POLITECNICO DI MILANO

Facoltà di Ingegneria Industriale e dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica

XeMPUPiL
Towards a performance-aware power capping orchestrator for the Xen hypervisor

Relatore: Prof. Marco Domenico SANTAMBROGIO

Correlatore: Dott. Ing. Rolando BRONDOLIN

Tesi di Laurea di:

Marco Arnaboldi

Matricola n. 833920

Anno Accademico 2016–2017

To my family

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

List of Abbreviations

aaS	as a Service
API	Application Programming Interface
CLI	Command Line Interface
DRAM	Dynamic Random Access Memory
DVFS	Dynamic Voltage and Frequency Scaling
IR	Instruction Retired
HPC	Hardware Performance Counters
MSR	Model Specific Register
RAPL	Running Average Power Limit
ODA	Observe Decide Act
OS	Operating System
PMU	Performance Monitoring Unit
pCPU	physical CPU
TTL	Time To Live
vCPU	virtual CPU
VM	Virtual Machine

MLR	Multinomial Logistic Regression
NASA	National Aeronautics and Space Administration
NPB	NAS Parallel Benchmarks
EP	Embarassingly parallel
BT	Block Tri-diagonal solver
MC	Memory Controller
SoA	State of the Art
CPU	Central Processing Unit
PMC	Performance Monitoring Counter
TDP	Thermal Design Power
EDA	Eletronic Desing Automation
SLA	Service Level Agreement
SoC	System on Chip

Abstract

In the era of Cloud Computing, applications and computational power are provided in an as a Service (aaS) fashion, reducing the need of buying, building and maintaining proprietary systems. In the last few years, many services moved from being proprietary and built in loco, to the as a Service paradigm. This was possible thanks to virtualization techniques, which allowed multiple applications to easily run on the same machine. However, the burden of costs optimization is left to the Cloud Provider, that still faces the problem of consolidating multiple workloads on the same infrastructure. As power consumption remains one of the most impacting costs of any digital system, several approaches have been explored in literature to cope with power caps, trying to maximize the performance of the hosted applications. These approaches were usually classified in two macro families, the software and hardware techniques. The former family is typically adopted when the goal consists in minimizing the power consumption, while providing the best possible performance for the running workloads. These approaches are characterized by obtaining high efficiency, but lacks in timeliness. Instead, the latter family is exploited when there are strict constraints regarding the power budget and the main goal consists in respecting them, while trying to maximize the performance of the running applications. In this case, the main characteristic consists in respecting the concept of timeliness, totally neglecting the concept of efficiency. In this thesis, we present results and opportunities obtained towards a performance-aware power capping orchestrator for the Xen hypervisor, that exploit a novel emerging family introduced in the literature: the hybrid approach. This fresh set of techniques aims to

adopt synergically and concurrently both hardware and software approaches in order to achieve at the same time the concept of efficiency and timeliness, masking the weak spots of the two common approaches when adopted alone. The proposed solution, called *XeMPUPiL*, uses the Intel RAPL hardware interface to set a strict limit on the processor's power consumption, while a software-level ODA control loop performs an exploration of the available resource allocations to find the most power efficient one for the running workload. We show how the *XeMPUPiL* methodology is able to allow the definition of two different policies: achieving higher performance under different power caps and minimizing power consumption while respecting a given SLA for almost all the different classes of benchmarks analyzed (e.g., CPU-, memory- and IO-bound).

Sommario

Al giorno d'oggi stiamo assistendo all'affermazione di un nuovo paradigma computazionale: il cloud computing. In questa nuova era, detta epoca dell'aaS, le applicazione e i servizi non vengono più eseguiti su macchine di proprietà, ma bensì su macchine (spesso virtuali) fornite da terzi: i così detti cloud provider. Dal punto di vista degli utenti questo permette loro di accedere a queste risorse computazionali in maniera elastica e scalabile, permettendo di dimensionare facilmente le loro necessità computazionali a seconda delle variazioni del mercato o anche solo all'interno della giornata lavorativa, riducendo in questa maniera i costi e i danni economici in caso di errore nella stima delle risorse richieste. D'altra parte questo "nuovo mondo" ha spostato tutti quelli che erano i costi di gestione delle macchine fisiche sulle spalle dei cloud provider. Inoltre l'attrattiva che questo nuovo paradigma porta con se, ha spinto sempre più clienti ad adottare approcci basati sul cloud computing. L'incremento di utenti affacciatesi a questo paradigma ha portato al sorgere di due sfide per i gestori del cloud, in particolare due sfide riguardanti la gestione dei datacenter. In modo da rispondere alla crescente domanda i gestori di datacenter devono aumetare la loro potenza computazionale e per farlo sono costretti ad aumentare il numero di server presenti nelle loro strutture rischiando quindi di saturare e sforare lo spazio fisico della struttura oppure di non rispettare i contratti con i fornitori energetici, incappando in penali o blackout. Per risolvere questi problemi diverse tecniche di ottimizzazioni delle risorse all'interno di un datacenter e di gestione dei carichi di lavoro sono stati promossi. Su tutti spiccano le tecniche di virtualizzazione, ormai diventate una prassi adottata da tutti i cloud provider. In questo modo è

possibile sfruttare una macchina fisica da più utenti, aumentandone così l'efficienza.

In questo scenario, il consumo di potenza rimane uno dei costi principali di ogni sistema digitale. Diversi approcci hanno provato, in letteratura, ad affrontare il problema dei consumi e limiti di potenza, provando a massimizzare le prestazioni delle applicazioni ospitate. Questi approcci sono comunemente classificati in due macro famiglie, quella software e quella hardware. La prima è tipicamente adottata quando l'obiettivo consiste in minimizzare il consumo di potenza e allo stesso tempo fornendo le performance migliori per i carichi di lavoro del sistema. Questa famiglia è caratterizzata dall'ottenimento di alta efficienza, ma dalla mancanza di tempestività. Al contrario, la seconda famiglia è usata quando ci sono vincoli stringenti riguardanti il budget di potenza e l'obiettivo principale consiste nel rispettarlo e contemporaneamente provare a massimizzare le performance delle applicazioni in esecuzione. In questo caso, la caratteristica principale consiste nel rispettare il concetto di tempestività, trascurando completamente il concetto di efficienza.

In questa tesi presentiamo una metodologia così detta ibrida, che cerca di sfruttare contemporaneamente sia un approccio software (un ciclo di controllo ODA) e un approccio hardware (Intel RAPL) in modo da nascondere i punti deboli dei due approcci quando presi singolarmente, ottenendo in questo modo sia efficienza the tempestività. Lo sviluppo di questa metodologia sfocia nel design di un orchestratore performance-aware e in grado di effettuare power capping svilluppato per l'hypervisor Xen, come prova di concetto. La soluzione proposta, chiamata *XeMPUPiL*, sfrutta la tecnologia RAPL di Intel tramite le sue interfacce hardware per definire un limite stringente sui consumi di potenza del processore, mentre a livello software un ciclo di controllo basato su strategia ODA si occupa di un'esplorazione delle possibili configurazioni riguardanti l'allocazione delle risorse ai vari carichi di lavoro, in modo da trovare quella corrispondente alla più power efficient. Per sfruttare al tecnologia RAPL siamo andati a sviuppare un tool stack che lavora a tutti e 3 i livelli della cononica pila di un ambienta virtualizzato, cioè: livello hardware, livello hypervisor e livello Virtual Machines

(VMs). Per far questo abbiamo usato una serie di meccanismi propri delle tecnologie di virtualizzazione in modo da ottenere privilegi sull'hardware, le così dette hypecall (simili per comportamneto alle syscall in un Operating System (OS) comune). Questo tools stack è poi stato sfruttatto nella fase di *Attuazione* del ciclo di controllo ODA in modo da definire e far rispettare il power cap. Sempre in questo stadio andiamo inoltre ad attuare la configurazione delle risorse decisa per questa iterazione del ciclo di controllo, ripartendo le risorse virtuali delle VMs su quelle fisiche. Nella fase di *Osservazione* invece siamo andati a sviluppare un sistema di monitoring agnostico che valuta come stanno performando le VM eseguite nel sistema tramite metriche hardware quali il numero di IR, recuperate dai Model Specific Registers (MSRs). Infine, nella fase *Decisionale* andiamo ad esplorare, tramite ricerca binaria in intervallo chiuso, la prossima configurazione delle risorse da esplorare in modo da massimizzare le risorse. In questa tesi mostreremo *XeMPUPiL* è in grado di raggiungere performance migliori sotto differenti power cap per diverse tipologie di carico analizzate (e.g., CPU-, memory- and IO-bound). Inoltre mostriamo anche come è possibile sfruttare la stessa metodologia per ribaltare il problema e quindi dato un SLA da rispettare cercare di minimizzare i consumi di potenza del sistema.

Il testo è organizzato come segue:

- Il Capitolo 1 fornisce una introduzione generale al lavoro;
- Il Capitolo 2 fornisce le definizione comuni usate in questo lavoro di tesi, insieme ad una descrizione nel dettaglio delle tecnologie sfruttate e da cui si è presa ispirazione per questo lavoro di tesi;
- Il Capitolo 3 definisce il problema affrontato dal lavoro di tesi e le motivazioni dietro ad esso;
- Il Capitolo 4 dà una visione d'insieme dello stato dell'arte in cui questo lavoro si inserisce;
- Il Capitolo 5 dettaglia la metodologia alla base di *XeMPUPiL*;

- Il Capitolo 6 presenta in dettaglio come è stato implemnetato l'orchestratore;
- Il Capitolo 7 presenta i risultati sperimentali ottenuti all'interno del lavoro di tesi;
- Infine, il Capitolo 8 riporta le conclusioni del lavoro, sottolineando il contributo e i possibili lavori futuri.

1

Introduction and motivations

Computing infrastructures changed considerably in the last few decades [31] moving from huge, private and centralized computing infrastructures, usually maintained by the client (e.g. universities, mid-range industries) of such infrastructures, that was at the same time both client and provider. Nowadays, such infrastructures are decentralized and easily scalable ones, decoupling the client from the provider. This change is due to a new demand coming from the market: accessing services everywhere and in an elastic fashion in order to keep the pace with the continuing evolving request [22]. Cloud computing has now become the leading paradigm, from Web services to batch and streaming computations, allowing companies to run their applications *"in-the-cloud"* instead of buying proprietary servers, providing a better response to the elastic and scalable needs of the datacenter clients. In this context cloud computing is an unprecedented opportunity for all the companies aiming either to grow fast or to consolidate their provided service. Thanks to this new computational paradigm is possible to move a lot of management and initial investment costs from the client to the cloud provider, which is now in charge to find ways in order to maximizes his/her profits by optimizing the usage of its hardware. In this context, *"virtualization"* enables cloud providers to run multiple applications on the same physical resources, still ensuring strong isolation to each of them [39, 8].

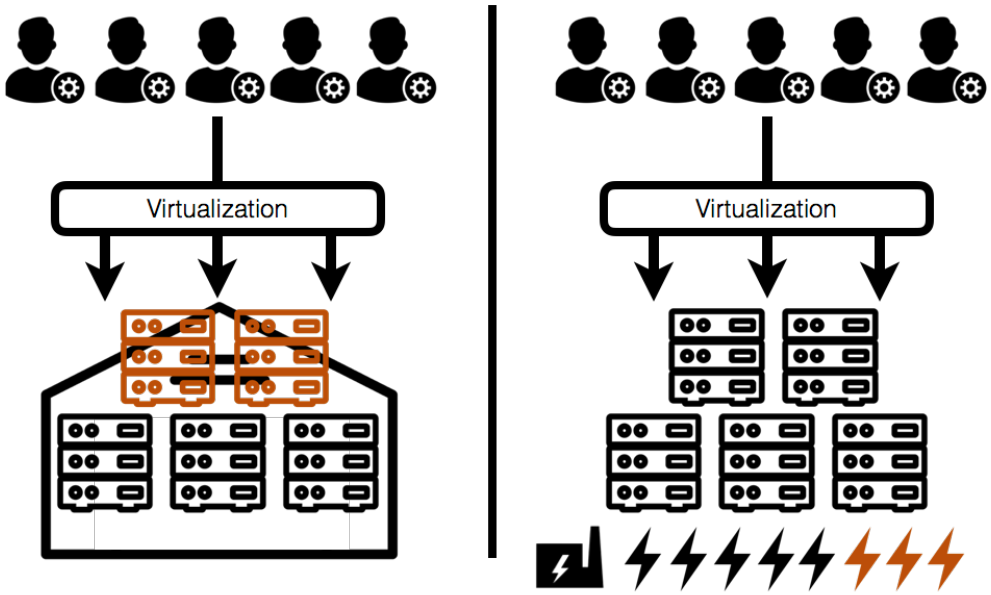


Figure 1.1: Major bottlenecks for current datacenters: on the left is represented the problem regarding limited physical datacenter available space, instead on the right the problem regarding energy consumption limits.

1.1 The challenge to growth

The growing number of users attracted by the rising of the aaS paradigm leads to an unprecedented number of workloads running on the servers composing the cloud, playing up to two major challenges regarding increasing the computational power of the datacenter facility: on the one hand while respecting the physical space constraints of the site on the other one while while respecting the contract stipulated with the energy provider. Both challenges are generated by the need of having more servers inside the datacenter in order to cope with the increasing number of workloads. The first challenge regards the physical space that such hardware occupies, which is limited inside a datacenter, as the left image in Figure 1.1 can suggest. The second one regards instead, the available power that such machines consume, since energy is not an endless resource and furthermore is defined in strict contract between datacenter owners and energy providers, as is suggested by the right image in Figure 1.1 . In particular the second challenge is the more concerning one as the study conducted by

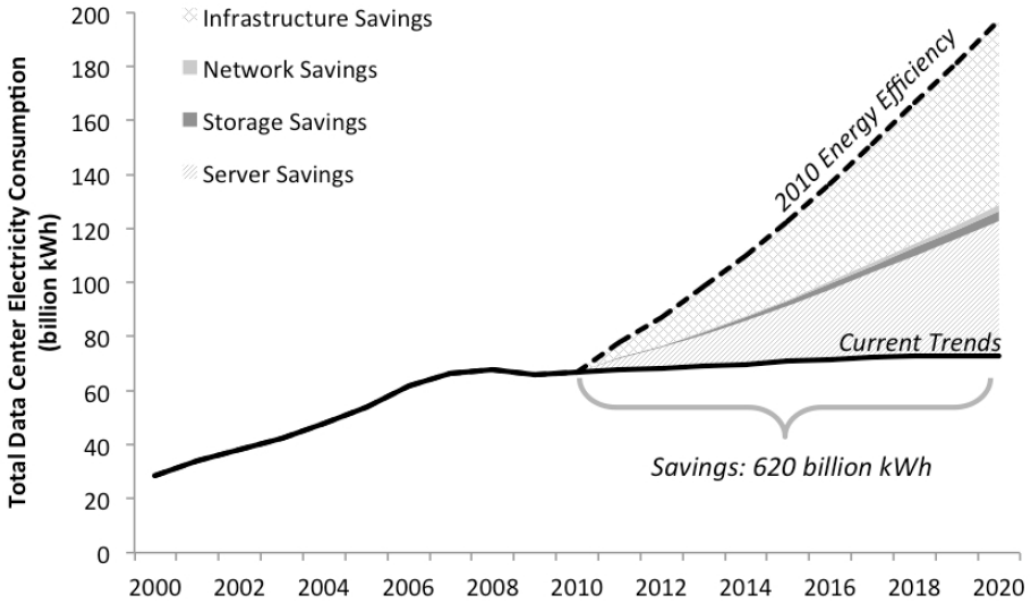


Figure 1.2: The chart shows past and projected growth rate of total US data center energy use from 2000 until 2020. It also illustrates how much faster data center energy use would grow if the industry, hypothetically, did not make any further efficiency improvements after 2010.

the US Department of Energy at Lawrence Berkeley National Laboratory shows in Figure 1.2. The chart presents a forecast of the power consumption trends in datacenter. The dotted line represents the power consumption prevision in case the same technology adopted in 2010 are still used nowadays. Instead, the black line at the bottom represents the power consumption forecast in case all the expected improvements in datacenter management technologies will be really effective. Then, the area delimited by these two lines represents a possible amount of power savings in a 10 year time window. In this area, it is also interesting to notice that the greatest amounts of savings are expected for infrastructure (more focused on material and electronic improvements) and servers.

Unfortunately, the need of having more and more computational power in a limited space can lead to a better utilization of the hardware platforms only if the hypervisor is able to perform a good consolidation of the tenants over the datacenter, as well as on the resources on every single machine [28, 42]. This task is made difficult by both hardware and software *heterogeneity*: the servers of the same datacenter may not be equipped with the same amount of memory and processors, as well as different tenants may be characterized by different

workload profiles (e.g., memory-bound, I/O-bound and/or CPU-bound).

Moreover, this scenario gets even worse when considering power consumption, a major concern for almost every digital system. Given the huge density of servers in modern data centers, the power grid may not be able to supply enough energy to run all of them at their peak performance, thus requiring tools and interfaces able to set a *power cap* on the whole system. To face this first requirement, Intel introduced the RAPL interface since its second generation of Sandy Bridge processors [16]: this interface enforces a strong and precise limit on the power consumption of a processor, i.e., the component that contributes the most on the *dynamic* power consumption of a server [44].

RAPL uses Dynamic Voltage and Frequency Scaling (DVFS) techniques to guarantee the desired power cap but is not aware of the impacts that these techniques have on the performances of the hosted applications. Of course, these performances need to be maximized even when a power cap is enforced: we want to find the most *power efficient* hardware configuration under a certain power cap, thus maximizing the performance-per-watt ratio. In order to accomplish our goal, a uniform metric of performance has to be defined, as well as a smart orchestration policy to guarantee the stability of the system as soon as the runtime conditions change. As the system may be composed of thousands of nodes, thus hosting hundreds of thousands of concurrently running applications, even a small optimization may lead to massive savings on the overall datacenter in terms of power consumed and, thus, money spent by the cloud provider.

In this thesis, we propose *XeMPUPiL*, a hybrid hardware and software power capping orchestrator for the Xen hypervisor (a common open source virtualization technology adopted by several datacenter providers), based on the PUPiL approach [45], that aims at enforcing two different and distinct policies: maximizing the performance of a running workload under a power cap and minimizing the power consumption given a SLA to respect. The main contributions of this thesis are the following:

1. we propose an *Observe* phase that takes into account a generic performance metric for all the hosted tenants, avoiding any instrumentation of the work-

loads;

2. we improved the decision phase of PUPiL, to deal with the resources available in a multi-tenant virtualized environment;

3. we implemented a new *Actuation* phase, to support all the *knobs* that Xen provides to control the resources assigned to each tenant.

The rest of the thesis is organized as follows:

- Chapter 2 gives the common definitions shared across this work, gives a description of the technologies and tools adopted in this thesis work;
- Chapter 3 introduces the problem we are coping with this thesis work;
- Chapter 4 describes the state of the art;
- Chapter 5 details the methodology behind *XeMPUPiL*;
- Chapter 6 digs into the implementation details of the orchestrator;
- Chapter 7 presents the experimental results that validates our approach;
- Finally, Chapter 8 draws the conclusions and presents the future directions of this work.

2

Background

In this chapter we are going to introduce some useful concepts that will help the reader in better understanding the work proposed in this thesis. In Section 2.1 the Xen project will be presented and information about virtualization will be provided. In Section 2.2 the *XeMPower* monitoring tool will be introduced in its key aspects. In Section 2.3 a full overview over the Intel RAPL interface for socket power management is provided.

2.1 Xen project

In this section we are going to introduce what is the Xen project, presenting some useful terminology that will let the reader better understand the next chapters of this thesis. The Xen hypervisor is an open-source type-1 or baremetal hypervisor, since it runs directly on the physical hardware resources. This allows it to run different instances of various operating systems in parallel on a single machine (usually called host). The main advantage of the Xen hypervisor is that it is the only type-1 hypervisor provided under open source license. Different commercial and open source application exploit it as their basis, examples of these applications are: security applications, server virtualization, desktop virtualization, Infrastructure as a Service (IaaS), embedded and hardware appliances. Indeed, different production virtualization technologies are powered by the Xen hypervisor, such as: Oracle VM Server [5] and Huawei FusionSphere [27]. In 2.1

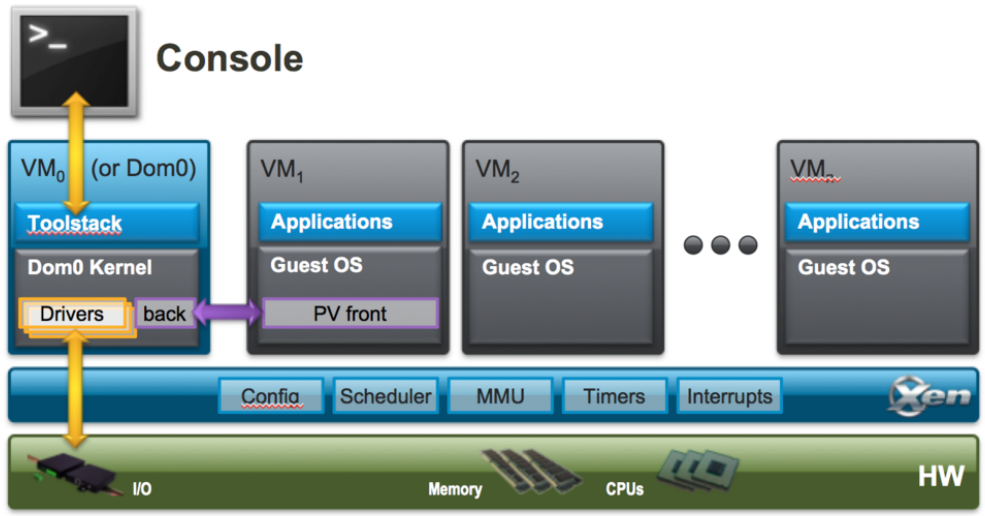


Figure 2.1: Representation of the Xen architecture

[7] is introduced a schema of the Xen Project architecture. CPU, Memory, and interrupts are handled by the hypervisor, since it lies directly on the hardware layer. After exiting the bootloader, the hypervisor is the first program running. The virtual machines run on top of the hypervisor. In Xen terminology a domain or guest is a running instance of a virtual machine. **Domain0** is a special domain containing the drivers for all elements in the system. The necessary control stack to manage virtual machines (e.g. creation, configuration, and destruction) is located in this domain. In detail, the components are:

The hypervisor is a thin[1] software level that lies directly on the hardware and is in charge to manage CPU, memory, and interrupts. It is the first program running after the bootloader exits.

Domains also called either Guest Domains or VMs, are virtualized spaces, each of them can run different OS and applications. Two kinds of virtualization are supported by the hypervisor: Paravirtualization (PV) and Hardware-assisted or Full Virtualization (HVM). On a single hypervisor, both guest types can be used concurrently. This kind of VM is defined as unprivileged domain (or DomU), the reason lies in the fact that it has no privilege to

[1]Xen hypervisor contains less than 150,000 lines of code

access directly hardware or I/O functionality, resulting in a total isolation from the hardware layer.

Domain 0, also called control domain, is a particular VM that has special rights, examples are: handling all access to the system's I/O functions, interacting with the other Virtual Machines and the capability to access the hardware directly. It also exposes a control interface to the system administrator, that allows the system control. It is not possible to use the Xen hypervisor without Domain 0, which is directly instantiated by the hypervisor after its initialization successfully terminates.

Command Line Interface (CLI) it is located in Domain 0 and drives a control stack (or Toolstack) allowing the management [2] of the running VMs.

Furthermore, will result useful for the reader to introduce some terminology related to the virtualization domain:

vCPU This is a virtual Central Processing Unit (CPU) assigned to a VM, usually known also as virtual processor. vCPUs permit multitasking to be performed sequentially in a multi-core environment.

pCPU It is a physical CPU, with all the circuitry and memory. It is capable of independent processing.

CPU-Pool The main idea behind CPU-pools consists in splitting into different pools the physical cores of the system. A separate CPU scheduler, with also different parameters, may be assigned to each of these pools. A pCPU can be assigned at any time to either no one or one of the defined pools. Similarly, a VM can be moved from one pool to another at any time, but must be always assigned to one and only one pool at a time.

2.2 XeMPower

In this section we introduce the *XeMPower* power monitoring tool [19]. XeMPower is a lightweight monitoring solution for Xen designed to: 1) provide pre-

[2] creation, destruction, and configuration

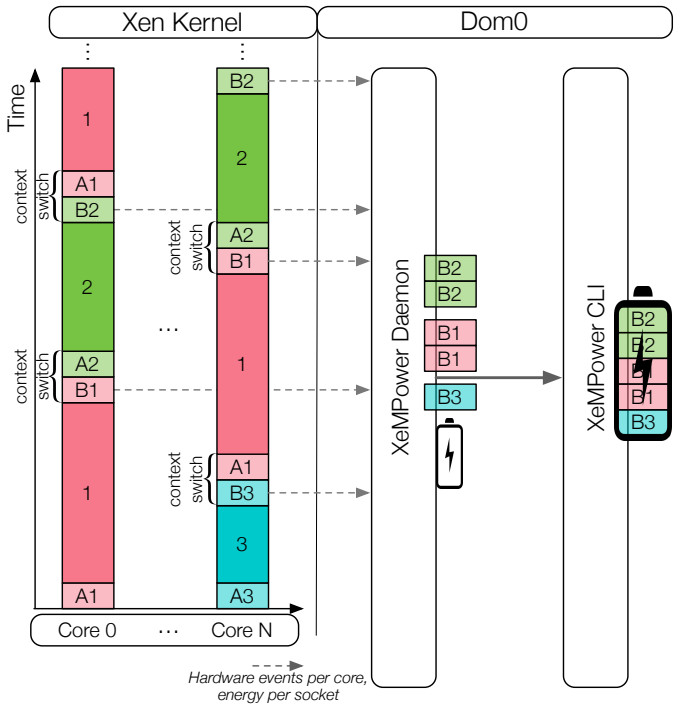


Figure 2.2: XeMPower design

cise attribution of hardware events to virtual tenants, 2) be agnostic to the mapping between virtual and physical resources, hosted applications and scheduling policies, and 3) add negligible overhead. Ferroni et al. approach uses hypervisor-level instrumentation to monitor every context switch between domains. More precisely, the monitoring flow proceeds as follows:

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

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

3. Steps 1 and 2 are performed at every context switch on every system CPU (i.e., physical core or hardware thread). The reason is that each domain may

have multiple vCPU. Socket-level energy measurements are also read (via Intel RAPL interface) at each context switch.

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

2.2 illustrates the monitoring flow described above. Steps 1 and 2 for domains 1, 2, and 3 are shown at every context switch on the left side of the figure. On the right side, steps 3 and 4 are performed by the XeMPower daemon and CLI program, both in Dom0.

2.3 Intel RAPL interface

RAPL is an interface provided by Intel consisting of non-architectural MSRs. Resources within each processor socket are divided into domains of power management. Usually these are the "Package domain", corresponding to the processor die, and the "Memory domain", corresponding to the directly attached Dynamic Random Access Memory (DRAM). Each domain has the following interfaces used to control its behaviour:

Power Limit Interface to specify power limit and its fine tuning such as the time window.

Energy Status Interface to retrieve information about the power consumption.

Performance Status It provides information about the effect due to the power limit. This interface is optional.

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.

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.

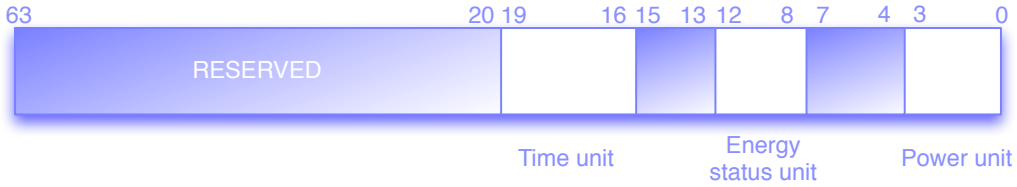


Figure 2.3: Representation of the MSR_RAPL_POWER_UNIT Register.

In this thesis work we targeted the first two, since they allow to set the power cap and subsequently to check it is correctly enforced. This interfaces are represented by three MSRs: the Power Unit MSR, the Package Power Limit MSR and the Package Energy Status. We will exploit the first and the last register in order to retrieve information on how the system is behaving. We need to read the right fields inside these registers and to aggregate the gained information accordingly to what is specified in the Intel Manual. Instead the second register will be written in order to define and to enforce the corresponding power cap. In the next sections we detail those register, their fields and how they are put together in order to enforce a power cap via hardware.

2.3.1 MSR Power Unit

As shown in 2.3, MSR_RAPL_POWER_UNIT is a 64 bits long register. It contains architecture specific information about the units adopted to measure power, energy and time. This register is used in read-only access mode, and it is composed by three fields:

Power Units 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. This information is contained in bits 3:0.

Energy Status Units Energy is measured in Joules and expressed in "number of energy units" inside the Package Energy Status register. As default this field is set to 10000b. This value indicates that each energy unit represents an increment of 15.3 micro-Joule. This information is contained in bits 12:8.

Time Units Time is measured in Seconds and expressed in "number of time units" inside the Package Energy Status register. As default this field is set to 1010b. This value indicates that each time unit represents an increment of 976 micro-seconds. This information is contained in bits 19:16.

The interesting information stored in this register is contained by the first two fields: power and energy. The register contains also info about time. This is not going to be useful, since there is no need to take into account information about this metric. It will be retrieved by the OS. Once the units are retrieved, it becomes possible to assess a relation between physical values and how they are represented at architectural level. These transformations are designed by 2.1 [21], where PU stands for Power Unit.

$$unitInWatts[WATTS] = \frac{1}{2^{PU}} \tag{2.1}$$

2.1 represents the relation between a power cap expressed in Watts and the number of power unit representing it. The initial value of this register is 011b, equivalent to 3 in base 10, applying 2.1 the resulting value will be $\frac{1}{2^3}$ so $1/8$. This explains why the default value corresponds to $1/8$ Watts. For what concerns energy the approach is the same. This time the equation ruling the relation between physical values and architectural ones is expressed by 2.2,where ESU stands for Energy Status Unit.

$$unitInJoule[JOULE] = \frac{1}{2^{ESU}} \tag{2.2}$$

This register is read just one time. It is read during the initialization phase previous the beginning of the ODA cycle. This is sufficient since once the metric (i.e. units) are retrieved the will remain the same during all the computation.

2.3.2 MSR Package Power Limit

As shown in 2.4, MSR_PKG_POWER_LIMIT is a 64 bits long register. It provides an interface in order to define and enforce a power limit. It allows to specify two power limits, corresponding to time windows of different sizes. The fields composing this 64 bits long register are:

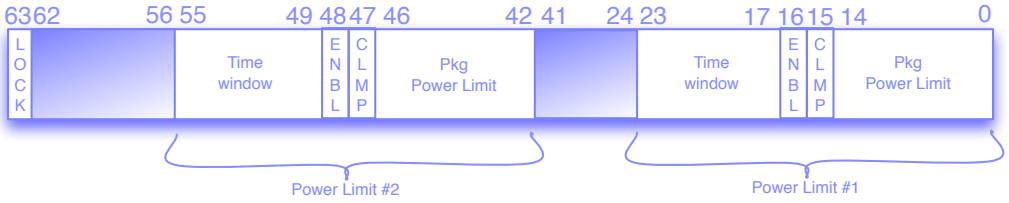


Figure 2.4: Representation of the MSR_PKG_POWER_LIMIT Register

Lock If this field is set to 1, the power limit settings are static and un-modifiable until next RESET. Corresponding to bit 63.

Package Power Limit The value of this field is specified in units, retrievable by the register presented in Section 2.3.1. Respectively bits 14:0 and 46:32 for the two imposable power limits.

Enable Once this field is settled to 1, the specified power limit is enforced.Respectively bit 15 and 47 for the two imposable power limits.

Package Clamping Limitation If this field is set (i.e. different from 0), the socket is allowed going below OS-requested P/T state. Respectively bit 1 and 48 for the two imposable power limits.

Time Window This field indicates the time window for the specific power limit. Respectively bits 23:17 and 55:49 for the two imposable power limits.

In the proposed approach only the "Package Power Limit" and the "Enable" fields are of interest, since the time window will be set to infinite and the "Lock" field will be unset in order to allow runtime modification at the power limit. In order to define a power limit the relation expressed in 2.1 is needed. This is necessary due to that physical measures are expressed in architectural specific units metrics inside the register. To define a cap in power units inside the correspondent field, 2.3 must be applied.

$$powerUnits = \frac{powerCap[Watt]}{\frac{1}{2^{PU}}} \tag{2.3}$$

2.3 states that in order to retrieve the number of power units corresponding to a physical value, it is necessary to divide it by the physical value corresponding

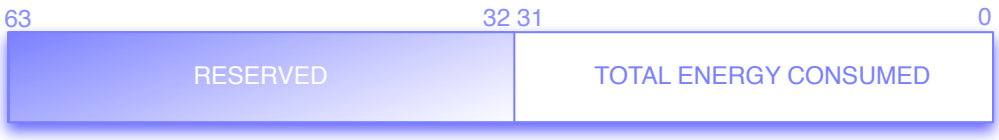


Figure 2.5: Representation of the MSR_PKG_ENERGY_STATUS Register

to a singular power unit. Also in this case the register is written just one time. This happens during the initialization phase too. The number of power units calculated is written in the corresponding field of one of the two power limits, than the time window is set to infinite and the enable bit is set to 1. All this fields are written at the same time. The fields corresponding to the other power limit are not written except for the enable bit that is set to 0 just to ensure that the second power limit will be disabled.

2.3.3 MSR Package Energy Status

The information contained in the register displayed in 2.5, represents the total energy consumed. This is an accumulator register, where the count starts from the moment it is cleared. "ENERGY_STATUS_UNIT" is the unit adopted to represents the value contained in this field, as specified by register "MSR_RAPL_POWER_UNIT" presented in Section 2.3.1. The total amount of energy consumed in a time window is designed by 2.4.

$$energyConsumed[Joule] = \text{TOTAL\_ENERGY\_UNITS} \times \frac{1}{2^{ESU}} \tag{2.4}$$

2.4 states that the energy consumed by the socket corresponds to the amount of number of energy units read from the register times the Watts value of a single energy unit, expressed in 2.2. Then, applying the physical relation between energy and power, it is possible to retrieve the power consumption over a time window, as stated in 2.5.

$$powerConsumed[Watts] = energyConsumed \times \Delta t \tag{2.5}$$

The time window is retrieved via OS calls. At the first time-call the register is erased, its bits are written to 0s. After a fixed time interval a second time-call is done, at this point the register is read and with the 2.4 and 2.5 the power consumption is retrieved. Δt in this case corresponds to difference between the time measured during the second time call and the first one. This register is used in order to periodically check that the given power cap is effectively enforced.

3

Problem Definition and goals

In this chapter we are going to describe the problem addessed by this thesis work and the main goals the proposed solution should achieve. In Section 3.1 we will present the power consumption problem in current datacenters, in particular how it can be addressed under two different points of view: respecting power budget while optimizing workloads performances and respecting a given SLA while optimizing power consumptions. In Section 3.2 the challenges of power capping in virtualized environments will be presented. Finally, in Section 3.3 we will introduce the goals of the proposed methodology.

3.1 Power consumption and power cap

Modern processors are constrained by dark silicon, as the abundance of transistors enables them to draw more power than they can safely sustain [18, 43]. This phenomena affects several classes of processors, from mobile System on Chip (SoC) to datacenter processors. On the one hand, mobile processors like the Exynos 5 (used in the Samsung Galaxy S4 phone) has a 5.5W peak power draw – nearly 2× its sustainable heat dissipation [41]. On the other hand, future exascale supercomputers have a predicted operating power budget of 20 MW [12], making power management a central challenge of supercomputer operating systems. In between this two borderline cases we have the datacenter challenge, since the power consumption estimated (only for servers) in a modern datacen-

ter is around 1MW [34]. Server compute performance has been increasing by a factor of three every two years, however, energy efficiency is only doubling in the same period. This means computational performance increased by a factor of 27 between 2000 and 2006. Energy efficiency has gone up as well, but by only a factor of eight during the same period. This means that while power consumption per computational unit has dropped dramatically in this six-year period (by 88 percent), the at-the-plug power consumption has still risen by a factor of 3.4 [13]. A constant rate of IT hardware spending results in increasing hardware power consumption at-the-plug, which, in turn, results in rapidly escalating site electric utility costs as part of the enterprise IT operating expense. This increase of the costs typically happens when an enterprise datacenter runs out of power and/or cooling capacity, and an unplanned (or sub-optimally planned) site capital investment is required to increase capacity. For large-scale enterprise data centers, these CapEx investments can now be in hundred million dollar increments. The physical constraints and the monetary constraints that affects modern enterprise datacenters create the need for power control systems which guarantee that the processors, the servers and the whole computational infrastructure operate within a strict power cap.

3.2 Virtualization challenges

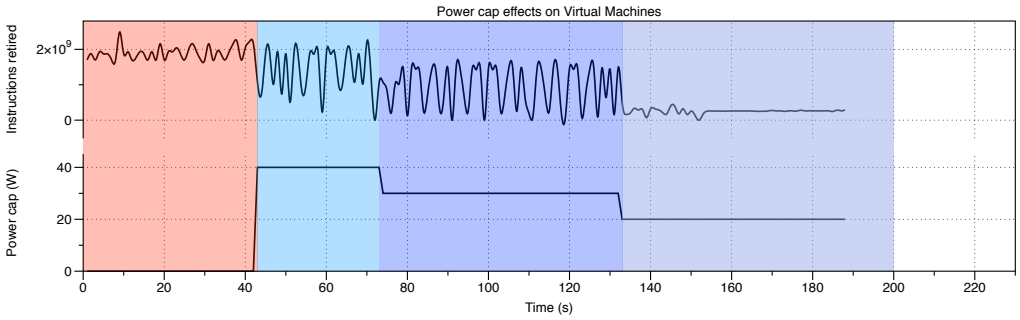


Figure 3.1: The graph shows how the performance (measured in IR) of a domain in Xen, running a high parallel application is affected by enforcing a power cap on the system.

The first technique (now the leading one) adopted in datacenter in order to increase the efficiency of the machines is virtualization. Thanks to this technology

it is possible to isolate and host concurrently multiple users on the same physical machine, enhancing in this way the utilization of the datacenter resources. However, when hardware resources are affected by power constraints, the virtualized ones are affected as well. The graph in Figure 3.1 shows how different power constraints affect the performance returned by a VM running in the virtualized environment. The application running in the VM is a high parallel random number generator taken from the NAS Parallel Benchmarks (NPB). It is possible to notice that in a system, where no power constrains (region 1) are defined, the performance (mesured as the number of IR) are optimal. The challenge rises when a power cap must be enforced on the system as shown in region 2. In this case it is possible to notice a slight downgrade in performance. This behaviour is even more clear when the power cap becomes stricter, like in regions 3 and 4. In these cases the performance downgrade is really significant. On the other hand it is also true that being able to exploit a power cap technique like the one in the graph (in this case was used RAPL) ensures a precise and strict power control of the system. Hence, the challenge consists in being able to apply this kind of power control and at the same time finding a way to reduce the amount of lost performance. Another challenge under this path is to find a strict power cap which allows a VM to consume less resources and power while respecting on a SLA.

3.3 Goals

The goals of this thesis work consist into the development of an orchestrator for workloads running in a virtualized environment, which will be based on the following concepts:

Timeliness At any change of the power cap for the system, the orchestrator should be able to enforce it in the strictest and quickest way possible, avoiding any oscillatory behaviour.

Efficiency Depending on the different running workloads in the system, the orchestrator should be able to identify which is the best resource set to assign

to each workload in order to maximize its performance.

Multi-tier The orchestrator should be able to work and access at each layer of the common virtualization stack. It should communicate with the hardware in order to manage the power consumption via RAPL, with the hypervisor in order to observe the running workload performance and with the VM tier in order to manage the resource assignment.

Workload agnostic Finally, the orchestrator should be able to provide all the listed behaviour without requiring any workload instrumentation, in order to ease the developers job and the spreading of such methodology.

4

State of the Art

Due to the increasing interest in the field of power monitoring tools, several works were produced in these years. They can be classified into two families of approaches: hardware approaches and the software approaches. The formers are built upon the concept of timeliness, trying to enforce the cap as faster and stricter as possible, exploiting hardware control circuits. The latter, instead, are built upon the concept of efficiency, searching for the best configuration possible in order to maximize the performance while reducing the power consumption. In this chapter we will present the analysis of the State of the Art that is of interest for this thesis work, focusing on the pure software/hardware and also on hybrid power capping approaches. In Section 4.1 a brief introduction about the methodology adopted in order to classify the State of the Art (SoA) is presented. Then, in Section 4.2 we are going to introduce in detail the hardware approaches and the concept of timeliness. Instead in Section 4.3 the software approaches and the concept of efficiency are presented. Finally in Section 4.4 a novel approach exploiting at the same time both hardware and software techniques, the so called hybrid approach, is analysed in detail.

4.1 Classification criteria

In several works proposed in literature and introduced in the next sections the approaches presented always exploit hardware techniques alongside soft-

ware techniques. This is due to the fact that hardware and software in the field of power management are strongly related. The criteria adopted to distinguish between pure software and hardware approach is based upon which of the two aspects is more prominent, which is exploited and used more and also which is the problem addressed. Usually the software approaches are typically used in problems where the main constraint is represented by the performance of the workloads with respect to a power cap, instead hardware approaches are used in problems where the constraint is represented by the power budget available. The novelty introduced by the hybrid approach consists basically into the fact that both approaches are equally valuable. In detail, it exploits software techniques in order to achieve better performance and hardware techniques to obtain a strict power control in a problem where the limit consists in the constrained power consumption. This approach results in real synergy able to exalt the best characteristics of both approaches, erasing completely the weak spots of the two techniques when adopted individually.

4.2 Hardware approaches

All the power capping techniques implying the use of on socket modules or interfaces in order to enforce a cap, can be classified as hardware approaches. These ones usually exploits socket control circuits in charge to manage the chip resources. This group of techniques guarantees that *timeliness* is reached, where timeliness is meant as the speed at which a new cap can be enforced. In general hardware reacts faster than software, thus, timeliness is achieved thanks to relatively simple circuits controlling key power indicators like processor voltage and frequency. To the best of our knowledge, the hardware approaches proposed in literature can be classified into three families: (i) DVFS, (ii)CPU Quota and (iii) RAPL.

DVFS This techniques exploits socket's frequency and voltage controllers. The main idea is to reduce dynamic power consumed per time unit via fre-

quency and voltage decrement using 4.1,

$$DynamicPower = C \times V^2 \times A \times f \tag{4.1}$$

where C is the capacitance being switched per clock cycle, V is the voltage, A is the Activity Factor indicating the average number of switching events undergoing between the transistors in the chip and f is the switching frequency. The technique ruled by 4.1 is profitable only if no concerns about the performance of the running workloads are taken into account. In the work proposed by Deng et al. in 2012 [17] the authors present MultiScale. This is the first technique that tries to manage DVFS in systems presenting multiple memory channel, devices, and Memory Controller (MC). This approach consists into monitoring workload bandwidth requirements across MC, under OS control. The information retrieved from the monitoring stage is then used by a heuristic in order to select the best frequencies combinations. These combinations try to minimize the overall system power consumption, while respecting the user-specified per-application performance constraints Instead in the work of Horvath et al.[26], the authors address DVFS in multistage service pipelines, unlike previous works that addressed DVFS on individual servers and on load-balanced server replicas.

CPU Quota This technique consists in assigning a limited amount of computational time (i.e. a quota) to the workload. Then the processor is in charge to schedule each workload in order to respect the allocated quota. Reducing the time will lead to a postponed workload, thus reducing the overall dynamic power. This approach is feasible for workloads with no priority and no strict Time To Live (TTL). The work presented by Fornaciari et. al in 2014 [32] represents a double value in the field of hardware power management and monitoring. The authors demonstrated how a user-space run-time resource manager, based upon CPU quota technique, can exploits Hardware Performance Counters (HPC) as performance metric. The obtained results consists in a resource manager able to optimize both energy consumption

and workloads execution time.

RAPL This interface provides a way to set power limits on processor packages and DRAM. The need behind this interface lies in the necessity to provide, to programs in charge to dynamically monitor and control, a mechanism in order to limit the max average power, matching the imposed power budget. Furthermore, power budgeting across the rack distribution is enabled by power limits in a rack. Power limits can be reassigned based on use and workloads, by dynamically monitoring the feedback of power consumption The control over the power limit can be defined also on short and long term averaging windows, whose size and limit can be defined dynamically. Also in this case, the proposed technique is profitable only if no performance interest is taken into account for the running workloads. The survey published in 2012 by the Intel sandy-bridge development team [38] provides useful information about the new features introduced in the sandy-bridge processors family. The Intel developer manual [21], volume 3B, section 14.9, shows that RAPL can be defined as an interface providing mechanisms to enforce power consumption limit. The usage of those interfaces has a huge importance for both client and server platforms. The provided interfaces ease the power limit control, especially in server platform adopted in data-centers, exploiting the following power and thermal related usages:

- Platform Thermal Management: this is a robust proactive or reactive mechanism to oversee platform and component thermal behaviour.
- Platform Power Limiting: system's power consumption is managed by a deterministic control model.
- Power/Performance Budgeting: enables the concept of efficiency, meaning to control the power consumed and the performance delivered within and across platforms.

The RAPL interfaces presents multiple domains of power managing within the socket, where each processor lies. In general, these RAPL domains may

be composed by:

- Package domain, which represents the processor die.
- Memory domain, which contains the directly-attached DRAM.

In order the case of multiple sockets, the consumed power can be managed via RAPL, programming individual power limit for each processor. The definition of specific RAPL power domain across multiple sockets is not supported yet. Detailed information about registers being part of this interface can be found in Section2.3.

4.3 Software approaches

Software approaches are thought in order to address the challenge represented by the concept of *efficiency*. Efficiency is meant as the performance delivered under the power cap. Software approaches have greater efficiency compared to hardware ones. They find the resources configuration providing the highest performance within the power limit. This capability is possible thanks to the time that this family of techniques spends during the exploration phase. Where all the possible configurations of the resources are studied and tested, considering complex interactions between them. In this way is possible to solve the constrained optimization problem, defined as finding the resource configuration which delivers the highest performance, while respecting the define power cap. Also in this case, at the best of our knowledge, the software approaches can be classified into three families: (i) thread migration, (ii) race-to-idle and (iii) model based monitoring.

Thread migration In this technique the different processing nodes are configured with various P and C states arrangements, one for each node. These arrangements are made in order to respect the overall power cap. Then a software technique analyses the threads requirements and resource requests and tries to pack them upon similarity and to schedule them on the computational nodes accordingly to a defined efficiency policy. This ap-

proach requires time in order to find the best packing and placing and it is profitable only if the cap is not strict, since this approach allows time windows where the cap is not respected. In this direction Cochran et al. presented in 2011 [15] an interesting work. Their approach, called "Pack & Cap", consists in a control technique, which aims to maximize performance while respecting a given power budget. This goal is achieve through a system designed to make optimal and thread packing control decisions. This chain of decisions leads to the selection of the optimal operation point, found by a Multinomial Logistic Regression (MLR) classier trained using a large amount of HPC, power, and temperature characterization data. They proved a decreasing of 51.6% of workload energy consumption compared to existing control techniques. On the other hand their most significant limitation consists in the fact that only the 82% of the time the power cap is respected.

Pacing This technique addresses the problem of power consumption under a different point of view. In this case the challenge is formulated as a minimization problem, where the objective function to minimize is the power consumption and the constraints are the application performance. The main idea in this case is to avoid computational peaks followed by idle states, in exchange trying to maintaining as constant as possible the busied computational resources, avoiding huge excursion in the processor's behaviour. This technique demonstrated that maintaining a constant pace reduces the power consumption, but this approach is profitable only if the power consumption is not though as the main constraint for the system. The work proposed by Hoffman et al. in 2013 [23] showed a detailed survey over different pacing heuristics, demonstrating their benefits and their constraints. In particular those heuristics can be divided into three families:

- *Race-to-idle*[11, 10, 23, 35]. In this approach all the resources are let available for a workload until it completes. Their consumption needs a runtime optimization , in order to limit the racing time. The aim of

this technique is to let the job to complete as quick as possible. In this way the time in which the system can stay in idle state waiting for another job can increase, reducing the power consumption.

- *Pace-to-idle*[23]. This is a simple, but effective heuristic. Its simplicity consists in the fact that no optimizations at runtime are required. When a task enters the system, it makes all resources available, going into idle state when the task completes.
- no-idle[23]. This strategy consists into finding a scheduling of the job tasks able to limit, and in the best scenario to eliminate, the idle states. Hence avoiding expensive transition phases between idle and race, saving the dynamic power necessary for this transition.

Model Based Monitoring In this set of techniques the main goal consists in reducing power consumption in datacenter [26, 33, 40] or in increasing the battery life of embedded systems [20, 30, 36]. Model based on previous observed behaviour of the application are used in order to predict the power and the resources required by a new unobserved running application in order to fine tune the resources' configuration, saving power. Once again this approach is profitable only if spending time in training the model is an affordable cost and if the power cap is not a strict requirements, but only a guideline.

4.4 Hybrid approaches

These approaches are defined *hybrid* due to their double nature. In particular the two natures work sinergically and concurrently to achieve a better technique where the strong points of one approach compensate the weak ones of the other. The two natures are the already presented hardware and software approaches. In detail, for a generic hybrid approach, the hardware technique is in charge to impose the given power cap as fast as possible regardless the application performance obtained, hence respecting the concept of timeliness. Concurrently a software approach begins to explore the space of feasible configuration in order

to maximize the performance while enforcing the power cap, hence respecting the concept of efficiency.

4.4.1 *PUPiL*

Among all the proposed works in literature, the one proposed by Zhang and Hoffmann [45] is the most remarkable one, since it has the same goal that we want to address: maximizing the performance and at the same time strictly respecting a given power cap. *PUPiL* is thought as an orchestrator for applications in a Linux bare metal OS. It totally embraces the definition of hybrid power consumption management technique. Its approach is composed by a software part (i.e. an ODA control loop) and an hardware one (i.e. Intel RAPL interface). By exploiting this hardware technique, it is possible to notice that enforcing the power cap simply consists in writing the registers presented in Section 2.3. In a bare metal Linux environment these registers are easily accessible by opening in read mode the files abstracting the CPUs. Then applying some transformations accordingly to the information provided by the Intel manual [21] its possible to impose a strict power cap for the entire socket. The *PUPiL* authors developed a simple, but really effective tool written in C in order to impose a power cap to address this challenge, exploiting all the abstraction that a bare metal native Linux OS provides to its users. For what concerns the software approach, it is composed by an ODA control loop written in Python. During the initialization phase before the beginning of the loop the first operation consists into invoking the tool in charge to enforce the power cap. Once it is defined the application is launched assigning it all the available computational assets. At the best of our knowledge in *PUPiL* the only resources managed, and hence assigned, is the number of cores on which running the application. Then the ODA control loop enters in its first iteration: the *observe* phase. In order to monitor the workload's performance the authors decided to instrument each application running under *PUPiL* orchestrator. They exploited the Heartbeats [25] library to retrieve information about application performance, in particular the throughput of the work. They also slightly modified the mentioned library in order to retrieve a throughput per

Watt metric in order to evaluate how the application is performing. This information is retrieved by the Hearbeats Application Programming Interface (API) during the observe phase and then transmitted to the decide phase. In this phase the performance regarding the current configuration of the workload is updated, then a new arrangement (i.e. a not already acted one) of the resources is decided to put in practice. The decision is lead by a binary tree search based algorithm. At the end of this iteration of the control loop, the decision is passed to the *act* phase. In this phase the decision is put in action via the Linux CLI commands providing a useful tool to change, while running, the number of cores assigned to a process, in this case to the running application. Even though the approach proposed by *PUPiL* is effective, we identified two non-negligible limitations of the proposed solution: first, the applications running on the system need to be instrumented with the *Heartbeat framework* [25, 24], in order to provide a uniform metric of throughput to the decision phase; second, the tool is meant to work with applications running bare-metal on Linux. Both these conditions might not be met in the context of a multi-tenant virtualized environment, in which a virtualization layer allows the execution of multiple workloads and ensures isolation to each of them. This is the case of the *Xen hypervisor* [9], a bare-metal type-1 hypervisor widely adopted in real production environments [6], that runs directly as an abstraction layer between the hardware and the hosted virtual machines, called *domains* in the Xen terminology. It is based on a microkernel design, providing services that allow multiple operating systems to concurrently run on the same hardware. A privileged domain, called *Dom0*, is in charge of managing the *DomU* unprivileged domains. In this context, the high isolation of each tenant, seen as a *black box*, makes any instrumentation of the code of the hosted applications not feasible in a real production environment.

5

Methodology

In this chapter we are going to introduce the methodologies adopted inside *XeMPUPiL* and the goals of the proposed approach. In Section 5.1 a brief introduction of the proposed approach is presented. In Section 5.2 a short comparison between the ODA control loop and a well known maximization problem solver technique is presented in order to ease the understanding of the overall approach. Finally in Section 5.3 the ODA methodology is explained in detail.

5.1 *XeMPUPiL*: a bird's eye view

XeMPUPiL is a hybrid power-aware orchestrator for the Xen hypervisor, since it exploits software and hardware techniques to achieve power control over the system. The work presented in this thesis takes inspiration from the idea exposed by Zhang and Hoffman [45] and detailed in Section 4.4, which proposes an hybrid power capping technique for application running in a native Linux environment. In this work we propose a new methodology based on *PUPiL*. We target a virtualized environment such as Xen, addressing all the problems and the challenges related to the isolation between the running jobs inside the guest OS and the underlying virtualized hardware.

The orchestrator is obtained thanks to an ODA control loop. It checks the running workloads and how they are performing, thanks to hardware performance metrics. After that it explores the space of all the feasible and interesting configu-

rations in order to find the one providing the best result for the defined objective function. These tasks are pursued exploiting the Intel RAPL interface ensuring that the power constraint over the system is respected. *XeMPUPiL* aims also at enforcing two different and distinct policies which are two faces of the same coin. The first one consists in the maximization problem concerning the performance function as objective function and power consumption power cap given as a constraints.Instead, the second one is nevertheless the dual problem of the previous one, where the objective function to be minimized is the power consumption and constraints are given over on SLA regarding workload performances. In this way, by just changing the decision policy inside the ODA loop, is possible to shift from one to another, since finding a methodology solving the first problem ensures a methodology able at least to provide an upper bound (*weak duality*) or in the best case an optimal solution (*strong duality*) also for the second one.

5.2 ODA as a gradient ascending algorithm

Data: a point $P(x_1, x_2, x_3, ..., x_n)$ and a concave N dimensional function f
Result: the point $P(x_1, x_2, x_3, ..., x_n)$ corresponding to the maximum
tmpPoint $\leftarrow$ P;
result $\leftarrow$ $f(\texttt{tmpPoint})$;
repeat
  tmpResult $\leftarrow$ result;
  tmpPoint $\leftarrow$ tmpPoint + $\gamma \bigtriangledown f(a)$;
  tmpResult $\leftarrow$ $f(\texttt{tmpPoint})$;
until *tmpResult - result* > 0;

Algorithm 1: Pseudocode for a gradient ascending algorithm

In the proposed approach the ODA control loop (Algorithm 2) was inspired by a gradient ascending algorithm (Algorithm 1). This specific formulation fits well with the proposed problem, since we are trying to solve a maximization [minimization] problem, hence finding the global maximum [minimum] of a target function. The objective function to maximize is the overall performance in our first case, instead in the second one the objective function to minimize is the power consumption. In order to express the problem under this formulation

strong assumptions were made:

- The performance function is a concave function, hence it contains a single global maximum, a unique point convergence;
- The power consumption function is a convex function, hence it contains a single global minimum, also in this case, a unique point convergence;
- Each resource is independent from the others, this allows to study how changing the amount of resource assigned to the workload modifies the target function.

Under these assumptions a gradient ascending algorithm follows the gradient in order to find the global maximum [minimum]. This approach can be divided in four phases:

Initialization A starting point (i.e. a configuration) according to the constraints defined in the problem formulation is selected, and the gradient is set to $-\infty$.

Function evaluation The target function is evaluated for the given point. The gradient is also calculated taking into account the difference between the old value of the target function and the new one.

Point update The new gradient is analysed. If it represents an increment [decrement] of the target function than the point is incremented by predefined fixed step. If there is no improvement then there is no increment and the convergence point is found.

```
Data: initial state of the system
Result: the best state according to the decision
configuration ← initialization();
repeat
    state ← act(configuration);
    metrics ← observe(state);
    configuration ← decide(metrics);
until configuration doesn not change;
```

Algorithm 2: Pseudocode for a generic ODA control loop

Repetition If no convergence was found, the new point is used to evaluate the function and repeating the second and third steps.

These phases can be easily mapped over the steps being part of a common ODA loop. In our approach we decide to proceed with the following mapping: the "Initialization" phase is totally mapped over the act stage, the "Function evaluation" one is partially mapped over the act (setting the point), observe (evaluate the function) and decide (calculate gradient) stages and at the end the "Point update" phase is totally managed by the decide stage. Obviously, the two methods are different, since they are based upon two different methodologies. The gradient ascending technique is based on a mathematical model, where the behaviour of the function is studied and analysed through the gradient, hence the derivative, resulting into an informed choice about where to move next. Instead, the ODA control loop, structured as in our case, is based upon a model free heuristic that observes how the target function changes, guessing where to move next. Further details about the methodology adopted thanks to this parallelism are provided in the next sessions.

5.3 *XeMPUPiL* ODA control loop

In this section we are going to present the software approach adopted in order to reach the properties of efficiency and timeliness. This technique is used in order to fine tune the resources assigned to each domain. The formulation is designed in order to support both maximization and minimization problems. In the first case, given a power cap as a constraint, the software approach maximizes the performance of the running application. Instead in the second one, given a SLA as a constraint,the software approach minimize the power consumption of the running applications. What we are going to use is the so called *PUPiL* approach. *PUPiL* was meant to be used in a bare metal OS environment, hence it was necessary to adapt its phases to work in a virtualized environment, in particular the observe, decide, act phases being part of its ODA control cycle as in 5.1. An ODA loop is a strategy planner technique. It is composed by three steps (i.e. Observe,

Decide, Act) plus an initialization phase. Once a first version of the strategy is put in practice during the initialization, the loop starts. During the observation phase feedbacks on how the running strategy is behaving are gathered. Then the information is passed to the decision step, where one or more decisional policies exploit it in order to take a choice on how to modify the strategy. Finally the new plan of action is communicated to the acting step, which is in charge to put it in practice. Then a new observation phase starts again. Furthermore to maintain this approach as portable as possible, we decided to implement the orchestrator logic at the highest level possible. To this aim, we inflated the control logic inside dom0, given that this VM is the first one instantiated in *Xen* every time the hypervisor is initialized. Moreover, this VM is the only one with privileged access to the underlying hypervisor. Arranging the orchestrator in such position allows to exploit the intrinsic privileges of dom0 inside the *Xen* architecture, since this domain can monitor other domains and also provide a CLI to manage the hypervisor and do some resource assignment. A brief description to the high-level flow is given:

- *XeMPUPiL observes* the power consumption of the system and a set of hardware events of interest for each running domain;
- the traced events are then used as metrics of performance, in order to *decide* which hardware configuration is the most power efficient for the current workload;
- finally, the *actuation* phase sets the system to the best configuration found, to maximize the performance under the desired power cap enforced through the RAPL interface.

In this section, we present the design and the implementation of the three ODA loop phases, describing the challenges faced while working in a virtualized environment.

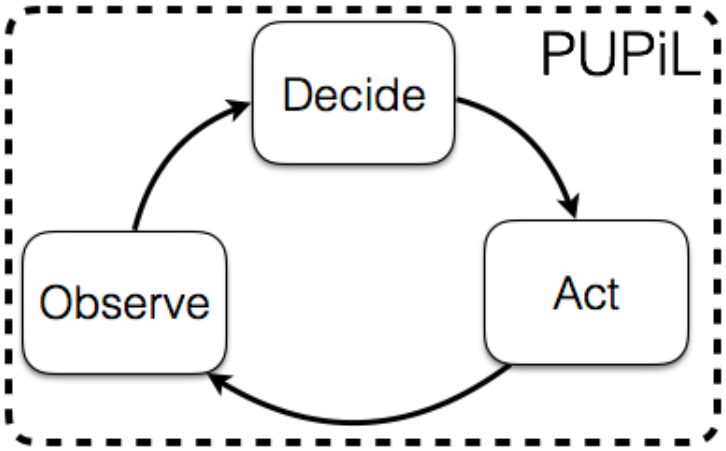


Figure 5.1: ODA logic composing the PUPiL orchestrator

5.3.1 Observe

Recalling the parallelism between this step and the phases of the gradient ascending algorithm, in this stage the goal is to evaluate, hence retrieving information about, the target function; in our case the performance. The observation phase is necessary in order to retrieve information about the system, and to understand how it is behaving under the current configuration. The main challenge consists in avoiding any instrumentation of the application in order to retrieve metrics on how it is performing. Addressing this challenges will lead us to obtain an approach which is as general and portable as possible, furthermore this does not require any additional effort by the application developers. Originally *PUPiL* was designed targeting workloads that are instrumented via "Heartbeats" library. In our adaptation we decided to change this approach, avoiding the need of workload instrumentation. We decided to use hardware event counters as low level metrics of performance, exploiting the Intel Performance Monitoring Unit (PMU) to monitor the amount of IR accounted to each domain in a certain time window. Among all the available hardware events that can be monitored, we chose to count the IR events on purpose, because these give an insight on how many microinstructions were completely executed (i.e., that successfully reached the end of the pipeline) between two samples of the counter, thus representing a reasonable indicator of performance [1] . The challenges here are three:

Figure 5.2: Workflow diagram leading the ODA cycle

1. provide precise attribution of hardware events to virtual tenants;
2. be agnostic to the mapping between virtual and physical resources, hosted applications and scheduling policies;
3. add negligible overhead.

In order to cope with these challenges we took inspiration from the *XeMPower* monitoring tool already being part of the Xen distro and developed by Politecnico di Milano in collaboration with the SwarmLab at University of California at Berkeley. In order to access the PMCs containing information about the IR we faced the necessity to work in hypervisor space Thus, we instrumented the Xen scheduler in order to read those values from the hardware registers and to empty them at each context switch. Then we exploited the deamon provided by *XeMPower* as is, in order to gather the information coming from the scheduler, to aggregate according to a per domain policy and to present them in a shared-memory region that will be read from the observe phase. Finally the retrieved information is sent to the *Decide* phase in order to drive the current decision policy.

5.3.2 Decide

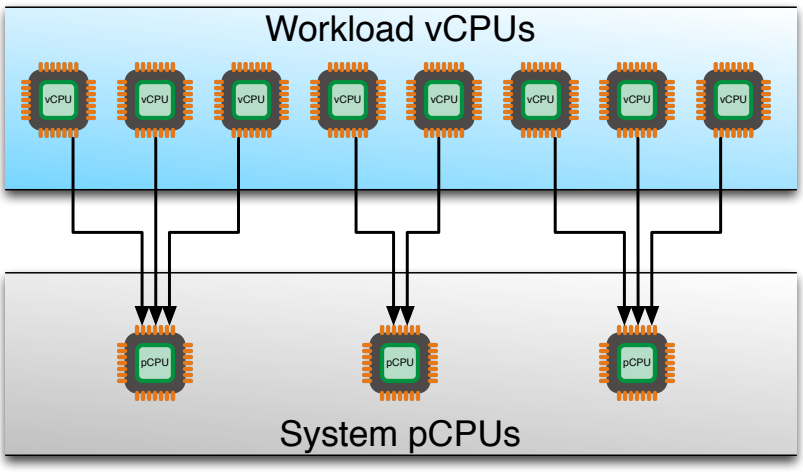


Figure 5.3: Graphical demonstration on how the vCPU are pinned over pCPU according to 5.1

Recalling again the coupling between this step and the phases of the gradient ascending algorithm, in this stage the goals are multiple: calculating the gradient

is the first one and updating the point, so selecting the next configuration to test is the second one. The gradient is trivially calculated by looking at the difference between the previous value of the obtained result and the just observed one. During this step we are going to work with the concept of resource, in particular the decision concerns how to assign different resources to a workload. A resource is defined as a computational significant asset which can slightly modify, either in better or in worse, the performance of running observable application. In particular the resources we are going to manage in *XeMPUPiL* are: the frequency of the cores, and the number of vCPUs pinned over the pCPUs ones. The *decision* phase is thought as we are working inside a "gradient ascending" algorithm. We made three meaningful assumptions in order to adopt this technique. The performance function is a concave function, the power consumption is a convex function and each resource is independent from each others. The first assumption ensures the spotting of a global maximum, hence an unique point of termination. The second one trivially ensures the spotting of a global minimum. The third one allows us to explore the domain of the target function one dimension at a time, hence one resource at a time, since the point of global maximum will be represented by the point where the gradient is 0 for each direction. To enable this approach the first things to do is to assign each resource to a priority queue, we decided to adopt the following one: the first resource to explore is the number of vCPUs to pin over a pCPU and the second one is the frequency. The gradient is simply expressed as the difference, in term of performance, from a previous measured point to the actually observed. If it will be positive, then the exploration will continue to another point, else, in case it will be negative or zero, there will be a rollback to the previous resource configuration and the exploration for this one is terminated. At the beginning all the resources are obviously not tested, hence the gradient for all of them is set to $+\infty$. As shown in 5.2, the decision phase is split into two steps. The first one consists into looking for a not already tested resource, this means that there is no information about the performance provided by the application once the current resource is set to a specific value, hence the one to be tested. The switching from exploring the different configuration happens when a

degradation or stabilization of the performance is encountered, hence a termination condition for the under examination resource is found. Then a new resource will be explored. For what concerns the allocation of resources to each domain, we chose to work at a core-level granularity: on the one hand, each domain owns a set of vCPU, while, on the other hand, we have a set of pCPU present on the machine. Each vCPU is mapped on a pCPU for a certain amount of time, while it may happen that even multiple vCPU can be mapped on the same pCPU. We wanted our allocation policy to be as fair as possible, covering the whole set of pCPU if possible; given a workload with M virtual resources and an assignment of N physical resources, to each pCPU we assign:

$$vCPUs(i) = \left\lceil \frac{M - \sum_{j=0}^{i-1} vCPUs(j)}{N - i} \right\rceil \tag{5.1}$$

where i is an integer between 0 and $N - 1$, i.e., it spans over the set of pCPU. This formula represents the following behaviour: if the system has 3 pCPU and a workload has 8 vCPU, respecting 5.1 leads to a partition of a pinning of the virtual ones over the physical ones of 3-3-2. Three vCPU pinned over the first pCPU, two over the second one and three over the third one, as shown in 5.3.

5.3.3 Act

This is the phase where the innovative hybrid approach truly takes place, the act step essentially consists in:

1. setting the desired power cap;
2. actuating the selected resource configuration.

On the one hand, we decided to implement the same hardware technique proposed by *PUPiL* to set the power cap, i.e., exploiting the Intel RAPL interface. This provides a fast and strict response to power oscillations, harshly cutting the frequency and the voltage of the whole CPU socket, and ignoring the perfor-

mance of the applications actually running on the system. On the other hand, we support the knobs made available by the hypervisor to assign resources to each domain. This second step allows a fine tuning of the resources to improve domains' performance, but it is of course slower than the hardware actuation in responding to power variations. This is the reason why we use both the approaches to provide a fast response, still trying to find the best resource allocation to maximize the performance of each domain under the power cap.

Hardware Power Cap

A bare metal operating system can easily access the RAPL interface to set a power cap on the system by writing data into the right MSR of the processor. The two registers of interest are `MSR_RAPL_POWER_UNIT` and `MSR_PKG_RAPL_POWER_LIMIT`: the former contains processor-specific time, energy and power units, which are used to scale each value read or written on the RAPL MSR. To obtain a valid power or energy measure. The latter,instead, can be written to set a limit on the power consumption of the whole CPU socket.

In a virtualized environment, these registers are not directly accessible by the virtual domains, even from the privileged tenant Dom0. However, this limitation can be overcome by invoking custom hypercalls that can directly access the underlying hardware. To the best of our knowledge, the Xen hypervisor does not natively support specific hypercalls to interact with the RAPL interface: as a consequence, we implemented our custom hypercalls to this purpose. In order to be generic enough, we implemented two hypercalls: `"xempower_rdmsr"` and `"xempower_wrmsr"`. The first one allows to read, while the second one allows to write a specific MSR from Dom0.

Each hypercall needs to be declared inside the kernel of the hypervisor, that runs bare metal on the hardware. The kernel keeps track of the list of hypercalls available and the input parameters they accept. For each of them, a callback function has to be declared and implemented to be accessible by the kernel at runtime: our implementation makes use of two *Xen* built-in functions to safely read and write MSR registers, i.e., `wrmsr_safe` and `rdmsr_safe`. These indications raise

exceptions if something goes wrong in accessing the registers, avoiding errors and faults that can undermine the kernel stability.

We then implemented our own CLI tools to access these hypercalls from Dom0: `xempower_RaplSetPower` to set and `xempower_RaplPowerMonitor` to read the power consumption of the socket. Arguments (e.g., the desired value of power cap and the power consumption measured) are passed through the whole stack using a set of buffers that allow a fast and safe communication between different hierarchical protection domains [29] (i.e. ring0 for *Xen* and ring3 for Dom0). The CLI tools are in charge of performing some checks on the input parameters, as well as of instantiating and invoking the *Xen* CLI to launch the hypercalls.

Software resource management

The current implementation of *XeMPUPiL* exploits two tools provided by the *Xen* hypervisor to tune the performance and assign resources to domains.

The first one is the *cpupool* tool: this is part of the *Xenxl* CLI and allows to cluster the physical CPUs in different pools. Once a pool is declared, it is possible to create a domain that uses that pool: a new scheduler is instantiated in order to manage the pool. It will then schedule the domain's vCPU only on the pCPU that are part of that cluster. Our approach exploits this tool to assign more pCPU to a domain at runtime: as a new resource allocation is chosen by the *decide* phase, we increase or decrease the number of pCPU in the pool and pin the domain's vCPU to these, to increase workload stability. The domain still has the same amount of virtual resources, that *XeMPUPiL* distributed over the maximum number of physical ones available, potentially causing more vCPU to be time-multiplexed on the same core.

The second tool supported is *xenpm*: this allows to set a maximum and minimum frequency for each pCPU. After a first evaluation, we decided to leave the actuation of the core frequencies out of the *decision* phase, as it may interfere with the actuation made by RAPL.

6

Implementation

In this chapter we will present the current implementation of *XeMPUPiL*. In Section 6.1 we will describe the architecture of the proposed solution. In Section 6.2 we will detail the implementation of the CLI needed to exploit the RAPL interface in the context of virtualized environment, while in Section 6.3 we will present the integration of RAPL with the ODA control loop.

6.1 Architecture design

In this section we will describe the architecture on which the *XeMPUPiL* approach is based on. From 6.1 is possible to notice the three layers composing the architecture. At the bottom there is the hardware layer, where the physical hardware resources lie. On top of this level there is the hypervisor, which is in charge of the virtualization of the underlying resources and to provide the respective virtualized one to the top layer, the domains level. In this last level the domains (i.e. the VM) are instantiated and can exploit the virtualized resources provided by the hypervisor. In this layer the domains containing the workloads will be instantiated and executed. The ODA control loop of the *XeMPUPiL* approach lies in the dom0, the first VM instantiated at hypervisor startup. In this way is ensured that the orchestrator will be always present in the system, and furthermore, it will be able to exploit the privileges provided by the hypervisor to this domain. The decision phase operates totally inside this domain, instead the act

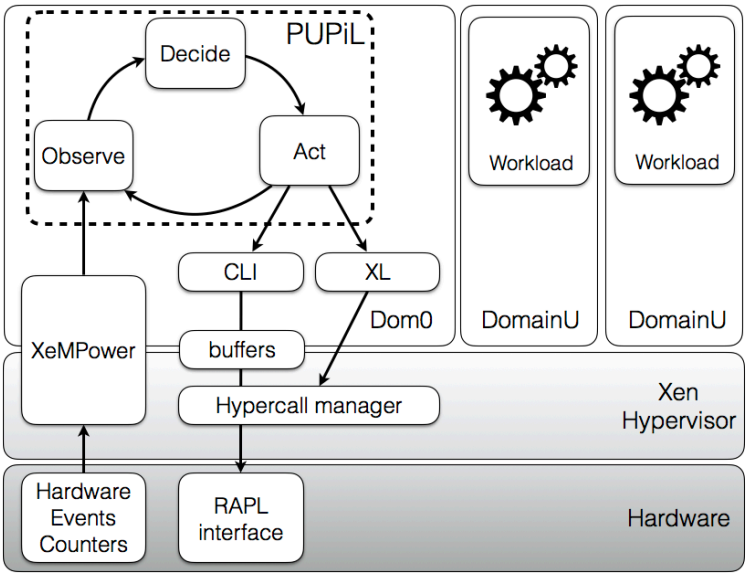


Figure 6.1: Overview of the *XeMPUPiL* architecture

and observe phases need to work among all the three different levels. The former will need to work at hardware level in order to exploit the RAPL interface, but to do so it needs access to the hardware, hence it is imperative to gain these access from the hypervisor level. This can be done exploiting the hypercall mechanism provided by Xen and addressed in details in Section 6.2. The chain of instructions is developed as follows: a tool providing a new set of commands for the Xen CLI is developed and exploited from dom0. This tool then exploits the hypercall and the buffer mechanism in order to move the computation into the hypervisor, gaining kernel rights. The hypercall manager deployed inside the kernel detects the called hypercall and manages it. This is enabled through the routines we will define as hypecall handlers for the new declared *XeMPUPiL* hypercalls. Finally, these routines will access the RAPL interface, setting the parameters according to the ones defined in uses space (i.e. dom0, act phase). Then, the act phase also needs to exploit privileged CLI commands in order to manage the resources assignments to domains, and these are provided at hypervisor level from the "xl" set of directives. The observe stage also need to access the hardware level in order to observe metrics retrieved from the HPC. To do so we exploit a modified ver-

sion of the *XeMPower* monitoring tool, a tool working among the three different layers, as explained in Section 2.2.

6.2 RAPL command line interface

The first step to achieve, in order to define the hardware power management technique that will exploit the RAPL interface, is the implementation of a working CLI that enables the exploitation of the RAPL interface also for a virtualized environment. In order to do so it is necessary to communicate with the hardware layer, thus leveraging the hypervisor. Since our software actor plays its role inside dom0, that is in user space, it is not possible to directly access the hardware resources from there. This is due to the Xen policies, made in order to maintain and respect the virtualization paradigm. In order to address this challenge, Xen provides a set of interfaces to its developers, which they are able to use when they want to gain access to the privileged level (i.e. the hypervisor level). These interfaces are called *hypercalls* and allow privileged command calls from the userland. A hypercall exploits a mechanism similar to the one used by system calls and OS. Making a parallelism between hypercalls and syscalls, an hypercall can be defined as a software trap from a domain to the hypervisor, in the same way a syscall is an interrupt from an application to the kernel. Privileged operations coming from the domain level can be requested only through a hypercall, that is synchronous and exploits event channels, a queue of asynchronous notifications, as return path to the domain. During the scheduling stage, when the domain has just been scheduled, if its queue of events is not empty, the event-callback (sited in the hypervisor) is called in order to take the related routine. They are provided through the `/proc/xen/privcmd` interface. In order to exploits the set of privileged calls, the user should compile dom0 or domU kernel with privileged configurations (i.e. CONFIG_XEN_PRIVILEGED_GUEST=y; in our case only dom0 is sufficient. Trying to be as general as possible, thus not focusing only on MSR for the RAPL interface, we will need two kind of hypercalls: one for reading a given MSR and one for writing a given MSR. In this direction the

first step consists into registering our hypercalls. To do so we need to modify the following file: `xen/arch/x86/x86_64/entry.S`. This assembly file contains all the hypercall and low-level fault handling routines. As shown in Listing 6.1,

```
1  .........
2  ENTRY(hypercall_table)
3  .quad do_set_trap_table     /*  0 */
4  .quad do_mmu_update
5  .quad do_set_gdt
6  .quad do_stack_switch
7  .quad do_set_callbacks
8  .........
9   .quad do_kexec_op
10 .quad do_tmem_op
11 .quad do_xempower_rdmsr    /* XeMPower MSR reading hypercall*/
12 .quad do_xempower_wrmsr    /* XeMPower MSR writing hypercall */
13 .rept __HYPERVISOR_arch_0-((.-hypercall_table)/8)
14 .quad do_ni_hypercall
15 .endr
16 .........
```

Listing 6.1: Code needed in order to declare a new hypercall

we added two hypercalls to the hypercall table. Each hypercall is declared as a `.quad` [1]. The second step, instead, consists into declaring the arguments that our hypercall routines will receive. In our case we will need two arguments, each one of 1 Byte size as shown in Listing 6.2.

```
1  ........
2  ENTRY(hypercall_args_table)
3  .byte 1 /* do_set_trap_table    */  /*  0 */
4  .byte 4 /* do_mmu_update        */
5  .byte 2 /* do_set_gdt           */
6  .byte 2 /* do_stack_switch      */
7  .byte 3 /* do_set_callbacks     */
8  ........
9  .byte 1 /* do_tmem_op           */
10 .byte 2 /* do_xempower_rdmsr    */ /* XeMPower MSR reading args */
11 .byte 2 /* do_xempower_wrmsr    */ /* XeMPower MSR writing args */
12 .rept __HYPERVISOR_arch_0-(.-hypercall_args_table)
13 .byte 0 /* do_ni_hypercall      */
14 .endr
15 .byte 1 /* do_mca               */  /* 48 */
16 ........
```

Listing 6.2: Code needed in order to declare the arguments passed to the hypercall managment routine

[1] For each expression into the current section, this assembly instruction generates an initialized word (64-bit). Each expression must evaluate to an integer value entry and must be a 64-bit value.[37]

The first argument represents the MSR on which the action will take place, while the second one will be used to pass the value to write (in the case of `do_xempower_writemsr`) or to pass the variable that will contain the value read from the given MSR (in the case of `do_xempower_readmsr`). The next step consists into assigning a constant for the new hypercalls. These constants are defined into the `xen/include/public/xen.h` file, containing the guest OS interface to Xen. The numbers choosen must be sequential with respect to the pre-existing hypercall definition in the file, as shown in Listing 6.3.

```
1 ........
2 #define __HYPERVISOR_kexec_op             37
3 #define __HYPERVISOR_tmem_op              38
4 #define __HYPERVISOR_xempower_rdmsr       39
5 #define __HYPERVISOR_xempower_wrmsr       40
6 #define __HYPERVISOR_xc_reserved_op       41 /* reserved for XenClient */
7 ........
```

Listing 6.3: Code needed in order to define the hypercall in the hypervisor interface

Then it is possible to declare the prototypes of our routines inside `xen/include/xen/hypercall.h`, the file containing all the prototypes for the system hypercalls as shown in Listing 6.5.

```
1  ........
2  extern long
3  do_tmem_op(
4  XEN_GUEST_HANDLE_PARAM(tmem_op_t) uops);
5  extern long
6  do_xenoprof_op(int op, XEN_GUEST_HANDLE_PARAM(void) arg);
7  extern long
8  do_xempower_rdmsr(unsigned long msr, XEN_GUEST_HANDLE_PARAM(uint64_t) u_val);
9  extern long
10 do_xempower_wrmsr(unsigned long msr, uint64_t val);
11 ........
```

Listing 6.4: Code needed in order to define the prototypes for the routines that will manage the hypercalls

As shown in Listing 6.5, the two prototypes are slightly different. Both of them return a long and are declared as extern. The first argument is also the same for both and it is an unsigned long representing the 64 bit address of the MSR. The real difference is in the second argument. Since the write just need to pass the value to the kernel level from the user space a simply passage by value is sufficient, because what matters is the value itself. Instead for what con-

cern the read, the value must be returned from this routine in kernel space to the user level, through the variable. A common passage by reference is not feasible, since in a virtualized environment the address space is disjoint between user and kernel space. So we exploit XEN_GUEST_HANDLE_PARAM() macro. XEN_GUEST_HANDLE_PARAM() represents a guest pointer, when passed as an hypercall argument. It is 4 bytes on aarch and 8 bytes on aarch64. In this way a mapping between guest and kernel memory addresses is possible, enabling the pass by reference technique. Once the prototypes are declared, it is possible to implement the body of the just defined hypercalls. To do so we need to modify the `xen/common/kernel.c` file. For what concerns the write and read routines we exploit the `wrmsr_safe` and `rdmsr_safe`, two functions provided by the Xen hypervisor in order to read and write those registers. In particular, the former needs as arguments the MSR address and the value to write, instead, the latter needs as arguments the MSR to read as well as the variable where to store the read value. Furthermore, in this routine we exploit the `copy_from_guest` and `copy_to_guest` functions in order to respectively retrieve and send the data into user space. Once the routines in charge to manage the hypercalls are defined at the kernel level, we can develop the interface based upon the *Privcmd* driver provided by Xen, providing access to those routines in user space. We declared and defined our user space hypercall invoker in `tools/libxc/xc_private.h`. This hypercall manager is in charge to identify the requested hypercall type (i.e. write or read) and the relative arguments to pass to the routine dispatcher in the Xen kernel. In particular, when a read is detected, an HYPERCALL_BUFFER_BOUNCE_BOTH is declared. This is a macro that creates a mapping between the address of a variable in user space and the address of the same variable in kernel space, enabling for a variable the passage by reference between these two disjoint address spaces. Now what remains to define are our two CLI tools in `tools/xcutils` and to modify accordingly the "Make" file corresponding to the Xen tools module. The tools developed are based upon the guidelines contained into the Intel manual and presented in Section 2.3. The pseudo code for the tool invoked via `xempower-set-rapl [power-cap]` is represented in

Algorithm 3.

```
Data: power cap
Result: completion value or exception
cap ← checkArgs(args);
openXenInterface();
powerUnit ← do_hypercall(MSR_RAPL_POWER_UNIT,
 null,RDMSR_HYPERCALL);
wattUnit ← transform(powerUnit);
capInUnit ← wattToUnit(wattUnit,cap);
result ←
 do_hypercall(MSR_PKG_RAPL_POWER_LIMIT,capInUnit,WRMSR_HYPERCALL);
```

Algorithm 3: Pseudocode for the xc_xempower_setRAPL tool

Instead the pseudocode for the `xempower-monitor-rapl` tool is represented in Algorithm 4. Where the loop is used to monitor that the cap is truly enforced.

```
Data: none
cap ← checkArgs(args);
openXenInterface();
powerUnit ← do_hypercall(MSR_RAPL_POWER_UNIT, null,
 RDMSR_HYPERCALL);
wattUnit ← transform(powerUnit);
while true do
    capInUnit ← do_hypercall(MSR_PKG_RAPL_POWER_LIMIT, null,
     RDMSR_HYPERCALL);
    result ← unitToWatt(wattUnit,capInUnit);
    print(result)
end
```

Algorithm 4: Pseudocode for the xc_xempower_monitorRAPL tool

6.2.1 Enabling RAPL in multi-socket architecture

In a single socket architecture is straightforward to set the correct value in the right MSR, since it is ensured that, independently from which core the hypecall is executed, the core will belong to the socket. Hence, the MSR hypercall will control the entire socket. Instead, in a multi-socket environment it's impossible to predict on which core the hypercall management routine will be executed. One first idea was to launch a random number of hyercalls, hoping that at some point in time at least one physical core per socket had executed the hypercall routine.

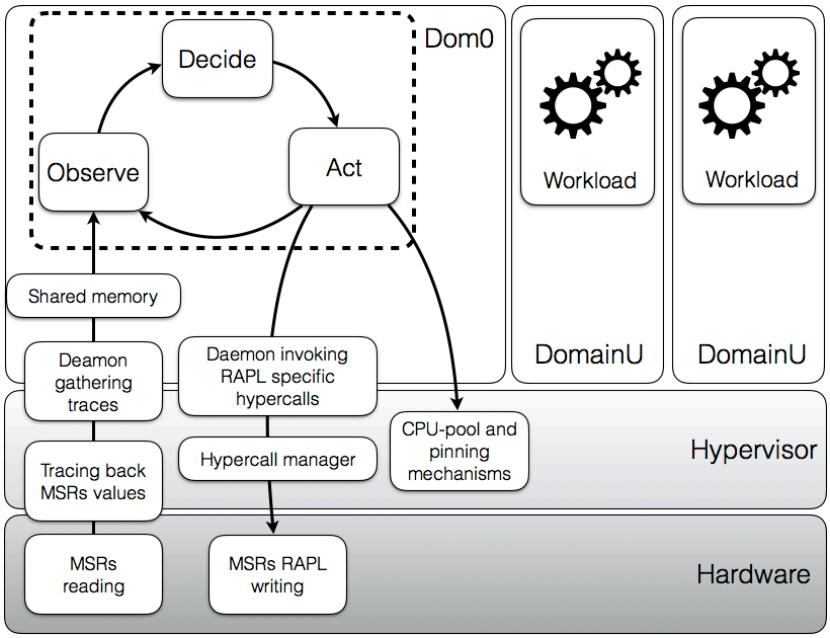


Figure 6.2: Overview of the *XeMPUPiL* architecture for multi-socket systems.

This approach is not optimal at all, since no guarantee about the enforcing of the power cap over all the sockets is provided. This is why we decided to exploits some elements provided by Xen under a different fashion. First of all we modified the *xempower_wrmsr* hypercall routine like in Listing 6.5. We exploited a macro already existing in Xen, the *FOR_EACH_ONLINE_CPU*. This macro is like the "for each" construct in high level languages like Java. In this particular case, it executes its body a number of times like the number of pCPUs actually online in the system, providing a driving variable containing the number of the actual CPU taken in consideration. The data initialization consists into defining a `msr\_data\_t` variable, that contains all the arguments that will be necessary to the auxiliary function launched on the new pCPU. The memory region is set to all zeros through the *memset* functionality. Then the values to be passed are stored in the variable and the hypercall is continued on each online CPU.

```
1 ........
2 DO(xempower_wrmsr) (unsigned long msr, uint64_t val)
3 {
4
5
6     unsigned int cpu;
7     msr_data_t data;
8     memset(&data,0,sizeof(data));
```

```
9       data.msr = msr;
10      data.value = val;
11
12      CUSTOM_LOGI("In xempower_wrmsr hypercall\n");
13      for_each_online_cpu(cpu){
14          continue_hypercall_on_cpu(cpu,hyppo_wrmsr_helper,(void *)&data);
15          CUSTOM_LOGI("Hypercall continued on cpu %d\n",cpu);
16      }
17  }
18  ........
```

Listing 6.5: New version of the routine managing the hypercall.

The second step, presented in Listing 6.6, consisted in the definition of the "writing MSR" function as a *tasklet*. In Xen, tasklets are dynamically-allocatable tasks run in either vCPU context (specifically, the idle VCPU's context) or in softirq context, on at most one CPU at a time. Softirq versus vCPU context execution is specified during per-tasklet initialisation. Defining the write MSR function as a tasklet allowed us to exploit it in the *continue_hypercall_on_cpu* already mentioned, which allows to continue a tasklet with hypercall priviledges on a defined CPU. In this way we were able to ensure that on each core, hence on each socket, the MSR controlling the power consumption is correctly defined.

```
1   ........
2   void hyppo_wrmsr_helper(void *data_passed){
3
4   msr_data_t data;
5   int ret;
6
7   memset(&data, 0, sizeof(data));
8   memcpy(&data,data_passed,sizeof(data));
9   ret = 0;
10
11  printk("[XeMPower Kernel Hypercall] Written value %"PRIu64" on CPU %d \n",
        data.value,smp_processor_id());
12
13  ret = wrmsr_safe(data.msr, data.value);
14  if(ret)
15  {
16  printk("[XeMPower Kernel Hypercall] custom wrmsr hypercall terminated with ERROR!\n");
17  return;
18  }
19  return;
20  }
21  ........
```

Listing 6.6: Tasklet defined in order to write a given MSR

Firstly, we exploit the *memset* and *memcpy* functions in order to set the memory

region of data to 0 and then to copy in the the values coming from the data region passed to the tasklet. Then we write the MSR and the value contained in the data passed. The new full architecture for multisocket architecture is represented in Figure 6.2. After several tests with this new implementation, conducted without rebooting the system, we discovered a significant downgrade of the performance of the hypervisor. This was due to a behaviour hidden by the *continue_hypercall_on_cpu* function. Digging deeper in its implementation we discovered that each time this function is called it kills the running hypercall and generates a new one, with return point in the killed one. This behaviour was leading us to have zombie hypercall tasklets running in the system, leading to worst performance. In order to solve this problem we moved to a new implementation where we exploit a *smp*[2] call, in order to launch a tasklet on each pCPU without the need of returning at some point, in this way avoiding any kind of zombie process in the system. The code of this final implementation is presented in Listing 6.7.

```
1
2  DO(xempower_wrmsr)(unsigned long msr, uint64_t val)
3  {
4
5  msr_data_t data;
6  memset(&data,0,sizeof(data));
7  data.msr = msr;
8  data.value = val;
9
10 CUSTOM_LOGI("In xempower_wrmsr hypercall\n");
11
12 on_each_cpu(hyppo_wrmsr_helper,(void *)&data,0);
13 return 1;
14 }
```

Listing 6.7: New version of the routine managing the hypercall.

[2]Symmetric multiprocessing (SMP) involves a multiprocessor computer hardware and software architecture where two or more identical processors are connected to a single, shared main memory, have full access to all I/O devices, and are controlled by a single operating system instance that treats all processors equally, reserving none for special purposes.

6.3 *XeMPUPiL* orchestrator

In this section we are going to describe how the three phases of the ODA control loop were implemented inside *XeMPUPiL*. The first important choice was to decide to write the orchestrator in Python in order to ease the developing and testing steps. The second choice, instead, consisted into developing it inside dom0, in order to exploit the privileges provided to this domain from the Xen hypervisor as mentioned in Section 5.3. We will start to analyse the ODA loop from the acting phase since it is the first one taking action in our implementation, starting from the initialization stage. Then, we will move to the observe phase and finally to the decision step. *XeMPUPiL* were designed as a Python class where each phase is represented by one or more methods. In this way the approach results extensible with respect to new implementations of the different phases, enabling modularity and isolation between them.

6.3.1 Act

This phase is divided into two different steps. The first one takes place during the initialization and consists into enforcing the power cap via RAPL exploiting the provided *xc* tool described in Section 6.2. It also consists into creating a number of CPU-Pools according to the number of instantiated workload. To achieve this second goal, we used the Xen "xl" interface(i.e. set of privileged CLI directives). The corresponding commands are exploited:

- `xl cpupool-create` in order to create a new pool for each running workloads;
- `xl cpupool-cpu-remove` in order to remove a pCPU from a given pool;
- `xl cpupool-cpu-add` in order to add a not assigned pCPU to a pool.

In this way, during initialization, we are able to define all the structures needed to manage the physical resources. During this initial stage, we exploit also the Xen CLI in order to create the domains (i.e. VM) on which the workloads will run. These initial activities are all executed inside the *RunApp* method. The other step

of the acting phase happens inside the ODA loop, just after a decision is taken. This step is represented by the method *AdjustConfig*. This function receives the results coming from the decision phase and puts them in practice. To do so, it exploit the `xenpm set-scaling-[max or min]freq` tool. This function is provided by Xen and allows a user to set the system frequency from dom0. By setting both min and max frequency at the same value it is possible to define a specific frequency instead of a range. The next phase being part of this stage consists into moving the pCPU from one pool to another according to the decision mode, exploiting the tools mentioned above. The mapping of all the pools regarding the pCPU over the vCPU is stored in a dictionary, where the key is the pool name and the value is a vector containing the pCPU actually assigned to it. This is helpful when the pool assignments change.

6.3.2 Observe

The observe stage is implemented through the *GetFeedBack* orchestrator method. This function retrieves information about how the workload is performing under the current configuration.

The first step consisted in instrumenting the Xen *Scheduler*. Xen instantiate an independent instance of the scheduler for each pCPU of the system, thus at this level it is not feasible to obtain an overall view of how a domain is performing, since each observation will be a fractal of the current pCPU (a domain may run on multiple pCPUs). In order to retrieve such information we are monitoring IA32_FIXED_CTR0 register, containing the number of IR in the Intel architecture. Then, the information provided by each instance of the schedule is traced to a higher level exploiting the xentrace [14], a lightweight trace capturing facility present in Xen that can record events at arbitrary control points in the hypervisor. We tag every trace record with the ID of the scheduled domain and its current VCPU, as well as a timestamp, in order to respect the mechanism implemented by the xempowermon deamon exploited to reconstruct the trace flow. Finally, we set to 0 the register, ready for the next observations. These informations are used by the xempowermon deamon (running at dom0 level) in order to reconstruct the

flow and to initially aggregate the results under a per domain policy and it saves them in a shared memory region that is read-only with respect to applications different from the deamon itself. The last step lies in the observer phase inside the orchestrator ODA logic, where exploiting Python we periodically scan the tumbling windows produced by the XeMPower deamon (in the shared memory region), and performs aggregations in the given time interval. This shared memory region is accessed by the *GetFeedBack* method, the new entries over the time window are read, the mean of IR over this interval is calculated and then a tuple is created. A new entry is meant as a new value arrived after the current configuration is changed. The tuple is composed by the current configuration and the calculated performance for it. In this way a coupling between them is made. Finally this structure is then added to the dictionary containing all the configuration-performance couple. Exploiting the dictionary data structure will speed up the search for the best configuration during the decision phase.

6.3.3 Decide

The *Decide* phase was implemented in two ways. The first one defining the condition necessary to solve the maximization problem regarding the performance of the workload under given power consumption constraints. The second one in order to solve the minimization problem regarding the power consumption given a SLA as constraint. The decision phase is implemented in the *Decision* method. The goal is simple: looking in all the possible configurations and finding the one associated with the best result. To do so we exploited a bounded binary search. Initially we explore always three configuration: (i) the one with minimum resources, (ii) the one with maximum resources and (iii) the one with an average amount of resources. For each of them we observe the behaviour of the workload, then some considerations are made. The overall idea consists into finding the two best and contiguous performances among these configurations and then defining a lower bound and an upper bound for the next iteration of the binary search. This reduces the space to be analysed, reaching in less iterations the convergence point. This is possible thanks to the nature of the dictionary containing the key-

value pairs made by configuration-performance tuples. Since these data can be seen as a sorted array without duplicates, it is possible to exploits two bounded binary searches in order to find the range of a given target value. Once these two configurations are defined they are passed to the function performing the binary search in an iterative way, where at each step the range becomes closer to the configuration point of convergence. The function is implemented in the *PerfB-search* method. The upper and lower bounds are used to define a new average configuration point, that is actuated and observed. Then the range is once again restricted adopting again the policy of best and contiguous performances. The search continues until the upper and lower bound are the same (convergence) or the new bounds performances are worse than the ones of the previous limits. In this second case the best between the previous ones is defined as the convergence configuration. Once the configuration providing the best performances for the workload is found the ODA control loop actuates it and terminates.

7

Experimental Evaluation

In this chapter we will present and discuss the results of the experimental campaign we conducted to validate the proposed methodology. The goals of our experiments are twofold: (i) we want to define a valid baseline that will be exploited to compare the obtained results and (ii) we want to show that *XeMPUPiL* is able to maximize the performance metric given a certain power limit. Then a comparison between pure RAPL and *XeMPUPiL* is done and discussed. This is possible thanks to the system setup, that is in common among the two steps. In this way, we will verify that the proposed methodology reaches better performance compared to the pure hardware one.

The chapter is structured as follows: Section 7.1 will present the benchmarks specification and also the experimental setup on which the experiments were conducted, including a description of the system. Section 7.2 will introduce the set of experiments conducted in order to define and validate the baseline. Finally in Section 7.3 the results obtained using the same set of experiments are presented when the hybrid methodology is applied and we will compare them with the baseline results.

7.1 Experimental setup and benchmarking

The evaluation of the proposed methodology has been carried out on a system equipped with an Intel Xeon E5-1410 processor. The CPU features 4 cores

clocked at 2.8 GHz, with 8 hardware threads. The evaluation was carried out with Turbo Boost and Hyper Threading disabled. The systems runs the Xen hypervisor version 4.4, with a paravirtualized instance of Ubuntu 14.04 as Dom0, pinned on the first core with 4GB of RAM assigned. For the benchmarking activity we decided to exploit four different benchmarks, each one representing a possible family of computational workloads having some bounds directly related to the resources of the system. In particular, we decided to investigate the following families: (i) CPU-bound, (ii) memory-bound, (iii) IO-bound and (iv) CPU-mem-bound workloads. Two of them are part of the set of benchmarks provided by the National Aeronautics and Space Administration (NASA), the NPB version 3.3 [3]. In the hera of supercomputers the NPB provides small, but valid, set of programs to evaluate the performance of these machines. The benchmarks consist of five kernel, derived from computational fluid dynamics (CFD) applications, and three pseudo-applications. This suite has been improved with computational grids, multi-zone applications, parallel I/O, and unstructured adaptive mesh. The sizes of the problems are predefine and represented as different classes. MPI and OpenMP are some programming models used to implement those benchmarks.

The two benchmarks taken from this suite are the Embarassingly parallel (EP) and the Block Tri-diagonal solver (BT) ones. The former adopts negligible interprocessor communication in order to provide estimates for the upper achievable limits for floating point performance. This can be defined as a CPU-bound workload. The latter, instead, is a pseudo application. In detail it provides solution of different, independent system of block tridiagonal, non-diagonally dominant equations. This application is both parallel and memory bounded. A third benchmark used to represent the IO-bound family is IOzone [2], a tool for benchmarking filesystems. A variety of file operation is measured and generated by this benchmark, which has been ported and able to run under different OSs. Finally, the fourth benchmark is used to represent the memory-bound application family. It is Cachebench [4], a performance test designed to test the memory and cache bandwidth performance, provided by the Low Level Characterization

Benchmarks suite. Each benchmark were run in a domain under a Debian OS. The configuration of each domain is independent from the workload running on it. A domain is configured as follow: 1 GB of RAM assigned and 3 vCPU. Before creating the domain, a CPU-pool is created with a number of pCPU equal to the virtual ones requested by the domain.

7.2 Baseline definition

In this section we are going to present and explain how the baseline were chosen and measured, exploiting some of the tools developed for *XeMPUPiL*. We tested the four benchmarks in different configurations of the system, selecting four power limits. First of all we measured the power consumption when no power limit was imposed. In order to have a valid comparison we exploited the WattsUp power meter. Knowing the power consumed by components other than the CPU socket, it was trivial to find the power consumption associated to the socket. During this phase we noticed that a totally busied CPU induces a power consumption of $\simeq$ 43 Watt, this was defined as the maximum power budged for the targeted socket. On the other hand on a totally idle CPU the consumed energy for the socket was measured as $\simeq$ 18 Watts, this was defined as the minimum power budget for the given system. In this way we were able to define the upper and the lower bound of the system's power consumption. Then, we decided to define four significant configurations in which studying the behaviour of the workloads. These configurations were defined as follows:

- `NO_RAPL`, in this configuration no power cap is defined, the workload is able to consume as much power as it needs, according to its resource assignment.
- `RAPL_40`, in this configuration a power cap is defined to 40W. The power limit is defined exploiting only the pure RAPL approach.
- `RAPL_30`, in this configuration a power cap is defined to 30W. Also in this case the only approach adopted in order to enforce this power cap is the

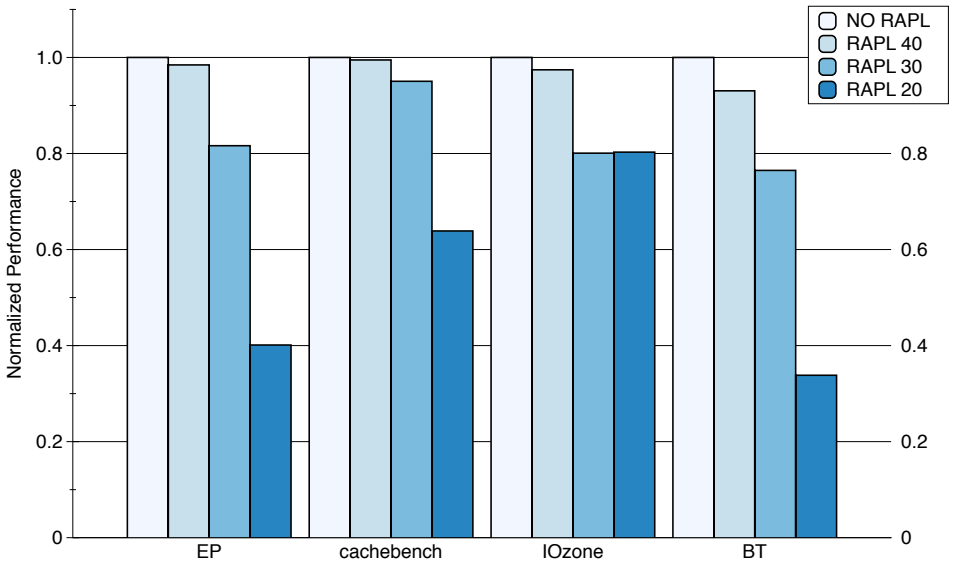


Figure 7.1: Baseline definition for the different configurations, displayed by benchmark

RAPL one.

- `RAPL_20`, in this configuration a power cap is defined to 20W. The same hardware power capping technique of the previous two cases is adopted to define this power limit.

Furthermore each domain is configured to run on a CPU-pool composed by three cores, where each vCPU is pinned over a different core, reaching a mapping of 1 on 1. The results are normalized with respect to the `NO_RAPL` configuration and displayed in 7.1.

From 7.1 it is possible to notice that the behaviour of the EP benchmark is as expected. When no constraint over the power consumption is enforced (i.e. `NO_RAPL` configuration), the CPU-bounded workload exploits the critical resources as much as it can, reaching the best performance possible. Instead, when a power cap is enforced (i.e. `RAPL_40, RAPL_30, RAPL_20` configurations) the performance decreases. This decrement is more significant as the cap becomes closer to the system lower bound. This is the expected behaviour since this CPU-bound workload is strongly dependent on the number of operation the processor can complete in a second. Hence, a reduction of frequency, along with a reduction of the computational units leads to worse performance. For what concerns the

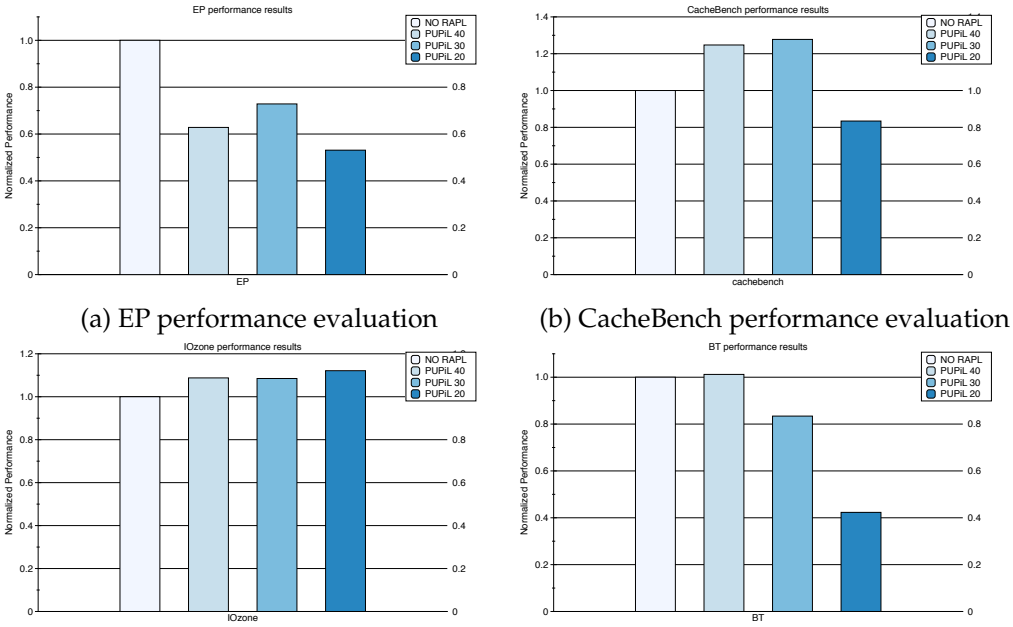
Cachebench benchmark, the trend slightly differs from the previous one. The decrement in performance is less significant now. This is due to the fact that in this case the performance strongly depends upon the memory and its response time. Hence, a decreasing in frequency is less significant for the high and middle range power caps. While applying limitations near the lower bound of the system the performance slightly decreases, mainly due to lower voltages assigned to memory components. A similar result is obtained for the IO-bound workload.. Once again the performance directly depends on resources different than the CPU. Finally, the performance of the BT benchmark workload depends both upon CPU and memory, indeed the measured behaviour is an average between the one observed in EP and Cachebench cases.

7.3 *XeMPUPiL* methodology evaluation

In this Section we are going to present the results obtained under the two different decision policies. In first place we conducted experiments on the maximization policy since its exploited later in the minimization one. We applied in the system a power cap by exploiting the developed tool stack for RAPL in Xen. Then, the ODA control loop is in charge to find the configuration obtaining the best performance in the current power configuration. These results were then compared to a pure hardware approach like RAPL. Then, we studied the behaviour of the minimization policy by looking at the resource assignment to the different classes of workload and the power consumption obtained for the system. In this case we defined different level of SLA, which correspond to fractions of the best performance, defined as the number of IRs obtained after the convergence of the maximization process.

7.3.1 Performance maximization given a power cap

In this set of experiments the results of the proposed hybrid approach are gathered and compared with respect to the defined RAPL baseline. Once again, the benchmarks exploited for these tests are the four presented in Section 7.1. In



(a) EP performance evaluation

(b) CacheBench performance evaluation

(c) IOzone performance evaluation

(d) BT performance evaluation

Figure 7.2: Results obtained for the four benchmarks under a power cap enforced through the proposed hybrid approach

this case the configurations explored are the following three:

- `pupil 40`, in this configuration a power cap is defined to 40W. The power limit is defined exploiting the RAPL interface, while the ODA control loop explores the set of all possible configuration, returning the best one providing the maximum performance.
- `pupil 30`, in this configuration a power cap is defined to 30W.
- `pupil 20`, in this configuration a power cap is defined to 20W.

The power caps are chosen according to the ones of the previous set of test in order to have a direct comparison between the results in the two cases. In Figure 7.2a the results obtained for the EP are shown. For what concern the IOzone benchmark the results are shown in Figure 7.2c, instead the behaviour of the memory-bound workload is represented in Figure 7.2b. Finally, the pseudo application BT is presented in 7.2d. All the results proposed are normalized with respect to the `NO_RAPL` configuration. In 7.3, 7.4 and 7.5 we compare the results obtained in the case of pure RAPL with the ones retrieved in the case of the hybrid

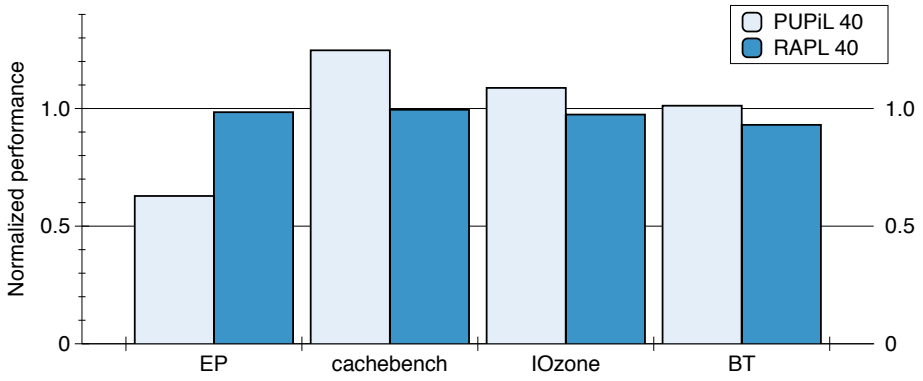


Figure 7.3: Comparison between the performance obtained enforcing a power cap of 40W in the hybrid and pure RAPL cases

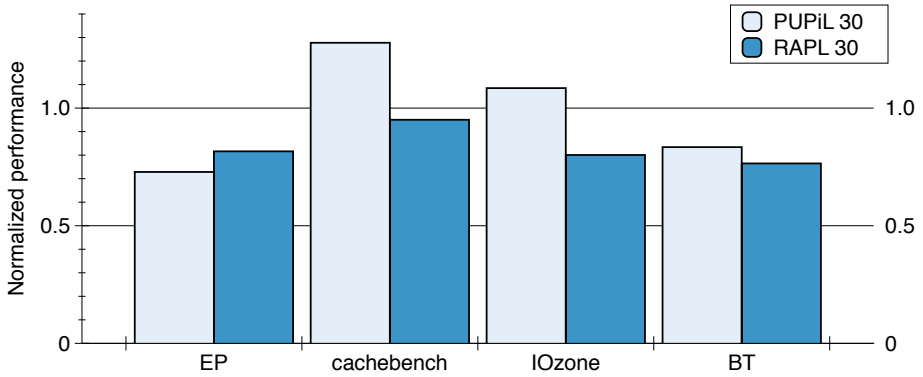


Figure 7.4: Comparison between the performance obtained enforcing a power cap of 30W in the hybrid and pure RAPL cases

approach. In all the three configurations is possible to notice how the proposed hybrid approach achieves better performance for the IO-bound, memory-bound and pseudo applications. This is due to the software approach represented by the ODA control loop, which finds the best configuration for the assignment of the resources, hence maximizing the overall performance. On the other hand it is also possible to notice that this behaviour is not valid for the EP workload too. This happens when the power cap is higher, while it decreases when the power budget diminishes. Looking at the configurations at which the ODA loop converges, it is possible to notice that the difference between RAPL and *XeMPUPiL* in this case lies in how the vCPU are pinned over the pCPU. This could be related to a misleading behaviour provided by some sort of optimization that Xen introduces during the initialization phase of the domain, when it automatically pins

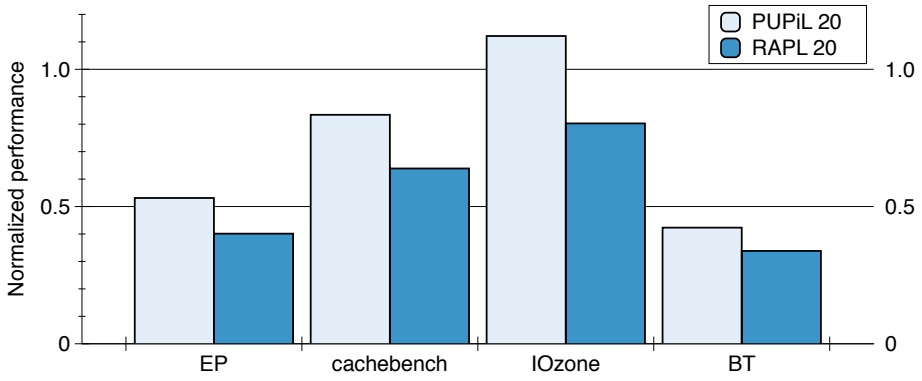


Figure 7.5: Comparison between the performance obtained enforcing a power cap of 20W in the hybrid and pure RAPL cases

the vCPU. This leads to a worst behaviour when the same mapping topology is adopted inside a CPU-pool, requiring a deeper study and a redesign of the policy adopted in *XeMPUPiL* in order to pin vCPU over pCPU while the workload presents a strong parallel behaviour.

7.3.2 Power consumption minimization under a SLA

In order to test both the minimization policy and the multisocket implementation, we moved the benchmark suite on a machine equipped as follow: dual socket Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz with 10 physical thread per socket and hyperthreading enabled. In this case we moved from Xen-4.4 to Xen-4.6 in order to exploit new functionalities introduced in this new version. The benchmarks and the VM running were the same. Since also in this case we exploited a 32-bit version of Ubuntu in order to carry out our tests, the maximum amount of vCPUs that the VM is able to manage is up to 8. We defined three SLAs: 90%, 80% and 70%, since from the previous set of experiments we noticed that at these steps is possible to notice measurable decrease of power consumption. The meaning of these values corresponds to the percentage of the maximum performance that must be at least returned. For instance, given a maximum performance of 1000 IRs obtained thanks to the maximization phase, defining for it a SLA of 80 means trying to minimize the power consumption until the performance downgrades under an absolute value of 800 IRs. The tests on the four

benchmarks were conducted as follows:

1. The power cap is imposed to default value. In this way is like imposing no power cap, since the maximum consumption for the sockets of the system is around 160W;

2. The maximization of the performance in a NO-CAP configuration is ran in order to find the resource configuration for the workloads providing the best performance;

3. The minimization of the power consumption is executed in order to respect the given SLA;

In Table 7.1 the results obtained for the EP benchmark are shown. It is possible to notice that in all the SLA cases, the maximization phase tends to converge to a configuration that assigns all the available resources to the VM. In these cases it is possible to notice that trying to respect a small SLA will result a greater standard deviation for what concerns the converged performance percentage, meaning that the orchestrator tends to disrespect the SLA defined in case is too small. A similar behaviour can be noticed from Table 7.2. In this test case the application is at the same time computational intensive and memory intensive and this is why less cores are assigned to it. This leads to try to assign to it less pCPUs, due to the memory intensive nature of the application.

Instead, in Table 7.3 and 7.4 two benchmarks respectively memory and IO intensive are presented. In this cases our methodology tends to assign less cores to the VMs. And it also possible to notice that the power consumption is decreased significantly. On the other hand these results presents a huge standard deviation. This is due to the IR metric adopted in order to evaluate the performance, since is

Table 7.1: Minimization results obtained for the EP benchmark

Sla			**Cores assigned**		**Power consumption (W)**		**Maximization time (s)**		**Minimization time (s)**	
Objective	*Obtained*	*Standard Deviation*	*Mean*	*Standard Deviation*	*Mean*	*Standard Deviation*	*Mean*	*Standard Deviation*	*Mean*	*Standard Deviation*
90	0,97	0,07	7,93	0,25	140,17	40,38	3,73	0,32	5,51	0,78
80	0,94	0,12	7,97	0,18	139,00	42,76	3,57	0,36	5,33	0,58
70	0,82	0,23	7,87	0,43	112,83	54,94	3,61	0,30	5,28	0,67

Table 7.2: Minimization results obtained for the BT benchmark

Sla			**Cores assigned**		**Power consumption (W)**		**Maximization time (s)**		**Minimization time (s)**	
Objective	*Obtained*	*Standard Deviation*	*Mean*	*Standard Deviation*	*Mean*	*Standard Deviation*	*Mean*	*Standard Deviation*	*Mean*	*Standard Deviation*
90	0,99	0,04	7,10	0,92	159,17	4,56	4,40	0,43	5,86	0,86
80	0,96	0,12	7,00	1,23	154,67	18,71	4,39	0,62	5,85	1,09
70	0,91	0,17	7,10	0,92	147,17	32,13	4,51	0,51	5,95	0,94

Table 7.3: Minimization results obtained for the CacheBench benchmark

Sla			**Cores assigned**		**Power consumption (W)**		**Maximization time (s)**		**Minimization time (s)**	
Objective	*Obtained*	*Standard Deviation*	*Mean*	*Standard Deviation*	*Mean*	*Standard Deviation*	*Mean*	*Standard Deviation*	*Mean*	*Standard Deviation*
90	1,00	0,05	5,60	2,40	147,17	35,40	4,51	0,50	6,04	0,79
80	0,99	0,12	5,43	2,28	135,50	46,37	4,62	0,50	6,27	0,93
70	1,00	0,05	5,10	2,09	147,67	35,10	4,68	0,44	6,48	0,99

really a low level measurement, very sensitive to each phase of the running application, even the smallest one. One last consideration regards the time needed in order to converge to the solution. This value is pretty stable for all the benchmark classes and its required only once for each iteration. Furthermore, this time can be further decreased since at the moment is left a time window of 2 seconds to the workloads to stabilize when the performance maximization phase is completed.

Table 7.4: Minimization results obtained for the IOzone benchmark

Sla			**Cores assigned**		**Power consumption (W)**		**Maximization time (s)**		**Minimization time (s)**	
Objective	*Obtained*	*Standard Deviation*	*Mean*	*Standard Deviation*	*Mean*	*Standard Deviation*	*Mean*	*Standard Deviation*	*Mean*	*Standard Deviation*
90	0,93	0,11	6,70	1,62	133,33	42,23	4,82	0,55	6,38	1,48
80	0,84	0,18	6,13	2,13	126,33	45,45	4,83	0,43	6,37	0,98
70	0,83	0,26	6,67	1,65	126,00	48,63	4,81	0,49	6,53	1,41

8

Conclusions and Future works

In this thesis, we presented *XeMPUPiL*, a performance-aware power capping orchestrator for the Xen hypervisor. We extended the current implementation of PUPiL [45] to make it work in a virtualized environment based on the Xen hypervisor. The methodology proposed in this work leverages three main concepts: (i) *efficiency*, (ii) *timeliness*, and (iii) *lack of workload instrumentation*. The first two concepts encapsulate the need of a system able to provide the best performance possible for the running workload (efficiency), while strictly and quickly respecting the defined power limit (timeliness). In order to achieve these two characteristics the proposed methodology totally embrace the novel hybrid approach, where software and hardware techniques work synergically in order to achieve at the same time both efficiency and timeliness. Instead, the third concept allows *XeMPUPiL* methodology to be portable and extensible as possible. This is feasible thanks to the monitoring the workload perfromances avoiding any instrumentation, resulting in less overhead pending on the workload developer to understand and use third parties APIs. Initially, we introduced in Chapter 3 the problem deriving from the switch to the new computational paradigm based on the cloud computing. We presented two problems deriving from this switch: maximizing performance given a power cap to respect and minimizing the power consumption while respecting a given SLA. In Chapter 4 we showed how the state of the art in the field of power management tackled power consumption optimization challenge by means of custom solutions, which can be classified

in two main families: the hardware and the software ones. The classification is based on the characteristic that each approach intrinsically enhances: timeliness for the hardware approaches and efficiency for the software ones. Then, we described a novel emerging technique that embraces the hybrid approach. However, to the best of out knowledge, a power capping hybrid approach in the field of virtualization requiring no instrumentation by the workloads is still missing. This is the why we introduced *XeMPUPiL*, a hybrid power capping orchestrator for a Xen virtualized environment. The methodology presented in Chapter 5 is built upon two components: the Intel RAPL interface and the ODA control loop. The former represents the hardware approach and is in charge to strictly enforce the power cap, instead the latter represents the software approach, which is in charge both to find the best resource configuration under the power cap that maximize the workload performance and minimizing the power consumption while respecting a given SLA, depending on which policy the user decided to put in practice. In Chapter 5 these two components are described inside the stages of the ODA control loop, where each challenge tackled in the three stages (i.e. observe - decide - act) is introduced and how they are addressed by *XeMPUPiL* is presented. Then, in Chapter 6 we went in detail on how to define and implement a set of tools in order to exploit the RAPL interface in the Xen virtualized environment. In particular, we explained how to take advantage of the hypercall mechanism, provided by virtualized envirorment, in order to gain control of the resources usually isolated (by the hypervisor) from the domains. Furthermore, the details regarding the implementation and the tools exploited by the ODA control loop are presented. Finally, in Chapter 7 we presented the system and the benchmarks adopted to validate the *XeMPUPiL* approach. At first, we described how the baseline was obtained through pure RAPL power capping. Then, these results were compared with the ones obtained when the power capping task is supervised by *XeMPUPiL*. We showed how *XeMPUPiL* is able to achieve higher performances under different power caps for almost all the different classes of benchmarks analyzed (e.g., CPU-, memory- and IO-bound ones). Finally, we conducted a study on the power consumption minimization policy,

showing that *XeMPUPiL* methodology is able to reduce power consumptions in almost all the benchmark classes tested.

Future Works The future works revolve around the development of a better decision algorithm to minimize the duration of the *decide* phase. In particular, it would be interesting to make studies about how different decision policies and techniques adopted in the the decision phase the ODA control loop may influence the convergence time of the software approach to the configuration providing highest performance. Furthermore, in this stage a redesign of the policy adopted to split the mapping of the vCPUs on the pCPUs is necessary in order to achieve at least the same performance obtained for pure parallel benchmarks in the case of pure RAPL. In this direction the mechanism behind the policies and techniques for vCPU repartition adopted by Xen during the domain creation must be better understood and then exploited. Moreover, we want to improve the *observe* phase, digging deeper into the XeMPower tool to weight the IR metric on the number of the “clock-ticks" in the observed interval, thus obtaining a cycle per instruction metric. Finally, we want to improve the actuation phase, implementing custom fine-grain tools, since the actual CLI provided by Xen allows only a limited set of resources to be tuned.

Bibliography

[1] Clockticks per instructions retired (cpi). https://software.intel.com/en-us/node/544403. Accessed: 2016-06-01.

[2] Iozone filesystem benchmark. http://www.iozone.org. Accessed: 2017-03-15.

[3] Nas parallel benchmarks. http://www.nas.nasa.gov/publications/npb.html#url. Accessed: 2017-03-15.

[4] Openbenchmarking.org. https://openbenchmarking.org/test/pts/cachebench. Accessed: 2017-03-15.

[5] Oracle vm user's guide = https://docs.oracle.com/cd/e35328_01/e35332/html/vmusg-ovm-intro.html, note = Accessed: 2017-04-16.

[6] The xen project - success stories. http://www.xenproject.org/users/success-stories.html. Accessed: 2017-03-15.

[7] Xen project wiki. https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview, note = Accessed: 2017-04-14.

[8] Ishtiaq Ali and Natarajan Meghanathan. Virtual machines and networks-installation, performance study, advantages and virtualization options. *arXiv preprint arXiv:1105.0061*, 2011.

[9] Paul R. Barham, Boris Dragovic, Keir A. Fraser, Steven M. Hand, Timothy L. Harris, Alex C. Ho, Evangelos Kotsovinos, Anil V.S. Madhavapeddy, Rolf Neugebauer, Ian A. Pratt, and Andrew K. Warfield. Xen 2002. Technical report, 2002.

[10] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

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

[12] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, et al. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15, 2008.

[13] Kenneth G Brill. The invisible crisis in the data center: The economic meltdown of moore's law. *white paper, Uptime Institute*, pages 2–5, 2007.

[14] David Chisnall. *The definitive guide to the xen hypervisor*. Pearson Education, 2008.

[15] Ryan Cochran, Can Hankendi, Ayse K Coskun, and Sherief Reda. Pack & cap: adaptive dvfs and thread packing under power caps. In *Proceedings of the 44th annual IEEE/ACM international symposium on microarchitecture*, pages 175–185. ACM, 2011.

[16] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. Rapl: Memory power estimation and capping. In *International Symposium on Low Power Electronics and Design (ISPLED)*, 2010.

[17] Qingyuan Deng, David Meisner, Abhishek Bhattacharjee, Thomas F Wenisch, and Ricardo Bianchini. Multiscale: memory system dvfs with multiple memory controllers. In *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*, pages 297–302. ACM, 2012.

[18] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 365–376. ACM, 2011.

[19] M. Ferroni, J. A. Colmenares, S. Hofmeyr, J. D. Kubiatowicz, and M. D. Santambrogio. Enabling power-awareness for the xen hypervisor. In *CEUR Workshop Proceedings*, volume 1697, 2016.

[20] Matteo Ferroni, Andrea Cazzola, Domenico Matteo, Alessandro Antonio Nacci, Donatella Sciuto, and Marco Domenico Santambrogio. Mpower: Gain back your android battery life! In *Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication*, pages 171–174. ACM, 2013.

[21] Part Guide. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 2011.

[22] Mohammad Haghighat, Saman Zonouz, and Mohamed Abdel-Mottaleb. Cloudid: Trustworthy cloud-based and cross-enterprise biometric identification. *Expert Systems with Applications*, 42(21):7905–7916, 2015.

[23] Henry Hoffmann. Racing and pacing to idle: an evaluation of heuristics for energy-aware resource allocation. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, page 13. ACM, 2013.

[24] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application heartbeats: A generic interface for expressing performance goals and progress in self-tuning systems. In *4th Workshop on Statistical and Machine learning approaches to ARchitecture and compilaTion (SMART)*, 2010.

[25] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application heartbeats for software performance and health. Technical report, August 2009.

[26] Tibor Horvath, Tarek Abdelzaher, Kevin Skadron, and Xue Liu. Dynamic voltage scaling in multitier web servers with end-to-end delay control. *IEEE Transactions on Computers*, 56(4), 2007.

[27] Huawei. *Huawei FusionSphere 3.1 Technical White Paper on Virtualization*.

[28] James M Kaplan, William Forrest, and Noah Kindler. Revolutionizing data center energy efficiency. Technical report, Technical report, McKinsey & Company, 2008.

[29] Paul Karger and Andrew Herbert. An augmented capability architecture to support lattice security and traceability of access. In *IEEE Symposium on Security and Privacy*, 1984.

[30] Minyoung Kim, Mark-Oliver Stehr, Carolyn Talcott, Nikil Dutt, and Nalini Venkatasubramanian. xtune: A formal methodology for cross-layer tuning of mobile embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 11(4):73, 2012.

[31] Rakesh Kumar and Shilpi Charu. Comparison between cloud computing, grid computing, cluster computing and virtualization. *International Journal of Modern Computer Science and Applications*, 3(1):42–47, 2015.

[32] Simone Libutti, Giuseppe Massari, Patrick Bellasi, and William Fornaciari. Exploiting performance counters for energy efficient co-scheduling of mixed workloads on multi-core platforms. In *Proceedings of Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, page 27. ACM, 2014.

[33] David Meisner, Christopher M Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F Wenisch. Power management of online data-intensive services. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 319–330. IEEE, 2011.

[34] Jennifer Mitchell-Jackson, Jonathan G Koomey, Bruce Nordman, and Michele Blazek. Data center power requirements: measurements from silicon valley. *Energy*, 28(8):837–850, 2003.

[35] Akihiko Miyoshi, Charles Lefurgy, Eric Van Hensbergen, Ram Rajamony, and Raj Rajkumar. Critical power slope: understanding the runtime effects of frequency scaling. In *Proceedings of the 16th international conference on Supercomputing*, pages 35–44. ACM, 2002.

[36] Shivajit Mohapatra, Radu Cornea, Hyunok Oh, Kyoungwoo Lee, Minyoung Kim, Nikil Dutt, Rajesh Gupta, Alexandru Nicolau, Sandeep Shukla, and Nalini Venkatasubramanian. A cross-layer approach for power-performance optimization in distributed mobile systems. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 8–pp. IEEE, 2005.

[37] CA 94065 U.S.A. Oracle Corporation 500 Oracle Parkway Redwood City. *x86 Assembly Language Reference Manual Developer's Manual*, 2010.

[38] 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.

[39] Amir Ali Semnanian, Jeffrey Pham, Burkhard Englert, and Xiaolong Wu. Virtualization technology and its impact on computer hardware architecture. In *Information Technology: New Generations (ITNG), 2011 Eighth International Conference on*, pages 719–724. IEEE, 2011.

[40] K. Shen, A. Shriraman, S. Dwarkadas, X. Zhang, and Z. Chen. Power containers: An os facility for finegrained power and energy management on multicore servers. In *IEEE 3rd International Conference on Cyber-Physical Systems, Networks, and Applications*. IEEE, 2015.

[41] Youngmin Shin, Hoi-Jin Lee, Ken Shin, Prashant Kenkae, Rajesh Kashyap, Dongjoo Seo, Brian Millar, Yohan Kwon, Ravi Iyengar, Min-Su Kim, et al. 28nm high-k metal gate heterogeneous quad-core cpus for high-performance and energy-efficient mo-

bile application processor. In *SoC Design Conference (ISOCC), 2013 International*, pages 198–201. IEEE, 2013.

[42] Arunchandar Vasan, Anand Sivasubramaniam, Vikrant Shimpi, T Sivabalan, and Rajesh Subbiah. Worth their watts?-an empirical study of datacenter servers. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–10. IEEE, 2010.

[43] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: reducing the energy of mature computations. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 205–218. ACM, 2010.

[44] C. Xu, Z. Zhao, H. Wang, and J. Liu. On the interplay between network traffic and energy consumption in virtualized environment: An empirical study. In *2014 IEEE 7th International Conference on Cloud Computing*, pages 392–399, June 2014.

[45] Huazhe Zhang and Henry Hoffmann. Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques. pages 545–559, 2016.

September 14, 2017

Document typeset with LaTeX