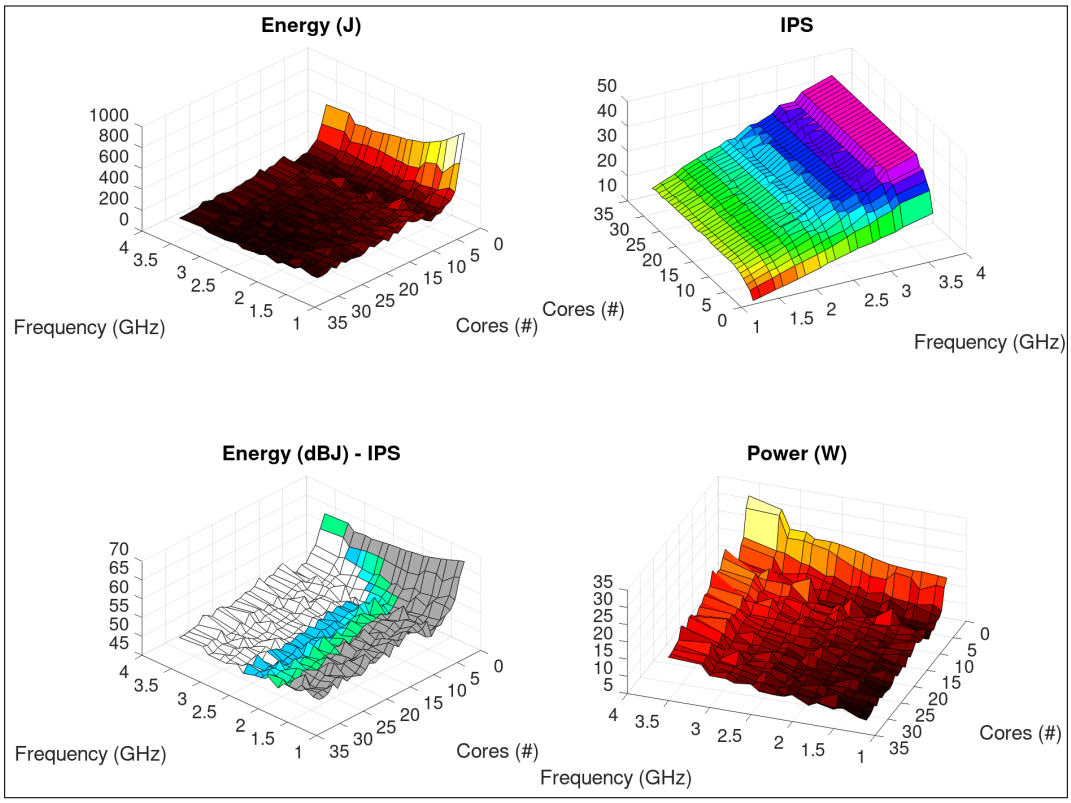


Figure 7. ML image inference: SqueezeNet - profiling on a dual-Intel Xeon E5 v4 system, using same color scheme of Figure 6.

5. Conclusions

In this work we proposed an energy and performance profiling tool—Containergy—based on software containers, DVFS, control groups and hardware performance counters. We verified that similar tools do not address the energy and performance profiling properly for current and future workloads. Particularly, we observed: lack of adequate support for container profiling; absence of mechanisms for automatic energy characterization over multiple system configurations; no consideration of the application output while evaluating performance, leading to incomplete results.

Considering these questions, we can summarize Containergy's main advantages. First, it provides a container-based profiling with low overhead. The tool can profile container-packed applications while isolating hardware performance metering by using control groups. Likewise, it extracts application metrics to evaluate performance. Energy measurements are also available. The use of software containers as a profiling methodology makes Containergy aligned to the current trend of software packaging in cloud computing. Also, it simplifies the profiling process, specially in cases of diverse workloads. Furthermore, the combination of control groups, multi-sampling and performance counters provides more accurate results with less interference in the target system. Finally, by cycling software and hardware through configuration space it provides a comprehensive application resultset containing energy and performance data.

As disadvantages, Containergy requires that a container engine (Docker [22], in current version) be available and operational on the target system. In addition, kernel support for control groups must already be enabled on the system kernel. We consider that these issues do not represent major concerns on the adoption of the tool, since support for software containers and control groups is default in most Linux distributions. However, we acknowledge that these requirements may restrict the applicability of the proposed solution on embedded systems with lean kernels.

A limitation of Containergy is the need for the application to be packaged in a software container. Typically, most current applications are made available together with a containerized version, or they

can be easily found in public repositories. In situations where this is not confirmed, which is becoming more and more unusual, it will be up to the user to package the application before profiling. In the case studies presented, we used the container version of the video transcoding application (Kvazaar), made available with the application's source code. For machine-learning image classification applications, we created the containers using the containerized version of TensorFlow as basis, which is publicly available in the Docker repository.

Containergy does not rely on code instrumentation, which implies that it performs assessment from a system perspective, outside the source code. So, it is not useful to identify inner performance bottlenecks or to assess energy on specific parts of the code. However, this makes Containergy quite suitable in situations where the source code is not available, or when the interest of analysis lies on the normal execution of the application.

Both limitations are results of design choices and we consider them as trade-offs. The lack of code instrumentation makes Containergy easier to apply, as there is no need for the user to have or know the source code of the application. This allows it to have a broader use. The use of software containers as workloads permits the proposed tool to fill a gap found on other profiling tools. In addition, it makes profiling simpler by avoiding all inconsistencies that may occur when installing software, since all libraries and dependencies are packaged with the application.

We evaluated the proposed tool with two case-studies: HEVC Video Transcoding and Machine Learning Image Classification. The profiling results were analyzed in terms of performance and energy savings under a Quality-of-Service (QoS) perspective. On HEVC Video Transcoding workloads, we verified an increase above 300% in energy consumption for the same task and QoS thresholds due to wrong choices in the configuration space (worst/best setups). This confirms that a poorly operation can turn software inefficient, especially under an energy point-of-view. The ML image classification case study indicated that the choice of the machine-learning algorithm and model impact significantly the energy efficiency. AlexNet and SqueezeNet, although equivalent in purpose and similar in accuracy, present distinct behaviors of energy and performance. Their profiling datasets, obtained with Containergy, indicated that SqueezeNet represents 55.8% in energy saving compared to the AlexNet, both using the same input. Our tests demonstrated that Containergy is able to provide energy and performance assessment, as well as act as a data generation tool for posterior modeling and statistical analysis of new generation workloads.

Author Contributions: Funding acquisition, S.X.-d.-S. and D.A.; Project administration, M.Z., S.X.-d.-S. and D.A.; Software, W.S.-d.-S.; Supervision, M.Z. and K.O.; Validation, A.I. and A.B.; Writing—original draft, W.S.-d.-S.; Writing—review and editing, M.Z., S.X.-d.-S., K.O. and D.A. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the EU FEDER and the Spanish MINECO number RTI2018-093684-B-I00, the Spanish CM number S2018/TCS-4423, the Swiss SERI Seed Money project number SMG1702, the EC H2020 RECIPE project number 801137, and the ERC Consolidator Grant COMPUSAPIEN number 725657.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Andrae, A. Total consumer power consumption forecast. *Nordic Digital Business Summit* **2017**, *10*.
2. Barroso, L.A.; Hölzle, U. The Case for Energy-Proportional Computing. *Computer* **2007**, *40*, 33–37. [CrossRef]
3. Wong, D. Peak Efficiency Aware Scheduling for Highly Energy Proportional Servers. *SIGARCH Comput. Archit. News* **2016**, *44*, 481–492. [CrossRef]
4. Pouwelse, K.L.J.; Sips, H. Energy priority scheduling for variable voltage processors. In Proceedings of the International Symposium on Low Power Electronics and Design, Huntington Beach, CA, USA, 6–7 August 2001; pp. 28–33.
5. Barros, C.; Silveira, L.; Valderrama, C.; Xavier-de Souza, S. Optimal processor dynamic-energy reduction for parallel workloads on heterogeneous multi-core architectures. *Microprocess. Microsyst.* **2015**, *39*, 418–425. [CrossRef]

6. cAdvisor: Analyzes Resource Usage and Performance Characteristics of Running Containers. Available online: https://github.com/google/cadvisor/ (accessed on 1 March 2019).
7. De Melo, A.C. The new Linux 'Perf' tools. In Proceedings of the 17th International Linux System Technology Conference, Nuremberg, Germany, 21–24 September 2010; pp. 1–42.
8. OProfile—A System Profiler for Linux. Available online: http://oprofile.sourceforge.net (accessed on 1 March 2019).
9. OProfile Manual. Available online: http://oprofile.sourceforge.net/doc/index.html (accessed on 1 March 2019).
10. Walcott-Justice, K. Exploiting Hardware Monitoring in Software Engineering. *Adv. Comput.* **2014**, *93*, 53–101. [CrossRef]
11. OProfile 1.3.0—Release Notes. Available online: https://oprofile.sourceforge.io/release-notes/oprofile-1.3.0 (accessed on 1 March 2019).
12. PAPI. Available online: http://icl.utk.edu/papi/ (accessed on 1 March 2019).
13. Terpstra, D.; Jagode, H.; You, H.; Dongarra, J.J. Collecting Performance Data with PAPI-C. In *Tools for High Performance Computing 2009—Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing*; ZIH: Dresden, Germany, 2009; pp. 157–173._11. [CrossRef]
14. Zhang, H.; Hoffman, H. A quantitative evaluation of the RAPL power control system. In Proceedings of the 10th Internation Workshop on Feedback Computing, Seattle, WA, USA, 13 April 2015.
15. Khan, K.N.; Hirki, M.; Niemi, T.; Nurminen, J.K.; Ou, Z. RAPL in Action: Experiences in Using RAPL for Power Measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.* **2018**, *3*, 9:1–9:26. [CrossRef]
16. Shende, S.S.; Malony, A.D. The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.* **2006**, *20*, 287–311. [CrossRef]
17. Schulz, M.; Galarowicz, J.; Maghrak, D.; Hachfeld, W.; Montoya, D.; Cranford, S. Open | SpeedShop: An open source infrastructure for parallel performance analysis. *Sci. Program.* **2008**, *16*, 105–121. [CrossRef]
18. PapiEx. Available online: https://bitbucket.org/minimalmetrics/papiex-oss (accessed on 1 April 2019).
19. The Linux Kernel Documentation—Control Group v2. Available online: https://www.kernel.org/doc/Documentation/cgroup-v2.txt (accessed on 1 April 2019).
20. Rosen, R. Namespaces and cgroups, the basis of Linux containers. In Proceedings of the NetDev 1.1: The Technical Conference on Linux Networking, Seville, Spain, 10–12 February 2016.
21. Pahl, C. Containerization and the PaaS Cloud. *IEEE Cloud Comput.* **2015**, *2*, 24–31. [CrossRef]
22. Merkel, D. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* **2014**, *239*, 2.
23. Joy, A.M. Performance comparison between Linux containers and virtual machines. In Proceedings of the 2015 International Conference on Advances in Computer Engineering and Applications, Ghaziabad, India, 19–20 March 2015; pp. 342–346. [CrossRef]
24. Yu, Y. OS-Level Virtualization and Its Applications. Ph.D. Thesis, Stony Brook University, Stony Brook, NY, USA, 2007.
25. Mouat, A. *Using Docker*, 1st ed.; O'Reily: Beijing, China, 2016.
26. Hankendi, C.; Coskun, A.K. Reducing the energy cost of computing through efficient co-scheduling of parallel workloads. In Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, Germany, 12–16 March 2012; pp. 994–999. [CrossRef]
27. Cochran, R.; Hankendi, C.; Coskun, A.; Reda, S. Identifying the Optimal Energy-efficient Operating Points of Parallel Workloads. In Proceedings of the International Conference on Computer-Aided Design, ICCAD '11, San Jose, CA, USA, 7–10 November 2011; IEEE Press: Piscataway, NJ, USA, 2011; pp. 608–615.
28. Zhu, H.; Erez, M. Dirigent: Enforcing QoS for Latency-Critical Tasks on Shared Multicore Systems. In Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, 2–6 April 2016; ACM: New York, NY, USA, 2016; pp. 33–47. [CrossRef]
29. May, J.M. MPX: Software for multiplexing hardware performance counters in multithreaded programs. In Proceedings of the 15th International Parallel and Distributed Processing Symposium, IPDPS 2001, San Francisco, CA, USA, 23–27 April 2001; p. 22. [CrossRef]

30. Bross, B. High Efficiency Video Coding (HEVC) text specification draft 9. In Proceedings of the 11th Meeting of Joint Collaborative Team on Video Coding of ITU-T SG16 WP3 and ISO/IEC JTC1/SC29/WG11—Document JCTVC-K1003_v13, Shanghai, China, 10–19 October 2012.
31. Viitanen, M.; Koivula, A.; Lemmetti, A.; Ylä-Outinen, A.; Vanne, J.; Hämäläinen, T.D. Kvazaar: Open-Source HEVC/H.265 Encoder. In Proceedings of the 24th ACM International Conference on Multimedia, MM '16, Amsterdam, The Netherlands, 15–19 October 2016; ACM: New York, NY, USA, 2016; pp. 1179–1182. [CrossRef]
32. Bossen, F. Common test conditions and software reference configurations. In Proceedings of the 2nd Meeting of Joint Collaborative Team on Video Coding of ITU-T SG16 WP3 and ISO/IEC JTC1/SC29/WG11—Document JCTVC-B300, Geneva, CH, Switzerland, October 2013; Volume 12, p. 7.
33. Chi, C.C.; Alvarez-Mesa, M.; Juurlink, B.; Clare, G.; Henry, F.; Pateux, S.; Schierl, T. Parallel Scalability and Efficiency of HEVC Parallelization Approaches. *IEEE Trans. Circuits Syst. Video Technol.* **2012**, *22*, 1827–1838. [CrossRef]
34. Krizhevsky, A.; Sutskever, I.; Hinton, G.E. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*; Pereira, F., Burges, C.J.C., Bottou, L., Weinberger, K.Q., Eds.; Curran Associates, Inc.: Red Hook, NY, USA, 2012; pp. 1097–1105.
35. Implementation of AlexNet with TensorFlow. Available online: https://github.com/felzek/AlexNet-A-Practical-Implementation (accessed on 1 March 2019).
36. Iandola, F.N.; Moskewicz, M.W.; Ashraf, K.; Han, S.; Dally, W.J.; Keutzer, K. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *arXiv* **2016**, arXiv:1602.07360.
37. SqueezeNet v1.1 Implementation Using Keras Functional Framework 2.0. Available online: https://github.com/rcmalli/keras-squeezenet (accessed on 1 March 2019).
38. Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G.S.; Davis, A.; Dean, J.; Devin, M.; et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. 2015. Available online: https://tensorflow.org (accessed on 1 April 2019).
39. Russakovsky, O.; Deng, J.; Su, H.; Krause, J.; Satheesh, S.; Ma, S.; Huang, Z.; Karpathy, A.; Khosla, A.; Bernstein, M.; et al. ImageNet Large Scale Visual Recognition Challenge. *Int. J. Comput. Vis. (IJCV)* **2015**, *115*, 211–252. [CrossRef]