POLITECNICO
MILANO 1863

POLITECNICO DI MILANO
DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
DOCTORAL PROGRAMME IN INFORMATION TECHNOLOGY

ON THE MANAGEMENT OF POWER AND PERFORMANCE TRADE-OFFS IN DISTRIBUTED CLOUD-NATIVE INFRASTRUCTURES

Doctoral Dissertation of:
Rolando Brondolin

Supervisor:
Prof. Marco D. Santambrogio

Tutor:
Prof. Daniel Florian

The Chair of the Doctoral Program:
Prof. Barbara Pernici

2020 – XXXII

Abstract

CLOUD computing is now the de-facto standard for the deployment of complex and scalable applications and systems at scale. In the last few years, cloud computing applications shifted from the monolithic architecture to a more flexible microservice-based architecture with the so-called *cloud-native* ecosystem. This shift allowed to separate concerns among different development teams, increased the scalability of the cloud applications and allowed to develop, test, and deploy each functionality almost independently from the rest of the system. Cloud-native applications fostered even more the growth of cloud computing and, for this reason, cloud providers have to manage an unprecedented amount of applications for a huge amount of users. This trend poses new challenges in the management of data-centers. In particular, the expected energy usage of data-centers will reach 8% of the whole energy consumption of the world by 2030. Moreover, power consumption represents 20% of the Total Cost of Ownership (TCO) of a data-center. If we consider that the CPU is currently the most power-hungry component of a server, there is the need to optimize how cloud applications are executed within cloud infrastructures to keep the cloud-computing growth sustainable.

Within this context, the goal of this thesis work is the design and development of power management techniques able to sustain the performances requested by cloud-native applications and workloads while reducing as much as possible the power consumption such applications generate. Given the complexity that microservice-based applications bring, we decided to design a fully automated system to manage power consumption and perfor-

mance leveraging the Observe Decide Act (ODA) autonomic control loop. This allowed us to focus on how to measure and monitor in a fine grain way performance and power consumption, on how to allocate power and performance, and on how to actuate the control decisions defined by the ODA loop. Within this thesis, we designed a fully black-box approach to attribute power consumption, measure resource usage, and monitor network performance. Such black-box approach imposes less overhead on the monitored applications than the state of the art in the field and provides less than 5% relative error for most of the collected metrics. We leveraged these metrics to define reactive control policies able to maintain CPU usage and latency near a user-specified target. Then, we enforced in a timely and precise way the power budgets derived by the performance constraints. The CPU usage ODA loop allows reducing by 25% on average the power consumption with a 5% Service Level Agreement (SLA) violation on average. The latency-aware ODA loop allows reducing by 37.13% on average the power consumption with a control error of 12.5% and of 1.5ms on average.

Finally, we explored how to improve energy efficiency of microservice-based applications by introducing heterogeneous architectures, merging together the elasticity of cloud-native applications and the performance and energy-efficiency of Field Programmable Gate Arrays (FPGAs). The results are: (1) an improvement in the FPGAs time utilization while maintaining the performance of the applications and (2) power savings w.r.t. a pure software system implementing the same functions. This work represents an interesting initial study and paves the way for more extensive research work on how to accelerate microservices and cloud-native workloads.

Sommario

Il cloud computing rappresenta lo standard di fatto per il deployment di applicazioni e sistemi complessi e scalabili di grand dimensioni. Negli ultimi anni le applicazioni cloud sono passate dall'architettura a monolite a un'architettura più flessibile basata su microservizi, dando vita al cosiddetto ecosistema *cloud-native*. Questa evoluzione ha permesso di separare le responsibilità dei diversi team di sviluppo, ha aumentato la scalabilità delle applicazioni cloud e ha permesso di sviluppare, testare e fare deploy di ogni funzionalità in maniera quasi indipendente dal resto del sistema. Questa evoluzione ha favorito ancora di più la crescita del cloud computing e, per questo motivo, i cloud provider si trovano oggi a gestire una quantità sempre maggiore di applicazioni per un numero enorme di utenti. Questa tendenza pone nuove sfide nella gestione dei data-center. In particolare, il consumo di energia previsto per i data-center raggiungerà l'8% dell'intero consumo energetico mondiale entro il 2030. Inoltre, il consumo di energia rappresenta ad oggi il 20% del costo totale di proprietà di un data-center. Se consideriamo che la CPU è attualmente il componente che consuma più energia in un server, è necessario ottimizzare il modo in cui le applicazioni vengono eseguite all'interno di queste infrastrutture per mantenere sostenibile la crescita del cloud computing nel tempo.

In questo contesto, l'obiettivo di questo lavoro di tesi è la progettazione e lo sviluppo di tecniche di gestione dei consumi in grado di sostenere le prestazioni richieste dalle applicazioni e dai carichi di lavoro cloud-native riducendo il più possibile il consumo di potenza generato da tali applicazioni. Data la complessità introdotta dalle applicazioni a microservizi, abbia-

mo deciso di progettare un sistema completamente automatizzato per gestire il consumo di energia e le prestazioni sfruttando il concetto di Observe Decide Act (ODA) loop. Questo ci ha permesso di concentrarci su come misurare e monitorare a grana fine le prestazioni e il consumo di potenza, su come allocare potenza e prestazioni, e su come attuare le decisioni definite dal sistema ODA. In questa tesi abbiamo progettato un approccio completamente black-box per attribuire il consumo di potenza a ciascun elemento del sistema, misurare l'utilizzo delle risorse e monitorare le prestazioni della rete. L'approccio di monitoraggio proposto impone meno overhead alle applicazioni rispetto allo stato dell'arte nel campo e garantisce meno del 5% di errore relativo per la maggior parte delle metriche raccolte. Abbiamo poi sfruttato queste metriche per definire politiche di controllo reattivo in grado di mantenere l'utilizzo della CPU e la latenza vicino a un target specificato dall'utente. Quindi, abbiamo applicato in modo tempestivo e preciso i budget di potenza derivati dai vincoli di performance. L' ODA loop basato sull'utilizzo della CPU consente di ridurre in media del 25% il consumo di potenza con una violazione media del 5% dei Service Level Agreements (SLAs). L'ODA loop basato sulla latenza consente invece di ridurre in media del 37,13% il consumo di potenza con un errore di controllo del 12,5% e di 1,5ms in media.

Infine, abbiamo esplorato la possibilità di migliorare l'efficienza energetica delle applicazioni basate sui microservizi tramite l'uso di architetture eterogenee, unendo l'elasticità delle applicazioni cloud-native e le prestazioni e l'efficienza energetica dei Field Programmable Gate Array (FPGA). I risultati sono: (1) un miglioramento del tempo di utilizzo degli FPGA mantenendo allo stesso tempo le prestazioni delle applicazioni e (2) una riduzione dei consumi di potenza rispetto ad un sistema puramente software che implementa le stesse funzioni. Questo lavoro rappresenta un interessante studio iniziale e apre la strada a lavori di ricerca più approfonditi su come accelerare microservizi e carichi di lavoro cloud-native.

Contents

1 Introduction **1**
1.1 Cloud computing landscape 1
1.2 Cloud computing and power consumption 3
1.3 Challenges and contributions 3
1.4 Thesis outline 6

2 A black-box power monitoring methodology for container-based environments **9**
2.1 Introduction 9
2.2 Related Work 11
2.3 Proposed approach 13
2.3.1 Per-thread power attribution 15
2.3.2 Kernel level data acquisition 17
2.3.3 User space power attribution 20
2.3.4 Cluster level metric aggregation 21
2.4 Experimental results 22
2.4.1 Experimental setup 22
2.4.2 Performance overhead 23
2.4.3 Power consumption overhead 26
2.5 Conclusion and future work 27

3 Towards a unified view of the cluster state **29**
3.1 Introduction 29
3.2 Related Work 31

3.2.1 Power monitoring 31
3.2.2 Application performance monitoring and network performance monitoring 32
3.3 Monitoring infrastructure 34
3.3.1 Monitoring agent 35
3.3.2 Cluster view and analysis 44
3.4 Evaluation 48
3.4.1 Experimental setup 48
3.4.2 Metrics accuracy 50
3.4.3 Agent overhead 57
3.4.4 Benchmark study 60
3.5 Conclusion and future work 63

4 Performance-aware power capping for cloud and containerized applications 65
4.1 Introduction 65
4.2 Related Work 67
4.3 Methodology 69
4.3.1 Observe: power and performance monitoring 71
4.3.2 Decide: performance-aware power allocation 71
4.3.3 Act: enforcing power allocation 73
4.4 Evaluation 75
4.4.1 Experimental setup 76
4.4.2 Experimental results 77
4.5 Conclusion and future work 82

5 Latency-aware power capping for cloud and containerized applications 85
5.1 Introduction 85
5.2 Related Work 87
5.3 System design 89
5.3.1 Power and performance monitoring 91
5.3.2 Graph-based service time estimation 93
5.3.3 Latency-aware reactive control 97
5.3.4 RAPL-based power allocation 98
5.4 Evaluation 98
5.4.1 Experimental setup 99
5.4.2 Experimental results 100
5.5 Conclusion and future work 105

6 Future directions: accelerating microservices to improve power efficiency 107
6.1 Introduction 107
6.2 State of the Art 109
6.3 System design 110
6.3.1 Remote OpenCL Library 111
6.3.2 Device Manager 112
6.3.3 Accelerators Registry 114
6.4 Experimental evaluation 116
6.4.1 System overhead 116
6.4.2 FPGAs time utilization 119
6.4.3 Power consumption 121
6.5 Conclusion and future work 123

7 Conclusion 125

Bibliography 127

List of Figures

2.1 Overall infrastructure of the proposed monitoring system on two servers (S1 and S2) with kernel level data collection through BPF, user-space monitoring agent that collects RAPL measurements and BPF data and back-end infrastructure with data visualization and power monitor APIs. 13

2.2 Power consumption of EP, MG and CG benchmarks from NPB with HT or without HT from 1 physical core to 10. HT experiments pin two threads on two logical cores mapped onto a single physical core 15

2.3 Monitoring agent structure: the BPF code communicates with the agent via the *thread map* and the *configuration map* and stores data about cores in the *processor topology map*. The agent collects data from BPF and RAPL and updates the internal thread table also with containers data. The runtime manager selects also the new aggregation window length, updates the selector for the new measure and sends the sample to the back-end when it is ready. 18

2.4 Average and 95% confidence interval of the benchmarks execution time when running with and without the *DEEP-mon* agent (lower is better). Figure 2.4(a) shows execution time of the NPB benchmarks, while Figure 2.4(b) shows execution time of the Phoronix test suite benchmarks. 24

2.5 Average benchmark scores with 95% confidence interval for pts/apache, pts/nginx and pts/postmark. The overhead of the monitoring agent is 3.3% for pts/apache, 0.3% for pts/nginx, 0.4% for pts/postmark. 25

3.1 Overall infrastructure of the proposed monitoring system with kernel level data collection through BPF, user-space monitoring agent that collects metrics and Docker and Kubernetes data, and back-end infrastructure with data visualization, monitor APIs, and graph analyzer. 34

3.2 Power and performance monitoring structure: the BPF code sends metrics to the agent leveraging the thread map and stores partial results within the processor topology map. The user space agent iterates over the thread map to collect the metrics and correlate cycles with RAPL power measurements. Thread metrics are stored inside the thread table ready to be sent to the backend via the runtime manager. 37

3.3 Hyper Thread aware event flow for power and performance monitoring with events order highlighted. Dotted arrows are for threads that starts execution, full arrows for threads that are stopping. 39

3.4 Network transaction over a TCP connection as seen by a client process and a server process. 40

3.5 Network monitoring structure: the BPF code sends metrics to the agent leveraging the network transaction map and stores partial network data in the endpoint map and in the connection map. The user space agent iterates over the network transaction map to collect the metrics and attribute them to each container. Network metrics are stored in the thread table to be sent to the backend via the runtime manager. 41

3.6 Backend pipeline structure: the REST collector (highlighted in blue) is the entry point that collects all the data coming from the agents, while metrics frontend, REST endpoint and Graph endpoint (highlighted in green) are the endpoints used to visualize and query data. Components are connected with a queuing system enabling distribution, replication and asynchronous communication. Arrows show the data flow. . 44

3.7 View of the social network application (DeathStarBench) while reading a home timeline: the graph shows the client, a Nginx reverse proxy, two pods that manage the home timeline and a redis database. Between client and reverse proxy we have only one edge as the client is outside the monitored machines. 46
3.8 Relative Error of cycles, IR, cache references, cache misses, execution time, and CPU usage of our tool w.r.t. the golden standard measured on 30 runs of EP, MG, and CG. Bars show 95% confidence interval. 51
3.9 Average RE between our tool and wrk2 for network performance metrics for the 3 workloads with different amount of requests. Bars show 95% confidence interval. 53
3.10 Average RE between our tool and wrk2 for 50th, 75th, 90th, 99th percentile latency for the read home timeline workload with different amount of requests (from 1 to 16 times w.r.t. the capacity of the samples reservoir).Bars show 95% confidence interval. 55
3.11 Average RE between our tool and wrk2 for network performance metrics for the 3 workloads (high configuration) with network delay of 10ms $\pm$ 5ms normally distributed. Bars show 95% confidence interval. 56
3.12 Normalized overhead (with 95% confidence interval) w.r.t. no monitored execution of weave scope, our solution, and sysdig cloud while running the compose post workload for three different workload levels. 58
3.13 Normalized overhead (with 95% confidence interval) w.r.t. no monitored execution of weave scope, our solution, and sysdig cloud while running the compose review workload for three different workload levels. 58
3.14 Normalized overhead (with 95% confidence interval) w.r.t. no monitored execution of weave scope, our solution, and sysdig cloud while running the read home timeline workload for three different workload levels. 59
3.15 Normalized overhead (with 95% confidence interval) w.r.t. no monitored execution of weave scope, our solution, and sysdig cloud while running the read home timeline workload over the social network benchmark with more connections (500 instead of 100) and less requests per second (2000 req/s for high, 1500 req/s for mid, 1000 req/s for low). 59

3.16 Critical path example of the DeathStarBench social network loaded with 10000 R/s with 60 open connections. Edge weights are the 90th percentile latency. 62

4.1 Representation of the opportunity gap for the Apache benchmark. The graph shows that the workload under utilizes the resources requested by it (e.g. 500% of the CPU, equivalent to 5 cores), presenting a gap that can be exploited in order to improve energy proportionality. 66

4.2 HyPPO ODA loop: we observe the status of each server, centralizing power and performance metrics. Then, the decision phase computes a new strategy towards the system goals, which is enforced by the actuators. 70

4.3 Bits organization inside the MSR_RAPL_POWER_UNIT . 74

4.4 Bits organization inside the MSR_RAPL_POWER_LIMIT . 75

4.5 The graphs represent the CPU usage percentage of the different benchmarks. 100% represent a single core fully utilized. A system with 40 cores has at most 4,000% CPU usage. The CPU request is set at 5 CPUs (black line, 500%). The blue continuous line shows the CPU usage without power capping. The red dotted line shows the CPU usage measured with the proposed hybrid orchestrator running. 78

4.6 The graphs represent the power consumption expressed in mW, of the different benchmarks. The blue continuous line represents the case in which the benchmark power consumed is measured in a system with no power cap enforced. Instead, the red dotted line represents the case in which the benchmark power consumed is measured in a system with the proposed hybrid orchestrator running. 80

4.7 Power consumption in the case of multi containers running in a Kubernetes pod. In this case two pts/apache container were executed concurrently. In blue continuous the power consumed in the case no power cap is enforced in the system. In red dotted the power consumed in case the hybrid orchestrator mange the power cap. 81

5.1 Power REgulator for Service Time Optimization (PRESTO) ODA control loop: we observe latency and power consumption of each Kubernetes pod in each server, centralizing metrics in a remote backend. Metrics are then passed to the Graph analyzer, which, starting from a latency requirement, defines the service times of each container. Then, the Power controller defines the power budget allocated to each server, which is enforced by the RAPL actuators. 90
5.2 Example of a network transactions over a TCP connection. . 92
5.3 Graph of microservices where pod i has a fan-out of 3 pods, each one with its own arrival rates, service times and network times. 94
5.4 Abstract representation of a microservice application. Vertices are pods, edges are connections. Each pod has a α coefficient associated according to Equation (5.2). β coefficients are computed following a breadth-first search of the graph starting from the internet vertex, which is outside of the cluster. 96
5.5 Comparison of power consumption of the 3 workloads with different arrival rates. Error bars represent 95% confidence interval. Lower is better. 102
5.6 Box plots of the percentile latency (50th, 75yh, 90th, and 99th) of the compose post workload with 400 req/s across 30 runs. 103
5.7 Box plots of the percentile latency (50th, 75yh, 90th, and 99th) of the read home timeline workload with 6000 req/s across 30 runs. 104
5.8 Box plots of the percentile latency (50th, 75yh, 90th, and 99th) of the compose review workload with 300 req/s across 30 runs. 105

6.1 High Level Overview of BlastFunction components (Remote Library in green, Device Manager in blue, and Accelerator Registry in red) and their connections. 110
6.2 OpenCL Remote Library Architecture, highlighting the steps performed in the asynchronous flow (dotted lines represent asynchronous responses). 112
6.3 Device Manager Architecture, with the command queue methods flow highlighted (dotted lines represent asynchronous responses). 113

6.4 Latency overhead w.r.t. input data size for R/W operations, sobel operator and MM kernel, graphs are in linear scale. For MM Native overlaps with BlastFunction shm. 117
6.5 Sobel filter performance in req/s and normalized performance/W (higher is better). 122
6.6 Matrix multiplication performance in req/s and normalized performance/W (higher is better). 122

List of Tables

2.1 Per thread average latency, requests per second, transfer rate and worst case 99th percentile tail latency among 30 runs of nginx stressed by wrk. 26

2.2 Power consumption average and variance for the benchmarks, with monitoring overhead. The average overhead for HPC workloads is 0.90%, while for cloud workloads is 1.74%. . . 26

3.1 Experimental setup for the social-network benchmark and the media-microsvc benchmark. For each workload we show the number of pods involved and the request rates for low, medium, and high configurations. 49

3.2 Average execution times (with 95% confidence interval) and overhead of the monitoring tool for the three NPB applications with problem size C and 40 threads each. 51

3.3 Execution time, Energy, and Power consumption values (with 95% confidence interval) of EP and MG (10 threads each) when executed in isolation or in co-location with a fully disjoint set of cores. 52

3.4 Example output of the operational analysis with pod name, utilization, arrival rate of each pod, maximum arrival rate for each pod, power consumption, number of threads, and HTTP and TCP average latency for a DeathStarBench social network benchmark loaded with 10000 R/s with 60 open connections. Arrival rate 1576 R/s, estimated saturation arrival rate 1626 R/s. 61

4.1 The mean and the standard deviation of the CPU utilized and power consumed by the elements composing the proposed hybrid orchestrator. 82

5.1 Experimental setup for the social-network benchmark and the media-microsvc benchmark. For each workload we show the number of pods involved and the requests rate for low, medium, and high configuration. 99
5.2 Experimental results for the 3 workloads with 3 different arrival rates for each workload. Table reports the average latency when the benchmark is not constrained, the latency target, the average latency achieved by PRESTO, the latency difference, the latency relative error, and the power savings w.r.t. the not constrained execution. 101

6.1 Tests configurations overview, showing how many requests per second were sent to each function for each benchmark. . 119
6.2 Multi-function test results for the Sobel acceleator in terms of average latency, FPGA time utilization (overall maximum 300%) and processed/target requests. 120
6.3 Multi-function test aggregate results for MM in terms of average latency, FPGA time utilization (overall maximum 300%) and processed/target requests. 121
6.4 Multi-function test aggregate results for PipeCNN (AlexNet) with average latency, FPGA time utilization (overall maximum 300%) and processed/target requests. 121

Acronyms

A

API	Application Programming Interface.

A

AWS	Amazon Web Services.

B

BCC	BPF Compiler Collection.

C

CG	Conjugate Gradient.
CNCF	Cloud Native Computing Foundation.
CNI	Container Network Interface.
CNN	Convolutional Neural Network.
CPU	Central Processing Unit.

D

DNAT	Destination Network Address Translation.
DVFS	Dynamic Voltage and Frequency Scaling.

E

eBPF	extended Berkeley Packet Filter.
EP	Embarassingly Parallel.

F

FD File Descriptor.
FPGA Field Programmable Gate Array.

H

HPC High Performance Computing.
HT Hyper Thread.
HTTP HyperText Transfer Protocol.

I

IaaS Infrastructure as a Service.
IR Instruction Retired.

J

JIT Just In Time.

M

MG Multi Grid.
MM Matrix Multiply.
MSR Model Specific Register.

N

NPB NAS Parallel Benchmark.

O

ODA Observe Decide Act.
OLDI On-Line Data Intensive.
OS Operating System.

P

PaaS Platform as a Service.
PDR Partial Dynamic Reconfiguration.
PI Proportional Integral.
PID Process ID.
PMC Performance Monitoring Counter.
PRESTO Power REgulator for Service Time Optimization.

Q

QoS Quality of Service.

R

RAPL	Running Average Power Limit.
RE	Relative Error.
RPC	Remote Procedure Call.
RTT	Round-Trip Time.

S

SaaS	Software as a Service.
SLA	Service Level Agreement.
SMT	Symultaneous Multi-Threading.

T

TCO	Total Cost of Ownership.
TCP	Transfer Control Protocol.

U

UDP	User Datagram Protocol.

V

VM	Virtual Machine.

W

WSC	Warehouse Scale Computer.

CHAPTER 1

Introduction

1.1 Cloud computing landscape

Since the first launch of Amazon Web Services (AWS) in 2006, cloud computing continuously evolved and grew towards what it is today: the de-facto standard solution to develop, deploy, and maintain complex systems and services at scale. Applications baked by cloud computing infrastructures offer scalable services to a variable amount of users with fast response times, supporting both latency-critical workloads as well as batch workloads. The cloud computing service offering is usually organized in three main categories: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). IaaS offers virtualized resources to be consumed on-demand, from compute resources (e.g. Virtual Machines (VMs), containers, serverless functions) to storage and networking. PaaS usually offers run-time execution environments where applications are managed automatically by the cloud provider. Finally, SaaS offers ready-to-go applications that can be accessed from a user interface.

Within the first decade of cloud computing, most of the cloud applications were designed with the classical three-tier architecture, which is composed of a presentation layer, an applicative layer, and a data storage layer.

The applicative layer handled the HyperText Transfer Protocol (HTTP) requests coming from the presentation layer, computed some operations following the application logic, and retrieved or stored some results from the data storage layer. All the operations performed by the applicative layer were executed within a single unit, forming the so-called monolithic architecture. Within this context, monolithic applications hide their complexity within this single unit. Unfortunately, scalability strategies are limited to vertical scaling of computing resources and, depending on how the application was designed, low to moderate horizontal scaling. Moreover, new releases require to build and test the whole application.

To overcome these limitations, in the last few years, cloud computing applications shifted from monolithic architectures to microservice-based ones. The microservice architectural style [46] is an approach that allows building an application as a suite of small and loosely-coupled services communicating with each other through lightweight networking mechanisms. This shift allowed to separate concerns among different development teams, increased the horizontal scalability of cloud applications and decoupled the deployment cycle of any given functionality. Unfortunately, the microservice architectural style moved the complexity that was previously hidden inside the monolith to the network layer, increasing the difficulty to build and manage complex and efficient applications at scale. Such complexity can lead to microservice dependency graphs that are almost unreadable without the help of observability tools and to environments that can be operated only with the extensive use of automation.

From a technology perspective, Docker and Kubernetes are the main building blocks of the so-called *cloud-native* applications. According to the Cloud Native Computing Foundation (CNCF), *"Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative Application Programming Interfaces (APIs) exemplify this approach."* [1]. Docker containers leverage kernel-level virtualization implemented with Linux *namespaces* and *cGroups* to limit visibility and provide resource isolation between microservices. On top of Docker, Kubernetes orchestrates resources across a cluster of machines, managing the containers' life-cycle and scaling them depending on performance needs. Kubernetes implements microservices as *pods*, where a pod is a group of containers providing a single functionality.

Cloud-native applications developed with the microservice pattern typically have a high degree of heterogeneity, as microservices can leverage

different languages, different run-time environments, and different storage databases. They also show a high degree of co-location, as many microservices can be hosted within a single server. Finally, given that they usually provide a user-facing service, their fundamental performance metric is the request's latency.

1.2 Cloud computing and power consumption

The development of cloud-native tools and techniques combined with the ability to exploit virtually infinite resources on-demand fostered the adoption of the cloud computing model. To sustain this growth, cloud providers are continuously increasing their offering by consolidating workloads, expanding the available data-centers, and building new ones. Of course, these efforts are increasing the number of available machines that, in turn, will increase the energy required to run them. As a consequence, according to [4], the energy usage of data-centers will reach $\simeq 8\%$ of the total energy consumption of the world by 2030. To make this growth sustainable, we need to carefully improve the energy efficiency of the data-center components.

Energy usage currently represents 20% of the TCO of a data-center [34] and a relevant portion of the energy usage is devoted to servers. As of today, the CPU represents one of the most power-hungry components of a server [12], and, as such, a thorough optimization process of the application performances' from a power consumption perspective is required. Unfortunately, modern servers are not energy proportional, meaning that their performance does not grow linearly w.r.t. their energy usage [13]. If we consider continuous batch workloads running on dedicated Warehouse Scale Computer (WSC) systems, it has been proved that the average utilization of the system is around 75% [12]. However, this percentage drastically decreases in the more common scenario where a mix of several types of workloads (e.g. batch plus on-line applications) run in such systems. In this case, the utilization varies between 10% and 50% [12]. To reduce the TCO in this last case it is important to improve the energy efficiency at low and moderate paces.

1.3 Challenges and contributions

It is clear that there exists a trade-off between performance and power consumption of cloud-native applications executed onto cloud infrastructures. Although a simple power-capping approach might be tempting, the effects that this activity might have on the performance of cloud-native applica-

tions are devastating. According to *Brutlag* [24], increasing the latency of the responses of the Google search service from 100ms to 400ms reduces the number of searches per user from -0.2% to -0.6%. Such reduction persists for some time even if the latency is restored to the original value. For these reasons, any system that wants to reduce the power consumption of cloud-native applications should keep into account performances as the main aspect to protect.

Within this context, the goal of this thesis work is the design and development of power management techniques able to sustain the performances requested by cloud-native applications and workloads while reducing as much as possible the power consumption such applications generate. Given the complexity of cloud-native applications and environments, an automated approach is required. In particular, we exploited the Observe Decide Act (ODA) control loop [54], which is an autonomic methodology that enables the applications to adapt their behavior depending on their state and on the observations of the surrounding environment. Such methodology poses many challenges that we addressed throughout the development of this thesis work:

1. how we can measure the behavior of cloud-native applications and the environment in terms of performance and power consumption;

2. how we can define performance throughout cloud-native applications and how we can define meaningful performance targets;

3. how we can effectively and precisely reduce power consumption while preserving performances of the running workloads.

The first challenge deals with observability and with the ability to achieve self-awareness. Unlike many other previous techniques [3,35,48,53,54,56], we decided to tackle this challenge with a black-box approach. This means that we observe and obtain knowledge of the application components without instrumenting the workloads. Although this approach poses some limitations in the kind of information we can collect, it provides huge flexibility as any future application component can be observed and monitored out-of-the-box without modifications.

The second challenge deals with the correct definition of what performance is for a cloud-native application and encompasses both the Observe and the Decide phases of the ODA control loop. In particular, we experimented with two different performance metrics: CPU usage and average latency. On the one hand, CPU usage is typically defined by application developers as a performance constraint within Kubernetes [2]. However,

setting CPU requirements correctly for each component is hard, thus there is room to optimize how the allocated CPU is used by the application components. On the other hand, latency is a performance constraint understandable both from developers as well as final users, as it defines the quality of the user experience. Moreover, latency allows using queuing theory to guarantee performance during execution.

Finally, the third challenge deals with how to enforce performance constraints to save power. Such challenge involves both the Decide and the Act phase of the ODA control loop. Within this thesis, we experimented with heuristic control based mainly on basic queuing theory and with fast and accurate power actuation, represented by Running Average Power Limit (RAPL). Within this context, RAPL is our source of information for power consumption as well as for power actuation.

Given the challenges described so far, the main contributions of this thesis can be summarized as follows:

- We designed and implemented DEEP-mon, which is a dynamic and energy efficient power monitor able to attribute power consumption and performance metrics to each container running on a given host. DEEP-mon leverages kernel features like extended Berkeley Packet Filter (eBPF) to avoid user-code instrumentation. The proposed approach is accurate and increases the system power consumption of 0.90%, representing a low overhead that is the current state of the art in the field.

- We designed and implemented a monitoring methodology able to capture network activity and its performance without instrumenting the user code. We integrated such methodology within DEEP-mon, improving at the same time the power attribution mechanism, to provide a unified view of the running applications, generating and analyzing the graph of all the network activities between containers deployed in a multi-node Kubernetes cluster. The resulting tool monitors for each container its CPU usage, power consumption, instruction retired, cycles, cache references, cache misses, network bandwidth, average network latency, and network latency percentiles (from 50th to 99th). Results show an overall measurement error that is below 5% for almost all metrics with a lower overhead w.r.t. similar approaches in the state of the art.

- We developed HyPPO, a Hybrid Performance-aware Power capping Orchestrator. HyPPO implements an ODA control loop that builds

upon the data we collect with DEEP-mon to maintain a given level of performance while reducing power consumption. In particular, we manage CPU usage, which is commonly set as a constraint by application developers to each container in a Kubernetes cluster. HyPPO acts by applying a hardware power cap with RAPL and controlling the increase in CPU usage up to the level specified by the application developers. This approach allows reducing by 25% on average the power consumption with an average SLA violation of 5%.

- We developed PRESTO, a latency-aware power capping system for cloud-native microservices. PRESTO implements an ODA control loop that exploits the power and performance data collected with DEEP-mon to maintain an average latency requirement while saving power. The latency requirement is split across all the microservices of the cloud-native application through queuing theory and a graph analysis that allows to precisely attribute service times to be enforced to each microservice. PRESTO leverages the same RAPL actuator of HyPPO and reduces the power consumption of 37.13% on average with a control error of 12.5% and of 1.5ms on average.

Moreover, we explored how we can improve the energy efficiency of cloud-native applications by bringing heterogeneous architectures equipped with FPGAs within this context. We explored this research opportunity with BlastFunction, which is an FPGA-as-a-Service system in the context of cloud-native applications able to share FPGAs among many serverless functions (or Docker containers). BlastFunction improves FPGAs time utilization while maintaining the performance of the applications and at the same time saving power w.r.t. a pure software system implementing the same functions.

1.4 Thesis outline

The outline of this thesis work fairly resembles the list of contributions presented above. Chapter 2 explores our first efforts towards the definition of a black-box power monitoring methodology for cloud-native applications backed by Docker and Kubernetes. Chapter 3 integrates the power attribution methodology defined in the previous chapter with an extensive black-box performance and network monitoring methodology. Chapter 4 shows the results we obtained by managing the CPU usage of cloud-native applications with the goal of reducing power consumption. Chapter 5 shows instead how we defined an ODA control loop able to derive the performance

constraints of each microservice starting from a single latency requirement expressed for the entry-point of the cloud-native application. Finally, Chapter 6 describes our efforts towards the definition of an infrastructure able to manage accelerated microservices towards performance improvement as well as power-efficiency improvement of cloud-native applications.

How to read this thesis

There is no need to go through all the chapters to fully understand the work done, as all the chapters are self-contained. If you are interested in the black-box monitoring methodology described in this thesis work, Chapter 2 provides an in-depth view of how we performed power attribution for each microservice, while Chapter 3 gives a comprehensive view of the approach and focuses more on network performance monitoring. If instead you are interested in the overall autonomic methodology and the ODA control loop, Chapter 4 and Chapter 5 provide details on how we implemented such methodology in the case of CPU usage and average latency respectively. The two chapters can be read in no specific order and provide also a brief overview of the monitoring features used to implement the Observe step. Thus, it is not necessary to read Chapter 2 and Chapter 3 to fully understand them. Finally, if you are only interested in the FPGA-as-a-Service system for accelerated microservice and serverless computing, it is sufficient to go through Chapter 6, as it is fully self-contained.

CHAPTER 2

A black-box power monitoring methodology for container-based environments[1]

2.1 Introduction

Energy efficiency and energy proportionality are becoming important aspects in the context of large scale computer systems [66]. Improvements in power consumption have been identified as a critical goal for reducing the Total Cost of Ownership (TCO) of a data center [17,52], which has been increasingly affected by provisioning costs deriving from power supply. Resource utilization inside the data center should be monitored and managed to avoid underutilization and to reduce inefficiencies from the single thread of an application to the entire cluster. For this reason, power and energy measurements become then necessary to provide a full picture of the sys-

[1]The work presented in this chapter was published in [23], for which Rolando Brondolin developed the whole methodology, most of the implementation, most of the experimental evaluation, and the whole paper writing. ©2018 IEEE. Reprinted, with permission, from: Rolando Brondolin, Tommaso Sardelli, and Marco D Santambrogio. Deep-mon: Dynamic and energy efficient power monitoring for container-based infrastructures. In Parallel and Distributed Processing Symposium Workshops, 2018 IEEE International, pages 676–684. IEEE, 2018.

tem behavior and to enable power-aware computing for the next generation of data centers. In this context, several previous works addressed power awareness, measuring power and performance metrics at several layers of the stack (as in [67]).

Given that the Central Processing Unit (CPU) is currently the largest contributor to server power consumption [12], precisely accounting power consumption of each workload running in the system may help to understand how the data center is behaving and may provide hints on how to optimize the running applications. In order to do so, there is the need for a monitoring tool able to capture the variety of current workloads, especially in case of different applications co-located on the same physical machine. At the same time data should be gathered keeping the pace of quickly evolving workloads and must be as fine-grained as possible.

Previous work on the subject [16,43,98] has started to address this problem in different contexts. In particular, HaPPy [98] addressed the problem of HyperThread-aware estimation and power attribution, using CPU performance counters (e.g. instruction retired, unhalted clock cycles) to attribute to each thread its power consumption. However, all this was done in a context of static workloads, pinning threads to logical CPUs that were known in advance. A different approach was adopted by XeMPower [43], which collects power and Performance Monitoring Counter (PMC) traces starting from context switches of virtual tenants (i.e. VMs) managed by the Xen Hypervisor. This approach enables to dynamically monitor and measure the power consumed by each tenant. Unfortunately, XeMPower sends raw data from the hypervisor level to the user-space one at each context switch, impacting on the performance of the virtual tenants.

All these approaches addressed VMs and standalone applications, however, in the last few years application containers gained interest both in cloud computing and in High Performance Computing (HPC) fields, thank to Singularity [50]. In this context, to the best of our knowledge, a systematic HyperThread-aware power monitoring approach for application containers is still missing.

Our work addresses all these limitations in order to generalize the process of collecting metrics to quantify the power consumption of single applications. *DEEP-mon* is able to assign precise information about power consumption to each thread running on a system, without any previous knowledge regarding the characteristics of the application and without any kind of workload instrumentation. Moreover, the monitoring tool is able to aggregate data for threads, application containers and hosts. The proposed methodology is designed to minimize the overhead of the monitoring solu-

tion, making its impact on the monitored applications negligible.

The results obtained open the way for a wide set of applications exploiting the capabilities offered by the monitoring tool, from power (and hence cost) metering of new software components deployed in the data center to fine grained power capping and power-aware scheduling and co-location. In this context, we are evaluating power and performance trade-offs in a distributed environment managed by Kubernetes [25], where containers are spread and replicated on a cluster of machines.

The rest of this chapter is organized as follows: Section 2.2 covers the related works, from monitoring tools to on how power measurements can be used at a fine grain level, Section 2.3 describes the proposed approach towards per thread and per container power attribution and monitoring, Section 2.4 shows the experimental evaluation of the monitoring tool, while Section 2.5 concludes the chapter and depicts some future work.

2.2 Related Work

The problem of power consumption in the data centers and its impact on the Total Cost of Ownership (TCO) has been addressed by different works in the last decade. An initial study of the impact of energy costs has been proposed by Fan et al. [42], where power usage characteristic of large collections of servers has been analyzed for a period of six months for different classes of applications. With their work they estimated that the more underutilized a facility is, the more expensive the cost of building the data center becomes in terms of fraction of the TCO. However, during the design activity of the datacenter, peak power consumption must be taken into account, even if the data center will be under-utilized for certain periods of time. This is fundamental, as short period of peak power consumption generated by software components can on rare occasions cause the entire data center infrastructure to breach the safety limits.

This problem is addressed by Bhattacharya et al. in [19], which highlights the importance of power capping techniques in mitigating underutilization as well as power limits violations, while also showing their limitations and possible improvements. We think that in these contexts, having a precise, fast and lightweight power monitoring agent providing fine-grained and real-time information is of paramount importance.

Several works tried to estimate power at different levels of the data center. One recent example is the work proposed by Maranthe et al. in [67]. This work proposed to integrate several probes at cluster level, rack level, node level and at the system level. Moreover, they carried out an analysis

of the efficiency of thermal and power management for a large scale data center using the proposed monitoring components. Our work relies on a different set of power measurements, as [67] does not cover per application power monitoring and attribution. However, our work is complementary to [67], as it can be integrated in a wider set of monitoring tools to perform better analysis at different levels of the stack.

Moving from data center power monitoring to a more fine grained monitoring scenario, there has been significant research into estimating power consumption for single applications from performance counters. One of the first works was done by Bellosa et al. [16], where they used an external power meter and found a correlation between measured power consumption and CPU performance counters such as *unhalted clock cycles*, *instructions retired* and *cache hits/misses*. However, this work addresses only coarse-grained power measurements using and external probe since Intel RAPL was not yet available at the time. More recent work can be found in Power Containers [88] where a per request power accounting solution was proposed but without taking into account the difficulties posed by Hyper Thread (HT) in precise energy attribution.

Such problem has been addressed by HaPPy [98] using a methodology that proportionally assigns power consumption to logical cores belonging to physical CPU cores, based on the number of weighted *unhalted clock cycles* measured for each logical core. HaPPy measures then the power consumption of each workload pinning the threads on fixed logical cores, thus knowing in advance workloads and resource assignments. *DEEP-mon* builds upon this work reversing the proposed methodology and accounting weighted cycles for each thread instead of logical CPUs. This is done as soon as a context switch happens, making the methodology resilient to time-shared CPUs.

A more generic approach, but within a different context, has been proposed by Ferroni et al. in XeMPower [43]. In this work they developed a monitoring solution for the Xen hypervisor that precisely accounts hardware events to Xen domains. XeMPower also enables attribution of CPU power consumption to individual tenants. One of the main drawbacks of the proposed methodology is related to the data transfer between the probes and the monitoring agent. XeMPower moves raw PMC traces at each vCPUs context switch, thus creating overhead for the virtual tenants managed by Xen. Our approach solves the overhead issue by aggregating the raw PMC traces as soon as possible in kernel, moving to user space just the aggregated data. XeMPower was then used to enable per-tenant power models [44] through a black-box methodology called MARC [45] that mod-

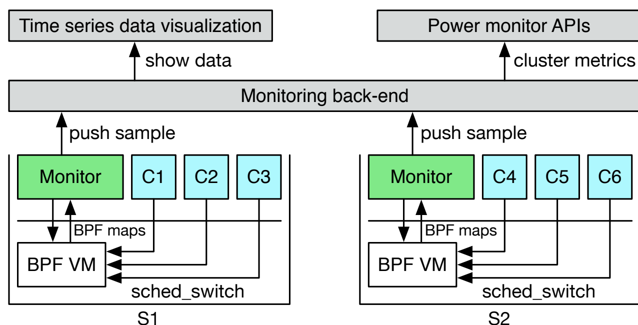


Figure 2.1: *Overall infrastructure of the proposed monitoring system on two servers (S1 and S2) with kernel level data collection through BPF, user-space monitoring agent that collects RAPL measurements and BPF data and back-end infrastructure with data visualization and power monitor APIs.*

els power consumption using PMC traces, identifying different working regimes at which a server operates and generating a model for each regime with model identification theory.

All these approaches can be applied not only for VMs or standard applications, but also for application containers. In the last few years the interest on this technology rose, and some works started to appear about power consumption metering and management, in particular for Docker containers[2]. Piraghaj et al. [84] proposed a framework and algorithm for energy efficient container consolidation in cloud data centers. Their work focuses on consolidation, and tests were carried out with CloudSim [27]. Of course, to implement the proposed methodology on real systems, a thorough monitoring activity of containers power consumption is necessary.

Finally, DockerCap [9] proposed an ODA loop able to cap power consumption of the system, assigning resources to each co-located container to guarantee the requested SLA. DockerCap leverages Linux perf to read RAPL measurements, obtaining a coarse-grain and low frequency view of the containers power consumption. Our work increases visibility and timeliness of the measurements while introducing negligible overhead.

2.3 Proposed approach

Power awareness is the first step towards the mitigation of data center power usage inefficiencies. To enable this feature, the data center should be aug-

[2]https://www.docker.com

mented with power monitoring tools. In order to achieve good visibility on data centers power usage, it is necessary to put in place not only power grid monitors, but also monitors able to trace power consumption at the single machine level. Unlike performance monitors (either at system level or at application level), power monitoring requires specialized hardware (e.g. external power monitors like WattsUp power meter [94]) or hardware interfaces (like Intel RAPL [86]) that are able to measure the power consumed instantaneously by the components of the system.

In order to provide new tuning knobs for autonomic schedulers and orchestrators, and given that a non negligible part of the power consumption is related to CPU tasks [12], there is the need for a fine grain power monitoring tool able to attribute power consumption to each thread and application running in the system. This kind of power attribution was tackled in the past by several works like [88] and [98], either not considering Symultaneous Multi-Threading (SMT) effects in the attribution or not narrowing down the proposed approach to the actual runtime power monitoring. Moreover, power attribution becomes a really complex task when we consider highly co-located environments, in particular when we consider dockerized applications. Docker and application containers are becoming increasingly popular not only in cloud computing, but also in HPC scenarios [36] (e.g. with the Singularity project [50, 63, 97]), as they allow to build stable and predictable running environments and allow to isolate applications in the system sharing only the Linux kernel implementation. In this context, to the best of our knowledge, a systematic approach to attribute power consumption to each application container running in a highly co-located environment is still missing.

DEEP-mon is a reliable and lightweight power monitoring solution for Linux threads and application containers designed to: 1) provide precise attribution of selected hardware events and power consumption to each thread and container, 2) be agnostic with respect to the running workloads, scheduling policies and resource mapping and usage, 3) add negligible overhead in the running system.

The monitoring tool is divided in three main components, as shown in Figure 2.1: 1) a kernel-level code that collects and aggregates performance counters data for each thread leveraging eBPF [15, 73]; 2) a user-space agent that collects the aggregated data, adding RAPL measurements and further aggregating for each container in the system; 3) a back-end application that collects the metrics coming from each agent in a cluster and processes the time-series data to show them to the user and to make them available to power-aware schedulers via APIs.

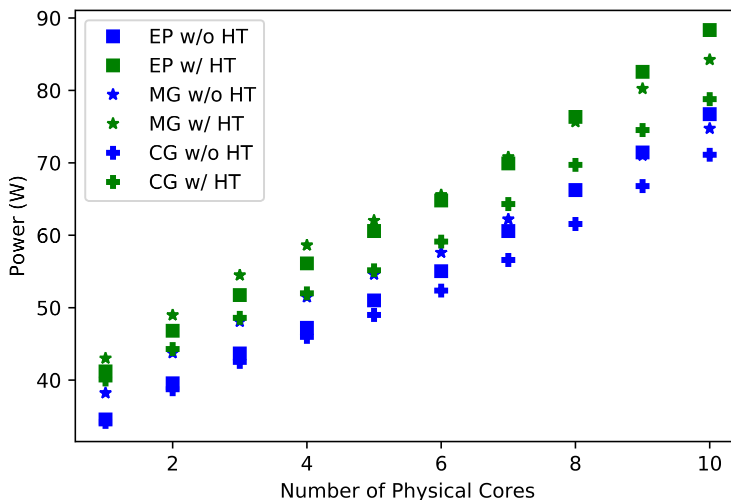


Figure 2.2: *Power consumption of EP, MG and CG benchmarks from NPB with HT or without HT from 1 physical core to 10. HT experiments pin two threads on two logical cores mapped onto a single physical core*

In the next sections, we will provide a detailed overview of on how to attribute power consumption to each thread and of each component of *DEEP-mon*, analyzing the trade-offs that we encountered during the design and development of the tool.

2.3.1 Per-thread power attribution

Intel RAPL interface and external power meters provide just a coarse grain measurement about the power consumption of a server. RAPL, which was introduced in Intel processors since the *Sandy Bridge* architecture, is able to measure the power consumption of cores, package and DRAM for each processor in a server. Starting from this ability, we want to attribute power consumption measurements to each thread depending on its usage of the processor.

Recent works show that there is a strong correlation between power consumption and PMCs [17], in particular with *UNHALTED_CORE_CYCLES* (0.99 linear correlation, as pointed out in [98]). From this evidence, we built our methodology starting from the work done in HaPPy [98] towards a per-thread power attribution that is HT aware and agnostic with respect to the running workloads and their assignment of resources. In fact, the goal

of *DEEP-mon* is to attribute power consumption even with CPUs that are time-shared across several workloads.

HT is the Intel implementation of SMT that allows to share the resources of a physical core, enabling two hardware threads to run simultaneously on the same core. The hardware threads are seen as logical cores by the Linux Operating System (OS), which can then schedule processes on each of them. When the resources of a physical core are shared across two different processes through HT, the power consumption derived from the execution of these threads will be different with respect to running the same threads on two different physical cores. Figure 2.2 shows the power consumption of three benchmarks of the NAS Parallel Benchmark (NPB) suite when their threads are pinned on different physical cores or on logical cores belonging to the same physical one. The figure shows on the X axis the number of physical cores, while on the Y axis we show the power consumption measured with RAPL. For the *no HT* experiments, we run N threads corresponding to N physical cores, while for the *HT* experiments we run 2N threads corresponding to N physical cores to measure the impact of two threads in HT with respect to just one thread per physical core. The benchmarks were run on a Dell PowerEdge r720xd equipped with 2x Intel Xeon E5-2680 v2 (10 cores + HT @ 2.80GHz), 380GB of RAM and with Ubuntu Linux 16.04 and kernel version 4.13. This first experiment shows that when executing threads on two logical cores mapped on the same physical core, the power consumption is $\simeq 1.15$ with respect to just one thread executing on that same physical core. This ratio, denoted as HT_r, gives us some hints on how to attribute power consumption among threads executing in the system, as we have to discriminate between execution of threads on logical cores or on physical ones.

To attribute power to threads, our model leverages the measurements of Intel RAPL core and of the *UNHALTED_CORE_CYCLES* performance counter. We collect also *INSTRUCTION_RETIRED* and *threads execution time* to assess the performance of the workloads throughout their execution. *UNHALTED_CORE_CYCLES* are measured for each logical core and for each physical core to account also for the execution of two threads on the same core through HT. For a given thread T_1, we denote $Cycle_{TA_1}$ as the number of unhalted core cycles where the thread executed alone on a physical core, while $Cycle_{TO_1}$ represents the number of unhalted core cycles when there is overlapping between two threads co-running on the same physical core via HT. Remembering the ratio HT_r described before, we can compute the weighted cycles that can be used to attribute power consumption to threads as shown in Equation (2.1).

$$Cycles_{TW_1}(t,s) = \sum_{i=0}^{N} Cycle_{TA_1}(s,i) + \frac{HT_r}{2} \cdot \sum_{j=0}^{M} Cycle_{TO_1}(s,j) \tag{2.1}$$

In this equation, for each discrete observation interval t and for a given socket s, the number of weighted cycles for thread $T1$ is the sum of two different contribution: in the first one we have the sum of the cycles measured during the N execution periods in which the thread run alone, in the second one we have the sum of the cycles measured during the M execution periods in which the thread was co-running on the same physical core via HT, weighted by the HT_r ratio and divided by 2 to equally divide the overlapping cycles among the two threads. In this context an execution period is defined as the time between context switches on the physical core where the thread is scheduled.

Starting from Equation (2.1), we can now attribute the power measured by RAPL for our thread $T1$ following Equation (2.2), where $|K|$ is the cardinality of the set K of threads running in the server in a given period of time and $|S|$ is the cardinality of the set S of sockets in the system.

$$P_{T1}(t) = \sum_{s=0}^{|S|} \left(RAPL_{core}(t,s) \cdot \frac{Cycles_{TW_1}(t,s)}{\sum_{k=0}^{|K|} Cycles_{TW_k}(t,s)} \right) \tag{2.2}$$

Starting from this result, the next sections will provide details on how we implemented power attribution for each thread and container running in the system.

2.3.2 Kernel level data acquisition

The power attribution model described in Section 2.3.1 needs a precise measurement of the performance counter values generated by each thread. Moreover, the monitoring tool should measure also the execution periods where two threads are running on the same physical core via HT. From the Linux OS perspective, on a logical core is either scheduled a thread or the idle process. Transitions between threads are identified by *context switch* events, as each time a thread is going to be scheduled on a logical core, the old context is saved and substituted with the one required for execution. If

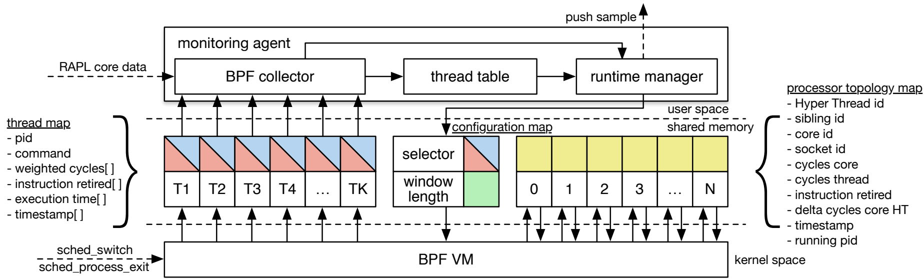


Figure 2.3: *Monitoring agent structure: the BPF code communicates with the agent via the* thread map *and the* configuration map *and stores data about cores in the* processor topology map*. The agent collects data from BPF and RAPL and updates the internal thread table also with containers data. The runtime manager selects also the new aggregation window length, updates the selector for the new measure and sends the sample to the back-end when it is ready.*

we are able to observe context switches, then we can measure precisely the performance counters for each thread.

Instead of developing a custom loadable kernel module or manually instrumenting the kernel, as it was done with the hypervisor code in [43], our approach leverages a different mechanism at the heart of the Linux kernel: eBPF [15, 73]. eBPF is a kernel level VM that runs in its extended version in every Linux distribution since kernel 3.15. The main advantage of the recent versions of eBPF stands in the ability to do Just In Time (JIT) compilation of well-formed codes that can get access to different events inside the Linux kernel. In order to guarantee the termination of the eBPF programs no loops are allowed, as well as backward jumps. In order to do JIT compilation and execution of eBPF code, at the time of writing, each eBPF program can have at most 4096 lines of eBPF assembly code and can allocate at most 512 bytes of stack. To ease the development of the monitoring tool, we leveraged the BPF Compiler Collection (BCC) tools [14], which allow to manage eBPF codes and to dynamically load and compile them, as well as managing the interconnection between the user-space agent and the kernel level VM.

We leveraged eBPF to observe Linux tracepoints [64], in particular the *sched_switch* and the *sched_process_exit* ones to observe context switch and process exit respectively. We do not observe process creation (e.g. clone, fork, execve) to reduce the impact on the monitored system, as each thread that need to execute and that will consume a certain amount of power has to do a context switch. Again, to lower the impact of the monitoring

tool we used tracepoints instead of Linux Kprobes even if tracepoints are less flexible, as Kprobes introduce more overhead in the system.

eBPF programs, when loaded in the VM, allow to communicate with an external application with several interfaces such as *bpf_perf_buffers* and *bpf_maps*: bpf_perf_buffers are used to send raw data from the kernel level VM to a user-space application when the data is generated, bpf_maps, instead, are key value maps that allow to send aggregated data to a user-space application asynchronously. Given that the rate of context switch events is really high, in particular in case of overloaded systems, we decided to aggregate events and performance counters inside the VM. Figure 2.3 shows the data structures of the bpf_maps used for *DEEP-mon* and the communication scheme between the VM and the user-space agent, as well as the structure of the monitoring agent. The eBPF program shares with the user-space agent a configuration map, the thread map with performance counters data and a processor topology map, which hosts data about the running PID and the partial measurements of the performance counters for each HT core in the system.

Algorithm 1 details how we collect cycles measurements for a given thread. Instruction retired and execution time are collected just for the running thread that did the context switch. The eBPF code executes on the same hardware thread and in the context of the thread that generated the observed event, sched_switch in this case. The code is divided in two sections: at first we look at the thread that is exiting from execution, then we look at the new thread being scheduled on a given hardware thread. Let us consider two HT cores mapped on the same physical thread. Each time a context switch happens on one of the two HT cores, the delta UNHALTED_CORE_CYCLES at the hardware thread level is measured for the thread who did the context switch. At the same time, the delta UNHALTED_CORE_CYCLES at the core level is measured for both threads (when the second one is not the idle process). This last measure is updated at first for the thread that did the context switch, then it is temporarily stored on the other HT core and accounted to the co-running thread at its next context switch. In this way, when a context switch happens, we can apply Equation (2.1) to compute the weighted cycles. This data is continuously updated until the user-space application read the data from the bpf_maps. After this step, we act on the starting thread reserving some space for it in the thread map and resetting the counters for the new measurement. Finally, when a thread finishes execution through the *sched_process_exit* event, we simply remove it from the thread map and we reset all the counters related to it in the processor topology map.

Algorithm 1 BPF pseudo code for cycles measurement

```
1: procedure SCHED_SWITCH(switch_args, thread_cycles, core_cycles, instruction_retired)
2:     //exiting thread
3:     HT_id ← bpf_get_smp_processor_id()
4:     socket ← topology[HT_id].socket_id
5:     //cc stands for core cycles
6:     Δcc ← 0
7:     scc ← 0
8:     sib_id ← topology[HT_id].sibling_id
9:     sibling_PID ← topology[sib_id].running_pid
10:    old_PID ← switch_args.old_PID
11:    new_PID ← switch_args.new_PID
12:
13:    if sibling_PID > 0 && old_PID > 0 then
14:        Δscc ← core_cycles − topology[sib_id].core_cycles
15:        topology[sib_id].Δcc ← topology[sib_id].Δcc + Δscc
16:        topology[sib_id].core_cycles ← core_cycles
17:
18:        Δcc ← core_cycles − topology[HT_id].core_cycles
19:
20:    //tc stands for thread cycles
21:    Δtc ← thread_cycles − topology[HT_id].thread_cycles
22:    weighted_cycles = Δtc + (HT_r / 2) · (Δcc + topology[HT_id].Δcc)
23:    old_weighted ← thread_map[old_PID].weighted_cycles
24:    thread_map[old_PID].weighted_cycles[socket] ← old_weighted + weighted_cycles
25:
26:    //entering thread
27:    if new_PID not in thread_map then
28:        add_new_thread(switch_args)
29:    topology[HT_id].running_pid ← new_PID
30:    topology[HT_id].thread_cycles ← thread_cycles
31:    topology[HT_id].core_cycles ← core_cycles
32:    topology[HT_id].Δcc ← 0
33:    return
```

2.3.3 User space power attribution

The user-space agent of *DEEP-mon* is in charge of setting up the eBPF program and attributing power consumption to each thread and container. Figure 2.3 shows the block diagram of the agent, which is detailed in three main components: *bpf-collector*, *thread-table* and *runtime-manager*. Bpf-collector handles the eBPF code, extracting data on a regular time basis. Then, for each thread collected from the *bpf_maps*, Equation (2.2) is applied. RAPL measurements are extracted at each time window using the *intel_rapl* module. At this point, the user-space agent updates the thread-table and for each new thread finds the proper folder in the */proc* folder of the Linux OS to read the *cgroup ID*. From the cgroup ID we can then identify which thread belongs to which container, further aggregating the monitoring data. When the agent collected all the data for each container into a sample, the runtime-manager selects the time window length for eBPF

Algorithm 2 time window length selection pseudo code

```
1: procedure TIME_WINDOW_SELECTION(sched_switch_count, old_window_length)
2:
3:     switch_per_second_per_core ← sched_switch_count / (#HT_cores·old_window_length)
4:
5:     if switch_per_second_per_core < 100 then
6:         return 4s
7:     else if switch_per_second_per_core < 200 then
8:         return 3s
9:     else if switch_per_second_per_core < 200 then
10:        return 2s
11:    else
12:        return 1s
```

data extraction and manages the communication between the agent and the monitoring back-end through Intel Snap[3].

Algorithm 2 shows how we select the dynamic window length used by the user-space agent to collect data from eBPF. In case of highly CPU intensive workloads where the number of threads are similar to the number of available HT cores, few context switch events are generated by the applications. In this case, the aggregated performance counters collected by eBPF can be accounted in the wrong time window, as the execution period length of the workloads can increase spanning two different time windows. Given that the frequency of context switch events can vary during the monitoring activity, we change the aggregation window lenght of the eBPF code at runtime to avoid imprecise measurements.

One last thing to notice about the monitoring system is that with BCC the agent is not allowed to send commands directly to the eBPF code, but it can just update data on the *bpf_maps* and wait for an event on the kernel VM to propagate it. For this reason and to avoid spurious reads of the *thread map* we introduced the *selector* flag shown in Figure 2.3 and we doubled some of the fields of the thread map. Before reading data on the bpf_maps, the user-space agent flips the flag so that the eBPF code can start writing on the cells of the map that the agent is not reading. The timestamp is always checked for each thread so that the eBPF code can notice if the selector flag changed more then once from a context switch to another.

2.3.4 Cluster level metric aggregation

The monitoring back-end is the component responsible for the final data aggregation. Each agent connects to this component and sends samples on a regular time basis. Given that the agents adopt a push model to send

[3] http://snap-telemetry.io

data to the back-end, no checks on the connections are done at this level. When a sample arrives at the back-end, the raw power and performance measurements are unpacked and used to build metrics that are segmented depending on container, host and cluster organization. In this context, the agent collects data about Kubernetes status and about Kubernetes PODs through Intel Snap plugins. These metrics are used to further aggregate the metrics in logical views that can be used for data visualization or as input for power-aware schedulers.

2.4 Experimental results

To enable power awareness in production systems, the overhead of the monitoring agents should be negligible. To this aim, in this Section we want to evaluate the proposed monitoring tool and its impact on the monitored system. On the one hand we are interested in the performance loss derived from the monitoring activity, while on the other hand we want to assess the impact of the tool on the overall power consumption. In Section 2.4.1 we will describe the experimental setup and the benchmark we selected for the evaluation while in Section 2.4.2 and Section 2.4.3 we will show the results for the performance overhead and the overall power consumption overhead analysis.

2.4.1 Experimental setup

We evaluated the proposed approach on a Dell PowerEdge r720xd equipped with 2x Intel Xeon E5-2680 *Ivy Bridge* with 10 cores each (20 HT) clocked at 2.80GHz and with 380GB of RAM. The evaluation platform represents a recent mid-range server. The host OS is an Ubuntu Linux OS 16.04 with kernel 4.13 and eBPF support enabled. Each workload runs inside a Docker container, with Docker runtime version 1.13.1. All the experiments are carried out with HT enabled without pinning the threads on any core, and we measure power with Intel RAPL both for our monitoring tool and for the power consumption overhead experiments.

To evaluate *DEEP-mon*, we selected three benchmarks from the NAS Parallel Benchmark (NPB) suite [11] version 3.3.1 [40], Embarassingly Parallel (EP), Multi Grid (MG) and Conjugate Gradient (CG):

- **Embarassingly Parallel (EP)** is a kernel that generates pairs of Gaussian random deviates and is a benchmark highly CPU intensive and CPU bound;

- **Multi Grid (MG)** is a memory intensive benchmark that performs a simplified multi grid calculation, it requires highly structured long distance communication and it tests both short and long distance data communication;
- **Conjugate Gradient (CG)** is a mixed CPU and memory intensive workload that performs an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix through a conjugate gradient method.

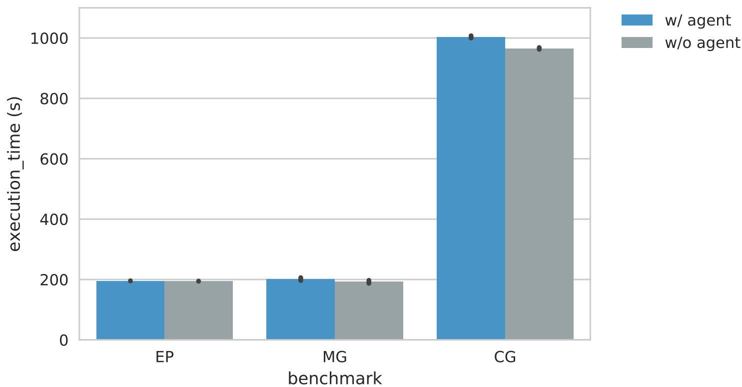
All the benchmarks of NPB were run using the maximum amount of logical cores available. Moreover, to have a complete picture of the proposed approach, we leveraged also three benchmarks from the phoronix test suite [61] to stress the tracepoint mechanism that triggers the aggregations performed by our eBPF code:

- **pts/apache** is a test of the *Apache Benchmark* program on an Apache2 web server that measures how many requests per second the web server is able to sustain when stressed with 1,000,000 requests with 100 requests performed concurrently;
- **pts/nginx** is a test of the *Apache Benchmark* program on an Nginx web server that measures how many requests per second the web server is able to sustain when stressed with 2,000,000 requests with 500 requests performed concurrently;
- **pts/postmark** simulates small-file testing similar to what is done by mail and web servers, performing 25,000 transactions with 500 files simultaneously with file sizes that ranges from 5 to 512 kilobytes.

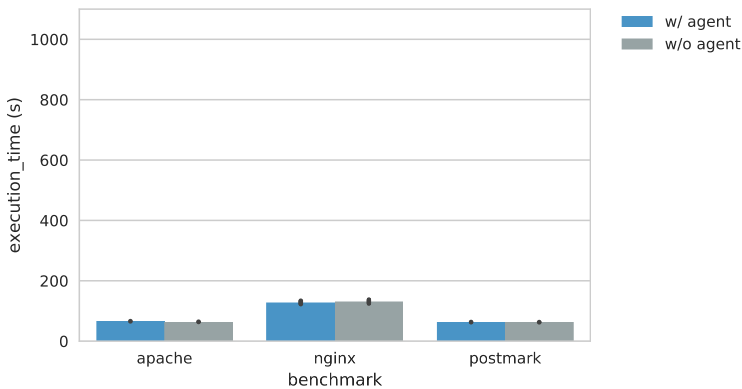
This last set of benchmarks are related to cloud computing applications, in contrast with the HPC benchmarks of the NPB suite. We decided to use both benchmark classes to evaluate the proposed approach with application containers in all the possible scenarios in which they are currently used.

2.4.2 Performance overhead

In this first set of experiments we are interested in evaluating the impact of *DEEP-mon* on the performance of the benchmarks, as the overhead of the monitoring system should be as low as possible. To this aim, we executed the benchmarks described in Section 2.4.1 30 times each and we collected data about the execution time of both classes of benchmarks and the benchmark score provided by the Phoronix test suite.



(a) Execution time (in seconds) of EP, MG and CG of the NPB suite, where the agent introduces an overhead of 0.4%, 4.2% and 3.8% respectively.



(b) Execution time (in seconds) of pts/apache, pts/nginx and pts/postmark, where the overhead introduced by *DEEP-mon* is 3.1%, -2.05% and 0.4%.

Figure 2.4: *Average and 95% confidence interval of the benchmarks execution time when running with and without the* DEEP-mon *agent (lower is better). Figure 2.4(a) shows execution time of the NPB benchmarks, while Figure 2.4(b) shows execution time of the Phoronix test suite benchmarks.*

Figure 2.4 details the execution time for both the classes of benchmarks comparing two different configurations: 1) the agent is running and monitoring the benchmarks power consumption, 2) the agent is not running. As we can see from the HPC-class benchmarks shown in Figure 2.4(a), the overhead of the monitoring agent is quite negligible in all the test cases. The tool increases the execution time of EP, MG and CG of 0.4%, 4.2% and 3.8% respectively. As for EP, the result is justified by the fact that the benchmark performs almost no synchronization and thus it rarely performs

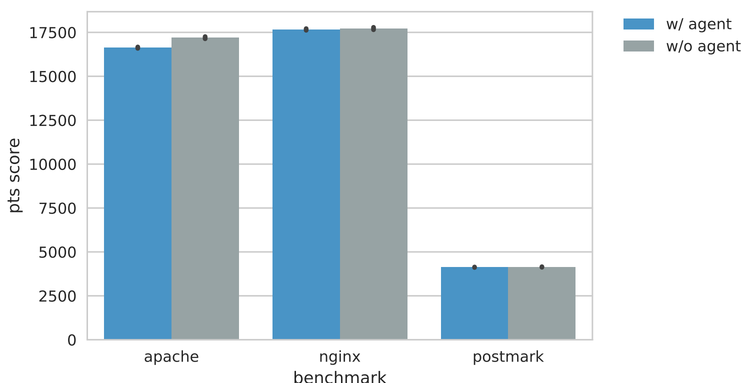


Figure 2.5: *Average benchmark scores with 95% confidence interval for pts/apache, pts/nginx and pts/postmark. The overhead of the monitoring agent is 3.3% for pts/apache, 0.3% for pts/nginx, 0.4% for pts/postmark.*

context switches. In this case, the monitoring impact is extremely low. As for MG and CG, their activity is more dependent on partial results of other threads. For this reason, context switches count increase, increasing the overhead as well. However, even in this case the impact of the monitoring tool is still low.

As for the cloud computing class of benchmarks, the graph of Figure 2.4(b) shows again good performance in terms of execution time overhead. In this context, pts/apache shows an increase in the execution time of 3.1%, pts/nginx of -2.05% and pts/postmark of 0.4%. A surprising result in this case is represented by pts/apache, which generates $\simeq 200.000$ context switches per second: even with this load on the eBPF code, the overhead in terms of execution time remains under 5%. Moreover, in this experiment, pts/postmark execution time remains almost unchanged. This happens because the benchmark continuously executes reads and writes on disk without performing many context switches. Finally, when the agent is running, pts/nginx reduced its execution time. This is due to the uncertainty of the workload, that shows high variability from one run to another. The results of Figure 2.4(b) are confirmed for the benchmark scores, highlighted in Figure 2.5. As we can see, the benchmarks perform almost identically for pts/nginx and pts/postmark, while there is a slight difference for pts/apache. Moreover, the agent does not introduce uncertainty in the performance of the benchmarks, as the 95% confidence interval tends to the average value.

Given that the pts/nginx exposed a variable behavior in Figure 2.4(b), we decided to investigate it further to better understand the agent impact.

Table 2.1: *Per thread average latency, requests per second, transfer rate and worst case 99th percentile tail latency among 30 runs of nginx stressed by wrk.*

configuration	nginx 20 threads with wrk 20 threads			
	latency	# requests	transfer rate	99th latency
w/o agent	41.62 ms	8793 req/s	141.08 MB/s	1130.0 ms
w/ agent	38.35 ms	8720 req/s	140.17 MB/s	1100.0 ms

Table 2.2: *Power consumption average and variance for the benchmarks, with monitoring overhead. The average overhead for HPC workloads is 0.90%, while for cloud workloads is 1.74%.*

benchmark	w/o agent		w/ agent		overhead
	μ	σ^2	μ	σ^2	
EP	152.00 W	1.53	154.38 W	0.47	1.56%
MG	130.58 W	7.11	131.81 W	4.05	0.93%
CG	126.71 W	1.02	126.99 W	0.99	0.22%
pts/apache	63.17 W	0.43	65.11 W	0.22	2.98%
pts/nginx	56.45W	1.87	56.76 W	3.03	0.45%
pts/postmark	34.15W	0.16	34.76 W	0.03	1.80%

We tested nginx with wrk[4], setting nginx concurrency level to 20 threads, while the wrk stress tool was set to 20 threads to have a CPU utilization of $\simeq 1$. In Table 2.1 we present the per thread average latency, number of requests per second, transfer rate and the worst case 99th percentile latency generated by 30 runs of the experiment. As we can see, this results shows again that the overhead generated by our agent is negligible. In particular, the 99th percentile tail latency measurement is pretty close, meaning that even in the worst case, the monitoring tool does not affect the execution of the benchmark.

2.4.3 Power consumption overhead

We run the same set of experiments described in Section 2.4.1 with the same configurations to assess the impact of the agent also in terms of power consumption. We observed the overall power of the test server with *perf* and we measured power consumption with *RAPL core*. The result of this analysis are shown in Table 2.2. In this case we measured the overhead as the increment of power consumption of the server when running a benchmark with our agent with respect to running that same benchmark alone.

The power consumption of the benchmarks shows a quite variegated scenario. On the one hand, the HPC benchmarks tends to consume as much power (and CPU) as possible, with EP consuming more than the others because there are few synchronizations points and wait periods. On the other

[4]https://github.com/wg/wrk

hand, the cloud workloads consume less, as the benchmarks stress the network latency and throughput of some web applications and the latency and throughput on files for a mail server. The overhead of the agent is 0.90% for the HPC workloads, while became 1.74% for the cloud workloads.

Overall, the power consumption of the monitoring agent is low, and the difference between the two classes suggests that this overhead is limited in all cases and becomes negligible when the utilization of the server increases. Pts/apache and CG are the most stressful workloads for the agent, both inducing a performance overhead equal or higher than 3%. When it comes to the server total power consumption, they are behaving in the opposite way: with CG the overhead is at most 0.24%, while with pts/apache it becomes 2.98%. This happens, again, because CG has a CPU utilization $\simeq 1$, while with pts/apache, the CPU usage drops down to $\simeq 0.20$.

Finally, if we consider the impact on power consumption for HPC workloads, XeMPower [43] obtained an overhead of $\simeq 1\%$, while our approach behaves slightly better with 0.90% of overhead. This result shows that the aggregation of PMCs inside the eBPF VM reduced the overhead on the overall system power, even if threads context switches are more fine grained than vCPU context switches. This headroom allowed us to introduce data collection about Kubernetes and to send data to the back-end still having comparable results with XeMPower, which simply aggregates by Xen domain and stores the data on disk.

2.5 Conclusion and future work

In this chapter we presented a novel approach for the attribution of power consumption to threads and application containers running in a Linux OS. The proposed approach is detailed in three different stages: 1) kernel-level data acquisition code, for which we leveraged eBPF and BCC tools, 2) user-space per-thread power attribution, that aggregates performance and power consumption data also for application containers, and 3) cluster level metric aggregation, which generates metrics that can be used by power-aware schedulers and orchestrators.

The experimental evaluation shows that our approach has a negligible impact on the performance of the workloads and on the overall power consumption. Moreover, our solution is designed to overcome the limitations of the previous works, in particular HaPPy [98] and XeMPower [43], in terms of time-shared CPUs support and monitoring overhead respectively.

One of the main issues we encountered during the development of the monitoring tool is related to eBPF. Even if eBPF allowed us to ease the

development of the performance counters data extraction, its limitations reduces the applicability of our approach. The maximum amount of assembly instructions and the limited stack size allowed us to test our approach only on a dual-socket server at most, as more cores in the system would saturate the eBPF stack. To mitigate this issue, as a future work, we will explore the improvement of the eBPF mechanism both at the eBPF application layer and at the kernel level, tuning and modifying the kernel VM.

CHAPTER 3

Towards a unified view of the cluster state[1]

3.1 Introduction

As we discussed in Chapter 1, the microservice architectural style combined with the proper technological tools allowed to move away from the standard monolithic pattern of legacy workloads. However, the complexity that was previously hidden inside the monolithic application moved as well, abruptly increasing the pressure on the network layer. With the growing complexity of deployments, it is becoming fundamental to monitor and measure the performance of microservices at scale to assess their functionality and to detect performance issues as soon as they appear in the system. Such issues may arise in several layers of the cluster's stack and can affect network performance, system performance, and resource usage. Moreover, as discussed in [47], microservices interact with CPU architectures differently w.r.t. monolithic workloads and these kinds of interactions can be captured by monitoring performance counters. If we leverage data

[1] The work presented in this chapter was published in [21], for which Rolando Brondolin developed the whole methodology, the whole implementation, the whole experimental evaluation, and the whole paper writing. Rolando Brondolin and Marco D Santambrogio. A black-box monitoring approach to measure microservices runtime performance. ACM Transactions on Architecture and Code Optimization (TACO), 17(4):1-26, 2020. DOI: https://doi.org/10.1145/3418899

about performance counters, system performance, network performance, and power consumption it is then possible to spot bottlenecks, improve performance, and increase energy efficiency. For these reasons, monitoring should take into account all these aspects, focusing on pure software performance metrics as well as on architectural ones.

Within this context, several works [55, 69] addressed the monitoring issue for Linux servers and in general for physical hosts and virtual machines. However, they lack the ability to drill down at the container level. Many commercial tools [35, 79] and open-source tools [3] overcame this limitation by instrumenting application code. Finally, [39, 89, 95] retrieve data using a black-box approach, however, they only provide metrics about the high-level performance of the applications, neglecting the low-level metrics needed to monitor how the workloads are exploiting the underlying architecture.

In this chapter, we present a novel black-box approach to monitor microservices at scale in the context of Docker containers managed by a Kubernetes orchestrator. The goal is to build a unified view of a microservice-based application from low-level architectural metrics to network performance. For this reason, the main contributions of this chapter are:

- The design of a monitoring methodology based on eBPF [15, 73] that collects data about network activity (e.g. number of requests, bytes sent and received, average latency, and from 50th to 99th percentile latency) for each Docker container, Kubernetes pod, and physical host;
- The integration and expansion of DEEP-mon (that we first described in Chapter 2) to measure network metrics, application performance metrics (e.g. CPU usage, execution time), and low-level performance metrics (e.g. cycles, Instruction Retired (IR), cache references, cache misses, and power consumption);
- The design of a metrics collection system that retrieves metrics from a set of monitoring agents, groups them depending on container, pod, service, and host, and provides a graph view and analysis of the monitored Kubernetes cluster that encompasses performance, power consumption and network activity of each microservice.

Within this chapter, we demonstrate how the proposed methodology generates less overhead on the monitored benchmarks w.r.t. similar approaches in the state of the art. At the same time, we are able to guarantee a reasonable accuracy over the measurements the monitoring system performs. Finally, we show a study of performance and power consumption of one

application of the *DeathStarBench* [47] benchmark suite, discussing the possible trade-offs between optimizing power consumption and maintaining the performance of the applications.

The rest of this chapter is organized as follows: Section 3.2 analyzes the related works in the field with a focus on power, performance and network traffic monitoring in microservices and cloud-based environments. Section 3.3 describes the proposed solution to monitor microservices behavior from the low-level eBPF module to the remote metrics collection system. Section 3.4 shows the experimental results we obtained with the proposed monitoring approach in terms of overhead and precision of the collected metrics. Section 3.4 derives also insights on how the benchmarks behaved during the experimental campaign thanks to our monitoring service. Finally, Section 3.5 draws the conclusion of the chapter and derives future works.

3.2 Related Work

According to the CNCF Cloud Computing Landscape [33], observability of cloud-native applications can be divided into three main areas: *monitoring*, *logging*, and *tracing*. Within this thesis work, we focus on monitoring microservices power and performance. *Monitoring* cloud applications allows capturing the run-time behavior of the system, enabling a thorough analysis of the collected data and highlighting performance issues that can lead to a poor end-user experience. Here we provide a brief view of the state of the art for power monitoring as well as application performance and network performance monitoring.

3.2.1 Power monitoring

Power consumption can be measured at different levels of the data-center stack, from the power grid level arriving to the thread level in a single host. In the last few years, some works focused on fine-grain power monitoring, from VMs to containers, to threads. One of the first works in this area is represented by *Bellosa et al.* [16]. Within this work, the authors found a first correlation between the power consumption of the server and some hardware performance counter measuring data at the level of the whole machine. At the VM level, a notable work is represented by XeMPower [43]. XeMPower allows to precisely attribute power consumption to each VM in a given host with low overhead on the system. The tool monitors both RAPL and various performance counters that are traced at the context switch of the *Xen* vCPUs. Power Containers [88] works at the per-request

level but does not take into account HT. This last aspect was taken into account by HaPPy [98] with a proportional power attribution at the thread level based on RAPL [86] and *cycles*.

Moving to containers and microservices, Piraghaj et al. [84] proposed a framework and algorithm for energy-efficient container consolidation in cloud data centers. DockerCap [9] proposes an ODA loop to cap power consumption while assigning resources to guarantee a SLA. Power monitoring in this context is done through RAPL at the socket level.

3.2.2 Application performance monitoring and network performance monitoring

To monitor hardware performance counters, two main tools are available: *Linux perf* [37] and *PAPI* [77]. Linux Perf is a performance tool that can measure performance counters, tracepoints, hardware, and software events and it can provide statistical profiling of the entire system performance. *PAPI*, instead, is a library to collect performance counters from within the monitored application. *Linux perf* is the standard tool for performance counter profiling, and, for this reason, we decided to use *perf arrays* within the eBPF code to retrieve them.

As for application performance monitoring, several works exist in the state of the art, both at the academic level as well as at the commercial level. For what concerns commercial and open source tools, here we describe briefly the most relevant. *Sysdig* [89] is a performance monitoring tool that collects data about processes, containers and Kubernetes clusters without instrumenting the user code. *Sysdig* introduces a kernel driver with several buffers that collect and analyze all the system calls to provide metrics about CPU usage, memory usage, network I/O and file I/O. Although our tool does not collect memory usage and file I/O, with Sysdig we are not able to collect low-level performance metrics and latency percentiles. *Weave scope* [95] collects data only about CPU usage and network connections without latency measures with eBPF and builds a network topology graph on top of it. *Datadog* [35] instruments the applications with custom integrations. Network analysis is performed through eBPF and it is based on *Weave scope*, however, it does not account for latency measures but just for bandwidth. Finally, *Prometheus* [3] is an open-source project within the CNCF that provides APIs to instrument the user code, a time-series database, and a visualization interface to show the metrics.

If we consider instead the research works in the field, one of the first works is represented by *Ganglia* [69], which is a distributed monitoring sys-

tem for clusters and grids typically used in HPC. Another interesting work is *Nagios* [55], which is an open-source monitoring tool for grid computing. Both *Ganglia* and *Nagios* focus on host metrics and grid metrics and do not take into account application-related metrics. Moving to microservice-aware research works, *Noor et al.* [80] designed a monitoring framework that can work in a multi-cloud, multi-virtualization, and multi-microservice setting. The authors retrieve microservice's performance from the *Linux* OS, group everything by microservice and expose the data through a REST interface. Although this work measures data at the microservice level, network performance is not considered. *Pina et al.* [83] instead focuses mainly on monitoring the network performance without user code instrumentation. To avoid instrumentation they introduce infrastructural components that are used by the application such as the API gateway, the service discovery and the load balancers. Unfortunately, these components are not mandatory in a microservice environment: our approach can retrieve the network metrics without instrumenting them. *Chang et al.* [30] specifically target Kubernetes and collect metrics about CPU usage, Memory usage, and Quality of Service (QoS) violation metrics. The proposed approach reads aggregate resource usage data from the *Linux* OS as [80] and retrieves network performance by stressing the applications with *Apache jMeter*[2]. Although this approach obtains network metrics similar to the one we provide, jMeter is extremely invasive w.r.t. the performance of the applications.

Another class of interesting work focuses on pure network performance monitoring. *Ntopng* [38] is a system for network traffic monitoring and characterization. It is mainly based on *libpcap* and *tcpdump* [90] and provides a *Lua* based scripting engine as well as a web server to observe network traffic and a data exporter. *Ntopng* can work efficiently with 10GbE networks without performance loss. The work of *Deri et al.* [39] is a Kubernetes and container-aware runtime security tool that leverages eBPF to gather connections' open and close events. It works both with Transfer Control Protocol (TCP) and User Datagram Protocol (UDP) and retrieves data exchange with *Ntopng*. The work of *Deri et al.* uses different Linux kprobes and tracepoints w.r.t. our work obtaining similar information. Unfortunately, given that the scope of [39] is on network security, no information about accuracy and overhead is provided. The work of *Cinque et al.* [32] provides a tool for network traffic analysis that accompanies a system for log collection that involves application instrumentation. Its *MetroFunnel* component is based on *pcap* and can pinpoint abnormal service operation with small overhead on the performance of the applications.

[2] https://jmeter.apache.org

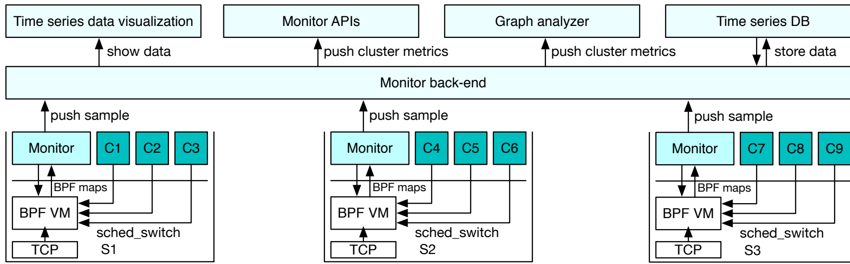


Figure 3.1: *Overall infrastructure of the proposed monitoring system with kernel level data collection through BPF, user-space monitoring agent that collects metrics and Docker and Kubernetes data, and back-end infrastructure with data visualization, monitor APIs, and graph analyzer.*

Finally, *ConMon* [76] monitors functions inside adjacent containers. Passive monitoring uses tools like *tcpdump*, while active monitoring injects network traffic to test the response of the workloads.

Among the reviewed state of the art so far, none of the described works was able to combine monitoring data on resource usage, application performance, network performance and application power consumption to provide a unified view of the application. A work that comes close to it is *Seer* [48]. *Seer* leverages application performance, resource usage, host power consumption and network tracing data to build a neural network model able to predict QoS violations across a cluster of microservices. Our work focuses more on data acquisition rather than performance predictions. Moreover, our work requires no instrumentation of the user code, while *Seer* tracing system requires a hook on the code that performs network operations. Another work in this area is *Rusty* [68], which uses power consumption and performance counters to predict the behavior of low level metrics of dockerized workloads. We believe that our approach can be used within *Rusty* to integrate also network performance in the prediction framework.

3.3 Monitoring infrastructure

Within this chapter, we propose a generalized black-box monitoring approach to measure the performance of cloud-native applications. We currently focus on microservices deployed with Docker containers and managed by a Kubernetes orchestrator, although the proposed methodology can be adapted to other container engines as well as orchestrators. For each container, we collect data about low-level performance (e.g. IR, cycles,

cache references, cache misses, and power consumption), application performance (e.g. CPU usage and total execution time), and network performance (e.g. number of connections, number of requests, bytes sent and received, average latency, and from 50th to 99th percentile latency). We designed the proposed monitoring tool following three main principles: *transparency*, *performance*, and *accuracy*. *Transparency* means to monitor cloud-native applications without requiring user intervention in terms of configurations and user-code instrumentation. *Performance* means to introduce the least overhead possible on the monitored applications and to use as few resources as possible on the monitored hosts. Finally, *accuracy* means to retrieve metrics in the most precise way without data loss.

Figure 3.1 shows the main components of the monitoring infrastructure: kernel-level instrumentation, user-space agent, and remote monitoring backend. For each host, we deploy a kernel-level instrumentation layer implemented with eBPF [15, 73]. On top of the kernel-level instrumentation, we run a user-space monitoring agent that periodically collects the metrics and enriches them with information about threads, processes, containers and Kubernetes pods and services. Once the monitoring sample is ready, the monitoring agent sends it to the remote backend each second. At this point, the backend computes and stores the metrics in a time-series database and builds the connection graph between pods. Finally, a user interface visualizes the metrics.

3.3.1 Monitoring agent

Starting from the principles depicted in Section 3.3, our goal is to build a monitoring system able to observe applications without code instrumentation, without providing significant impact on performance, and with the highest accuracy possible. Unfortunately, to avoid code instrumentation, we need to resort to the OS to retrieve the data that we need. The common way to collect data about the running workloads without instrumenting them is to develop a custom kernel module [89]. However, this is usually a time consuming and error-prone task. To overcome this limitation, we resorted to eBPF, which is a kernel-level VM that runs inside the Linux kernel. eBPF allows to do JIT compilation of C code that can get access to different events inside the Linux kernel. The kernel-level VM enforces security measures to prevent system instability: for instance, to load an eBPF program, a user needs to have privileged access to the host system. Moreover, the kernel-level VM accepts only codes that do not have loops, that have at most 4096 eBPF assembly instructions, and that do not use

more than 512 bytes of stack. These measures are adopted to avoid infinite loops and to limit the footprint on performance and memory usage of the eBPF programs. Within this context, to ease the development process, we leveraged the BCC [14] tools to compile, load, and manage eBPF codes at run-time. BCC provides a wrapper to an LLVM compiler that is able to compile C code to eBPF assembly, a set of helper functions in C for the eBPF programs, and a set of helper functions for python codes. These last helper functions allow to load and remove eBPF programs and perform R/W operations on eBPF data structures accessible both from the python code as well as from the eBPF code.

The peculiarities of eBPF and BCC allows us to design monitoring tools with the following general approach: (1) extract metrics from data as soon as the data is generated, (2) move metrics only in aggregated form to limit the impact on the overall system, (3) correlate metrics coming from different hosts without coordination among distributed agents. Previous works that leverage custom kernel modules are not able to provide aggregation at the kernel-level due to security and performance issues, and, as such, they encounter performance and accuracy degradation when the monitored system is overloaded.

Starting from the general approach, we designed two different eBPF programs able to collect performance data and network data respectively. These two programs react to Linux tracepoints and kprobes that are specific to the data the monitoring tool has to collect. Once the eBPF VM receives an event from the Linux kernel, the event is passed to the proper program that processes it and stores the output on a hash map that can be accessed by a user-space agent. All the processing activity on the single event is performed within the eBPF VM and the hash map stores only aggregated results. This allows avoiding to send the raw events from kernel-space to user-space, reducing the user-space agent load that can otherwise result in high overhead and data loss in case of system saturation. At this point, the only goal the user-space agent has is to enrich the aggregated data with context information from Docker and Kubernetes and then send all the data to a remote backend to achieve cluster-level visibility.

Power and performance monitoring

Within Intel processors, HTs belonging to the same core share some resources. This means that executing two threads on different cores has a different impact on the system's power consumption than executing the same two threads on two HTs that share the same core. To perform the power attribution, we expanded and improved the work presented in Chapter 2.

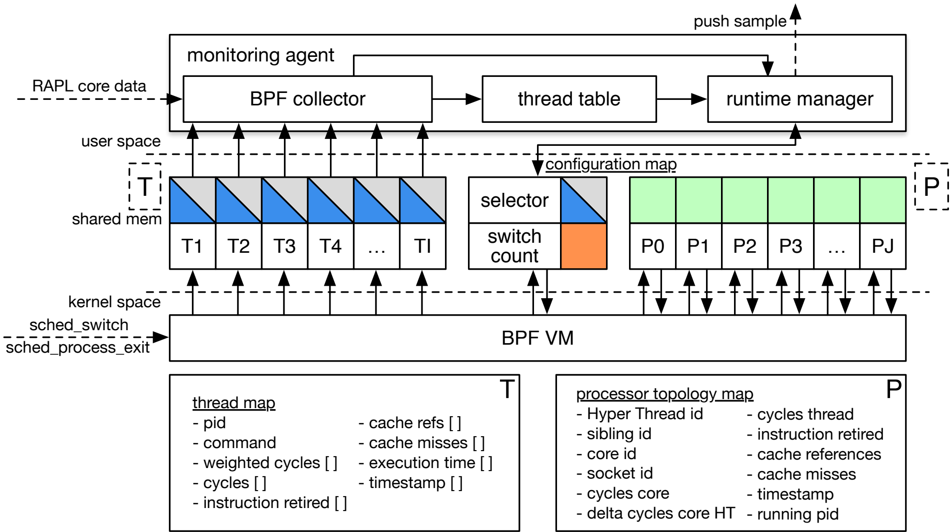


Figure 3.2: *Power and performance monitoring structure: the BPF code sends metrics to the agent leveraging the thread map and stores partial results within the processor topology map. The user space agent iterates over the thread map to collect the metrics and correlate cycles with RAPL power measurements. Thread metrics are stored inside the thread table ready to be sent to the backend via the runtime manager.*

In particular, Chapter 2 show that there is a strong correlation between *cycles* and *RAPL core power consumption* and that there is a ratio (of 1.1 for *Sandy Bridge* and *Ivy Bridge*) that can be used to weight *cycles* to perform a fair attribution of power consumption across threads. We denote this ratio as HT_r and we derive the weighted cycles in Equation (3.1).

$$Cycles_{W_T}(t,s) = Cycle_{A_i}(s,t) + \frac{HT_r}{2} \cdot Cycle_{O_i}(s,t) \tag{3.1}$$

The weighted cycles for a given thread i on a given socket s during the observation time t is the sum of two contributions: (1) the cycle measurements $Cycle_{A_T}(s,i)$ when no threads are running in the sibling HT; (2) the cycle measurements $Cycle_{O_T}(s,j)$ when there is a thread running in the sibling HT. This second contribution is scaled by the HT_r ratio divided by 2 to avoid accounting twice for the same cycles measurements on the two sibling threads.

To attribute the power consumption to a thread i for a given observation interval t, it is sufficient to divide the power consumption measured with RAPL by all the weighted cycles of all the threads K that run in a given time interval t. This result can be then multiplied by the weighted cycles of

thread i. This operation should be performed for each socket, as shown in Equation (3.2).

$$P_i(t) = \sum_{s \in S} \left(RAPL_{core}(t,s) \cdot \frac{Cycles_{W_i}(t,s)}{\sum_{k \in K} Cycles_{W_k}(t,s)} \right) \tag{3.2}$$

To perform power and performance monitoring, we need to track *context switches* to monitor how threads are scheduled and executed on the host processor. Context switch events are exposed with a *tracepoint* by the Linux kernel and with eBPF, we can attach to it to know when a thread starts its execution and when it ends. In this way, we can measure the execution time and performance counters and we can collect data to attribute power consumption. Figure 3.2 shows the main components of the kernel-level data acquisition for low level and application-related performance metrics. The eBPF virtual machine communicates with the user-space agent with two hash maps: *thread map* and *processor topology map*. The former stores the aggregated metrics for each thread observed in the system, while the latter works as a scratchpad for each HT to keep partial results during the observation. The *configuration map*, instead, stores the amount of context switch observed and the selector field used to switch between two memory areas: one is used by the eBPF code to write metrics, while the other is used by the user-space agent to retrieve the metrics.

Figure 3.3 shows the events flow during metric collection for power and performance metrics with two threads. When a thread has to be executed on a given HT (HT 1 in the case of Figure 3.3), the scheduler performs a context switch that is captured by our monitoring tool. Each context switch provides information about the new thread (e.g. thread ID, command, priority) as well as the old thread. Once the context switch arrives, we measure *cycles*, *IRs*, *cache references*, and *cache misses* on the HT that is going to host the thread using the eBPF *perf maps*. We then store these data along with the timestamp to the *processor topology map* (step 1). Eventually, a new thread is scheduled on HT 2. In the case of Figure 3.3, we measure the performance counters for HT 2 and we store them in the *processor topology map* (step 2a). To account for the shared execution on the core, we count also the *cycles* of HT 1 and we update this information in the *processor topology map* (step 2b). When the first thread stops the execution, a new context switch happens. We measure again the performance counters of HT 1, we account to the thread the difference between the first measurements and the second ones (step 3a) and we increment the weighted cycles for thread 1 following Equation (3.1). As we did in step 2b, we account for the

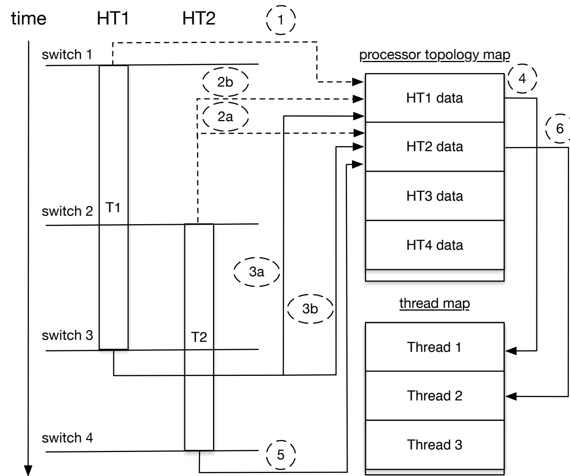


Figure 3.3: *Hyper Thread aware event flow for power and performance monitoring with events order highlighted. Dotted arrows are for threads that starts execution, full arrows for threads that are stopping.*

shared *cycles* in step 3b. Thread metrics are then stored in the *thread map* (step 4). We repeat the same operations for the second thread when it stops execution (step 5), updating its metrics in the thread map (Step 6). When a thread exits, a *process exit* tracepoint triggers the thread removal from the *thread map* and the cleaning of the *processor topology map* for the HT that was running.

Unfortunately, all these measurements are subject to variability since context switches depend on how the application is interacting with the system. Applications that heavily exploit network communication like user-facing cloud workloads will be subject to more context switch activity than applications that focus on batch computations like *HPC* or *BigData* workloads. To reduce this uncertainty, each second we generate a software event that is captured by the eBPF code. This event allows computing the metrics for each running thread without waiting for a context switch, thus having an updated view of the thread metrics before collecting them in user-space.

Network monitoring

From an OS perspective, networking activity can be captured and analyzed at different layers of the stack. In particular, the OS handles communication until layer 4, while layer 7 is handled directly by the applications. To achieve *transparency*, we decided to monitor network traffic within the OS

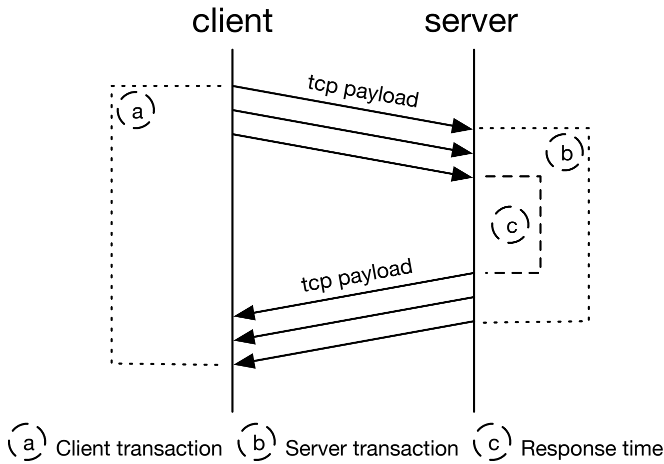


Figure 3.4: *Network transaction over a TCP connection as seen by a client process and a server process.*

at layer 4 (e.g. Transfer Control Protocol (TCP)) trying also to detect plain text data that can reconstruct layer 7 protocols (e.g. HTTP). To this aim, we decided to leverage TCP *kprobes* and *tracepoints* to capture the entire TCP payload and to avoid reconstructing the TCP flow. Within this context, we are mainly interested in network performance metrics like bandwidth and latency. Once we have the network connection and the TCP payload, the effort to measure bandwidth is quite low. On the contrary, measuring latency requires to introduce the concept of network transaction.

As shown in Figure 3.4, a *network transaction* is a data exchange in an established TCP connection between a client process and a server process where the client initiates the transaction by sending one or more TCP payloads and the server replies by sending one or more TCP payloads. A new network transaction begins (and the previous one ends) when the client process starts again to send one or more TCP payloads after receiving the TCP payloads sent by the server during the previous transaction. Starting from a network transaction, we can define two measurements: the server response time and the transaction Round-Trip Time (RTT). The server response time measures the time required by the server to build a response to a given request, while the transaction RTT measures the time from the first TCP payload sent by the client to the last TCP payload received by the client. On the one hand, if we are monitoring a client process, we measure the transaction RTT. On the other hand, if we are monitoring a server

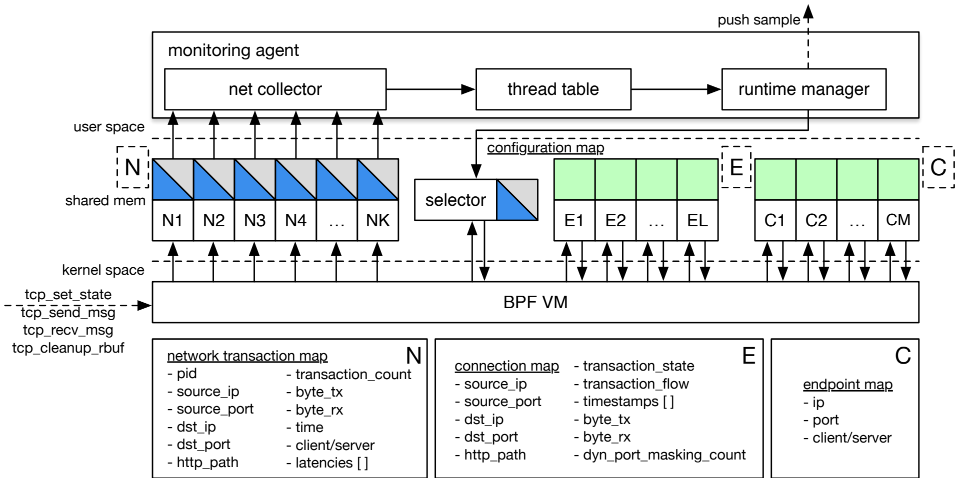


Figure 3.5: *Network monitoring structure: the BPF code sends metrics to the agent leveraging the network transaction map and stores partial network data in the endpoint map and in the connection map. The user space agent iterates over the network transaction map to collect the metrics and attribute them to each container. Network metrics are stored in the thread table to be sent to the backend via the runtime manager.*

process, we measure the server response time.

Figure 3.5 shows the main components behind the kernel-level data acquisition for network metrics. The eBPF code reacts to TCP kprobes like *tcp_set_state*, *tcp_send_msg*, *tcp_recv_msg* and *tcp_cleanup_rbuf* and supports both IPv4 and IPv6 communications. The kernel calls *tcp_set_state* function when there is a state change in the TCP socket (e.g. listen, syn sent, established, closing) and we capture it to track the state of each connection in the system. *Tcp_send_msg* is called when the kernel wants to send data over the TCP connection, while *tcp_recv_msg* is called when one or more TCP payloads need to be received. Kprobes and tracepoints can be attached at function invocation as well as at function return, providing respectively the input parameters and the return value of the given function. Unfortunately, while *tcp_send_msg* provides socket data, message content and message size at function invocation, *tcp_recv_msg* provides the socket data and an empty message pointer at function invocation and the amount of data read at function return. Even if we instrument both invocation and return of *tcp_recv_msg*, we will not be able to assign the message size to the connection as we do not have the socket data at function return. For this reason, we decided to attach a kprobe also at the invocation of *tcp_cleanup_rbuf*, which is a function called mainly by *tcp_recv_msg* at the end of the mes-

sage retrieval. *Tcp_cleanup_rbuf* provides at function invocation both the socket and the size of the data (but not the message pointer), allowing us to reconstruct for a given socket the message read (through the message pointer saved from the *tcp_recv_msg* kprobe) and its size.

We record all the information we collect from the kprobes according to the mechanism of Figure 3.4 in the *network transaction map* and we leverage some helper maps to update the state of all the connections in the system. In particular, Figure 3.5 shows two helper maps: the *endpoint map* and the *connection map*. Each time a connection is established, we store the endpoints in the *endpoint map* with information on the endpoint role (i.e. client or server) to attribute the correct timing when capturing data exchange between clients and servers. The *connection map* is used instead as a scratchpad to keep track of the open connections and to keep track of the transaction state and flow for each connection. Within this context, a connection is identified by source IP and port, destination IP and port, and the HTTP path that we extract from the message payload in case of plain-text transmission. The *network transaction map*, instead, stores for each connection the data about all the transactions recorded in the last observation interval. In particular, we store the Process ID (PID) of the last thread that interacted with the connection, the number of transactions, the amount of bytes received and transmitted, the average latency and a sample of the latency measures (i.e. 240 items, configurable). Once a transaction exceeds the size of the latency measures array, a new item is inserted in a uniformly random position using a reservoir sampling technique. Data in the *network transaction map* uses the same selector mechanism described for the power and performance monitoring activity.

Given that we use source IP, source port, destination IP, destination port, and HTTP path when available to identify a given connection in our system, we may have a lot of connections between the same two endpoints where the only difference is in the client port. This is particularly true for HTTP, where each request can require a new connection with new parameters unless the client decides to recycle the already open connections. This behavior severely affects our data collection system, as we need to generate a new item in the *network transaction map* for each connection we track for just a few network transactions. To avoid *network transaction map* saturation, we decided to mask the client port in case we track the HTTP protocol and in case the two endpoints exchange few transactions (currently less than 10) during the connection lifetime. This trade-off still allows to have a detailed view of the network activity in terms of granularity of connections, allows to better use the latency sampling technique and reduces the

pressure on the *network transaction map*.

User-space data collection

The user-space agent is the first data aggregation layer in our monitoring infrastructure. Its main role is to manage the kernel-level instrumentation and to collect the metrics coming from the maps located in shared memory. Each second, the user-space agent switches the selector in the configuration maps of the network monitor and of the power and performance monitor and collects the RAPL measurements for power attribution. Then, it starts iterating over the *thread map*. For each entry of the *thread map*, the user-space agent collects all the metrics (i.e. power consumption, CPU usage, execution time, cycles, IR, cache references and cache misses), applies Equation (3.2) for power attribution and creates an entry in the *thread table*. It is worth noticing that the contents of the *thread map* are not deleted after data collection as the entries are evicted in the eBPF code when the *process exit* tracepoint is called. After the user-space agent finished collecting the power and performance metrics, it scans the */proc* folder to find the *cgroup* of each recorded thread. If the cgroup was generated by docker, the user-space agent creates a container entry in a *container table* and aggregates the metrics of all the threads belonging to that *cgroup*. The threads that do not have a cgroup are grouped in the *container table* as *other* to keep track of their activity.

When the user-space agent finished building the *container table*, we have the list of its PIDs for each container; thanks to this information we can attribute network transactions to each container. To this aim, the user-space agent starts to iterate over the *network transaction map*. For each entry in the *network transaction map*, the user-space agent creates a transaction group object that contains the aggregate network metrics (i.e. transaction count, byte transmitted, byte received and average latency) and the raw latency samples. From the raw latency samples of each transaction group, we compute the latency percentiles using *DDSketch* [70], which is a library able to compute quantiles and percentiles over large data-sets. Aggregate network metrics and raw latency samples are grouped by container looking at the PID list, then we apply again *DDSketch* to compute the percentiles for each monitored container. After this step, we clear the *network transaction map* to prepare it for the next data collection, whereas *endpoint map* and *connection map* entries are evicted by the eBPF code when the connection is closed by one of the two endpoints. At this point, the agent connects to the *kube-api-server* to retrieve Kubernetes state data (i.e. pods, containers associated to the pods, services, namespaces, and hosts), packs all these

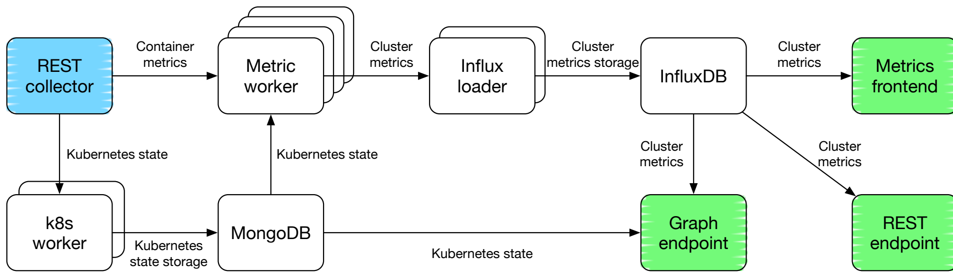


Figure 3.6: *Backend pipeline structure: the REST collector (highlighted in blue) is the entry point that collects all the data coming from the agents, while metrics frontend, REST endpoint and Graph endpoint (highlighted in green) are the endpoints used to visualize and query data. Components are connected with a queuing system enabling distribution, replication and asynchronous communication. Arrows show the data flow.*

data along with the metrics of the *container table*, and submits the sample to the monitoring backend.

3.3.2 Cluster view and analysis

The backend is the second data aggregation layer in our infrastructure and its goal is to provide a cluster-level view and an analysis of the performance of the monitored applications. Figure 3.6 shows the steps the backend takes to manipulate the metrics to achieve this goal. Metrics arrive at the *REST collector* and are propagated to other components that transform them. The final output can be consumed from three components: the *metric frontend*, the *REST endpoint*, and the *graph endpoint*. We will first describe how we ingest, aggregate and visualize data, then, we will detail how we build and analyze the graph of microservices.

Data ingestion, aggregation and visualization

We designed the backend to be able to process metrics from different clusters of different users to provide a general service. The *REST collector* is the entry point for the metrics that we collect with the monitoring agents and receives both container metrics and Kubernetes state metrics. Container metrics are sent directly to the *metric workers*, while Kubernetes state metrics are sent to the *k8s workers*. Communication with the components of the backend happens through a queuing system to work asynchronously and to keep samples in the queues storage to avoid data loss in case of system saturation.

The *k8s worker* takes the new Kubernetes state information and updates our internal view of the cluster that is stored inside the backend *MongoDB*[3] instance. At the same time, the *metric worker* takes the container metrics and aggregates them depending on the cluster data that we processed with the *k8s worker*. In particular, container metrics are aggregated by pod and by host. Then we add information about the namespace to segment them appropriately. Network transaction group metrics follows also the *metric worker* step. Once the *metric worker* finished processing the sample, it sends everything to the *influx loader*, which prepares the data and stores them inside the backend *InfluxDB*[4] instance. At this point, the metrics are ready to be consumed by the backend endpoints: *metric frontend*, *REST endpoint* and *graph endpoint*. For the *metric frontend*, we leveraged *Grafana*[5] to directly query the time-series database to show the data, although the system is general enough to support other dashboard frontends (e.g. [72]). The *REST endpoint* provides metrics that can be used to control the performance of the observed workloads. Finally, the *graph endpoint* computes and analyzes the interactions between microservices.

Graph analysis

Once we processed and grouped the metrics depending on the different layers of the infrastructure, we can build a graph that allows analyzing the behavior of the microservices in a distributed scenario. In particular, we are interested in:

1. the distributed interactions between microservices,
2. the ability of microservices to promptly respond to the user requests,
3. how the performance of the distributed system components constraint the overall performance of the system.

The first step to obtain this information is to build the graph, where we consider pods as microservices. For each transaction group (TCP or HTTP) we have the connection information that allows to link microservices and we have the transaction group performance metrics. We query the *MongoDB* instance to get the list of pods and the list of hosts in the cluster with their IP addresses and we query *InfluxDB* to get the transaction groups data. We then use *JGraphT* [78] to build the graph, where pods and hosts are the vertices, while the connections between them are the edges.

[3] https://www.mongodb.com
[4] https://www.influxdata.com/products/influxdb-overview/
[5] https://grafana.com

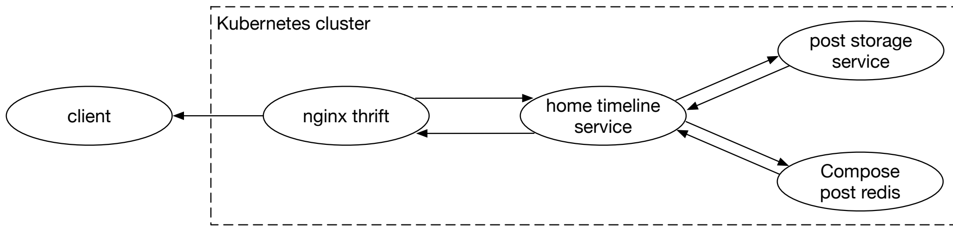


Figure 3.7: *View of the social network application (DeathStarBench) while reading a home timeline: the graph shows the client, a Nginx reverse proxy, two pods that manage the home timeline and a redis database. Between client and reverse proxy we have only one edge as the client is outside the monitored machines.*

One of the issues we found when building the graph is represented by the network manipulation performed by the Kubernetes Container Network Interface (CNI) plugin. In particular, Kubernetes CNI plugins like *calico*[6] and *flannel*[7] route packets leveraging *IPTables* and Destination Network Address Translation (DNAT). This means that the IP addresses of the pods and the destination IPs may be different. This happens because Kubernetes implements the *service* concept within the cluster. A pod usually cannot connect directly to another pod because it does not know its destination address. Instead, it uses the address of a known *service* endpoint. The *service* implements a round-robin layer 4 load balancer that internally translates the IP address of the service with the IP address of the pod selected for that connection. The translation is done at the *IPTables* layer without sending out network packets, meaning that we do not have visibility of the server endpoint when we look at the client-side connection. To solve this issue, we store in the backend the service IPs to translate back the IP addresses of the pods. Of course, in the case of multiple server pods behind the service, we connect the client pod to all the server pods, but only when there is a server connection that links back the server pod to the client pod. An example of the resulting graph showing the distributed interactions between pods can be seen in Figure 3.7. The graph shows a *DeathStarBench* [47] *social network* benchmark stressed with a *read home timeline* workload. Connections between pods are represented by multiple pairs of edges, where each pair has an edge going from the client to the server and an edge going from the server to the client (both with the related metrics).

The graph is the starting point for two analysis that helps in understanding the behavior of the system. At the vertex level, we can analyze

[6]https://www.projectcalico.org
[7]https://github.com/coreos/flannel

the performance metrics we collect to understand which pod represents a bottleneck. If we consider the graph as an open model, we can analyze the asymptotic bounds on performance thanks to operational analysis [62]. Open models with infinite customers that can arrive at any given time do not have a pessimistic bound on response time. However, we can look at the throughput bound, trying to estimate the load of each pod and deriving the maximum throughput that the system is able to sustain before saturation and as such before response time starts to increase abruptly. All the quantities are evaluated as average values and we consider as arrival rate of the system $\lambda(t)$ the sum of the arrival rates generated by elements of the graph that are outside the distributed application. To correctly perform the operational analysis, we modified the equations provided in [62] to properly model multithreaded workloads co-located on the same multicore processor, as the original equations consider only single core processors serving one workload each.

Equation (3.3) shows how we compute the utilization $U_i(t)$ of each pod i for a given time t. In Equation (3.3) we scale the $CPU_usage_i(t)$ of pod i by the minimum between the number of threads of the pod and the number of cores of the host machine to obtain a value that falls in the range $[0..1]$ bounds included.

$$U_i(t) = \frac{CPU_usage_i(t)}{\min(\#cores_j, \#threads_i)} \tag{3.3}$$

Then, we compute the service demand $D_i(t)$ for each pod i for a given time t. The service demand indicates how much time a job stays inside a given pod. In our case, $D_i(t)$ is the ratio between the pod utilization and the arrival rate $\lambda(t)$.

$$D_i(t) = \frac{U_i(t)}{\lambda(t)} \tag{3.4}$$

Finally, we compute the estimated arrival rate of saturation $\lambda_{sat}(t)$ of the overall distributed application at a given time t by taking the inverse of the maximum service demand $D_{max}(t)$. The arrival rate of saturation $\lambda_{sat}(t)$ is a theoretical bound that indicates which microservice is close to throughput saturation in a given time and can be used to decide whether it is necessary to scale that microservice to sustain the load of all the incoming customers.

$$\lambda_{sat}(t) = \frac{1}{D_{max}(t)} \tag{3.5}$$

If instead, we look at the edge level, we can find the critical path between the entry-points of the distributed application and its pods. To do so,

we resorted again to *JGraphT*. We modified the graph structure, pruning the edges by selecting only the server edges with maximum latency percentiles (we can choose the weight among 50th, 75th, 90th, and 99th percentile latency). From this graph, we take the entry-points, which are all the vertices without outgoing edges and we compute the widest path from each entry point to each vertex in the graph. From this set of widest paths, we select the one with the highest score as the critical path.

3.4 Evaluation

In this section, we evaluate the proposed monitoring approach. First, we assess the accuracy of the monitoring approach for all the metrics we collect w.r.t. a set of tools that can represent a golden standard for the measurements (Section 3.4.2). Then, we evaluate the cost of data collection. In particular, we compare our work with state of the art tools that are able to collect a similar set of metrics (Section 3.4.3). Finally, we leverage our tool to analyze some applications of the *DeathStarBench* benchmark suite [47].

3.4.1 Experimental setup

Hardware platform

We evaluated the proposed approach on a small cluster composed of two Dell PowerEdge r720xd servers, each one equipped with 2x Intel Xeon E5-2680 *Ivy Bridge* with 10 cores each (20 HT) clocked at 2.80GHz and with 380GB of RAM. The hardware platform represents a recent mid-range setup. The host OS is a Ubuntu Linux 16.04 with kernel 4.15 with eBPF enabled. On each machine, we installed Docker Community Edition with version 18.06.2 and Kubernetes with version 1.17.3. All the experiments were carried out with HT enabled.

Benchmarks and goals of the experimentation

We leverage a different set of workloads to evaluate different aspects of the proposed approach. We centered our evaluation on the *DeathStarBench* benchmark suite [47]. At the time of writing, DeathStarBench provides implementations for Kubernetes environments for the social-network benchmark and the media-microsvc benchmark. The *social-network* benchmark is composed of an *Nginx* web server that works as an entry point for many other microservices. Each microservice covers a particular functionality of a social network deployment like user management, post management, social graph management, URL shorten and advertising, read and write

Table 3.1: *Experimental setup for the social-network benchmark and the media-microsvc benchmark. For each workload we show the number of pods involved and the request rates for low, medium, and high configurations.*

benchmark	workload	# pods	low $\lambda(t)$	mid $\lambda(t)$	high $\lambda(t)$	duration
social-network	compose post	21	300 req/s	400 req/s	500 req/s	120 s
social-network	read home timeline	5	5000 req/s	6000 req/s	8000 req/s	120 s
media-microsvc	compose review	27	250 req/s	300 req/s	350 req/s	120 s

timeline, contents management, search. The benchmark deploys many database instances (e.g. MongoDB, Redis, and Memcached) to support each microservice functionality, where there is no shared database instance between microservices. The *social-network* benchmark provides two workloads, one that simulates users reading their home timeline and one that simulates users composing posts. The *media-microsvc* benchmark, instead, implements a media reviewing, renting ,and streaming platform. This application is composed of identification services (users and movies), a review composition service, a page composition service, services for users and reviews consultation, and a video streaming service. As in the social-network application, the microservices store data within databases and in-memory stores like MongoDB, Redis, and Memcached. The *media-microsvc* benchmark provides one workload that simulates users composing reviews. These two applications are able to effectively resemble the interactions between microservices in a complex deployment, and, as such, they can provide interesting insights on how to monitor them. Table 3.1 shows how we configured the benchmarks: for each workload, we test 3 different levels of intensity named *low*, *mid*, and *high*. We used these configurations to evaluate the overhead and the network metrics accuracy of the proposed solution.

To evaluate the accuracy of the low-level measurements, the execution time, the CPU usage, and the power consumption metrics, we leveraged three micro-benchmarks from the NPB suite [11]. Although they are not commonly referred to as representative of microservices, they provide stable benchmarks to evaluate the measurement mechanisms (particularly for low-level metrics). Here we provide details about the benchmarks selected from the NPB suite:

- **EP** is a kernel that generates pairs of Gaussian random deviates and is a benchmark highly CPU intensive and CPU bound;

- **MG** is a memory-intensive benchmark that performs a simplified multi grid calculation, it requires highly structured long-distance communication and it tests both short and long-distance data communication;

- **CG** is a mixed CPU and memory intensive workload that performs an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix through a conjugate gradient method.

Before resorting to the NPB benchmarks, we tested the accuracy of the low-level metrics with the two aforementioned benchmarks of the *DeathStarBench* benchmark suite. To do so, we selected the *perf* [37] tool to provide the golden standard for the performance metrics. Although perf can attribute metrics to each running *cGroup*, the overhead generated heavily affected the execution of the benchmarks (+12.93%, +14.6%, and +9.16% perf overhead w.r.t. our solution for read home timeline, compose post, and compose review workloads respectively). This prevented us to compare the results obtained with perf w.r.t. our solution when executing the tools separately. Unfortunately, it is not possible to run the tools concurrently, as perf resets the counters after reading them.

3.4.2 Metrics accuracy

We leveraged both the NPB and the three DeathStarBench workloads to assess the accuracy of each metric we collect. In particular, we used the NPB with problem size *C* and 40 threads to measure the accuracy of cycles, IR, cache references, cache misses, execution time, and CPU usage on a multi-socket server. We leveraged *perf* as a golden standard for cycles, IR, cache references, and cache misses, while the Linux tool *time* was used as the golden standard for execution time and CPU usage. We then used the NPB with problem size *C* also to assess whether the power attribution mechanism described in Section 3.3.1 provides good results. Finally, to measure the accuracy of the latency metrics (i.e. average latency, request count, byte transmitted, 50th, 75th, 90th, and 99th percentile) we monitored the workloads described in Table 3.1 and we compared our results with what we obtained from the load generator. We run each experiment 20 times.

Performance metrics

Figure 3.8 shows the Relative Error (RE) between our measurements and the golden standard results for the cycles, IR, cache references, cache misses, execution time, and CPU usage metrics. EP metrics gave us a result that is always below 4% in RE. For CG and MG, the maximum RE is near 5% for the cache references and cache misses metrics (and IR for MG), while the cycles metric has a RE below 4.5%. Finally, execution time has a RE below 3% for all experiments and CPU usage has a RE near 2% for EP and CG,

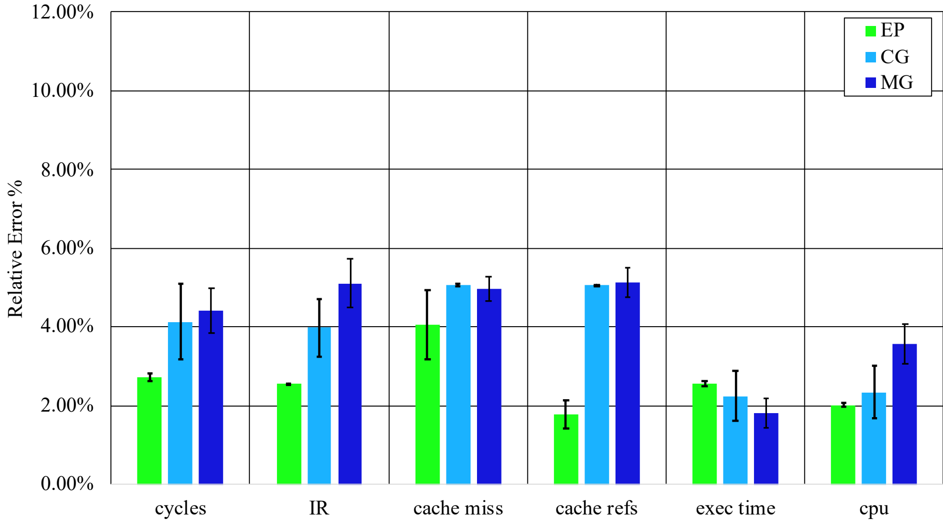


Figure 3.8: *Relative Error of cycles, IR, cache references, cache misses, execution time, and CPU usage of our tool w.r.t. the golden standard measured on 30 runs of EP, MG, and CG. Bars show 95% confidence interval.*

Table 3.2: *Average execution times (with 95% confidence interval) and overhead of the monitoring tool for the three NPB applications with problem size C and 40 threads each.*

	monitor	**no monitor**	**overhead**
EP	6.947s ± 0.004	6.950s ± 0.003	-0.05%
CG	21.116s ± 0.140	20.207s ± 0.192	4.49%
MG	8.435s ± 0.051	8.303s ± 0.038	1.59%

while MG has a RE of 3.56%. The results we obtained allow us to say that the monitoring tool is accurate on those metrics considering the goal of the proposed approach, that is monitoring and not profiling. We can find the reasons for the RE values on the different overheads posed by the golden standard tool and our implementation. As we can see from Table 3.2, our tool has a negligible impact on EP, while MG and CG have overheads that are comparable with the RE values shown in Figure 3.8. Moreover, our tool does not compute metrics on process exit, as these metrics are not actionable anymore. In particular, if the monitoring tool is used in a control loop to act on the monitored system, when a process exits the control loop cannot act anymore on the process as its effects are no longer visible.

Table 3.3: *Execution time, Energy, and Power consumption values (with 95% confidence interval) of EP and MG (10 threads each) when executed in isolation or in co-location with a fully disjoint set of cores.*

	configuration	exec time (s)	energy (J)	power (W)	time delta	energy delta	power delta	power scaled (W)	power scaled delta
EP	**co-location**	25.06s $\pm$ 0.13	1168.23 $\pm$ 11.45	46.62 $\pm$ 0.43	7.09%	-9.30%	-15.30%	49.93	-2.87%
	isolation	23.40 $\pm$ 0.12	1288.08 $\pm$ 15.86	55.04 $\pm$ 0.55				51.40	
MG	**co-location**	16.79 $\pm$ 0.09	736.33 $\pm$ 9.33	43.84 $\pm$ 0.47	21.23%	-15.78%	-30.54%	53.16	2.10%
	isolation	13.85 $\pm$ 0.10	874.33 $\pm$ 6.20	63.12 $\pm$ 0.23				52.07	

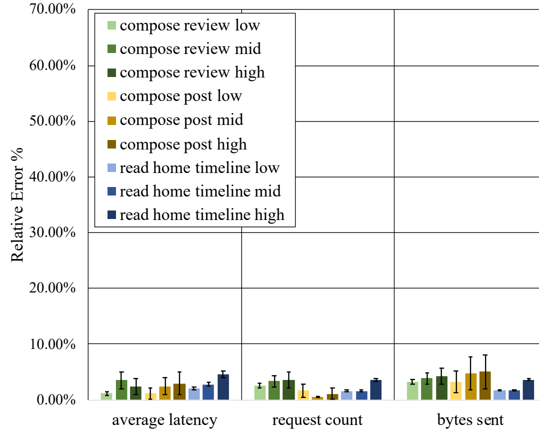
Power consumption metrics

Table 3.3 shows the results of EP and MG for what concerns the power attribution metrics. Unfortunately, a golden standard for power attribution in case of workloads co-location does not exist. Within this context, we can try to evaluate if power attribution provides reasonable results using the NPB micro-benchmarks, although formal correctness cannot be verified. We configured EP and MG to run with 10 threads and we pinned them to a disjoint set of HT of the first socket of the test server: 10 EP threads on 5 cores and 10 MG threads on the other 5 cores. Then, we run the two benchmarks in co-location as well as in isolation. We decided to not use CG because the benchmark has a higher execution time w.r.t. EP and MG and most of its execution would have happened in isolation.

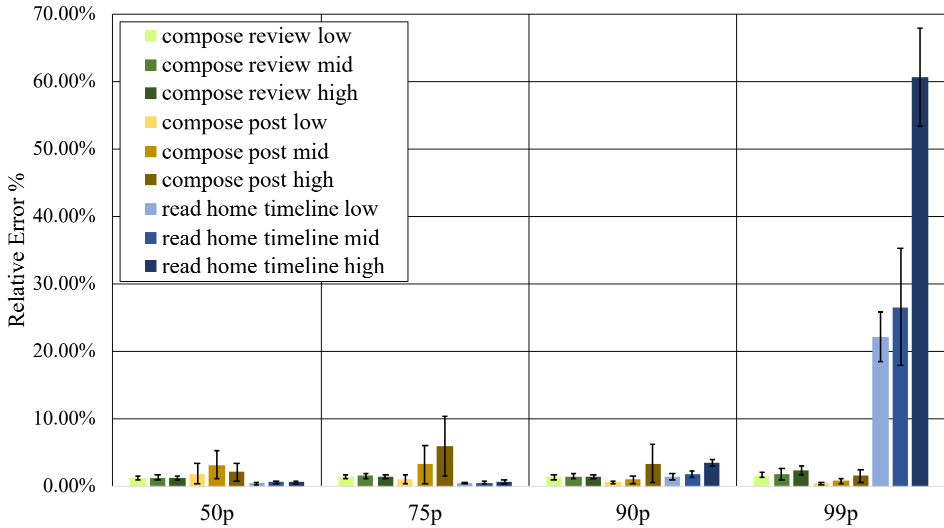
As we can see from Table 3.3, both benchmarks show an increase in execution time and a decrease in total energy consumed when running in co-location. It is clear that co-location increase contention and slows-down the access to shared resources, however, if we compute the *scaled power consumption* (i.e. we compute the power consumption of a benchmark run in a given configuration using its energy and the execution time of the other configuration), we can see that the difference between the power consumption of the two configurations is of $\simeq$2% both for EP and MG. This happens as both EP and MG are compute-bound micro-benchmarks and their operations are regular. This means that changing their pace can change almost linearly the execution time and the energy.

Network metrics

Figure 3.9 shows the results we obtained running the workloads of Table 3.1 and comparing the network metrics collected by our solution with the ones gathered by our golden standard: *wrk2*. Looking at the results related to average latency, requests count, and bytes sent by the workloads (shown in Figure 3.9(a)), we can see that the proposed tool is able to measure them with an error that is less than 5% on average. The 95% confidence interval shows that the worst case for this set of metrics is represented by the



(a) Average RE between our tool and wrk2 for avg latency, requests and bytes transmitted.



(b) Average RE between our tool and wrk2 for 50th, 75th, 90th, 99th percentile latency.

Figure 3.9: *Average RE between our tool and wrk2 for network performance metrics for the 3 workloads with different amount of requests. Bars show 95% confidence interval.*

compose post workload with the bytes sent metric, where the error falls in the range 5% $\pm$ 3.08%. Part of these errors are due to the different measurement mechanism: wrk2 measures the metrics on the client-side within the application, while our monitoring tool measures them within the kernel on the server-side. These errors are limited, as the proposed tool is able to compute those metrics considering all the requests and not just a subset. Figure 3.9(b) shows instead the measurement error for the latency percentiles, which are computed from a reservoir of 240 items collected each second for each open connection. Before looking at Figure 3.9(b), it is fundamental to keep in mind that the results are computed for a single HTTP endpoint and with a single client sending all the requests to stress test the data collection system. In a more general setting, the requests sent to the workloads of Table 3.1 would have been generated by multiple clients. In this case, our tool would have been able to create multiple reservoirs, one for each TCP connection and one for each HTTP endpoint requested by each client, increasing the accuracy. In general, it is always possible to increase the reservoir size, although it is necessary to evaluate the trade-off between accuracy and measurement overhead. Looking at Figure 3.9(b), for what concerns the compose review workload, the worst case error is 2.35% on average for the 99th percentile metric for the high configuration. The compose post workload, instead, has the errors on all the metrics below 3.5% on average except for the 75th percentile of the high configuration, where the error is 5.97% on average. For what concerns these two workloads, the reservoir sampling is behaving in a good way, providing samples that are representative of the requests performed by wrk2. The read home timeline workload instead, shows a different behavior. While for the for 50th, 75th, and 90th percentile latency metrics the errors shown in Figure 3.9(b) stays below 3.5% on average, the 99th percentile latency shows errors of 22.17% for the low configuration, 26.61% for the mid configuration, and 60.70% for the high configuration. This error is due to the sampling activity performed by our tool. In particular, the read home timeline workload generates for each second requests for the high, mid, and low configurations that are respectively 33, 25, and 20 times higher w.r.t. the number of samples collected each second by our tool. The 99th percentile metric becomes unreliable with this load level.

To investigate to which extent our tool can be effectively used to measure the 99th percentile latency, we run the read home timeline workload with an increasing number of requests (up to 16 times the size of the reservoir) and we show the measurement errors in Figure 3.10. We report also the 50th, 75th, and 90th percentile latency for completeness. As we can see

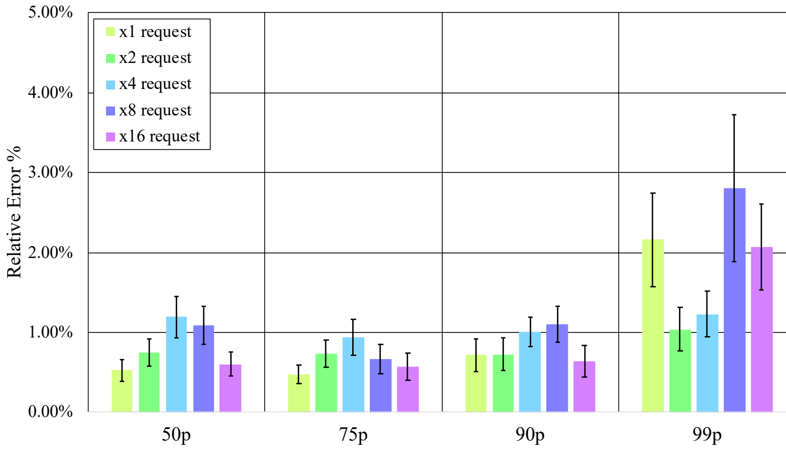
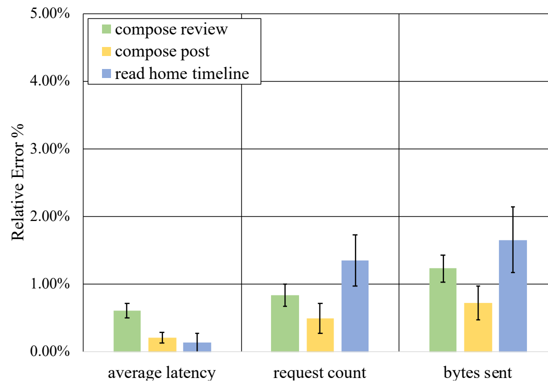


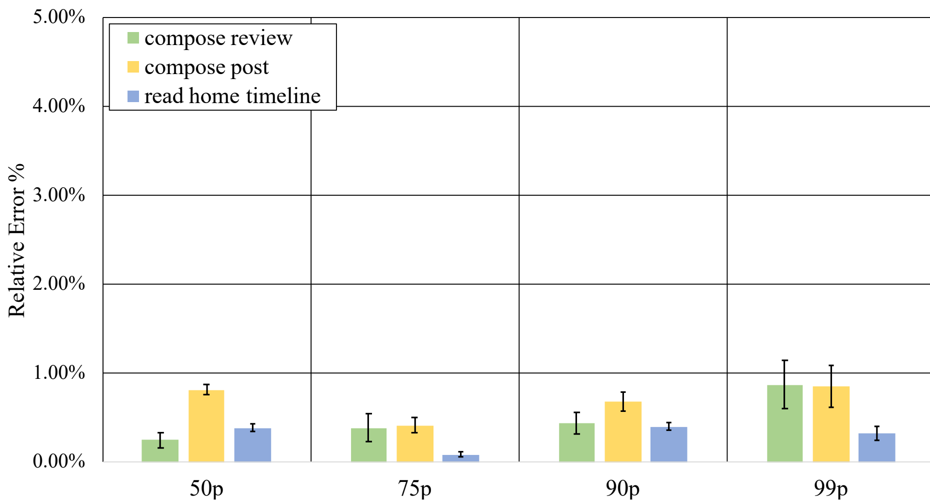
Figure 3.10: *Average RE between our tool and wrk2 for 50th, 75th, 90th, 99th percentile latency for the read home timeline workload with different amount of requests (from 1 to 16 times w.r.t. the capacity of the samples reservoir).Bars show 95% confidence interval.*

from Figure 3.10, the proposed tool is able to measure the 99th percentile latency up to 16 times the reservoir size with an error that is always below 3% on average. When we increase the load further, as shown in Figure 3.9(b), such measure becomes unreliable.

To demonstrate that the main source of uncertainty in the quality of the measurements is the number of requests collected each second for each connection, we tested the workloads of Table 3.1 in a more variable setup. In particular, we added a 10ms $\pm$ 5ms normally distributed delay on all the network requests performed by wrk2 and the workloads using the Linux tool *tc*. The results of this experiment are shown in Figure 3.11. Here we report only the high configuration, as the workloads in the three configurations behaved in the same way. As we can see from Figure 3.11(a), the errors for the average latency, request count, and bytes sent are limited and always below 2% on average. We obtained the same behavior for 50th, 75th, 90th, and 99th percentile latency (shown in Figure 3.11(b)) with an error always below 1.5%, even with the read home timeline workload. The reason why we obtained this result is twofold. On the one hand, the delay reduced the number of processed requests per second, allowing the tool to build a more comprehensive view of the performance of the workloads. On the other hand, the way in which we perform the measure is able to account for all the network delays and errors of the TCP protocol, thus providing very accurate results.



(a) Average RE between our tool and wrk2 for avg latency, requests and bytes transmitted.



(b) Average RE between our tool and wrk2 for 50th, 75th, 90th, 99th percentile latency.

Figure 3.11: *Average RE between our tool and wrk2 for network performance metrics for the 3 workloads (high configuration) with network delay of 10ms $\pm$ 5ms normally distributed. Bars show 95% confidence interval.*

3.4.3 Agent overhead

Monitoring activity should be carried out without imposing too much overhead on the monitored applications. Within this section, we will evaluate the overhead of the proposed monitoring tool w.r.t. other two similar state of the art solutions: *Weave scope* [95] and *Sysdig cloud* [89]. We selected *Weave scope* as a baseline because it monitors just CPU, memory usage (by scanning */proc*), and the number of open connections between pods. Its overhead should be always lower than the one we introduce in the system, as *Weave scope* does not monitor performance counters and network transactions. *Sysdig cloud*, instead, is a monitoring tool that measures resource usage, network I/O, and file I/O using *system calls* captured by a kernel module that sends to user-space all the raw system call traces. To have a fair comparison, we disabled all the metrics that *Sysdig cloud* collects and we do not collect, with the only exception of file I/O, as some of the network operations can be performed with file-related system calls with a network File Descriptor (FD) as a parameter [20]. We measured the overhead of our approach leveraging the workloads detailed in Table 3.1. The overhead is computed as the increase in latency of the network requests of the benchmarks, obtained by wrk2 from the average latency measurement and the 50th, 75th, 90th, and 99th percentile latency measurements. We deployed each application on the two test machines using Kubernetes and we pinned each pod to a given machine to avoid too much noise in the resulting data. Each test was executed 20 times and here we show average results with 95% confidence interval.

Although the testing setup is small, it can be considered a relevant setup as the overhead of the proposed monitoring tool depends only on the instrumentation performed through eBPF. The instrumentation adds execution time to the network operations performed by the benchmarks and to the context switches the OS performs during execution. For this reason, if we consider a microservice benchmark executed on a small cluster w.r.t. the same benchmark with the same amount of containers executed on a larger one, the smaller cluster is a far more challenging environment. This happens because the number of network requests captured by our tool remains the same across the two setups, but in the small cluster setup the pressure on our monitoring tool is higher as the network requests and the context switches are not spread across many monitoring agents but are instead condensed into few ones.

Figure 3.12 shows the normalized overhead w.r.t. a no monitor execution of the compose post workload of *Weave scope*, our solution, and *Sysdig*

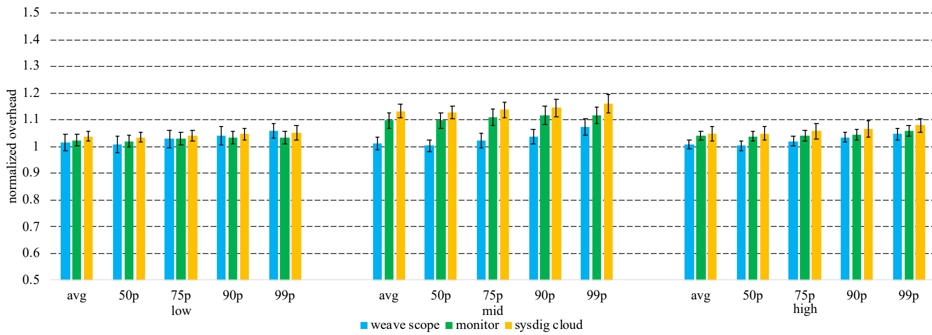


Figure 3.12: *Normalized overhead (with 95% confidence interval) w.r.t. no monitored execution of weave scope, our solution, and sysdig cloud while running the compose post workload for three different workload levels.*

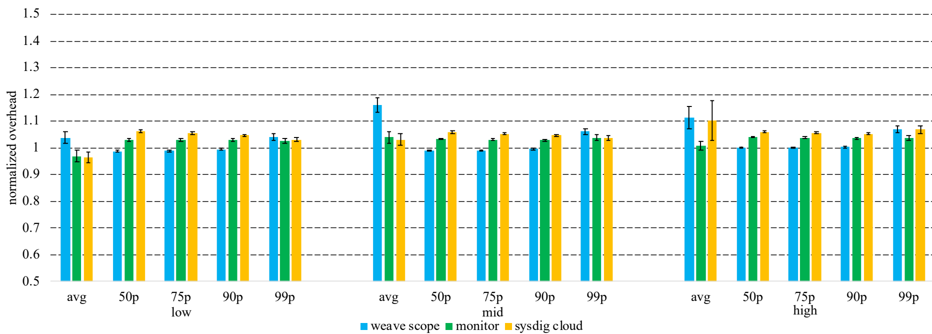


Figure 3.13: *Normalized overhead (with 95% confidence interval) w.r.t. no monitored execution of weave scope, our solution, and sysdig cloud while running the compose review workload for three different workload levels.*

cloud for the three levels of the benchmark we described in Table 3.1. The results of Figure 3.12 confirms that *Weave scope* provides very low overhead in all configurations. Our solution is able to outperform *Sysdig cloud* in all the configurations in all cases, providing an overhead of $\simeq$ 10% at most, obtained the mid configuration. The 95% confidence interval shows that the execution of our tool was sufficiently regular across all the runs.

Figure 3.13 shows the normalized overhead w.r.t. a no monitor execution of the compose review workload of *Weave scope*, our solution, and *Sysdig cloud* for the three levels of the benchmark we described in Table 3.1. This experiment shows an interesting behavior w.r.t. the previous one. In general, our solution always outperforms *Sysdig cloud* also in this case, but it is also able to outperform *Weave scope* in terms of impact on the

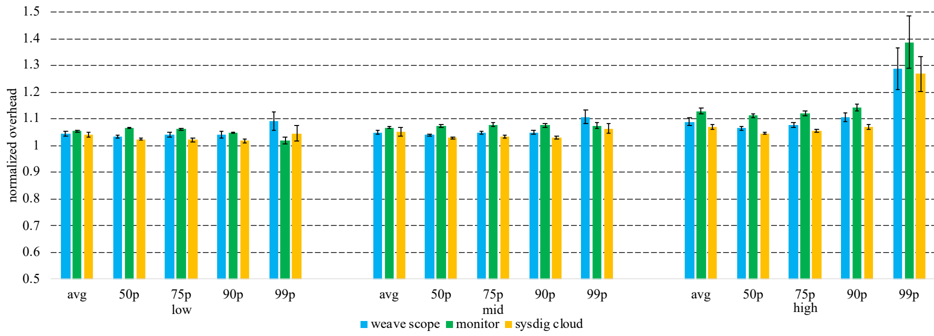


Figure 3.14: *Normalized overhead (with 95% confidence interval) w.r.t. no monitored execution of weave scope, our solution, and sysdig cloud while running the read home timeline workload for three different workload levels.*

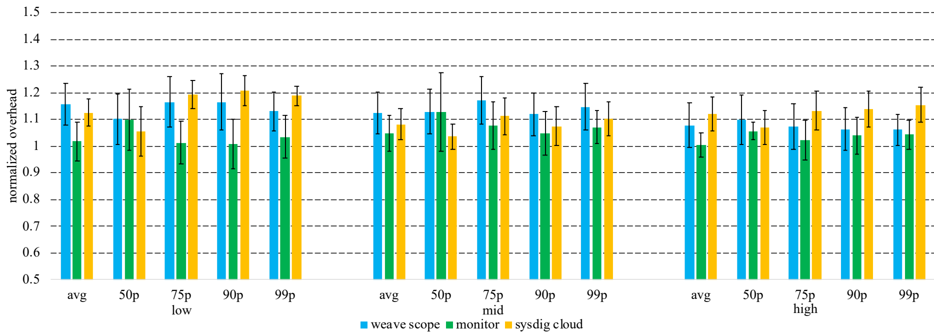


Figure 3.15: *Normalized overhead (with 95% confidence interval) w.r.t. no monitored execution of weave scope, our solution, and sysdig cloud while running the read home timeline workload over the social network benchmark with more connections (500 instead of 100) and less requests per second (2000 req/s for high, 1500 req/s for mid, 1000 req/s for low).*

average latency and on the 99th percentile latency of the three workload configurations. On the contrary, *Weave scope* behaves better in the 50th, 75th, and 90th percentile latency.

Figure 3.14 shows the normalized overhead w.r.t. a no monitor execution of the read home timeline workload of *Weave scope*, our solution, and *Sysdig cloud* for the three levels of the benchmark we described in Table 3.1. In this case, our solution behaved unexpectedly: *Sysdig cloud* was able to outperform it in all cases for all configurations. It is worth noticing that the overhead introduced by our solution w.r.t. a no monitor execution is, in the worst case, of 0.2 ms on the average network latency metric and

2ms in the 99th percentile latency metric. To investigate further this result, we increased the number of parallel connections (from 100 to 500) and we reduced the number of requests per second (2000 for high, 1500 for mid, 1000 for low). The idea is to reduce the request per connection ratio to assess the overhead on the requests themselves and the connection setup. Figure 3.15 shows the result of this new experiment. Our solution behaves better than *Weave scope* and *Sysdig cloud* in all the cases except the 50th percentile latency metric for the low and mid configuration. This result is extremely important as it shows that our solution behaves better in case of many parallel connections, which is far more common w.r.t. what we presented in Figure 3.14.

3.4.4 Benchmark study

To give an overview of the insights we can provide, we decided to stress test the social network benchmark with a compose post workload with 10000R/s divided across 60 connections. We configured the *graph analysis* to retrieve the last connection data of each pod in a time interval of 5 seconds and we report the output data in Table 3.4 for what concerns the operational analysis and in Figure 3.16 for what concerns the critical path. We do not report the entire graph due to space limits.

Although we loaded the benchmark with 10000R/s, the system can respond only to 1576R/s. To investigate the source of this bottleneck, we resorted to the operational analysis formulas described in Section 3.3.2. As we can see from Table 3.4, the *compose-post-redis* pod is almost saturated with a $U_i(t)$ equal to 96.9%. Moreover, *compose-post-redis* is responding to 11138 R/s coming from the other pods in the application. If we apply equation (3.5) to this case, we find that $\lambda_{sat}(t)$ is 1626 R/s, which is extremely close to the current $\lambda(t)$. If we take another look at Table 3.4, we can see that the application entry-point represented by *nginx-thrift* and most of the other application components have very low utilization, meaning that a strategy to replicate the *compose-post-redis* database could solve the bottleneck. Another interesting insight that we can obtain from Table 3.4 is that the average latency of *nginx-thrift* is higher w.r.t. most of the other containers within the microservice application. This happens as *nginx-thrift* serves as an entry-point and reverse proxy for the other pods of the application. For this reason, its latency is affected by the latency of the pods it has to connect to in order to build a full response that is sent to the client as a response to an HTTP request. Such connections may happen in parallel or sequentially depending on the data dependencies of the

Table 3.4: *Example output of the operational analysis with pod name, utilization, arrival rate of each pod, maximum arrival rate for each pod, power consumption, number of threads, and HTTP and TCP average latency for a DeathStarBench social network benchmark loaded with 10000 R/s with 60 open connections. Arrival rate 1576 R/s, estimated saturation arrival rate 1626 R/s.*

pod name	$U_i(t)$ %	$\lambda_i(t)$	max $\lambda_i(t)$	power (W)	# threads	average latency HTTP (ms)	TCP (ms)
compose-post-redis	96.90%	11138	11493.57	7.40	1	-	1.36
user-timeline-redis	7.00%	1348	19249.45	1.03	1	-	0.01
social-graph-service	5.56%	401	7210.87	7.43	8	-	0.57
nginx-thrift	5.53%	1571	28365.00	13.41	32	39.34	19,27
social-graph-redis	5.14%	596	11574.56	0.55	1	-	0.08
user-memcached	4.06%	1288	31663.31	3.37	8	-	0.00
write-home-timeline-rabbitmq	3.03%	741	24401.31	3.55	15	-	0.23
text-service	2.53%	1461	57553.38	8.48	231	-	27.39
user-mention-service	2.51%	2845	113337.91	7.98	61	-	4.87
compose-post-service	2.36%	2899	122493.18	17.27	296	-	13.76
user-timeline-service	1.51%	471	31116.93	4.33	14	-	0.53
post-storage-service	1.46%	714	48796.94	4.40	24	-	0.52
url-shorten-service	1.08%	1420	130699.35	4.23	112	-	13.07
unique-id-service	0.83%	879	105375.93	3.75	62	-	12.67
media-service	0.81%	777	94905.51	3.40	62	-	12.49
user-timeline-mongodb	0.51%	962	188029.60	2.17	30	-	0.16
post-storage-mongodb	0.28%	481	168390.95	0.92	26	-	0.24
url-shorten-mongodb	0.25%	537	214770.44	1.12	46	-	0.20
user-service	0.19%	480	242366.64	1.76	60	-	16.12
user-mongodb	0.16%	537	318997.27	1.01	49	-	0.13
social-graph-mongodb	0.14%	228	152210.67	0.35	22	-	0.21

application. Finally, Table 3.4 shows also that the most utilized pod in the application is not the one with the highest power consumption. Solving the bottleneck could change the power profile of the system: this aspect should be considered when building applications that aim to be efficient in terms of performance, resource usage, and power consumption.

To further analyze the relations between pods in the social network application, we found the critical path between the entry-point and all the other pods. We configured the graph to have as edge weights the 90th percentile latency of the connection that is represented by the given edge. Of course, two pods may have multiple connections between one another, however, given that we are using a percentile latency, we decided to prune the graph by taking the slowest edge connecting each pair of pods. Figure 3.16 shows the critical path with the chain of connections from the entry-point to the last pod in the path. As we can see, within the chain we have also the same *compose-post-redis* pod that we found out to be the bottleneck of the application. Solving the bottleneck means that the increased amount of requests will affect all the pods in the chain, and, for this reason, it is fundamental to verify whether the pods in the chain will be able to sustain the new load.

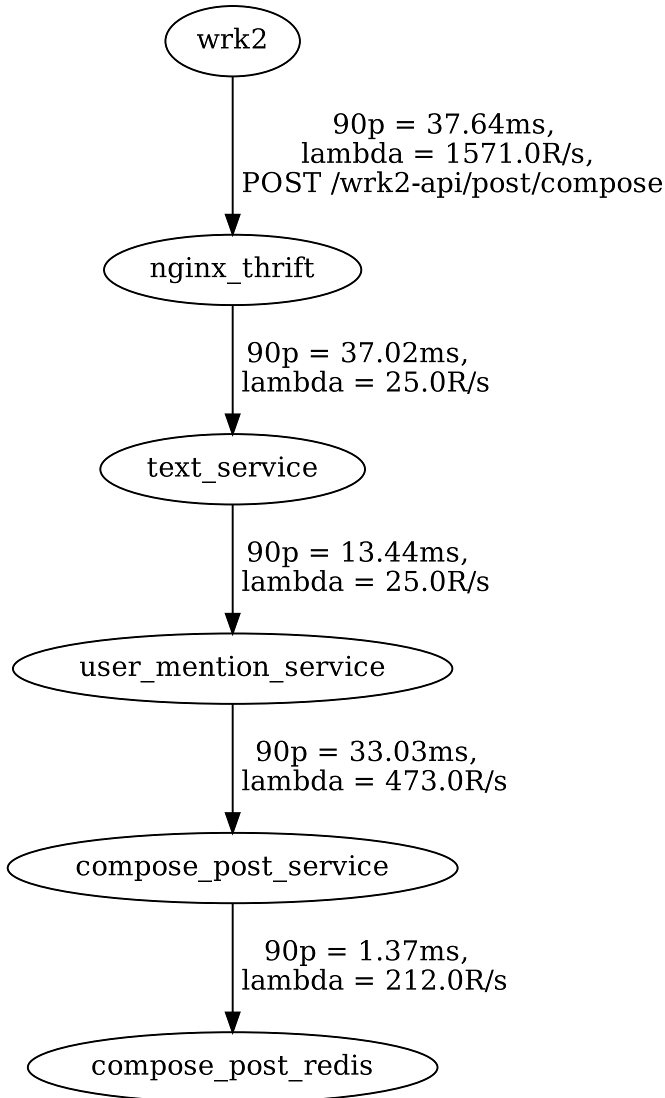


Figure 3.16: *Critical path example of the DeathStarBench social network loaded with 10000 R/s with 60 open connections. Edge weights are the 90th percentile latency.*

3.5 Conclusion and future work

In this chapter, we proposed a novel black-box monitoring approach to build a unified view from the architectural level to the application performance level of a microservice-based application. The proposed approach leverages eBPF to generate the metrics at the kernel level to transparently build a view that aims to be accurate and with low overhead on the monitored applications. We showed how we collect data about resource usage, performance counters, power consumption and network performance without instrumenting the user code. We then used such data to analyze the microservice-application w.r.t. its interaction with the external world as well as w.r.t. the internal connections and dependencies. The experimental campaign showed that the proposed approach is reasonably accurate w.r.t. its goal, which is performance monitoring. We also showed how the proposed approach can introduce less overhead in the monitored application w.r.t. a solution that is based on similar technologies but with a different data extraction strategy.

As a future work, the proposed approach will be integrated with more data sources to build a complete view of the monitored applications. In particular, monitoring file and disk I/O will allow detecting bottlenecks that depend on the performance of the disks. Moreover, we will introduce predictive models w.r.t. the measured latency to detect in advance bottlenecks that will saturate the capacity of the system. This will allow a timely reaction when such situation occurs.

CHAPTER 4

Performance-aware power capping for cloud and containerized applications [1]

4.1 Introduction

Inside cloud environments we can find several classes of workloads, from interactive services to batch workloads. On the one hand, interactive services have to be extremely reactive to the user input, which have arrival rates usually unpredictable. This characteristic usually makes them latency sensitive and latency critical. On the other hand, batch workloads should execute in a fast way to compute data that can be accessed by the interactive services, and in the end by the final users.

In this context, power consumption is a fundamental aspect that can not be neglected: on the one hand, power consumption is accounted for the 20% of the TCO [34] of a data-center, while, on the other hand, cloud servers are energy inefficient at low and medium loads. If we consider continuous batch workloads running on dedicated WSC systems, it has been proven

[1] The work presented in this chapter was published in [5], for which Rolando Brondolin developed part of the methodology, part of the implementation, and part of the paper writing. ©2018 IEEE. Reprinted, with permission, from: Marco Arnaboldi, Rolando Brondolin, and Marco Domenico Santambrogio. Hyppo: Hybrid performance-aware power-capping orchestrator. In 2018 IEEE International Conference on Autonomic Computing (ICAC), pages 71–80. IEEE, 2018.

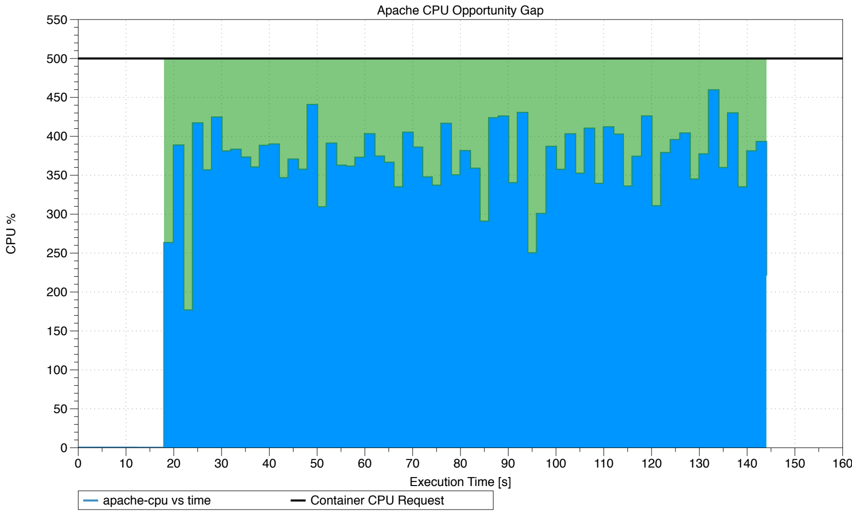


Figure 4.1: *Representation of the opportunity gap for the Apache benchmark. The graph shows that the workload under utilizes the resources requested by it (e.g. 500% of the CPU, equivalent to 5 cores), presenting a gap that can be exploited in order to improve energy proportionality.*

that the average utilization of the system is around 75% [12]. However, this percentage drastically decreases in the more common scenario where a mix of several types of workloads (e.g. batch plus on-line application) run in such systems. In this case the utilization varies between 10% and 50% [12]. In order to reduce the TCO in this last case it is important to improve the energy efficiency at low and moderate paces.

Several works were proposed in the literature to cope with this issue [65, 74]. These techniques usually exploit either resource consolidation in order to turn off un-utilized machines or delay techniques like idle injection, in order to force the activation of the deep sleep mode on the servers. Unfortunately, these techniques are not applicable for most of the cloud workloads. On the one hand, consolidation is not always effective when dealing with highly variable interactive workloads because load spikes can saturate the resources and increase latencies and execution times. On the other hand, cloud and interactive workloads should be able to serve each request with a response time of tens of milliseconds at most: this prevents to batch requests to introduce sufficiently long idle periods to be able to save power. In this context, to the best of our knowledge, an autonomic system able to manage performance and power consumption of cloud workloads running in Docker containers in a distributed environment is still missing.

In this chapter, we propose HyPPO, a Hybrid Performance-aware Power-capping Orchestrator that tackles performance requirements and power optimization of workloads in Kubernetes [25] and Docker [41] based environments. Our research work aims to tackle the new challenges introduced by the variable nature of cloud workloads, exploiting fine grain Dynamic Voltage and Frequency Scaling (DVFS) techniques and considering *power consumption* as a control variable in order to achieve energy proportionality. We present an orchestrator based on a distributed ODA control loop able to enforce a power cap while being aware of how this impacts the performance of the running workloads. In particular we aim to exploit the opportunity gap that cloud workloads intrinsically present, as shown in Figure 4.1. The proposed methodology is designed to be as extensible as possible. Our goal consist in improving energy proportionality of different kind of containerized workloads running in a Kubernetes cluster [18]. This is achieved by reducing power consumption while respecting the given SLA, defined as the CPU request assigned to each container in their configuration.

The rest of the chapter is organized as follows: Section 4.2 presents related work on hybrid power capping in containerized infrastructures; Section 4.3 describes the proposed approach based on a distributed ODA control loop and details its components; Section 4.4 shows the experimental evaluation of the proposed orchestrator, while Section 4.5 concludes the chapter and derives the future work.

4.2 Related Work

Achieving energy proportionality in virtualized and/or containerized data-center environments has become a major topic for researcher, increasing the number of works proposed in this field. An interesting work in this direction is the one presented by Piraghaj et al. in [84]. They proposed a framework and an algorithm for energy efficient container consolidation in cloud data-centers. The developed framework was designed for working in a virtualized environment, in which the containers were running inside VMs deploying Docker. They defined the SLA as the workload requested CPU utilization and modelled the optimization as a minimization operative research problem. The function to minimize in this case was the overall power consumption of the data-center. Under this perspective, our hybrid orchestrator chooses a greedier approach, minimizing the power consumption per node. Finally, the authors evaluated the proposed methodology only through simulation via CloudSim [27]. That allowed the author to design and to integrate in the proposed framework any kind of monitoring

task.

Another work trying to achieve energy proportionality in a virtualized environment is the one proposed by Arnaboldi et al. with their *XeMPUPiL* orchestrator [6]. The authors developed an orchestrator based on a ODA control loop able to reduce power consumption in a pure Xen[2] virtualized environment. The results were achieved thanks both to resource consolidation at VM level and power capping through RAPL at system level. Even in this case the problem was defined as an operative research task. In the proposed solution the SLA can be defined under two different perspective: 1) a power budget that can not be violated and under which we want to maximize the performance and 2) a minimum performance requirement under which it is possible to minimize the power consumption. In both cases the term performance refers to the number of IR that the workloads returned in a time window. This lets the proposed approach to work only with batch workloads. Furthermore, the authors required to instrument the hypervisor in order to perform both the monitoring and actuation tasks of the proposed ODA approach.

For what concerns works regarding containerized infrastructures, Asnaghi et al. [9] proposed a methodology to minimize power consumption in a Docker environment based on a ODA control loop. The described methodology consisted in different control policy. These were achieve through monitoring the performance via parsing *perf*[3] in order fed data to a Proportional Integral (PI) controller and finally, to actuate the obtained configuration via CPU Quota and C-groups. In terms of precision, the monitoring and acting stage of the DockerCap approach represent the main bottleneck for the methodology, not allowing to export the approach on multiple nodes. In fact, the tests were conducted only in single node mode.

Finally, an outstanding work in the field of achieving energy proportionality in WSC is the one proposed by Lo et al. in [66]. They characterized the behaviour of two On-Line Data Intensive (OLDI) application running in Google clusters: *search* and *memkey*. Thanks to this first step they were able to identify the opportunity gap that are going to exploit in order to reduce power consumed by such workloads. To do so they exploited the *PEGASUS* controller, a dynamic multi layer bang-bang one, in order to fine-grain enforce the power cap at server level via RAPL. The methodology is designed to respect a SLA defined on the admissible maximum latency that such workloads should provide. The results were gathered on the Google production clusters and showed that the methodology was able

[2]https://www.xenproject.org/

[3]https://github.com/torvalds/linux/tree/master/tools/perf

to achieve significant energy proportionality with negligible SLA violation. The only drawback consisted that the proposed approach were tested only for services composed by homogeneous tasks. All the described works represent a starting point from which the development of our solution takes inspiration in order to overcome the aforementioned limitations.

4.3 Methodology

Modern data-centers TCO is composed of several expenditures, from buildings to servers and maintenance. One of the most important operative costs of a datacenter is represented by power consumption, which can impact for $18\% - 20\%$ of the TCO [34]. The correct management of power consumption at run-time then becomes a key feature for modern data-centers, allowing to reduce energy waste and as such reducing energy bills in the long run. Of course, setting the power consumption of a system disregarding the performance of the applications running on it introduces performance issues that will lead to SLA violations. A different approach that can be used towards the goal of improving the energy efficiency in containerized environments could be the opposite: try to guarantee performance requirements and applications SLAs while optimizing power usage in an energy proportional fashion. In this sense, energy proportionality is defined as the ability of a system to be energy efficient in all possible working conditions, and in particular in the typical operating region. This property was introduced by *L.A. Barroso* [13] in 2007 and is the key concept behind HyPPO, which applies it transparently w.r.t the applications: this is done because the system should be able to autonomously monitor and adapt itself towards performance and power goals.

Overall, HyPPO is an autonomic orchestrator that manages workloads in a distributed environment backed by Kubernetes. The main goal of HyPPO is to guarantee performance required by the containers while optimizing power usage of the workloads on the physical servers. In this context, the performance requirement of each container is expressed as resource requests and limits, which is a feature already supported in the description of pods in kubernetes. What HyPPO adds is a hybrid [99] power-aware resource management that allows to reduce power usage when containers are under-utilizing their resources. The hybrid feature of HyPPO means that to guarantee performance and optimize power we leverage hardware techniques such as Intel RAPL [86] and at the same time software techniques to control how the power budget is allocated.

Figure 4.2 shows the overall architecture of HyPPO: it is composed by

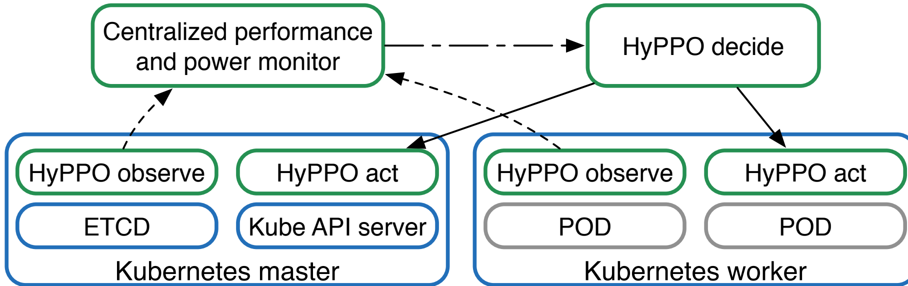


Figure 4.2: *HyPPO ODA loop: we observe the status of each server, centralizing power and performance metrics. Then, the decision phase computes a new strategy towards the system goals, which is enforced by the actuators.*

several actors, all belonging to an ODA loop structure. At first, HyPPO collects data from physical hosts connected to the Kubernetes cluster. In this context we are interested in performance metrics (e.g. CPU usage, PMCs measurements) and power measurements at the finest grain possible. All these data are then moved from each host to a centralized performance and power monitor, which is able to aggregate metrics having cluster visibility. Then, the *Decide* component takes this information when needed and elaborates a new strategy tailored for the cluster and its performance goals. The outcome of this step is then passed to the *Act* phase, which enforces the new strategy. The system is then able to converge given that each applied decision is observed again, leading to a new strategy towards the system goals. In this context, we designed HyPPO having two goals in mind:

1. monitor and guarantee measurable SLAs, as the autonomic system behavior should be understandable and accountable w.r.t. business and technical objectives,

2. build a low latency ODA loop, as a fast response in case of sudden load spikes can reduce the SLA violations.

In the next Sections, we will describe how autonomic energy proportionality can be applied in distributed container-based environments. In Section 4.3.1 we will cover how we *Observe* workloads and extract informations used by the *Decide* phase described in Section 4.3.2. Finally, Section 4.3.3 will describe how we *Act* to adapt the system towards the desired goals.

4.3.1 Observe: power and performance monitoring

The first activity that should be performed to achieve energy proportionality is to make the system aware of how it is behaving in terms of power consumption and energy. To this aim we leveraged DEEP-mon, the monitoring tool described in Chapter 2, for which we provide a brief description here. DEEP-mon is a lightweight performance and power monitoring tool based on BCC [14] and eBPF [15, 73]. DEEP-mon collects *IR* and *cycles* PMCs, CPU usage and execution time at each *context switch* using Linux tracepoints [64], collecting data for each thread in the system and aggregating for each running container. Cycles measurements are weighted depending on HT usage and are used to segment power measurements collected with Intel RAPL [86]. DEEP-mon collects also the status of the Kubernetes cluster using Intel Snap[4] plugins, sending all the data to a remote back-end on a regular time basis. In order to monitor the entire Kubernetes cluster, one containerized instance of the DEEP-mon agent is deployed for each physical server through a Kubernetes *DaemonSet*.

The monitoring back-end is composed of several stages. At first, the monitoring data arrives at the *Collector*. DEEP-mon sends two kind of samples to the collector: 1) Kubernetes cluster state with container names and 2) performance and power monitoring metrics. The former data is stored in MongoDB for a later use, while the latter are instead unpacked inside the metrics workers. In this stage, the backend takes the metrics segmented by container and aggregates them by pod, by namespace and pod, by host and pod and, finally, by host, namespace and pod. These new set of metrics are then passed to the *Influx loader*, which stores the time series data into the InfluxDB database. Two other components then access the time series database: the *metrics frontend* and the *REST endpoint*. The former is a web dashboard which queries the database to obtain real-time monitoring data while the latter is a REST service that exposes the APIs used by the *Decide* phase. Among the others, the REST endpoint exposes APIs to gather CPU usage, Power consumption, IR, cycles, execution time and Kubernetes requests and limits, segmenting these metrics by container and host.

4.3.2 Decide: performance-aware power allocation

The *Decide* stage is the "thinking" component of our orchestrator. The process of taking a decision is composed by three steps: retrieving information, exploiting such data to take a decision and finally, communicating

[4] http://snap-telemetry.io

the outcome to the actuator agents. All of them are done by implementing a *proportional controller* exposing an API to the actuators.

The first step is achieved by exploiting the *REST endpoint* exposed by the monitoring backend in a polling fashion (the poll rate is defined in the configuration of the controller). In detail, the controller asks for three kinds of information: 1) container CPU request and limits, 2) container power consumption and 3) container CPU usage. The first set of informations is rearranged in order to create a dictionary containing, for each user in the system, a map indicating which is the requested CPU usage for each container, thus defining the SLA to be respected. Then, the second kind fo information is aggregated on a per node basis in order to create a map representing the power consumption for each node in the cluster. Finally, the last information is exploited in order to perform the controlling task, hence taking the final decision the will be communicated.

The second task of this stage is the controlling policy itself. As already mentioned, the controller is implemented in a proportional fashion. Its duty lies in defining the power limit for each node where the controlled containers are running. This behavior is achieved through Equations (4.1) and (4.2). Equation (4.1) represents how the power cap for a given node n is computed, where $power_n$ is the power limit for the n-th node of the cluster, $power_{n,c}$ is the power consumed by the c-th container running on the n-th node.

$$power_n = P_{idle} + \sum_{c=0}^{C} (power_{n,c} + i(c)) \tag{4.1}$$

Equation (4.1) defines the power for the n-th node as the sum of the idle power P_{idle} and the sums of all the powers consumed by the c-th container running on the n-th node, plus a contribute $i(c)$. The contribution can be positive or negative and is expressed in Equation (4.2), where $cpu_request_c$ represents the CPU request expressed for the c-th container, cpu_usage_c represent the actual CPU consumption for the c-th container and P represents a proportional factor that can be defined in controller configurations. Each container CPU usage data point passes through the controller represented by Equations (4.1) and (4.2). In this way, it contributes positively or negatively to the total power consumption of the node on which the container is actually running.

$$i(c) = \begin{cases} (cpu_usage_c - cpu_request_c) * P & \text{if } \exists cpu_usage_c \\ 0 & \text{if } \nexists cpu_usage_c \end{cases} \tag{4.2}$$

The P parameter was chosen after several experiments and represent the

pace at which the controller tries to fill the opportunity gap. It is defined in the configuration file of the controller as 10000 mW (10W). In case a container does not have a defined $cpu_request$, according to Equations (4.1) and (4.2), it will contribute to the total power of the node only for the amount it consumes, since the proportional contribute will be 0. In this way, thanks to the proportional controller, it will be possible to exploit the opportunity gap. Once the gap is discovered, Equation (4.2) introduces a negative contribute which results in a stricter power cap on the node. This reduced power cap will influence the amount of CPU utilized by the workload, increasing it. Obviously this behavior will be true only for data intensive workload. For what concerns batch workloads the behaviour will be different and presented in Section 4.4. Finally, the last task of the *Decide* stage is carried out by a REST endpoint exposed by the controller, providing in this way a set of APIs exploitable by the actuator in order to impose the decided power limits for each nodes. This is achieved by delivering the decide stage inside a deployment in the kube-system namespace. To let its APIs be accessible from the actuators in a preconfigured way, the controller is wrapped in a Kubernetes service. This allow to expose the APIs to the cluster on a fixed URL and port. The actuators are able to sent requests to the controller via a GET HTTP request to the decide endpoint.

4.3.3 Act: enforcing power allocation

The *Act* stage is where the output of the decide stage is put in practice. The decision consists in enforcing a power cap on the node. This task is carried out by the actuators running on each node of the Kubernetes cluster. In order to define a power cap for the node we leveraged the RAPL interface, which provides a set of counters providing energy and power consumption information. The measurements provided by RAPL are obtained by a software model based on the PMCs developed by Intel. It has been proved that such measurements match the ones obtained through analog power meters [87]. RAPL provides a way to set power limits on processor packages and DRAM. This will allow a monitoring and control program to dynamically limit maximum average power, to match its expected power budget.

In RAPL, platforms are divided into domains for fine grained reports and control. A RAPL domain is a physically meaningful domain for power management. They are commonly divided in 4 kinds of domain: 1) DRAM, 2) PPO (i.e. core devices), 3) PP1 (i.e. uncore devices), and 4) Package, which is a domain comprehending both PP0 and PP1. We leveraged the *Package* domain, since we are using CPU usage as control variable in the

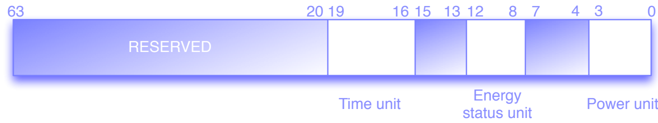


Figure 4.3: *Bits organization inside the MSR_RAPL_POWER_UNIT*

decide stage. RAPL exposes an interface for each domain that is exploitable by the developers in order to access this technology. This interface consists in five sub-interfaces:

1. *Power Limit*. Interface to specify power limit and its fine tuning such as the time window.
2. *Energy Status*, Interface to retrieve information about the power consumption.
3. *Performance Status*. It provides information about the effect due to the power limit. This interface is optional.
4. *Power Info*. It provides information about the range of parameters describing a given domain, such as min-power, max-power etc. This interface is optional.
5. *Policy*. It is used in order to describe a policy on how to divide budget between sub-domains in a parent domain. This interface is optional.

In our actuator we targeted the first two, since they allow to set the power cap and subsequently to check it is correctly enforced. These interfaces are accessible through three non-architectural Model Specific Registers (MSRs): 1) the Power Unit MSR shown in Figure 4.3, 2) the Package Power Limit MSR shown in Figure 4.4, and 3) the Package Energy status MSR. We exploit the first two in order to retrieve the units of measure used in the systems and then to enforce the power cap defined in the retrieved units. What we read from the RAPL_POWER_UNIT MSR is the *Power Unit* value contained in bits 3:0. Power is measured in Watts and expressed in "number of power units" inside the Package Energy Status register. This value is an unsigned integer. As default it is set to 001b. This value indicates that each power unit represents an increment of 1/8 Watt [51]. This information is used to transform the power limit retrieved by the REST endpoint exposed by the controller. According to the Equation (4.3) defined by Intel manual [51]:

$$unitInWatts[WATTS] = \frac{1}{2^{PU}} \tag{4.3}$$

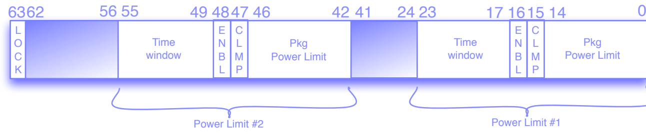


Figure 4.4: *Bits organization inside the MSR_RAPL_POWER_LIMIT*

where PU are the power units retrieved by the MSR. Equation (4.3) can be rearranged in Equation (4.4) in order to represent a power cap expressed in Watts in a power cap expressed in power units.

$$powerUnits = powerCap[Watt] \cdot 2^{PU} \tag{4.4}$$

This result is written in bits 14:0 of the MSR represented in Figure 4.4 in order to enforce the power cap on the package.

The actuator behavior can be summarized as follow: in a polling fashion an actuator queries the decide endpoint about the power limit to enforce on the node where its deployed. Once a valid response by the controller is retrieved, the actuator transforms the power cap expressed in Watts to one expressed in power unit by reading the power unit MSR and applying Equation (4.4). Finally, the obtained value is written in the Package Power Limit MSR of each core of the system. In order to have an actuator running on each node of the Kubernetes cluster, they are delivered inside a *DaemonSet* as the monitoring agents. In order to provide the actuator container with the access to the RAPL MSRs, it is defined as *Privileged* in the DaemonSet configuration file and the folder */dev/cpu* is mounted in it, enabling read and write of the MSRs.

4.4 Evaluation

To enable power efficiency in the workload orchestration, the orchestrator must provide some gains in terms of power, trying to reduce the opportunity gap without violating the CPU request assigned to the container. Finally, the orchestrator components impact on the system should be negligible. In Section 4.4.1 we are going to introduce the system on which the orchestrator was evaluated, the benchmark used and how they were configured in containers. In Section 4.4.2 we are going to presents the obtained results in terms of power savings and opportunity gap reduction.

4.4.1 Experimental setup

We evaluated the proposed methodology on a cluster of homogeneous machines, composed by two Dell PowerEdge r720xd equipped with 2x Intel Xeon E5-2680 *Ivy Bridge* with 10 cores each (20 HT) clocked at 2.80GHz and with 380GB of RAM. The evaluation platforms represent a recent mid-range servers. The host OS is an Ubuntu Linux OS 16.04 with kernel 4.13 and eBPF support enabled. Kubernetes version 1.10 was installed directly on the two nodes. Each workload runs inside a Docker container with a CPU request of 5 CPUs and a limit of 10 CPUs, with Docker runtime version 1.13.1. All the experiments are carried out with HT enabled without pinning the threads on any core, and we measure power, workloads CPU usage and performance counters data with DEEP-mon. The Kubernetes pods[5] affinity was set in order to prioritize the containers scheduling on the slave node. To evaluate our orchestrator we leverage six benchmarks from the phoronix test suite [61], version 7.6. In detail we tested our methodology with:

- **pts/apache-1.6.1** is a test of the *Apache Benchmark* program on an Apache2 web server that measures how many requests per second a given system can sustain when carrying out 1,000,000 requests with 100 requests being carried out concurrently;
- **pts/fio-1.4.0** is an advanced disk benchmark that depends upon the kernel's AIO access library. The test was executed using the examples/net.fio file included in the suite
- **pts/nginx-1.1.0** is a test of the *Apache Benchmark* program on a Nginx web server that measures how many requests per second a given system can sustain when carrying out 2,000,000 requests with 500 requests being carried out concurrently;
- **pts/postmark-1.1.0** simulates small-file testing similar to the tasks endured by web and mail servers and performs 25,000 transactions with 500 files simultaneously with the file sizes ranging between 5 and 512 kilobytes;
- **pts/dbench-1.0.0** Dbench is a benchmark designed within the *Samba project*, containing only file-system calls for testing the disk performance and 48 instances trying to perform them concurrently;
- **pts/iozone-1.8.0** tests the hard disk drive / file-system performance. It performs reads and writes in chunks of 512kB over 4GB of data.

[5] www.kubernetes.io/docs/concepts/workloads/pods/pod/

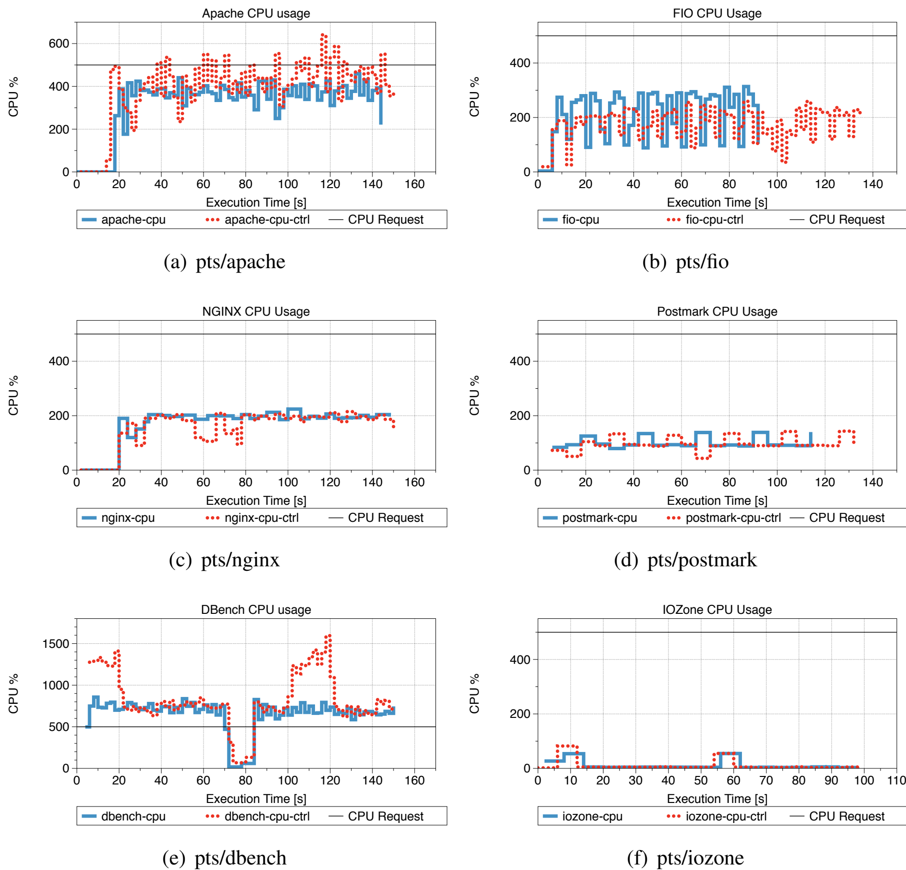
The measured CPU usage will be expressed in terms of %. In particular, the 100% will represent the fully utilization of a single core in the system. Thus, the maximum amount reachable in our test environments is 4,000%. This is due to the fact that each node provides 40 cores. This is also the reason why the CPU request for the containers is defined as 500%, since each container running a benchmark presents a CPU request of 5 cores.

4.4.2 Experimental results

The first set of experiments consisted in the evaluation of how good are we at exploiting the opportunity gap in different workloads. The obtained results are shown in Figure 4.5. It is possible to notice that in workloads where the stress is mainly on the CPU, like pts/Apache in Figure 4.5(a), we are able to exploit the opportunity gap, while violating the defined CPU request of the container at maximum the 5% of the time. Despite the pts/nginx benchmark is similar to the pts/apache one, they differ in the crucial aspect of scaling, since the former is bounded in the number of threads (2 as we can see from 4.5(c)). This allow to reduce the power consumed by the workload even without an increase in terms of CPU utilization. This happens because currently the HyPPO actuation mechanism does not consider workload parallelism, which is an aspect that should be taken into account when designing workload request and limits.

For what concerns the pts/dbench workload shown in Figure 4.5(e), we can notice that from the beginning (blue continuous line) it requires more than the requested CPUs. Before the beginning of the benchmark, the system is in idle state (the only containers utilizing CPU are the ones of Kubernetes system and the ones of our orchestrator), so the power cap defined by the controller and enforced on the node is the minimum possible. Once the benchmark starts, due to the node power configuration, it requires more CPU than the requested one. Our orchestrator discovers that according to Equation (4.2) and introduces a positive contribute to the power cap for the node. In this way, with a less strict power cap on the system, the workload can achieve its target performance. The same behavior happens in the switch between the first and the second stage of the pts/dbench benchmark, since there is an idle period in between the two.

Finally, for what concerns the other three benchmarks shown in Figures 4.5(b), 4.5(d) and 4.5(f) there is no clear exploitation of the opportunity gap, since these workloads are strongly dependent on the performance of the disks and IO peripherals. In particular, it is possible to notice that for what concerns pts/fio benchmark, our orchestrator introduce a delay in the



(a) pts/apache

(b) pts/fio

(c) pts/nginx

(d) pts/postmark

(e) pts/dbench

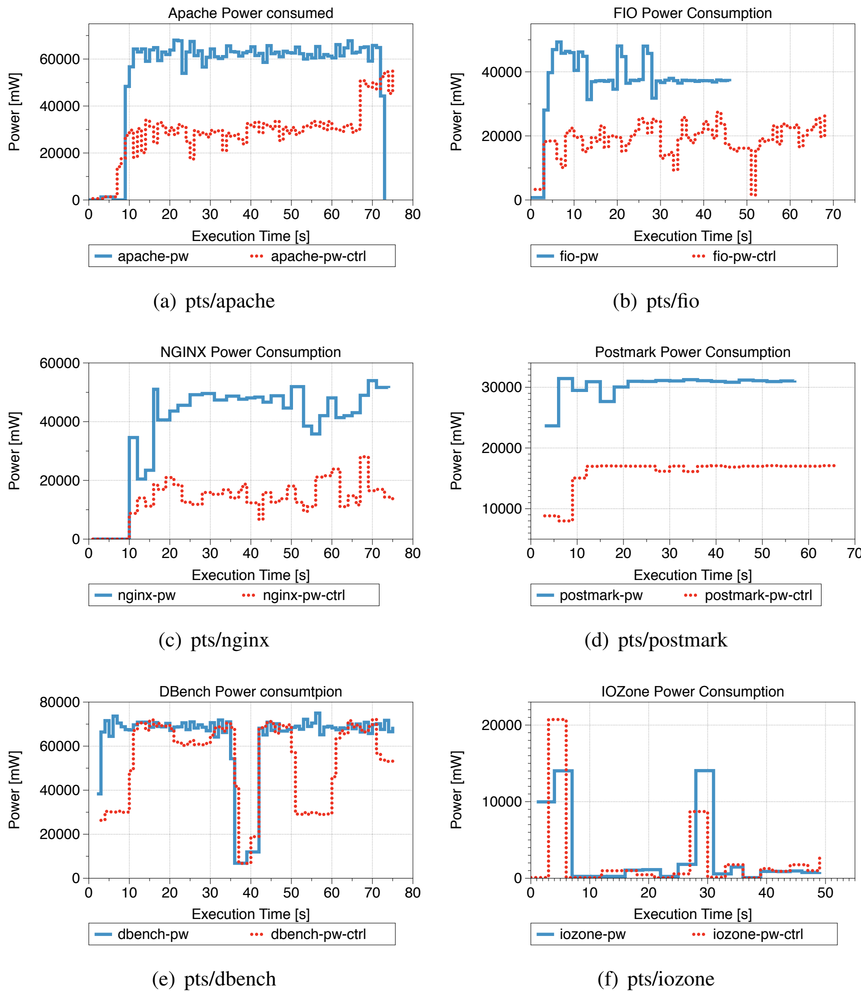
(f) pts/iozone

Figure 4.5: *The graphs represent the CPU usage percentage of the different benchmarks. 100% represent a single core fully utilized. A system with 40 cores has at most 4,000% CPU usage. The CPU request is set at 5 CPUs (black line, 500%). The blue continuous line shows the CPU usage without power capping. The red dotted line shows the CPU usage measured with the proposed hybrid orchestrator running.*

completion of the task. This depends on the nature of the fio benchmark, which is a batch workload. In the case of batch workloads our orchestrator tends to delay their execution in order to save power. In a real production environment the set of workloads running in a Kubernetes cluster is a mix of batch workloads (throughput oriented) and traffic intensive service (response oriented). This is why we conducted tests also on batch workloads in order to obtain a full concept of the behavior of the proposed orchestrator.

In Figure 4.6 it is represented the power consumed by the workloads expressed in mW. For what concern CPU dependent applications like pts/apache and pts/nginx in Figure 4.6(a) and Figure 4.6(c), the power gain is significant. If we consider the joint behavior of CPU usage and power of the pts/apache workload (Figure 4.5(a) and 4.6(a) respectively), we can see that the CPU usage increases despite the power consumption decreases. This happens as the power capping technique applied to all the workloads leverages Intel RAPL, which is able to precisely set a power cap for a given socket. To enforce this power cap, RAPL applies fine grain DVFS, which reduces the frequency of the cores. Of course the amount of work that pts/apache has to perform does not change, however, each request takes more time to be executed, increasing the CPU usage accordingly and slightly increasing the overall execution time of the benchmark. This is fine if we consider parallel and interactive workloads, as their goal is to complete requests as fast as possible and then free the CPU. However, if we consider CPU intensive workloads, applying this technique leads to an increased overall execution time, which could lead to an increase in the overall energy consumption.

Figure 4.6(e) shows how the orchestrator tries to save power for pts/dbench. When the workload begins, the power cap of the system is set to the minimum possible since nothing is actually running. This leads to an increase in terms of CPU utilization as we notice in Figure 4.5(e). The controller discovers this behavior and it consequently decides to increase the power budged allowing the workload to run at its best. Once the first part of the dbench benchmark terminates, the same behavior happens due to the idle period that exists between the two parts of the benchmark. This also points out that our controller requires almost 10s to take a decision and enforcing it in the worst case scenario. This is mainly due to the polling communication strategy that we adopted in order to let the three components of the orchestrator communicate. The polling time-outs are the main contributors to this delay. This amount can be reduced in the future, since we are planning to shift to a pub/sub communication strategy, where the delay will be mainly introduced by the computation time required by the



(a) pts/apache (b) pts/fio

(c) pts/nginx (d) pts/postmark

(e) pts/dbench (f) pts/iozone

Figure 4.6: *The graphs represent the power consumption expressed in mW, of the different benchmarks. The blue continuous line represents the case in which the benchmark power consumed is measured in a system with no power cap enforced. Instead, the red dotted line represents the case in which the benchmark power consumed is measured in a system with the proposed hybrid orchestrator running.*

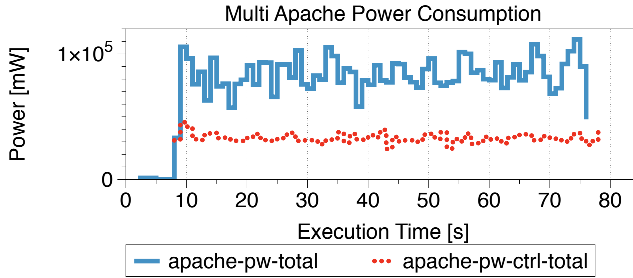


Figure 4.7: *Power consumption in the case of multi containers running in a Kubernetes pod. In this case two pts/apache container were executed concurrently. In blue continuous the power consumed in the case no power cap is enforced in the system. In red dotted the power consumed in case the hybrid orchestrator mange the power cap.*

three components instead of the communication time.

For what concerns the pts/fio workload presented in Figure 4.6(b), we are able to save power at the cost of delaying the completion of the workload. Finally, we obtain the same power consumption on the pts/iozone benchmark, since it does not significantly exploit cores resources, but its boundary is on memory. Finally, for what concerns pts/postmark and pts/iozone benchmark we are able to save power since we cap core resources instead of DRAM ones, which are the most utilized by these workloads.

The second set of experiments aimed to study and validate the behaviour of the proposed orchestrator in case of multiple running containers. The obtained results are reported in Figure 4.7, representing the power consumption of a pod composed by two containers each of them running an instance of the pts/apache benchmark. In this case we can observe that the power consumed is almost doubled with respect the case in which a single container running pts/apache was deployed in the system. Even in this case our orchestrator discover an opportunity gap in both the container that can be exploited. This results in power savings, while the CPU utilization behavior becomes similar to the one showed in Figure 4.5(a), for both the containers.

The last set of measurements was conducted in order to discover the impact of the proposed hybrid orchestrator on the system under two different perspectives: power consumed and CPU utilized. The results are reported in Table 4.1.

From Table 4.1 it is possible to notice that all the components of the orchestrator present a negligible impact in terms of CPU utilized in the system, even if considering the standard deviation. The same assertion held true analyzing the power consumed by the *Controller* and *Actuator* com-

Table 4.1: *The mean and the standard deviation of the CPU utilized and power consumed by the elements composing the proposed hybrid orchestrator.*

	CPU [%]		Power [mW]	
Component	μ	σ^2	μ	σ^2
Monitor	12,97	7,32	2536,20	3028,69
Controller	0,61	1,08	44,27	295,29
Actuator	0,13	0,14	15,28	145,76

ponents. Instead, the *Monitor* component consume slightly more, but still a negligible amount considered in the overall picture, as shown in Chapter 2. The huge standard deviation in all the reported measurements it is mainly due to the nature of the different workloads. Since workloads that will require to adjust the power cap often due to their behavior will require more interventions coming from the three components. On the other hand, applications requiring less adjustments will leave the Controller and Actuation components in an idle state, waiting for significant notification coming from the Monitoring components. This also explains the reason why this last component is the most expensive one, since among the three, it is the one that is always required to be active all the time, independently by the state of the system.

4.5 Conclusion and future work

In this chapter we presented a novel hybrid performance-aware power-capping orchestrator enabling better energy proportionality in a distributed containerized environment governed by Kubernetes. The described methodology does not require any kind of instrumentation of Docker, Kubernetes and the workloads running in the cluster. The proposed approach is based on an ODA control loop strategy composed by three stages: 1) the monitoring task performed by the DEEP-mon agents, 2) a controller in charge to reduce power consumption while respecting the define SLA, and 3) the actuation performed by the actuator agents, in charge to enforce the power cap on the single nodes exploiting RAPL interfaces. The results obtained during the experimental evaluation show that our approach is able to consume less power in almost all the adopted benchmarks with a negligible impact on the system.

Although the work proposed in this chapter is able to reduce power consumption of cloud workloads depending on their CPU requirements, more can be done to optimize such workloads and to improve the HyPPO methodology. The actuation scheme proposed in this work strongly de-

pends on Intel RAPL, and even if it does not reduce the applicability of the methodology with other CPU architectures and vendors, it does not consider workload parallelism. Future work will introduce *thread affinity* as a tuning knob to idle cores and to address also workloads with limited parallelism.

CHAPTER 5

Latency-aware power capping for cloud and containerized applications [1]

5.1 Introduction

Within the previous chapters we discussed how to measure microservices performance, as well how to attribute power consumption to each of them in a transparent way. Then, we started to analyze how to leverage such data to modify the behavior of the system, and in particular to keep the CPU near a target value defined by the developers for each microservice. Within this chapter we want to move forward by defining a different and more comprehensive approach.

As discussed in Chapter 1, cloud-native applications are usually heterogeneous, highly co-located and latency-sensitive. To manage power and performance for this kind of applications, we need to take into account all these aspects. Heterogeneity means that we need to take into account that each microservice is different from the others, with different workloads,

[1]The work presented in this chapter was published in [22], for which Rolando Brondolin developed the whole methodology, the whole implementation, the whole experimental evaluation, and the whole paper writing. ©2020 IEEE. Reprinted, with permission, from: Rolando Brondolin, and Marco Domenico Santambrogio. PRESTO: a latency-aware power-capping orchestrator for cloud-native microservices. In IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS), pages 11–20. IEEE, 2020.

different loads, and different roles. Co-location means that on the same physical server we may find microservices with different power profiles and different performance requirements. Finally, latency-sensitivity means that they privilege latency w.r.t. throughput. Given all these characteristics, we need to manage latency directly if we want to reduce power consumption in a meaningful way.

Within this context, this chapter focuses on the design of a power capping orchestrator able to manage power consumption while keeping the performance of the workloads near a predefined latency SLA. As already presented in the previous chapters, this work focuses on microservices executed within Docker containers and managed by a Kubernetes cluster. We decided to pursue this goal with a black-box approach to avoid any kind of instrumentation of the target workloads, as well as of the orchestrators and the run-time environments.

Several works addressed this challenge in the last few years. The most notable examples are PEGASUS [66], Rubik [59], and Copper [56]. PEGASUS [66] provides a feedback control loop to manage latency SLAs at scale for OLDI applications and actuates by reducing power consumption with RAPL [86]. Rubik [59] introduces a statistical model to manage latency-critical workloads along with batch workloads without degrading tail latency. Finally, Copper [56] provides a control scheme to manage power consumption with RAPL providing guarantees on performance requirements. Unfortunately, they lack some aspects concerning the problem at hand. PEGASUS does not consider the heterogeneity of workload components within the same server. Rubik leverages DVFS that, according to [56], is now difficult to integrate with current hardware. Finally, Copper [56] instruments each workload component to monitor performance and requires to explicitly set the latency goal for each one.

Given the current research opportunities, in this chapter, we present PRESTO. PRESTO takes into account microservices co-location and heterogeneity, provides a fully black-box approach that avoids modifying the user code, and leverages modern power capping techniques. In particular, the contributions of this chapter are the following:

- the design and development of a graph-based analysis to attribute service time requirements to each microservice in the cluster starting from a latency SLA that is enforced at the cluster entry-point level,
- the design and development of an ODA control loop able to monitor latency and power consumption, define service time goals based on the observed metrics, transform such goals in power budgets, and enforce

them to all the machines in the cluster through the RAPL hardware power-capping interface.

The proposed methodology reduces the power consumption of the target workloads by 37.13% on average with a control error that is below 12.5% and below 1.5 ms on average.

The rest of this chapter is organized as follows: Section 5.2 describes the main related works in the field, with a focus on microservice and Kubernetes based solutions. Section 5.3 details the step we followed to build PRESTO and its components. Section 5.4 shows the experimental result we obtained with PRESTO using the *DeathStarBench* suite [47]. Finally, Section 5.5 concludes and derives future work.

5.2 Related Work

Several works in the past addressed the challenge of minimizing power consumption while keeping workloads performance near a certain SLA. If we consider latency as a SLA, the fundamental aspects to take into account are the definition of the correct latency target as well as the ability to maintain the target latency. *Brutlag* [24] showed that increasing the latency of a Google web search reduces the number of searches performed by each user from 0.2% to 0.6%. Although 0.2% seems negligible, at the Google scale it causes the loss of millions of searches per day with durable effects on user satisfaction. For this reason, business-critical application developers should carefully tailor the requested performance, while a power management system should be designed to precisely track those performance requirements.

For what concerns power management with latency guarantees, here we provide a brief view of the main works in the field.

PowerNap [74] is an energy-conservation approach that eliminates expensive idle state with near-zero power idle states leveraging server usage traces. The authors showed that servers used for user-facing workloads typically have CPU usage below 30% or even below 10% in some cases. For this reason, the authors defined a scheme to sleep servers and activate them without significant degradation on the response time of the applications. This approach can no longer be used in the case of OLDI workloads [75], where coordinated full-system active low-power modes provide better results.

Rubik [59] is a fine-grain DVFS scheme for latency-critical workloads that adapts voltage and frequency depending on a statistical model of the application performance. Rubik works at a sub-millisecond granularity, however, latency requirements must be explicitly set to all the applications

being controlled. Moreover, according to [56], software DVFS is going to be replaced by hardware interfaces like RAPL [86].

PEGASUS [66] is a feedback-based controller that improves the energy proportionality of WSC. It adapts latency without violating SLAs and uses RAPL as an actuation system. Although PEGASUS provides interesting results with a single latency requirement for the whole cluster, it focuses on WSC, thus it does not support heterogeneity of workloads within the same server.

Copper [56] is a control scheme based on *Kalman* filtering that manages the performance of applications while reducing power consumption to achieve the requested performance. Performances are indicated as execution times and the adaptive controller approximates the non-linearities involved in this kind of power management technique. Copper is simple and reliable, but it needs to instrument the applications with the *Heartbeats* library [53]. Our approach, instead, does not need to instrument the applications' code.

To better analyze the current state of the art, here we provide the main works on power management with general performance guarantees in the context of microservices and application containers. Seer [48] is a performance debugging tool that employs deep learning to analyze the performance traces of all the microservices running in a cluster to provide predictions about SLA violations. The approach of this paper is extremely interesting for our case, as predicting the microservices' performance can improve the quality of the control activity and can further reduce the power consumption.

The work of *Piraghaj et al.* [84] is a framework for energy-efficient container consolidation in cloud data-centers. Containers are executed inside VMs and the goal is to minimize the overall power consumption guaranteeing a SLA. This work provides interesting insights using CPU usage as SLA, which unfortunately does not fully describe the applications' performance from the end-user perspective.

Dockercap [9] is an ODA control loop to manage power consumption guaranteeing a defined level of CPU usage within Docker containers. Performances are monitored parsing the output of various monitoring tools like *perf* [37], while control and actuation are limited to the single node level. Our approach scales to systems composed of several machines, supporting latency instead of CPU usage as the main performance metric.

The work of *Townend et al.* [91] studies the integration of Kubernetes clusters in the more complex data-center system, building a scheduler for the container orchestrator that takes into account both software and hard-

ware models. Their approach reduces power from 10% to 20%. Our approach does not add components in the Kubernetes cluster, instead, it provides power capping after scheduling decisions.

Power Shepherd [60] leverages both RAPL and a CPU quota mechanism to actuate in a distributed fashion over a cluster of compute nodes. It differentiates between ideal, usable and unusable states for the application performance and moves the cluster power budget across the various servers to boost performance when needed. The authors state that Power Shepherd does not provide good results in the case of underutilized servers. Our power capping approach, instead, is mainly designed to work on medium to low server utilization, while it releases the power cap in case of high utilization.

5.3 System design

PRESTO is a power capping orchestrator whose goal is to reduce the power consumption of each server belonging to a Kubernetes cluster while maintaining a predefined SLA for the running cloud-native application. We decided to focus on microservices average latency as the target performance metric as it allows to express easily the performance the system should be able to achieve at any given time both from the developer perspective as well as from the end-user perspective. To have full control of the performance of each component, previous solutions required to set a latency goal for each of them. However, most of the microservice-based applications react to external inputs and show connected components that depend on each other operations [47]. For this reason, PRESTO allows setting just one latency requirement that will be translated dynamically and at run-time to latency requirements for all the components. The automatic definition of latency goals for each application component allows optimizing the power usage of each server when the application is underutilizing its resources, meaning that we will be able to slow down the application until its latency will be near the target SLA. To support this capability, we designed PRESTO following three main principles:

1. **Autonomicity**, as the proposed system should be able to operate with the least external intervention possible;

2. **Performance**, as microservices support business-critical applications and, as such, their functionality should not be compromised by power-saving systems;

3. **Transparency**, as workload components may vary frequently, and, as

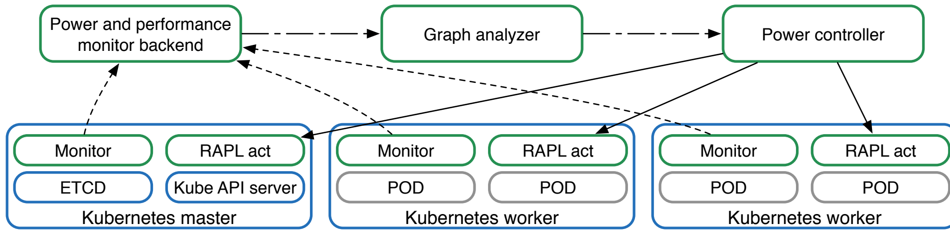


Figure 5.1: *PRESTO ODA control loop: we observe latency and power consumption of each Kubernetes pod in each server, centralizing metrics in a remote backend. Metrics are then passed to the Graph analyzer, which, starting from a latency requirement, defines the service times of each container. Then, the Power controller defines the power budget allocated to each server, which is enforced by the RAPL actuators.*

such, the use of PRESTO should not be limited to just a small set of instrumented workloads.

Figure 5.1 shows the main components of PRESTO, highlighted in green. From now on we will refer to microservices as *pods*, as they are the basic unit of work of Kubernetes and are defined as a collection of Docker containers providing a single functionality. To support *autonomicity*, we designed the proposed system leveraging the concept of ODA control loop. For each second, PRESTO *observes* the behavior of each pod running in each physical server without instrumenting the user code, supporting *transparency*. The *Monitor* component collects metrics about resource usage (e.g. CPU), low-level performance metrics (PMC like cycles, IR, cache references, and cache misses), power consumption (for each container, pod, and physical host), network latency (e.g. average latency and from 50th to 99th percentile latency), and network bandwidth. All these metrics are sent to the *Power and performance monitor backend*, which groups them depending on the pod, the service, and the namespace. Metrics are then sent to the *Graph analyzer*, which is the first step in the *Decide* phase. The Graph analyzer builds a graph of the pods starting from their network connections and, given a latency requirement for the cluster entry-point, derives the network times and the service times each pod should provide at the next time interval to guarantee the *performance* principle. The *Power controller*, which is the second step in the *Decide* phase, takes the service times defined at the previous step and computes the power budget to be allocated to each physical server. Finally, the *RAPL act* components *actuate* for each host in the cluster the allocated power budget.

In the next sections, we will detail each step, starting from the monitor-

ing tool in Section 5.3.1. Section 5.3.2 details how we define the service and network times starting from the single latency requirement. Then, Section 5.3.3 describes the heuristic control we perform to define the power budget, which is actuated on the system with the RAPL act component described in Section 5.3.4.

5.3.1 Power and performance monitoring

To observe the behavior of the cluster we resorted to the monitoring tool described in Chapter 3, for which here we provide the aspects that are relevant for this Chapter. The monitoring tool computes the fine-grain attribution of power consumption for each Docker container and Kubernetes pod. It operates by measuring PMC values with eBPF [73], aggregating such values at the kernel level and exporting one sample to a user-space agent. The sample represents how the CPU was used in the last second by all the threads running in the system. RAPL core power values are then attributed proportionally to each thread using the weighted cycles measurement. This allows accounting for the concurrent execution of threads in HTs, which show a different power profile w.r.t. isolated execution on different physical cores [98]. Metrics are then aggregated by container within the user-space agent and then by pod, service, and namespace inside a remote backend.

To enforce a latency requirement we first need to measure the network performance of each pod. Given that our goal is to let the user express a single requirement for the whole application, we also need to study the interactions between pods. Thus, we need to track the network performance across all the connections performed by the pods with the external world and between each other. To solve this issue, we leveraged eBPF to track all the network operations performed by the pods (both IPv4 and IPv6). In particular, we instrumented the network stack leveraging the following Linux *kprobes*: *tcp_set_state*, *tcp_send_msg*, *tcp_recv_msg*, and *tcp_cleanup_rbuf*. We use *tcp_set_state* to detect changes in the connection status. We use *tcp_send_msg* and *tcp_recv_msg* kprobes to track incoming and outgoing TCP communications, while *tcp_cleanup_rbuf* is used to gather the data size read by a previous *tcp_recv_msg* kprobe invocation.

All the kprobes we used provide just raw data about the network connection, its endpoints and the data size exchanged over TCP. To compute the network latency we resorted to the concept of *network transaction*. A network transaction is an exchange of data between a client and a server within a TCP connection, where the client starts by sending one or more messages and the server replies to them. A new transaction starts (and the

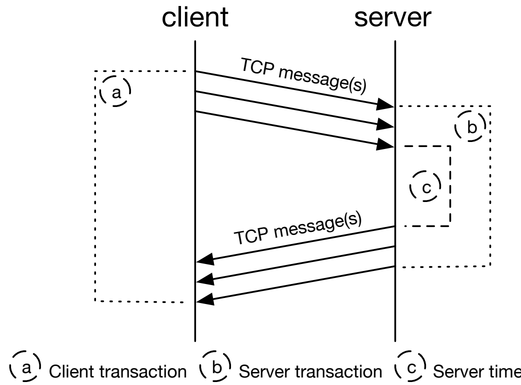


Figure 5.2: *Example of a network transactions over a TCP connection.*

previous one ends) when the client starts again to send messages after the reply from the server. The latency of a request is the time elapsed from the first to the last message in the transaction. Figure 5.2 shows an example of a network transaction, where we have highlighted the execution time of such transaction from the client perspective and the server perspective. The time elapsed between the last client message and the first server message is the time required to create the response and we denoted it as *server time* (which is different from the service time if, to build the response, the server has to communicate with other application components).

Following the network transaction mechanism, we collect metrics for each network connection, where each connection is characterized by its *source IP*, its *source port*, its *destination IP*, and its *destination port*. When the connection is a plain HTTP connection, we collect the HTTP endpoint in place of the client port. We differentiate between client transaction and server transaction to collect different transaction times, as shown in Figure 5.2. Within the eBPF code, for each connection, we collect the bandwidth, the average latency and a sample of all the network latencies (currently a reservoir sampling of 240 items per second, configurable). On the user-space side of the monitoring agent, we collect those metrics and we compute the 50th, 75th, 90th, and 99th percentile latency and we send all the data to the remote backend. The connection data, as well as the performance data and the power consumption data, are used to build a graph of all the network interactions between pods and all the network interactions between the external world and the pods.

5.3.2 Graph-based service time estimation

The graph generated in the previous step is analyzed every 2 seconds in the *graph analyzer* component, which is deployed in the remote backend. If we focus on synchronous applicative protocols (e.g. HTTP, Remote Procedure Call (RPC) over TCP), for each pod we may find two different kind of queues: the TCP protocol queues and the client application queues. On the one hand, the TCP protocol queues provide too low-level information to be effectively used in the control of the average latency from the microservice perspective. On the other hand, client application queues are not available for the clients out of the cluster and in general cannot be accessed due to the *transparency* principle.

Given these limitations, we defined the average latency of a given pod i according to Equation (5.1). The latency of a pod is the sum of its average service time and the average time necessary to obtain a response from the pods connected to pod i (denoted by the set $dPOD_i$ that contains the downstream pods of pod i). For each pod j connected to pod i, the time necessary to obtain a response from j starting from i is the sum of the average latency of j and the network time needed to reach it. Given that not all the downstream pods are always involved in the computation of the response of pod i, we weight each contribution by a factor denoted as α_j.

$$L_i(t) = S_i(t) + \sum_{j \in dPOD_i} \alpha_j(t) \cdot \big(L_j(t) + N_{ij}(t)\big) \tag{5.1}$$

Figure 5.3 helps us to understand how to design the α_j factor. First of all, we decided to have α_j as a function of time t, as the usage of the downstream pods can change depending on the loads and on the different requests coming to pod i. As we can see from Figure 5.3, pod i has 3 downstream pods ($j1$, $j2$, and $j3$), each one with a client connection and a server connection. Both connections are denoted by an average arrival rate and an average latency. For a given pair of connected pods (i, j), the network time $N_{ij}(t)$ can be computed as the difference between the average client latency and the average server latency. This is because on the client-side we measure from the first client request to the last server response for each transaction, while on the server-side we measure just the *server time* without the network overhead. At this point, two main cases may arise (other cases are a combination of the two):

1. for each request coming to pod i, all the downstream pods are invoked to build the response of pod i,

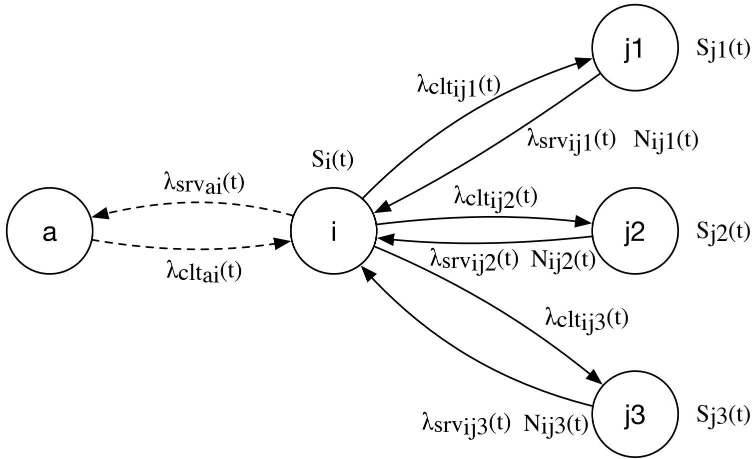


Figure 5.3: *Graph of microservices where pod i has a fan-out of 3 pods, each one with its own arrival rates, service times and network times.*

2. for each request coming to pod i, just one downstream pod is invoked to build the response of pod i.

In the first case, the arrival rate of each downstream pod will match the arrival rate of pod i. For this reason, to weight correctly the impact of each downstream pod, the sum of all the $\alpha_j(t)$ for pod i should be equal to $\frac{1}{|dPOD_i|}$. On the contrary, in the second case, the sum of the arrival rates of the downstream pods will match the arrival rate of pod i. Thus, the sum of all the $\alpha_j(t)$ for pod i should be equal to 1.

To enforce this behavior over the $\alpha_j(t)$ factor, we need to use the number of requests that are handled by pod i and by the downstream pods. In particular, we can denote $\lambda_{srv_i}(t)$ and $\lambda_{srv_j}(t)$ the sum of all the server requests handled respectively by pod i and pod j. We can do the same for the client requests, where, for instance, $\lambda_{clt_i}(t)$ is the sum of all the requests generated by pod i as a client to all the downstream pods. Starting from these values, we define $\alpha_j(t)$ fo a given pod j as in Equation 5.2, where $\alpha_j(t)$ is the product of two different contributions. The first contribution is the ratio between the requests served by pod i to other pods ($\lambda_{srv_i}(t)$) and the client requests generated by pod i to the downstream pods. This contribution weights the impact of pod i w.r.t. the rest of the pods in the application. The second contribution is the ratio between the number of requests served by pod j to other pods ($\lambda_{srv_j}(t)$) and the client requests generated by pod i to the downstream pods. This second contribution weights the impact of the downstream pod j w.r.t. the requests performed by pod i to all the

downstream pods.

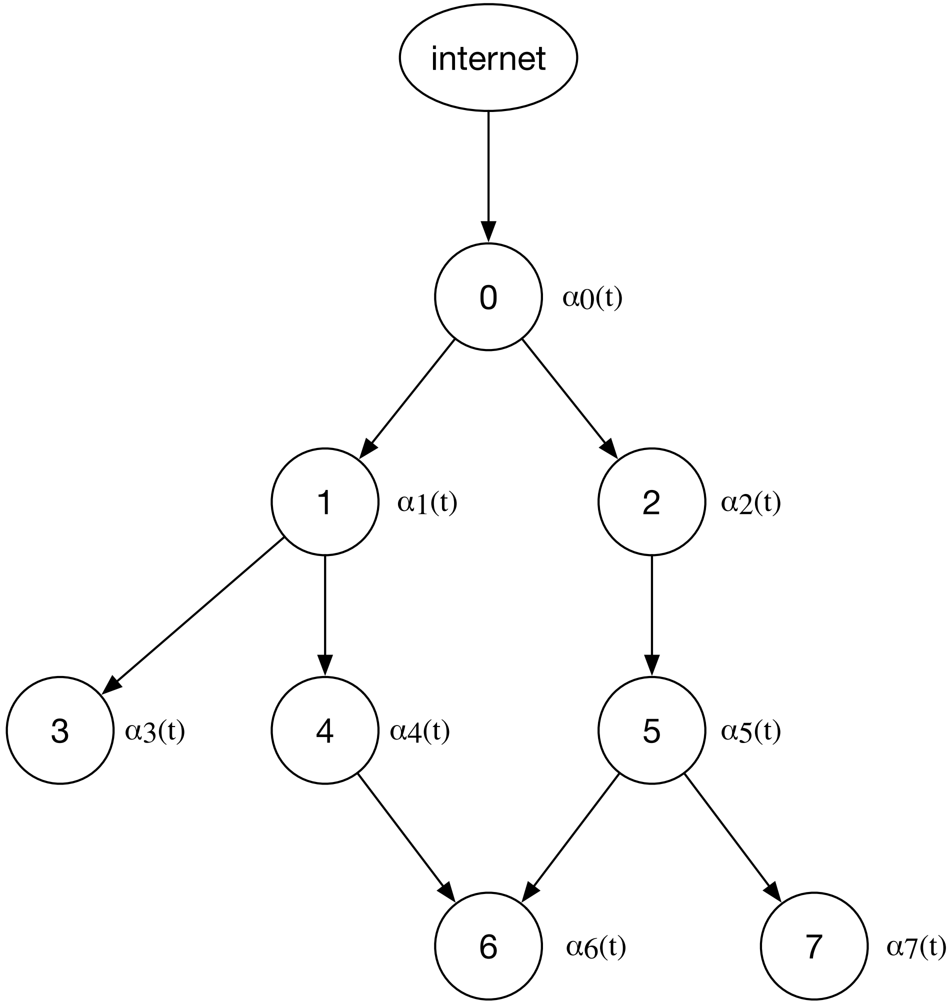
If the pods are behaving as in case 1, the first contribution will weight $\frac{1}{|dPOD_i|}$, while the second contribution will weight $\frac{1}{|dPOD_i|}$ as well. If we sum up the $\alpha_j(t)$ we will obtain $\frac{1}{|dPOD_i|^2}$ $|dPOD_i|$ times, which will lead to the expected $\frac{1}{|dPOD_i|}$. If, instead, we are in case 2, the first contribution will fall down to 1, while the second contribution will lead to $\frac{1}{|dPOD_i|}$ when the $\alpha_j(t)$ factors are summed up for pod i.

$$\alpha_j(t) = \frac{\lambda_{srv_i}(t)}{\lambda_{clt_i}(t)} \cdot \frac{\lambda_{srv_j}(t)}{\lambda_{clt_i}(t)} = \frac{\lambda_{srv_i}(t) \cdot \lambda_{srv_j}(t)}{\lambda_{clt_i}(t)^2} \tag{5.2}$$

Once we defined the effects of $\alpha_j(t)$, we can estimate the latency of the whole microservice-based application to then try to enforce the new latency target. To do so, we need to solve Equation (5.1) for a given entry-point. Figure 5.4 shows an example of a microservice-based application graph where vertices and edges are pods and client connections respectively. The *internet* vertex represents all the client endpoints outside of the cluster. The first pod in the path that starts from the internet vertex is our entry-point, for which we define the average latency of the microservice application. We can then explore the graph with a breadth-first search. Within this exploration, we compute a new factor that we denote $\beta_i(t)$ that keeps into account the path each request should follow from the entry-point, where i is a pod in the application. In the case of multiple visits of the same pod i, we select the highest $\beta_i(t)$ among the visits to enforce a harder requirement on the final average latency. After the exploration of the graph, we can estimate the latency at the entry-point as in Equation (5.3). Having computed $\beta_i(t)$ factors, the latency at the entry-point $L_{out}(t)$ is just the sum of two contributions. On the one hand, we have the sum of the service times of all the pods i multiplied by $\beta_i(t)$ (this first contribution can be summarized as the total service time of the application $S_{out}(t)$). On the other hand, we have the sum of the network times $N_{ij}(t)$ between all pod pairs (i, j) that are connected by an edge of the graph, multiplied by the highest available $\alpha_j(t)$ (this second contribution can be summarized as the total network time of the application $N_{out}(t)$).

$$L_{out}(t) = \sum_{i \in POD} \Big(\beta_i(t) \cdot S_i(t)\Big) + \sum_{(i,j) \in C} \Big(\alpha_j(t) \cdot N_{ij}(t)\Big) \tag{5.3}$$

Once we have the estimation of the entry-point latency, we can compute the requirements each pod should satisfy to meet a given SLA. Unfortu-



$\beta_0(t) = \alpha_0(t) = 1$

$\beta_1(t) = \alpha_1(t)$

$\beta_2(t) = \alpha_2(t)$

$\beta_3(t) = \beta_1(t) \cdot \alpha_3(t) = \alpha_1(t) \cdot \alpha_3(t)$

$\beta_4(t) = \beta_1(t) \cdot \alpha_4(t) = \alpha_1(t) \cdot \alpha_4(t)$

$\beta_5(t) = \beta_2(t) \cdot \alpha_5(t) = \alpha_2(t) \cdot \alpha_5(t)$

$\beta_6(t) = \max(\beta_4(t) \cdot \alpha_6(t), \beta_5(t) \cdot \alpha_6(t))$

$\beta_7(t) = \beta_5(t) \cdot \alpha_7(t) = \alpha_2(t) \cdot \alpha_5(t) \cdot \alpha_7(t)$

Figure 5.4: *Abstract representation of a microservice application. Vertices are pods, edges are connections. Each pod has a α coefficient associated according to Equation* (5.2)*. β coefficients are computed following a breadth-first search of the graph starting from the internet vertex, which is outside of the cluster.*

nately, we are not able to control the network times, thus, we will focus on the service times of each pod to enforce the latency requirement. Within this context, if we define the target latency SLA as $\overline{L_{out}}$, the service time $S_i(t+1)$ for a given pod i can be computed according to Equation (5.4). This equation simply scales the last value of the service time $S_i(t)$ by the ratio between the expected service time and the current service time $S_{out}(t)$. The expected service time is computed as the difference between the latency target $\overline{L_{out}}$ and the current network latency $N_{out}(t)$.

$$S_i(t+1) = S_i(t) \cdot \frac{\overline{L_{out}} - N_{out}(t)}{S_{out}(t)} \tag{5.4}$$

5.3.3 Latency-aware reactive control

Once we computed the service time each pod should guarantee at the next time interval, we can then try to enforce it through the *power controller*. We designed a heuristic controller, placed inside the cluster as a normal pod, that reacts to changes in the arrival rate of the application and to changes in the enforced service times. We decided to resort to the utilization based *Little's law* to translate the enforced service time into a requested utilization. To do so, for each physical server k in the Kubernetes cluster, for each second, we select the pod that has maximum utilization, as shown in Equation (5.5). We target the pod with maximum utilization because, otherwise, if we increase too much the power cap, such pod will be the first to saturate reducing the performance in an uncontrolled way.

$$U_c(k,t) = \max_{i \in POD}(U_i(k,t)) \tag{5.5}$$

Starting from the candidate utilization $U_c(k,t)$ we define the control error over the utilization as in Equation (5.6). We define this error as the difference between the current candidate utilization $U_c(k,t)$ and the target utilization that this pod should achieve, according to the *Little's law*.

$$e_u(k,t+1) = U_c(k,t) - \lambda_c(t) \cdot S_c(t+1) \tag{5.6}$$

We then define the latency error of the overall latency application according to Equation (5.7). The latency error is the absolute value of the percentage error between the target average latency $\overline{L_{out}}$ and the current average latency $L_{out}(t)$.

$$e_l(t) = \left| \frac{\overline{L_{out}} - L_{out}(t)}{\overline{L_{out}}} \right| \tag{5.7}$$

Starting from the utilization error and the latency error we define the candidate power budget of server k as in Equation (5.8). The candidate power budget $P_c(k, t+1)$ is the sum of the current core power consumption of the server $P(k,t)$ and the actionable power $P_{act}(k,t)$ multiplied by the utilization error $e_u(k, t+1)$ and the latency error $e_l(t)$. The actionable power $P_{act}(k,t)$ is the power consumption generated by the pods that can be controlled by PRESTO.

$$P_c(k, t+1) = P(k,t) + e_u(k, t+1) \cdot e_l(t) \cdot P_{act}(k,t) \tag{5.8}$$

Finally, we define the final power budget for server k as in Equation (5.9). The power budget $P(k, t+1)$ is equal to $P_c(k, t+1)$ if the current latency of the microservice-based application $L_{out}(t)$ is less than $2 \cdot \overline{L_{out}}$, otherwise the maximum power budget $P_{max}(k)$ for the server is enforced. This condition is necessary to let the system to move away from the equilibrium in case of violation of the SLA requirement.

$$P(k, t+1) = \begin{cases} P_{max}(k) & L_{out}(t) > 2 \cdot \overline{L_{out}} \\ P_c(k, t+1) & otherwise \end{cases} \tag{5.9}$$

5.3.4 RAPL-based power allocation

Once the *power controller* has computed the power budgets for all the servers in the Kubernetes cluster, it makes this data available through a REST interface. Each *RAPL act* pod is executed as a Kubernetes *DaemonSet* on each host. To enforce the power cap, each pod is executed in privileged mode and we mount the */dev/cpu* folder to get access to the MSR files. The *RAPL act* pod retrieves the data coming from the controller and transforms the power cap expressed in Watt into power units by reading the power unit MSR and applying the relative formula described in the Intel manual [51]. Then, the power budget is enforced on the *package* domain by writing on the proper MSR.

5.4 Evaluation

Within this Section, we evaluate the performance of PRESTO w.r.t. the ability to keep the average latency of the applications near the latency target and wr.t. the power savings that this activity can generate. Section 5.4.1 will detail the experimental setup we built to validate the proposed methodology, while Section 5.4.2 will describe the results obtained w.r.t. power savings and performance of the applications.

Table 5.1: *Experimental setup for the social-network benchmark and the media-microsvc benchmark. For each workload we show the number of pods involved and the requests rate for low, medium, and high configuration.*

benchmark	workload	# pods	low $\lambda(t)$	mid $\lambda(t)$	high $\lambda(t)$	duration
social-network	compose post	21	300 req/s	400 req/s	500 req/s	300 s
social-network	read home timeline	5	5000 req/s	6000 req/s	8000 req/s	300 s
media-microsvc	compose review	27	250 req/s	300 req/s	350 req/s	180 s

5.4.1 Experimental setup

The experimental campaign was conducted on a small cluster composed of two Dell PowerEdge r720xd equipped with 2x Intel Xeon E5-2680 Ivy Bridge with 10 cores each (20 HT) clocked at 2.80GHz and with 380GB of RAM. The host OS is an Ubuntu Linux 16.04 with kernel 4.15, eBPF, Docker 18.06.2 community edition, and Kubernetes v1.17.4. We tested the proposed methodology with two benchmarks from the *DeatStarBench* benchmark suite [47], an open-source[2] suite of microservice-based benchmarks. We chose the *social-network* and the *media-microsvc* benchmark because, at the time of writing, those benchmarks have a supported Kubernetes distribution. The social-network benchmark represents a small social-media application with microservices deputed to user management, user timeline management, home timeline management, post composition, media storage, social graph management, search, and URL shortening. When needed, microservices store data inside MongoDB[3] servers, Redis[4] servers, and Memcached[5] in-memory stores. The media-microsvc benchmark, instead, implements a media reviewing, renting and streaming platform and is composed of identification services (users and movies), a review composition service, a page composition service, services for users and reviews consultation, and a video streaming service. Also, the media-microsvc benchmark leverages MongoDB, Redis, and Memcached data stores. Both benchmarks combine an HTTP front-end with a Thrift-based RPC system between pods and TCP connections to data stores.

Table 5.1 shows how we set-up the tests for the experimental campaign. The social-network benchmark provides two different workloads: on the one hand, we have scripts for the post composition, while, on the other hand, we can read the home timeline of each user in the system. The media-microsvc benchmark, instead, has just the compose review workload. For

[2]https://github.com/delimitrou/DeathStarBench
[3]https://www.mongodb.com
[4]https://redis.io
[5]https://memcached.org

each workload, we set 3 different arrival rates, identified as low, mid, and high. For the compose post we set 300 req/s, 400 req/s, and 500 req/s for low, mid, and high respectively. For the read home timeline workload, we set 5000 req/s, 6000 req/s, and 8000 req/s for low, mid, and high respectively. Finally, for the compose review workload we set 250 req/s, 300 req/s, and 350 req/s for low, mid, and high respectively. The workloads for the social-network benchmark were run for 5 minutes, while the workload of the media-microsvc benchmark was run for 3 minutes. For what concerns the compose post and the compose review workloads, unfortunately, we were not able to push the benchmarks over 10 and 4 parallel connections each because of a race condition in the UniqueID microservice.

For each workload and each arrival rate, we perform 30 runs of the experiment both for the workload equipped with PRESTO, as well as the unconstrained workload (denoted as *alone*). Each run is executed thanks to a *wrk2*[6] load generator that is launched from a third machine identical to the previous two. For each run, we collect the average latency, the power consumption, the energy usage throughout the experiment, and the percentile latencies of the workload. The experimental results are presented in Section 5.4.2.

5.4.2 Experimental results

Table 5.2 shows a synthetic view of the results we obtained running the experiments described in Section 5.4.1. If we consider the compose post workload, we can see that the unconstrained run generated an average latency of 10.08ms, 9.27ms, and 8.83ms for the low, mid, and high arrival rates respectively. Given the average latency showed by the benchmark, we decided to bring the latency target to 13ms, which represents an increase of $\simeq$40% w.r.t. the normal behavior of the workload. Table 5.2 shows that the proposed methodology was able to increase the average latency, with a RE of -16.44%, -13.21%, and 14.69% for the low, mid, and high arrival rates respectively. Although the RE is not negligible, the difference between the latency target and the achieved latency is bounded to $\simeq$2.6ms. Within this context, PRESTO was able to reduce power consumption by 7.01%, 35.15%, and 23.99% for the low, mid, and high arrival rates.

If we look at the read home timeline workload, we can see that the unconstrained execution has an average latency of $\simeq$2ms. In this case, we decided to bring the target latency to 8ms, which is a 5x increase on average latency. Table 5.2 shows that PRESTO was able to increase the average

[6]https://github.com/giltene/wrk2

Table 5.2: *Experimental results for the 3 workloads with 3 different arrival rates for each workload. Table reports the average latency when the benchmark is not constrained, the latency target, the average latency achieved by PRESTO, the latency difference, the latency relative error, and the power savings w.r.t. the not constrained execution.*

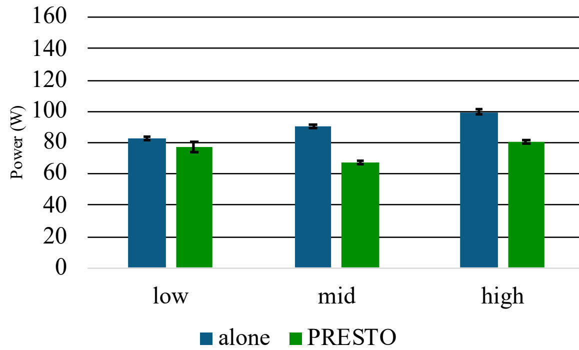
	configuration	$\lambda(t)$	alone avg latency	latency target	avg latency	Δ latency	relative error	power saving
compose post	low	300 req/s	10.08 ms	13 ms	15.66 ms	-2.66 ms	-16.44%	7.01%
	mid	400 req/s	9.27 ms	13 ms	15.25 ms	-2.25 ms	-13.21%	35.15%
	high	500 req/s	8.83 ms	13 ms	11.43 ms	1.57 ms	14.69%	23.99%
read home timeline	low	5000 req/s	1.92 ms	8 ms	9.04 ms	-1.04 ms	-11.15%	46.26%
	mid	6000 req/s	1.89 ms	8 ms	10.22 ms	-2.22 ms	-21.65%	42.52%
	high	8000 req/s	2.10 ms	8 ms	10.62 ms	-2.62 ms	-24.52%	42.69%
compose review	low	250 req/s	9.72 ms	15 ms	14.54 ms	0.46 ms	5.94%	43,93%
	mid	300 req/s	10.24 ms	15 ms	15.51 ms	-0.51 ms	-2.70%	47.53%
	high	350 req/s	10.32 ms	15 ms	14.83 ms	0.13 ms	2.14%	45.09%
Averages (on absolute values)						1.50 ms	12.49%	37.13%

latency also in this case, with a RE of -11.15%, -21.65%, and -24.52% for the low, mid, and high arrival rates respectively. Again, if we look at the difference between the latency target and the achieved latency, we can see that it is bounded to $\simeq$2.6ms also in this case. If we consider instead the power savings, we can see that PRESTO reduces the power consumption of 46.26%, 42.52%, and 42.69% for the low, mid, and high arrival rates.

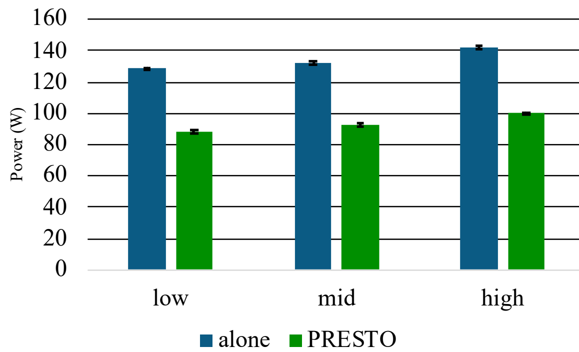
Finally, Table 5.2 shows also the results for the compose review workload. In this case, the benchmark exposes an average latency of $\simeq$10ms and we enforce a less strict latency target: 15ms. Within this case, the proposed methodology achieves a RE that is below 6% in all cases, with power savings that goes from 43.93% for the low arrival rate, to 47.53% for mid and 45.09% for high.

Although we obtained good results on the power savings side, the RE exposed by PRESTO is not negligible in some cases. This is mainly due to the actuation mechanism. Given that we leverage RAPL, the power cap enforced is valid for all the cores in the system. This is taken into account by the controller that defines the power cap depending on the pod with the highest utilization within each physical server. However, this poses some challenges: first of all, the pod with the highest utilization within a server often works near its saturation, thus, small and frequent changes in the power cap lead to high oscillations of the performance of the pod. To solve this issue, we decided to change the power cap of the server only if the new power cap has a difference of at least 1W. This, of course, results in the possibility to reach the equilibrium at a value that is slightly higher or smaller than the target latency. Read home timeline and compose post exposes this behavior, although it is bounded to $\simeq$2.6ms, while compose review reaches the equilibrium with a smaller error.

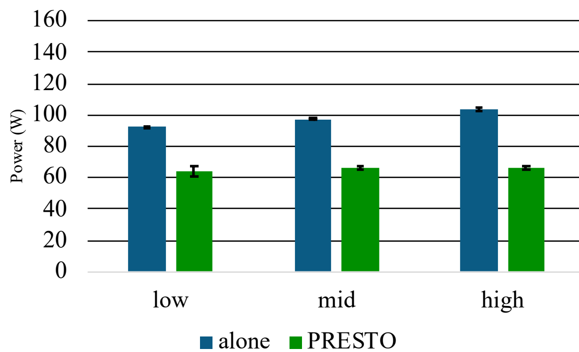
To further analyze the results we obtained, Figure 5.5 shows the power consumption of the three workloads executed with and without PRESTO.



(a) Compose post power consumption with arrival rate of 300 req/s, 400 req/s, and 500 req/s.

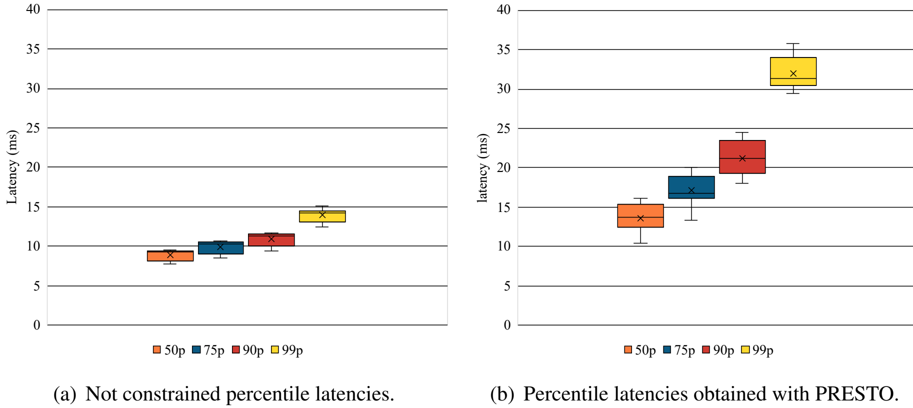


(b) Read home timeline power consumption with arrival rate of 5000 req/s, 6000 req/s, and 8000 req/s.



(c) Compose review power consumption with arrival rate of 250 req/s, 300 req/s, and 350 req/s.

Figure 5.5: *Comparison of power consumption of the 3 workloads with different arrival rates. Error bars represent 95% confidence interval. Lower is better.*



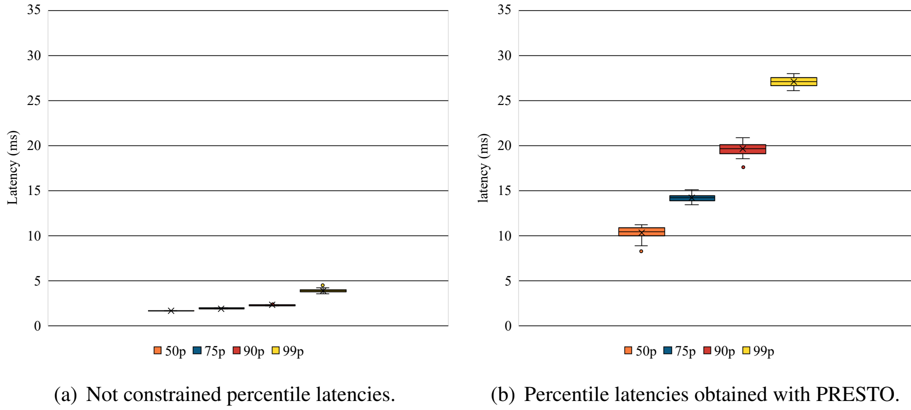
(a) Not constrained percentile latencies.

(b) Percentile latencies obtained with PRESTO.

Figure 5.6: *Box plots of the percentile latency (50th, 75yh, 90th, and 99th) of the compose post workload with 400 req/s across 30 runs.*

Figure 5.5(a) shows the power consumption of the compose post workload for different arrival rates. As we can see, the difference between the power consumption increased from low to mid, but then slightly reduces from mid to high. This is because in the low case PRESTO hits the minimum of the RAPL actuator for one server, which is set to 30W. Then, in the high case, power savings decrease because the controller sets a slightly lower latency w.r.t. the target. In Figure 5.5(b) the power consumption for the read home timeline workload is reported. This workload is the one that has the highest power consumption among the three analyzed workloads. As we can see, the power consumption increases for both the unconstrained execution as well as for PRESTO when we increase the arrival rate. Power savings for this case remain similar among the different arrival rates. Finally, Figure 5.5(c) shows the power consumption of the compose review workload for the different arrival rates that we described in Table 5.1. Here we can see a slight increase in power consumption for both the unconstrained execution as well as the one managed by PRESTO. This is because the increase in the arrival rate is less steep w.r.t. the other cases. Again, the power savings remain fairly similar across the different runs, indicating a system that is stable and that can control the performance accurately within this case.

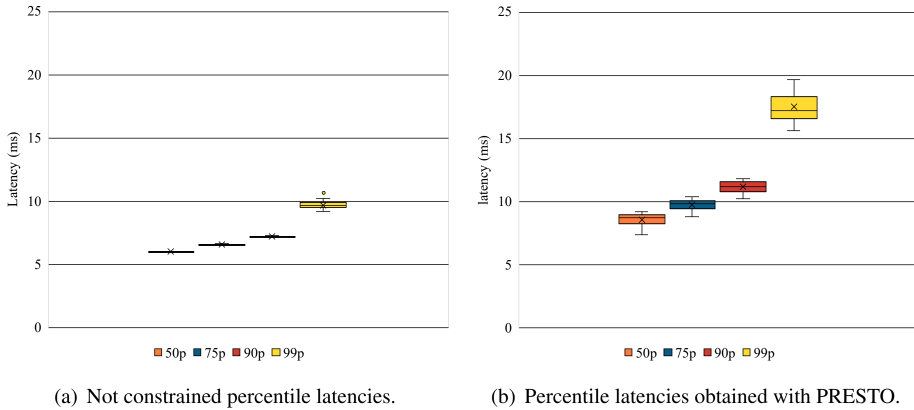
To better analyze the effects of PRESTO on performance, we decided to collect data also on the latency percentile provided by the workloads during the experiments. In particular, we leverage the *uncorrected latency* measurements of *wrk2*. Here we report only the box plots for all the workloads we analyzed for the mid arrival rate case, as similar considerations



(a) Not constrained percentile latencies. (b) Percentile latencies obtained with PRESTO.

Figure 5.7: *Box plots of the percentile latency (50th, 75yh, 90th, and 99th) of the read home timeline workload with 6000 req/s across 30 runs.*

can be made for the other configurations. Figure 5.6 shows the 50th, 75th, 90th, and 99th percentile latency for the compose post workload for the unconstrained execution (Figure 5.6(a)) and the one managed by PRESTO (Figure 5.6(b)). The power capping imposed by PRESTO increased not only the average latency but also all the latency percentiles. In particular, we can see that the average and the median of the 50th percentile are in line with the latency target, while the other latency metrics increase with similar paces w.r.t. the unconstrained ones. As expected, PRESTO slightly increases the variability of the latency results due to the reactive control activity. Similar considerations can be made for the latency percentiles of the read home timeline workload (shown in Figure 5.7). Within this case, however, the 50th percentile is always higher than the latency target (except for one outlier). This is due to the lower latency of the workload w.r.t. the compose post one and the 5x latency increase required by the latency target that poses more challenges in the control activity. The variability induced by PRESTO is lower w.r.t. the compose post case, because the workload involves fewer microservices, as indicated in Table 5.1. Finally, Figure 5.8 shows the results for the compose review workload. As we can see, the 50th percentile and the 75th percentile latency metrics are below the latency target. This indicates that the average latency is heavily affected by the tail latency. Within this case, PRESTO correctly increases the average latency, modifying the latency percentiles accordingly without abrupt changes in the application behavior.



(a) Not constrained percentile latencies.

(b) Percentile latencies obtained with PRESTO.

Figure 5.8: *Box plots of the percentile latency (50th, 75yh, 90th, and 99th) of the compose review workload with 300 req/s across 30 runs.*

5.5 Conclusion and future work

Within this chapter, we presented PRESTO, a latency-aware power capping orchestrator specifically designed to manage the heterogeneity of workloads based on microservices. The proposed ODA control loop is based on the transparency, performance, and autonomicity principles, and provides a fine-grain monitoring tool combined with a graph-based analysis of the running workloads, a reactive and heuristic controller, and a fast actuation based on RAPL. The proposed approach reduces the power consumption of 37.13% w.r.t. an unconstrained execution, with a RE in the control activity that is 12.49% on average. The absolute error on average remains below 1.50ms. Future works will extend the proposed methodology by adding tuning knobs like CPU quota and pinning, by introducing prediction mechanisms within the controller and by improving the resiliency of the system in case of diurnal traffic patterns.

CHAPTER 6

Future directions: accelerating microservices to improve power efficiency[1]

6.1 Introduction

Compute intensive workloads may require performance that current CPUs are not able to provide and, for this reason, heterogeneous computing is becoming an interesting solution to continue to meet SLAs in the cloud. Within this context, in addition to GPU-enabled instances, cloud vendors offer FPGA-based compute instances (directly as in AWS F1 instances[2] or through managed services like Catapult [28] and Brainwave [31]). FPGA-based algorithms, if well designed, provide an optimal performance-per-Watt ratio w.r.t. current CPUs. For this reason, FPGAs are good candidate architectures to continue to improve the energy efficiency of cloud data-centers and infrastructures. Cloud workloads such as web search [85], image processing [7], database operations [82], neural network inference [96],

[1]The work presented in this chapter was published in [10], for which Rolando Brondolin developed part of the methodology, part of the experimental evaluation, and the whole paper writing. ©2020 IEEE. Reprinted, with permission, from: Marco Bacis, Rolando Brondolin, and Marco D Santambrogio. Blastfunction: an fpga-as-a-service system for accelerated serverless computing. In Design, Automation & Test in Europe Conference & Exhibition (DATE), 2020, pages 852–857. IEEE, 2020.

[2]https://aws.amazon.com/ec2/instance-types/f1/

and many others can benefit from the use of FPGAs to timely react to the end-users' requests.

To exploit FPGAs at their best in the cloud, hardware accelerators should be designed to meet latency requirements while optimizing throughput [28]. Requests from the outside network can come at unpredictable rates and they usually cannot be batched, and for this reason minimizing latency becomes fundamental. Moreover, the unpredictability of the requests can lead to an underutilization of the FPGAs, thus reserve one FPGA for each service that needs it will result in a waste of resources. From a cloud provider perspective, a possible solution would be to share the FPGA across different tenants to improve its time utilization.

Within this context, the serverless computing paradigm can be leveraged to share an FPGA at a fine-grain level with other tenants to improve its time utilization. Serverless computing [58] is an architectural pattern for cloud applications where server management is delegated to the cloud provider. Each application functionality is deployed by the user as a function and scheduled, executed, scaled, and billed depending on the exact need of the moment. In this way, cloud-native applications can benefit from FPGAs to accelerate compute-intensive workloads such as neural network inference, web searches, and image processing in a seamless way. To support these goals, in this chapter we propose BlastFunction, a distributed and *transparent* FPGA sharing system for the acceleration of microservices and serverless applications in cloud environments. We decided to focus on a *time sharing* approach to maximize the devices' utilization (in terms of accelerator execution time) and optimize the use of devices from the cloud provider perspective. The contribution of BlastFunction are:

- the design and implementation of a *scalable* system enabling *multi-tenancy* for cloud FPGAs in containerized and serverless environment,

- the design and implementation of a *transparent* layer that allows integrating applications and hardware accelerators written in OpenCL without code rewriting.

The rest of this chapter is organized as follows: Section 6.2 describes the State of the Art in the field, Section 6.3 details the design and implementation of BlastFunction, Section 6.4 evaluated the proposed design, while Section 6.5 draws the conclusion of this chapter.

6.2 State of the Art

Here we present the analysis of the State of the Art of interest for this work, focusing on the integration of FPGAs in cloud environments. We classified the literature by *communication method*, *sharing mechanism* and *computational model*.

The first distinction among the analyzed works is the *communication method* used to access the shared or virtualized device. In particular, we classified works focusing on *PCIe-Passthrough*, *Paravirtualization*, *API Remoting* and *Direct Network Access*. *PCIe-Passthrough* is the lowest-level method available, as it works by directly connecting a single VM or container to the FPGA device. This communication level is used by the AWS F1 instances. With *Paravirtualization* the requesting application VM is connected to a host device driver which virtualizes the access to the resources. This mechanism is used by *pvFPGA* [93]. The *API Remoting* mechanisms is the most used in the analyzed state of the art [8,57,71,100], and it works by defining a custom API to remotely access the device. It allows multiple applications to control the shared device and to perform both space and time sharing. A special API Remoting technique is represented by the work in [81], as in this case the system exposes a microservice for each accelerator, and not a general API for the entire system. Finally, *Direct Network Access* is used by *Catapult* [29]. This method works by exposing the FPGA through its network interface, thus enabling a low-latency access.

The second classification of the works analyzed is based on the *sharing mechanism*, and, in particular, we distinguish between *space-sharing* and *time-sharing*. *Space-Sharing* [8,26,100] employs FPGA virtualization through the use of *Partial Dynamic Reconfiguration (PDR)* or *Overlays* to run multiple accelerators on the same FPGA, which are used by different applications. *Space-sharing* allows to use the entire resources on the device (in terms of logic blocks), but requires careful handling of the accelerators to minimize the reconfiguration time. *Time-Sharing* [29,57,71,81,93] works by *multiplexing* multiple requests from different applications on the same accelerator in the FPGA board. In this case, the challenge is to efficiently schedule the incoming requests to minimize latency on the application side and managing memory accesses to fit in the I/O bandwidth.

The last classification is related to the *computational model*: we distinguish between *batch* and *service* based nature, where the batch/service terms are related to how the FPGAs are seen by the system. In a *Batch System* [8, 26, 93, 100], the workloads (and the connection to the FPGA) are seen as limited in time. Therefore the scheduling and allocation of

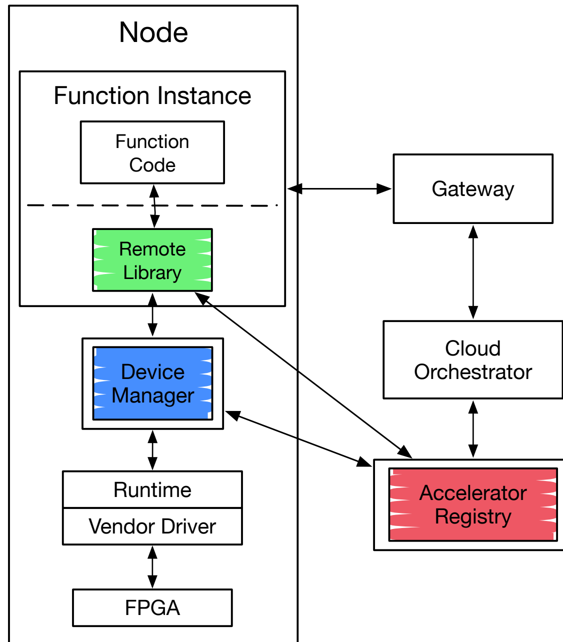


Figure 6.1: *High Level Overview of BlastFunction components (Remote Library in green, Device Manager in blue, and Accelerator Registry in red) and their connections.*

workloads are computed based on the *lifetime* of each job. In *Service-Based* [29, 57, 71, 81] systems, instead, the FPGAs are continuously working by receiving and processing requests from the system or the application. Works such as [81] directly expose the underlying FPGA accelerator as a standalone service.

Within the analyzed literature, BlastFunction leverages *API Remoting* techniques in a transparent way with OpenCL, focusing on *Time-Sharing* of FPGAs for *Service-Based* workloads like micro-services and serverless functions.

6.3 System design

BlastFunction is an FPGA sharing system for the acceleration of microservices and serverless applications in cloud environments. The system allows multiple applications to concurrently execute kernels on the same FPGA without changing the underlying host code. This is achieved leveraging OpenCL as the accelerators' runtime support system. The system allocates the available devices as requested by the applications leveraging runtime metrics collected by the system itself.

Figure 6.1 shows the main components of BlastFunction: the *Remote OpenCL Library*, the *Device Manager* and the *Accelerators Registry*. The *Remote OpenCL Library* allows the client application (either microservice or serverless function) to integrate with BlastFunction; it is a custom implementation of OpenCL that transparently abstracts the remote device access protocol and the communication with the *Accelerators Registry* and the *Device Managers*. Each *Device Manager* is connected to a FPGA in the system and provides the time-sharing mechanism. It exposes a service to remotely access the device functionalities, providing isolated access for multiple application containers. Finally, the *Accelerators Registry* is the central controller of the system. It tackles the allocation and reconfiguration of the available devices using runtime performance metrics. We use network protocols like *gRPC*[3] for control and *shared memory* for data transfers between the *Remote OpenCL Library* and the *Device Manager*. Data exchange between the *Accelerators Registry* and the other components is done through gRPC.

BlastFunction integrates with other external components to reach its goals. The *Cloud Orchestrator* (Kubernetes [25] in our case) is used by the *Accelerators Registry* to control the cluster resources and their allocation to the nodes. The *Gateway* is the serverless (OpenFaaS[4]) system's endpoint, which forwards the requests to the functions and handles autoscaling.

6.3.1 Remote OpenCL Library

The *Remote OpenCL Library* is a custom implementation of an OpenCL host library developed to integrate the applications with BlastFunction and to isolate them from the execution of the hardware accelerator. In particular, the *Remote OpenCL Library* implements most of the methods used to control an FPGA accelerator and can be linked both statically and dynamically to the application. The Remote OpenCL Library implements a central router component, which keeps the list of the available platforms. In particular, it gets the address of the selected *Device Manager* (or managers if multiple addresses are provided) and creates a connection to it through gRPC.

The system allows for both synchronous/blocking OpenCL calls to the remote runtime as well as asynchronous/non-blocking calls. Both the synchronous and asynchronous flows are designed with *asynchronous events*. An event in the system is composed by a set of subsequent asynchronous calls to the device manager service, a state machine to control the steps that

[3] https://grpc.io/
[4] https://www.openfaas.com/

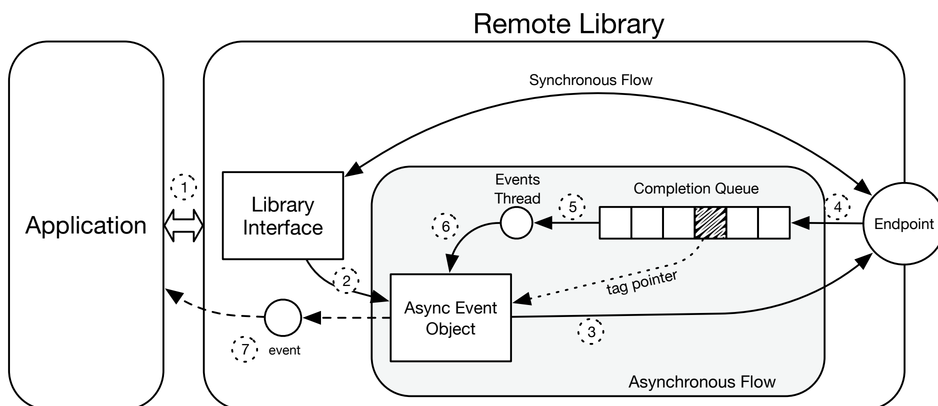


Figure 6.2: *OpenCL Remote Library Architecture, highlighting the steps performed in the asynchronous flow (dotted lines represent asynchronous responses).*

the event must follow and an OpenCL status for the event that is updated while the event is processed. In this way, the remote library supports event polling (e.g. clWaitForEvents, clGetEventInfo) like the standard OpenCL specification. We will explain the asynchronous flow by following Figure 6.2, which shows the main components of the Remote Library.

When the Remote library receives an asynchronous OpenCL call like a clEnqueueReadBuffer from the application (step 1), it creates an event (step 2) and performs a first asynchronous request through the network stack (step 3). The request encapsulates a *tag*, which is the pointer to the newly created event. When the device manager responds (step 4), the network runtime pushes the tag into the completion queue of the client. Then, the connection thread pulls the tag and retrieves the corresponding event (step 5). The thread then calls the event state machine and updates its state and the OpenCL event status (step 6). Finally, the application is notified when the event changes the OpenCL status (step 7). For instance, to perform the clEnqueueReadBuffer function, the event state machine contains 4 states. The INIT state sends the call metadata (buffer size, buffer id, offset); the FIRST step waits for the command to be enqueued by the manager; the BUFFER step actually sends the buffer data when the manager is available, and the COMPLETE step signals the call completion.

6.3.2 Device Manager

The *Device Manager* shown in Figure 6.3 controls and manages a single board inside BlastFunction. In particular, along with the Remote OpenCL Library, it is the basic block of the *sharing mechanism* presented in this

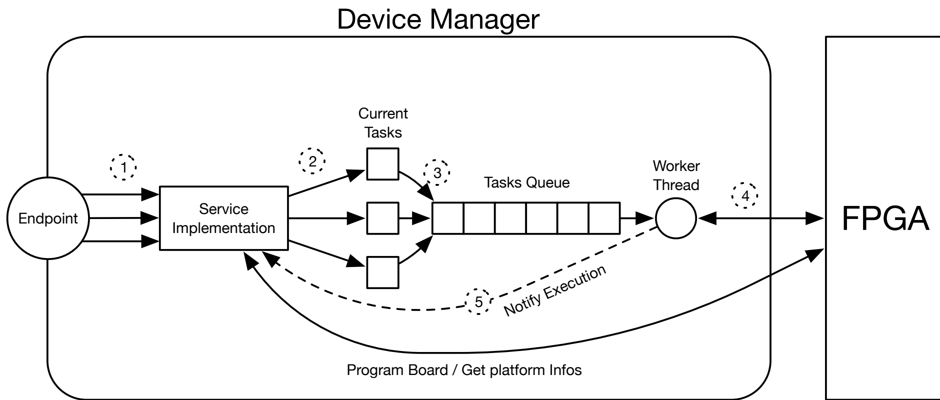


Figure 6.3: *Device Manager Architecture, with the command queue methods flow highlighted (dotted lines represent asynchronous responses).*

work, allowing many services to access the FPGA concurrently. The *Device Manager* separately controls each client's resources pool to enforce isolation between multiple clients.

There are two kind of methods exposed by the service: *context and information methods*, and *command-queue methods*. The *context and information methods* are executed synchronously as they do not involve execution on the FPGA. This group of requests includes the creation (on the client side) of kernels, the creation of platforms and contexts, the request of information related to the device and the buffer management requests. The *board reconfiguration request* represents the only exception in this group, as it blocks the execution of other operations to reprogram the board with the given bitstream. The other group of requests is represented by *command-queue methods*, which are composed by operations that must be executed in the order decided by the client application and might require to use the FPGA exclusively. An example is the kernel execution request, which might be interleaved with buffer reads and writes on one or multiple queues. For this kind of requests, if any operation is received or executed in the wrong order by the *Device Manager*, the results of the execution will change breaking the application consistency.

To ensure the in-order execution of *command-queue* methods, the *Device Manager* employs *multi-operation tasks*. We define a *task* as the atomic unit of execution of BlastFunction, composed of a sequence of operations that should execute atomically on the FPGA. Whenever the *Device Manager* receives a *command-queue call* (step 1 in Figure 6.3), the requested operations are added to the task related to that particular client

(step 2). After that, if the client sends a flush command (either by calling a blocking method or clFinish/Flush/EnqueueBarrier), the current task is sent to the central queue of the manager (step 3). Once the task arrives to the central queue of the manager, a worker thread pulls and executes it on the FPGA in a First-In-First-Out order (step 4). Each operation in the task is linked with a OpenCL event and when the operation is completed the event is notified to the caller (step 5). In this way, the client is notified punctually, even if the operations are executed in groups.

For what concerns data exchange with the *Remote OpenCL Library*, the *Device Manager* allows two different mechanisms: network based (using gRPC) or shared memory. The Device Manager employs gRPC if the client application is not on the same node, or if it is not possible to create a shared memory area. Although gRPC is a powerful protocol for data exchange over network, we found performance issues utilizing it locally due to serialization overhead and due to multiple data copies. For this reason we limit its use whenever possible, leveraging instead shared memory. This improves performance and reduces additional data copies (from four to one) at the cost of having the client function together with the device manager on the same node with enough permissions. We still need one data copy to maintain full OpenCL compatibility, as a direct access to the shared memory would require to define additional functions not available in the OpenCL specification.

6.3.3 Accelerators Registry

The *Accelerators Registry* is the master component of the system: it registers functions and devices, it aggregates performance metrics, it allocates devices to functions and it validates reconfiguration operations. The *Registry* offers two endpoints, each backed by a different service. The *Devices Service* collects and manages information about the devices (e.g. platform, configured bitstream and connected instances). The *Functions Service* contains data about the serverless functions (e.g. identifier, location, device, created instances).

Data collected through the Device and Functions Services are integrated by the *Metrics Gatherer*, which receives Device Managers performance metrics from a Prometheus[5] service. Data like the FPGA time utilization (defined as the time spent by the device computing OpenCL calls in a given amount of time) are used to improve allocation of functions.

To match function instances and available devices, the Registry per-

[5]https://prometheus.io

Algorithm 3 Devices allocation algorithm

1: **procedure** ALLOCATE($instance, devs, metrics_order, metrics_filters$)
2: $devs \leftarrow filterby_compatibility(devs, instance.devicequery)$
3: $devs \leftarrow filterby_metrics(devs, metrics_filters)$
4: $devs \leftarrow orderby_metrics_and_acc(devs, metrics_order)$
5: $i \leftarrow 0$
6: **if** $not_compatible(devs(i))$ **then**
7: **while** $not_redistributable(devs(i))$ **do**
8: $i \leftarrow i + 1$
9: **if** $i < len(devs)$ **then**
10: $chosen_device \leftarrow devs(i)$
11: **else**
12: $raise\ error$ "$device\ not\ found$"
13: $instance.devs \leftarrow \{chosen_device\}$
14: **if** $instance.node ==$ "" **then**
15: $instance.node \leftarrow chosen_device.node$
16: **return**

forms an *online* allocation algorithm when a new instance is created. To do so, the Registry integrates with Kubernetes to intercept function creation and deletion in the cluster. When the cluster notifies the creation of a new function, the allocation algorithm patches the notified operation (e.g. adds environment variables, volumes for shared memory and forces the host allocation). The allocation algorithm is presented in Algorithm 3. It takes as input the function instance that must be matched, all the available devices in the system and a list of metrics to be taken into account. First, the procedure filters the devices based on their compatibility with the application requests (in terms of vendor, platform and accelerator) and the performance metrics (e.g. filtering out highly utilized devices). The devices are then sorted by metrics and by accelerator compatibility to ensure an optimal and consistent allocation. The metrics priority can be chosen depending on the system and applications SLA (e.g. device utilization, connected functions, latencies). The *accelerator compatibility* instead checks if the device should be reconfigured by looking at the currently configured bitstream. When compatible accelerators are missing, the algorithm checks which workloads can be redistributed to other compatible devices. If at least one device is found, it is flagged for reconfiguration and the Registry allocates it to the requesting function instance.

When a reconfiguration is required, BlastFunction checks the redistribution of instances and then *migrates* them with the Kubernetes API if necessary. In particular, when a function instance sends a reconfiguration request, the Registry verifies the allocation of the requesting function instance and checks if the device needs to be reconfigured. In that case, it deletes all the functions connected to that device. Kubernetes creates new instances before deleting the previous ones: in this way the Registry can patch and schedule them on a different node.

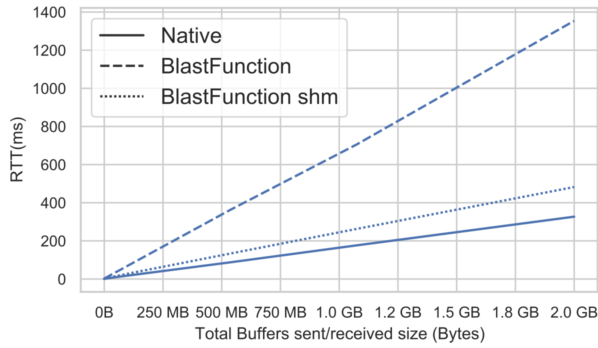
6.4 Experimental evaluation

The goals of the experimental campaign are to assess whether BlastFunction introduces an acceptable overhead (Section 6.4.1) and if it can improve device's time utilization (Section 6.4.2) w.r.t. the maximum theoretical performance scenario represented by a native execution that has direct access to the FPGAs. Moreover, we would like to understand whether the FPGA-based applications improve the performance-per-Watt ratio w.r.t. a standard CPU execution (Section 6.4.3). We leveraged three accelerated cloud functions available in the State of the Art: the Sobel edge detector and the Matrix Multiply (MM) kernel from the *Spector* benchmark suite [49] and *PipeCNN* [92], which is an open-source implementation of an FPGA accelerator for Convolutional Neural Networks (CNNs). This benchmark calls several kernels iteratively with multiple parallel command queues to compute the CNN output. According to the Spector benchmark, we synthesized the Sobel edge detector (also called *Sobel operator*) with 32×8 blocks, 4×1 window with no SIMD applied and a single compute unit, as it results in the best latency performance. For MM, we found from [49] that the best design is with 1 compute unit, 8 work items for each unit, and a completely unrolled block of 16×16 elements. Finally, we synthesized PipeCNN with AlexNet as in [92].

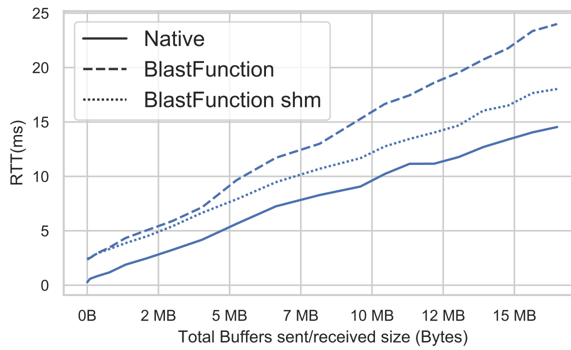
The experimental platform is composed of three nodes. The master node (node A) contains a 2.80Ghz Intel® Xeon®W3530 CPU, with 8 threads (4 cores) and 24GB of DDR3 RAM. Each worker node (nodes B and C) is equipped with a 3.40Ghz Intel® Core™ i7-6700 CPU, with 8 threads (4 cores) and 32GB of DDR4 RAM. Each node is connected to the local network through a 1Gb/s ethernet link. Each node contains a Terasic DE5a-Net FPGA board with an Intel®Arria 10 GX FPGA (1150K logic elements), 8GB RAM over 2 DDR2 SODIMM sockets and a PCI Express x8 connector (version 3 for the workers, version 2 for the master).

6.4.1 System overhead

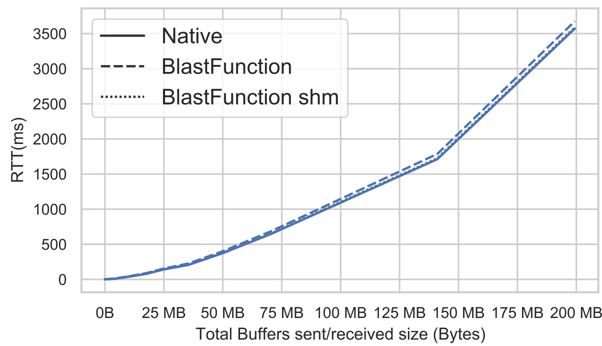
We evaluated the system overhead on a single node, deploying one instance of the Device Manager with a Docker Container connected to the FPGA. The host code was deployed on another Docker Container on the same node. Our system leverages the local virtual network stack + shared memory and PCI Express, while Native execution needs PCI Express only. We run each test by increasing the input and output size to see the impact of the Remote Library communication mechanism (both gRPC and shared memory) and the Device Manager queue. We tested each input size 40



(a) Latency overhead for read and write operations.



(b) Latency overhead for Sobel operator.



(c) Latency overhead for MM accelerator.

Figure 6.4: *Latency overhead w.r.t. input data size for R/W operations, sobel operator and MM kernel, graphs are in linear scale. For MM Native overlaps with BlastFunction shm.*

times averaging results and waiting 200ms after each call to have independent measures. We skip PipeCNN here because it does not allow to change the input size. Results are presented in Figure 6.4.

Figure 6.4(a) shows the RTT for a write-read operation (first write, then read synchronously) with total size from 1KB to 2GB for Native, BlastFunction and BlastFunction with shared memory. The pure gRPC implementation ("BlastFunction" in Figure 6.4(a)) shows a total latency of four times w.r.t. the Native execution. This is due to protobuf overheads and 3 copies of the data buffers. The shared memory implementation ("BlastFunction shm" label) shows an improvement in terms of latency and overhead, with a maximum overhead of 155ms when transferring 2GBs. Most of the overhead is composed by the single memory copy operation, while a smaller part ($\sim 2ms$) is given by the gRPC control signals, which are used in both systems.

Figure 6.4(b) shows the latency measurements for the Sobel operator. In all cases, the kernel has a linear behaviour w.r.t. the input size. The Native RTT starts from 0.27ms with a 10×10 image (800 bytes sent and received), up to 14.53ms for the largest image (1920×1080 pixels, read/write of $\sim 8MB$). BlastFunction starts with a overhead of 2.46ms and reaches 24ms with the largest image. BlastFunction shm, instead, has a constant $\sim 2ms$ overhead w.r.t. Native in all the experiments.

Figure 6.4(c) shows the latency measurements for the MM kernel. The MM accelerator is compute-intensive and the execution overhead between the Native and remote execution is low for both communication systems (still remaining lower in the shared memory system). The Native runtime shows a minimum RTT of 0.45ms for the smallest matrices (16×16 in both input and output matrices), but quickly rises up to 3.571s (for 4076×4096 matrices). As in the Sobel results, both BlastFunction and BlastFunction shm show a minimum RTT of $\sim 2ms$ given by the control signals. BlastFunction then reaches a maximum of 3.675s, while BlastFunction shm stops at 3.588s, which is only 17ms more than Native.

The results of Figure 6.4 show that the overall impact of our system depends on the complexity and operational intensity of the accelerator. When the majority of the execution time is spent in kernel execution the overall overhead is low (as in the MM example with a relative overhead of 0.27% for shared memory). Instead, with lower operational intensity, the I/O latency impacts more on the task even in the shared memory case (as in Sobel with a relative overhead of 24.04%). This derives from the fact that the Native system does not execute any additional data copy, while BlastFunction needs at least one copy to maintain full OpenCL compatibility.

Use-Case	Configuration	1st	2nd	3rd	4th	5th
Sobel	Low load	20 rq/s	15 rq/s	10 rq/s	5 rq/s	5 rq/s
	Medium Load	35 rq/s	30 rq/s	25 rq/s	20 rq/s	15 rq/s
	High Load	60 rq/s	50 rq/s	35 rq/s	30 rq/s	15 rq/s
MM	Low load	28 rq/s	21 rq/s	14 rq/s	7 rq/s	7 rq/s
	Medium Load	49 rq/s	42 rq/s	35 rq/s	28 rq/s	21 rq/s
	High Load	84 rq/s	70 rq/s	49 rq/s	42 rq/s	21 rq/s
AlexNet	Medium load	6 rq/s	3 rq/s	3 rq/s	3 rq/s	3 rq/s
	High Load	9 rq/s	9 rq/s	6 rq/s	6 rq/s	3 rq/s

Table 6.1: *Tests configurations overview, showing how many requests per second were sent to each function for each benchmark.*

.

6.4.2 FPGAs time utilization

To test the FPGAs utilization, we run a set of multi-application, multi-node experiments. The goal is to check if BlastFunction is able to increase the FPGAs time utilization and the number of total requests served without significant losses for the single tenant. Here we leverage BlastFunction with shared memory, wrapping each benchmark in a OpenFaaS function both for Native and BlastFunction. For each experiment (Sobel, MM and PipeCNN with AlexNet) we deployed 5 identical functions for *BlastFunction*, while we could deploy only 3 functions in the Native scenario (one for each device). We tested each function using Hey[6] (a tool for HTTP load testing), running the experiments multiple times with one connection per function and collecting data about latency and FPGA utilization. Table 6.1 shows all the configurations used for the BlastFunction runtime, while for the native scenario we consider only the first 3 columns as only 3 functions can be deployed during these tests.

We show the per-function results for Sobel in Table 6.2. The results are divided by scenario (BlastFunction vs Native), configuration, and tested function. In the low load configuration both runtimes keep up with the target throughput with latency between 20-30ms. Results are in line with the overhead results, with BlastFunction improving device's utilization. In the medium load configuration BlastFunction has better latency for *sobel-1*, *sobel-2* and *sobel-3* and the other two functions effectively increases the board's time utilization. Finally, in the high load configuration, BlastFunction still improved the overall FPGAs time utilization with comparable latency results for *sobel-1* and *sobel-3*. However, *Node A* saturated in both

[6]https://github.com/rakyll/hey

Type	Configuration	Function	Node	Util.	Latency	Processed	Target
BlastFunction	Low Load	sobel-1	B	21.95%	21.43 ms	17.25 rq/s	20.00 rq/s
		sobel-2	A	22.57%	24.23 ms	15.00 rq/s	15.00 rq/s
		sobel-3	C	13.22%	19.01 ms	10.00 rq/s	10.00 rq/s
		sobel-4	A	7.49%	31.98 ms	5.00 rq/s	5.00 rq/s
		sobel-5	B	6.48%	27.16 ms	5.00 rq/s	5.00 rq/s
	Medium Load	sobel-1	B	40.95%	19.45 ms	32.93 rq/s	35.00 rq/s
		sobel-2	A	39.40%	23.62 ms	26.30 rq/s	30.00 rq/s
		sobel-3	C	32.85%	18.28 ms	24.98 rq/s	25.00 rq/s
		sobel-4	A	29.85%	26.99 ms	19.98 rq/s	20.00 rq/s
		sobel-5	B	18.76%	22.94 ms	14.97 rq/s	15.00 rq/s
	High Load	sobel-1	B	60.31%	18.95 ms	49.58 rq/s	60.00 rq/s
		sobel-2	A	39.15%	32.05 ms	26.63 rq/s	50.00 rq/s
		sobel-3	C	45.75%	17.82 ms	34.96 rq/s	35.00 rq/s
		sobel-4	A	38.44%	22.56 ms	26.11 rq/s	30.00 rq/s
		sobel-5	B	18.39%	21.74 ms	15.00 rq/s	15.00 rq/s
Native	Low Load	sobel-1	A	30.41%	25.02 ms	19.49 rq/s	20.00 rq/s
		sobel-2	B	19.74%	21.50 ms	14.74 rq/s	15.00 rq/s
		sobel-3	C	13.73%	24.34 ms	9.75 rq/s	10.00 rq/s
	Medium Load	sobel-1	A	51.48%	26.04 ms	33.11 rq/s	35.00 rq/s
		sobel-2	B	37.19%	23.33 ms	27.95 rq/s	30.00 rq/s
		sobel-3	C	34.22%	23.48 ms	24.23 rq/s	25.00 rq/s
	High Load	sobel-1	A	58.10%	26.77 ms	38.36 rq/s	60.00 rq/s
		sobel-2	B	54.69%	23.95 ms	41.80 rq/s	50.00 rq/s
		sobel-3	C	44.81%	24.75 ms	32.61 rq/s	35.00 rq/s

Table 6.2: *Multi-function test results for the Sobel acceleator in terms of average latency, FPGA time utilization (overall maximum 300%) and processed/target requests.*

cases as it is not able to keep-up with the target throughput. Regarding the requests throughput, Native has a difference w.r.t. the target of 2.25% in the low load configuration, 5.23% and 22.22% for the medium and high load conditions respectively. BlastFunction has instead averages of 5.01%, 4.67% and 19.85% respectively. Although BlastFunction supports more load, the response of the two systems are still comparable.

Table 6.3 shows the aggregate results for MM. We do not show the detailed results for brevity as similar considerations can be made. The Native scenario presents a higher difference between target and processed requests w.r.t. BlastFunction, with slightly higher latencies and a similar utilization. The average difference for BlastFunction is of 0.04%, 0.05% and 1.22% for the low, medium and high load configurations. Meanwhile, Native reaches 3.97% with a low load, 15.19% and 39.97% in medium and high load conditions.

Finally, we show the aggregate results for AlexNet with the PipeCNN accelerator in Table 6.4. Because of the low number of requests that the accelerator is able to serve, we decided to test only two configurations, with medium and high load conditions. The results show that Native has an average latency of 94.29ms for medium load and 91.74ms for high load,

Type	Configuration	Utilization	Latency	Processed	Target
BlastFunction	Low Load	43.49%	12.55 ms	76.96 rq/s	77 rq/s
	Medium Load	98.53%	11.57 ms	174.90 rq/s	175 rq/s
	High Load	144.18%	10.69 ms	262.73 rq/s	266 rq/s
Native	Low Load	50.87%	21.12 ms	60.49 rq/s	63 rq/s
	Medium Load	103.22%	22.81 ms	106.84 rq/s	126 rq/s
	High Load	122.97%	24.25 ms	121.85 rq/s	203 rq/s

Table 6.3: *Multi-function test aggregate results for MM in terms of average latency, FPGA time utilization (overall maximum 300%) and processed/target requests.*

Type	Configuration	Utilization	Latency	Processed	Target
BlastFunction	Medium Load	124.68%	132.89ms	17.88 rq/s	18 rq/s
	High Load	202.08%	124.52ms	29.81 rq/s	33 rq/s
Native	Medium load	96.22%	94.29ms	11.91 rq/s	12 rq/s
	High Load	189.82%	91.74ms	23.57 rq/s	24 rq/s

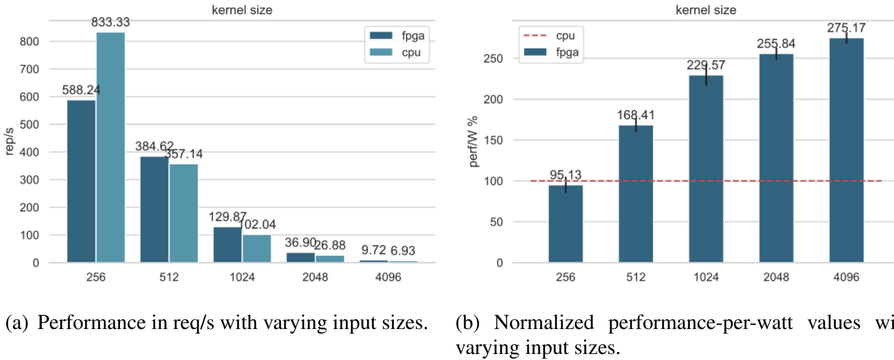
Table 6.4: *Multi-function test aggregate results for PipeCNN (AlexNet) with average latency, FPGA time utilization (overall maximum 300%) and processed/target requests.*

while BlastFunction presents a higher latency (132.89ms for medium and 124.52ms for high load). This difference derives from the concurrent access to the device with long execution times, which raise the probability of having a request waiting for the previous one to complete. Regarding the difference between sent and processed requests, we have 0.63% for BlastFunction and 0.68% for Native in medium load conditions, while in high load conditions Native behaves better (1.79% vs 9.64%). However, in both configurations, sharing allows BlastFunction to reach a higher utilization and number of processed requests.

6.4.3 Power consumption

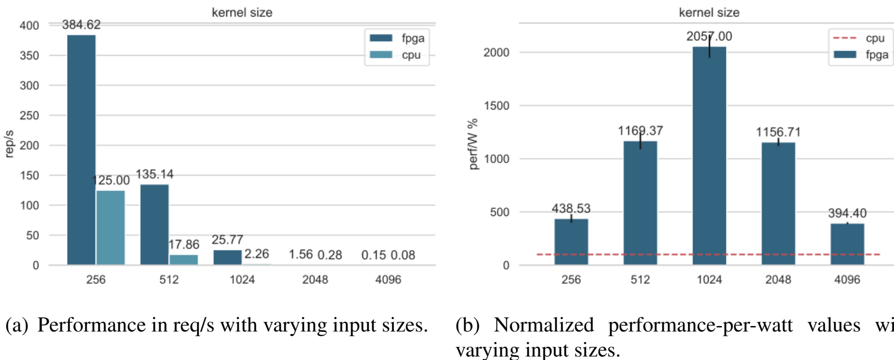
To test the improvements provided by the FPGAs over the power consumption of the system, we instrumented the test machines with a Watts up power meter [94]. We then run the Sobel and MM applications both on the CPU as well as on the FPGA, loading them with 10,000 requests, where each run has different input sizes. Again, here we skip PipeCNN as we are not able to change the input size among the various experiments.

Figure 6.5 shows the results in terms of performance and performance-per-Watt for the Sobel filter (Figures 6.5(a) and 6.5(b) respectively). As for pure performance, with the only exception of input size 256x256, we can see that when we increase the input size, the FPGA-based Sobel filter is able to process more requests per second w.r.t. the software one. Of course,



(a) Performance in req/s with varying input sizes. (b) Normalized performance-per-watt values with varying input sizes.

Figure 6.5: *Sobel filter performance in req/s and normalized performance/W (higher is better).*



(a) Performance in req/s with varying input sizes. (b) Normalized performance-per-watt values with varying input sizes.

Figure 6.6: *Matrix multiplication performance in req/s and normalized performance/W (higher is better).*

increasing the input size means increasing also the transfer time. This, in turn, leads to a decrease in the number of requests per second that both implementations are able to deliver. If we look at Figure 6.5(b), we can see that the performance-per-Watt ratio increases when we increase the input size. In particular, in the case of the smallest input size, we can see that the performance-per-Watt ratio of the two implementations is close, while the FPGA reaches a 2.75x improvement with the largest input size we tested.

Figure 6.6 shows the results in terms of performance and performance-per-Watt for the MM kernel (Figures 6.6(a) and 6.6(b) respectively). Within this case, we can see that the hardware implementation outperforms the software one even in the case of input size 256x256. However, the performance speed-up decreases while increasing the input size after reaching a

peak. This is because MM is a memory-bound algorithm and it reaches the memory bandwidth limits earlier on the FPGA than on the CPU. This behavior can be seen also in Figure 6.6(b), where the performance-per-Watt ratio is always higher w.r.t. the CPU one, but it increases up to 20.5x before decreasing due to reaching the memory bandwidth limit.

6.5 Conclusion and future work

We presented BlastFunction, a distributed FPGA sharing system for the acceleration of microservices and serverless applications in cloud environments. The proposed system design is based upon the goals of *multi-tenancy* and *scalability*, with a focus on the *transparency* of the resulting software library. BlastFunction adds limited overhead and reaches higher utilization and throughput w.r.t. native execution thanks to device sharing. Moreover, it shows that the FPGAs can be effectively used to improve not only the performances, but also the performance-per-Watt ratio of the accelerated cloud workloads.

Future work will address the integration with AWS F1 and its APIs to provide autoscaling of nodes depending on system load, as well as mechanisms to support space-sharing alongside the BlastFunction time-sharing capability.

CHAPTER 7

Conclusion

Within this thesis work, we presented our efforts towards the design and development of power management techniques able to sustain the performances requested by cloud-native applications while reducing as much as possible the power consumption such applications generate. To achieve these goals we resorted to autonomic techniques like the ODA control loop to Observe power and performance behavior of heterogeneous and highly co-located workloads, Decide how to give power budgets to workloads depending on performance constraints, and Act to reach such performance constraints while reducing power consumption.

Chapter 2 presented our first approach towards the definition of a black-box power and performance monitoring methodology. We attributed power consumption to each microservice depending on the architectural characteristics of each microservice and we started to collect metrics about CPU usage and low-level performance. Then, we integrated and improved such data collection system with network performance monitoring in Chapter 3. The resulting monitoring approach provides a unified view of the Kubernetes cluster state, analyzing performance, saturation limits and the graph of interconnections between all the microservices in the cluster.

Then, we leveraged all the metrics we collected to manage power and

performance of cloud-native applications. Chapter 4 presented a CPU-based ODA control loop able to maintain each microservice near the CPU usage defined by the application developer while at the same time reducing power consumption. Chapter 5, instead, leverages a higher level target: application latency. The Resulting ODA loop keeps the given average latency of the whole application by computing requirements for all its components and reactively enforcing them.

Finally, Chapter 6 shows a different approach towards the improvement of power efficiency of cloud-native applications. In particular, we explored how to merge together cloud-native infrastructures with accelerated cloud-native workloads. The result is a system able to improve the performance and the performance-per-Watt ratio w.r.t. CPU-based microservices and improve utilization of FPGAs by sharing resources.

Limitations and future work of each component of this thesis are discussed in their respective chapters. In a more broader view, the future work of this thesis revolves around the design of a fully *integrated* system, where power management systems can cooperate at different levels of the data-center stack towards a truly green cloud. Taking into account also server components other than CPUs (e.g. disks, network cards, memories) will enable to extract even more power savings by analyzing access patterns and by improving current optimization strategies. Specialized resources like FPGAs should be then exploited extensively to improve power efficiency of compute-intensive cloud-native workloads. Then, the introduction of power and performance-aware ODA control loops will help in the optimization process at the infrastructure level. Finally, coordination with power grids and other power-management systems will provide the best strategy towards the sustainable management of cloud-computing infrastructures.

Bibliography

[1] Cloud-native definition. `https://github.com/cncf/toc/blob/master/DEFINITION.md`.

[2] Kubernetes cpu requests and limits. `https://kubernetes.io/docs/tasks/configure-pod-container/assign-cpu-resource/`.

[3] Prometheus monitoring system. `https://prometheus.io/`.

[4] Anders SG Andrae and Tomas Edler. On global electricity usage of communication technology: trends to 2030. *Challenges*, 6(1):117–157, 2015.

[5] Marco Arnaboldi, Rolando Brondolin, and Marco Domenico Santambrogio. Hyppo: Hybrid performance-aware power-capping orchestrator. In *2018 IEEE International Conference on Autonomic Computing (ICAC)*, pages 71–80. IEEE, 2018.

[6] Marco Arnaboldi, Matteo Ferroni, and Marco D Santambrogio. Towards a performance-aware power capping orchestrator for the xen hypervisor. *ACM SIGBED Review*, 15(1):8–14, 2018.

[7] Shuichi Asano, Tsutomu Maruyama, and Yoshiki Yamaguchi. Performance comparison of fpga, gpu and cpu in image processing. In *2009 international conference on field programmable logic and applications*, pages 126–131. IEEE, 2009.

[8] Mikhail Asiatici, Nithin George, Kizheppatt Vipin, Suhaib A. Fahmy, and Paolo Ienne. Designing a virtual runtime for FPGA accelerators in the cloud. *FPL 2016 - 26th International Conference on Field-Programmable Logic and Applications*, 2016.

[9] AMEDEO Asnaghi, M Ferroni, and MD Santambrogio. Dockercap: A software-level power capping orchestrator for docker containers. In *Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES), 2016 IEEE Intl Conference on*, pages 90–97. IEEE, 2016.

[10] Marco Bacis, Rolando Brondolin, and Marco D Santambrogio. Blastfunction: an fpga-as-a-service system for accelerated serverless computing. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2020*, pages 852–857. IEEE, 2020.

[11] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, 1991.

[12] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool Publishers, 2013.

[13] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12), 2007.

[14] Bpf compiler collection (bcc). `https://www.iovisor.org/technology/bcc`, 2020. [Online; accessed 25-Jan-2020].

[15] Andrew Begel, Steven McCanne, and Susan L Graham. Bpf+: Exploiting global data-flow optimization in a generalized packet filter architecture. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 123–134. ACM, 1999.

[16] Frank Bellosa. The benefits of event: driven energy accounting in power-sensitive systems. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*, pages 37–42. ACM, 2000.

[17] Anton Beloglazov, Rajkumar Buyya, Young Choon Lee, Albert Zomaya, et al. A taxonomy and survey of energy-efficient data centers and cloud computing systems. *Advances in computers*, 82(2):47–111, 2011.

[18] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[19] Arka A Bhattacharya, David Culler, Aman Kansal, Sriram Govindan, and Sriram Sankar. The need for speed and stability in data center power capping. *Sustainable Computing: Informatics and Systems*, 3(3):183–193, 2013.

[20] Rolando Brondolin, Matteo Ferroni, and Marco Santambrogio. Performance-aware load shedding for monitoring events in container based environments. *ACM SIGBED Review*, 16(3):27–32, 2019.

[21] Rolando Brondolin and Marco D Santambrogio. A black-box monitoring approach to measure microservices runtime performance. *ACM Transactions on Architecture and Code Optimization (TACO)*, 17(4):1–26, 2020.

[22] Rolando Brondolin and Marco Domenico Santambrogio. Presto: a latency-aware power-capping orchestrator for cloud-native microservices. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 11–20. IEEE, 2020.

[23] Rolando Brondolin, Tommaso Sardelli, and Marco D Santambrogio. Deep-mon: Dynamic and energy efficient power monitoring for container-based infrastructures. In *Parallel and Distributed Processing Symposium Workshops, 2018 IEEE International*, pages 676–684. IEEE, 2018.

[24] Jake Brutlag. Speed matters for google web search, 2009.

[25] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Communications of the ACM*, 59(5):50–57, 2016.

[26] Stuart Byma, J. Gregory Steffan, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. FPGAs in the cloud: Booting virtualized hardware accelerators with OpenStack. *Proceedings - 2014 IEEE 22nd International Symposium on Field-Programmable Custom Computing Machines, FCCM 2014*, pages 109–116, 2014.

[27] Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience*, 41(1):23–50, 2011.

[28] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A cloud-scale acceleration architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 7. IEEE Press, 2016.

[29] Adrian M Caulfield, Others, Eric S Chung, Puneet Kaur, Joo-young Kim Daniel, Lo Todd, and Massengill Kalin. A cloud-scale acceleration architecture. *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.

[30] Chia-Chen Chang, Shun-Ren Yang, En-Hau Yeh, Phone Lin, and Jeu-Yih Jeng. A kubernetes-based monitoring platform for dynamic cloud resource provisioning. In *GLOBECOM 2017-2017 IEEE Global Communications Conference*, pages 1–6. IEEE, 2017.

[31] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro*, 38(2):8–20, 2018.

[32] Marcello Cinque, Raffaele Della Corte, and Antonio Pecchia. Microservices monitoring with event logs and black box execution tracing. *IEEE Transactions on Services Computing*, 2019.

[33] Cncf cloud native landscape. `https://landscape.cncf.io`, 2020. [Online; accessed 10-Feb-2020].

[34] Yan Cui, Charles Ingalz, Tianyi Gao, and Ali Heydari. Total cost of ownership model for data center technology evaluation. In *Thermal and Thermomechanical Phenomena in Electronic Systems (ITherm), 2017 16th IEEE Intersociety Conference on*, pages 936–942. IEEE, 2017.

[35] Datadog. `https://www.datadoghq.com`.

[36] Maximilien de Bayser and Renato Cerqueira. Integrating mpi with docker for hpc. In *Cloud Engineering (IC2E), 2017 IEEE International Conference on*, pages 259–265. IEEE, 2017.

[37] Arnaldo Carvalho De Melo. The new linux perf tools. In *Slides from Linux Kongress*, volume 18, pages 1–42, 2010.

[38] Luca Deri, Maurizio Martinelli, and Alfredo Cardigliano. Realtime high-speed network traffic monitoring using ntopng. In *28th Large Installation System Administration Conference (LISA14)*, pages 78–88, 2014.

[39] Luca Deri, Samuele Sabella, and Simone Mainardi. Combining system visibility and security using ebpf. In *ITASEC*, 2019.

[40] NASA Advanced Supercomputing Division. Nas parallel benchmarks (npb). `http://www.nas.nasa.gov/publications/npb.html`, 2018.

[41] Docker. `https://www.docker.com`, 2020. [Online; accessed 25-Jan-2020].

[42] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 13–23. ACM, 2007.

[43] Matteo Ferroni, Juan A Colmenares, Steven Hofmeyr, John D Kubiatowicz, and Marco D Santambrogio. Enabling power-awareness for the xen hypervisor. *ACM SIGBED Review*, 15(1):36–42, 2018.

[44] Matteo Ferroni, Andrea Corna, Andrea Damiani, Rolando Brondolin, Juan A Colmenares, Steven Hofmeyr, John D Kubiatowicz, and Marco D Santambrogio. Power consumption models for multi-tenant server infrastructures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(4):1–22, 2017.

[45] Matteo Ferroni, Andrea Corna, Andrea Damiani, Rolando Brondolin, John D Kubiatowicz, Donatella Sciuto, and Marco D Santambrogio. Marc: A resource consumption modeling service for self-aware autonomous agents. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 12(4):21, 2017.

[46] Martin Fowler and James Lewis. Microservices a definition of this new architectural term. *URL: http://martinfowler. com/articles/microservices. html*, page 22, 2014.

[47] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[48] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 19–33, 2019.

[49] Quentin Gautier, Alrie Althoff, Pingfan Meng, and Ryan Kastner. Spector: An OpenCL FPGA benchmark suite. *Proceedings of the 2016 International Conference on Field-Programmable Technology, FPT 2016*, pages 141–148, 2017.

[50] M. Kurtzer Gregory, Sochat Vanessa, Bauer Michael, and Bockelman Brian. Singularity project. `http://singularity.lbl.gov`, 2018. [Online; accessed 25-Jan-2018].

[51] Part Guide. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part*, 2, 2011.

[52] James Hamilton. Cost of power in large-scale data centers. `http://perspectives.mvdirona.com/2008/11/cost-of-power-in-large-scale-data-centers/`, 2008. [Online; accessed 21-Jan-2018].

[53] Henry Hoffmann, Jonathan Eastep, Marco D Santambrogio, Jason E Miller, and Anant Agarwal. Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In *Proceedings of the 7th international conference on Autonomic computing*, pages 79–88, 2010.

[54] Henry Hoffmann, Martina Maggio, Marco D Santambrogio, Alberto Leva, and Anant Agarwal. Seec: A framework for self-aware computing. 2010.

[55] Emir Imamagic and Dobrisa Dobrenic. Grid infrastructure monitoring system based on nagios. In *Proceedings of the 2007 workshop on Grid monitoring*, pages 23–28. ACM, 2007.

[56] Connor Imes, Huazhe Zhang, Kevin Zhao, and Henry Hoffmann. Copper: Soft real-time application performance using hardware power capping. In *2019 IEEE International Conference on Autonomic Computing (ICAC)*, pages 31–41. IEEE, 2019.

[57] Anca Iordache, Guillaume Pierre, Peter Sanders, Jose Gabriel de F Coutinho, and Mark Stillwell. High performance in the cloud with fpga groups. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*, pages 1–10. ACM, 2016.

[58] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[59] Harshad Kasture, Davide B Bartolini, Nathan Beckmann, and Daniel Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 598–610. IEEE, 2015.

[60] Jakub Krzywda, Ahmed Ali-Eldin, Eddie Wadbro, Per-Olov Östberg, and Erik Elmroth. Power shepherd: Application performance aware power shifting. In *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 45–53, 2019.

[61] Michael Larabel and M Tippett. Phoronix test suite. *Phoronix Media,[Online]. Available: http://www. phoronix-test-suite. com/.[Accessed June 2016]*, 2011.

[62] Edward D Lazowska, John Zahorjan, G Scott Graham, and Kenneth C Sevcik. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984.

[63] Emily Le and David Paz. Performance analysis of applications using singularity container on sdsc comet. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, page 66. ACM, 2017.

[64] Linux tracepoints. `https://www.kernel.org/doc/Documentation/trace/tracepoints.txt`. [Online; accessed 03-Oct-2016].

[65] Liang Liu, Hao Wang, Xue Liu, Xing Jin, Wen Bo He, Qing Bo Wang, and Ying Chen. Greencloud: a new architecture for green data center. In *Proceedings of the 6th international conference industry session on Autonomic computing and communications industry session*, pages 29–38. ACM, 2009.

[66] David Lo, Liqun Cheng, Rama Govindaraju, Luiz Andre Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 301–312. IEEE, 2014.

[67] Aniruddha Marathe, Ghaleb Abdulla, Barry L Rountree, and Kathleen Shoga. Towards a unified monitoring framework for power, performance and thermal metrics: A case study on the evaluation of hpc cooling systems. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*, pages 974–983. IEEE, 2017.

[68] Dimosthenis Masouros, Sotirios Xydis, and Dimitrios Soudris. Rusty: Runtime interference-aware predictive monitoring for modern multi-tenant systems. *IEEE Transactions on Parallel and Distributed Systems*, 32(1):184–198, 2020.

[69] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[70] Charles Masson, Jee E Rim, and Homin K Lee. Ddsketch: a fast and fully-mergeable quantile sketch with relative-error guarantees. *Proceedings of the VLDB Endowment*, 12(12):2195–2205, 2019.

[71] Stelios Mavridis, Manolis Pavlidakis, Ioannis Stamoulias, Christos Kozanitis, Nikolaos Chrysos, Christoforos Kachris, Dimitrios Soudris, and Angelos Bilas. VineTalk: Simplifying software access and sharing of FPGAs in datacenters. *2017 27th International Conference on Field Programmable Logic and Applications, FPL 2017*, pages 2–5, 2017.

[72] Benjamin Mayer and Rainer Weinreich. A dashboard for microservice monitoring and management. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 66–69. IEEE, 2017.

[73] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX winter*, volume 93, 1993.

[74] David Meisner, Brian T Gold, and Thomas F Wenisch. Powernap: eliminating server idle power. In *ACM Sigplan Notices*, volume 44, pages 205–216. ACM, 2009.

[75] David Meisner, Christopher M Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F Wenisch. Power management of online data-intensive services. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 319–330. ACM, 2011.

[76] Farnaz Moradi, Christofer Flinta, Andreas Johnsson, and Catalin Meirosu. Conmon: An automated container based network performance monitoring system. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 54–62. IEEE, 2017.

[77] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, volume 710, 1999.

[78] Barak Naveh et al. Jgrapht. *Internet: http://jgrapht. sourceforge. net*, 2008.

[79] Newrelic. `https://newrelic.com`.

[80] Ayman Noor, Devki Nandan Jha, Karan Mitra, Prem Prakash Jayaraman, Arthur Souza, Rajiv Ranjan, and Schahram Dustdar. A framework for monitoring microservice-oriented cloud applications in heterogeneous virtualization environments. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 156–163. IEEE, 2019.

[81] S Ojika, A Gordon-Ross, H Lam, B Patel, G Kaul, and J Strayer. Using fpgas as microservices: Technology, challenges and case study. In *9th Workshop on Big Data Benchmarks Performance, Optimization and Emerging Hardware (BPOE-9)*, 2018.

[82] Philippos Papaphilippou and Wayne Luk. Accelerating database systems using fpgas: A survey. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 125–1255. IEEE, 2018.

[83] Fábio Pina, Jaime Correia, Ricardo Filipe, Filipe Araujo, and Jorge Cardroom. Nonintrusive monitoring of microservice-based systems. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pages 1–8. IEEE, 2018.

[84] Sareh Fotuhi Piraghaj, Amir Vahid Dastjerdi, Rodrigo N Calheiros, and Rajkumar Buyya. A framework and algorithm for energy efficient container consolidation in cloud data centers. In *Data Science and Data Intensive Systems (DSDIS), 2015 IEEE International Conference on*, pages 368–375. IEEE, 2015.

[85] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News*, 42(3):13–24, 2014.

[86] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Eliezer Weissmann, and Doron Rajwan. Power-management architecture of the Intel microarchitecture code-named Sandy Bridge. *IEEE Micro*, 32(2):20–27, March 2012.

[87] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Eliezer Weissmann, and Doron Rajwan. Power-management architecture of the intel microarchitecture code-named sandy bridge. *Ieee micro*, 32(2):20–27, 2012.

[88] Kai Shen, Arrvindh Shriraman, Sandhya Dwarkadas, Xiao Zhang, and Zhuan Chen. Power containers: An os facility for fine-grained power and energy management on multicore servers. In *ACM SIGPLAN Notices*, volume 48, pages 65–76. ACM, 2013.

[89] Sysdig. `https://sysdig.com`.

[90] Tcpdump. `http://www.tcpdump.org`.

[91] Paul Townend, Stephen Clement, Dan Burdett, Renyu Yang, Joe Shaw, Brad Slater, and Jie Xu. Improving data center efficiency through holistic scheduling in kubernetes. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 156–15610. IEEE, 2019.

[92] Dong Wang, Ke Xu, and Diankun Jiang. Pipecnn: An opencl-based open-source fpga accelerator for convolution neural networks. In *2017 International Conference on Field Programmable Technology (ICFPT)*, pages 279–282. IEEE, 2017.

[93] Wei Wang, Miodrag Bolic, and Jonathan Parri. PvFPGA: Accessing an FPGA-based hardware accelerator in a paravirtualized environment. *2013 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2013*, 2013.

[94] Watts up? plug load meters. `https://www.wattsupmeters.com/secure/products.php?pn=0`. Accessed: 2017-06-07.

[95] Weave scope. `https://www.weave.works/oss/scope/`.

[96] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. Automated systolic array architecture synthesis for high throughput cnn inference on fpgas. In *Proceedings of the 54th Annual Design Automation Conference 2017*, page 29. ACM, 2017.

[97] Andrew J Younge, Kevin Pedretti, Ryan E Grant, and Ron Brightwell. A tale of two systems: Using containers to deploy hpc applications on supercomputers and clouds. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 74–81. IEEE, 2017.

[98] Yan Zhai, Xiao Zhang, Stephane Eranian, Lingjia Tang, and Jason Mars. Happy: Hyperthread-aware power profiling dynamically. In *USENIX Annual Technical Conference*, pages 211–217, 2014.

[99] Huazhe Zhang and Henry Hoffmann. Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques. *ACM SIGARCH Computer Architecture News*, 44(2):545–559, 2016.

[100] Zhuangdi Zhu, Alex X. Liu, Fan Zhang, and Fei Chen. FPGA Resource Pooling in Cloud Computing. *IEEE Transactions on Cloud Computing*, PP(c):1, 2018.