XeMPUPiL: Towards a Performance-aware Power Capping Orchestrator for the Xen Hypervisor

BY
MARCO ARNABOLDI
B.S, Politecnico di Milano, Milan, Italy, 2014

THESIS
Submitted as partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Chicago, 2017

Chicago, Illinois

Defense Committee:
Piotr Gmytrasiewicz, Chair and Advisor
Weijing Rao
Marco D. Santambrogio, Politecnico di Milano
ACKNOWLEDGMENTS

Several people made this work possible, they gave me inspiration, suggestions, support, precious time, guidance, help and patience. First of all I want to thank my advisor, Marco Domenico Santambrogio, who encouraged me and stimulated my interest and passion on our work. Without his guidance this thesis would not have been possible. A thank you goes also to Matteo Ferroni and Rolando Brondolin for the support and precious help they gave me. Thank you also to all the people in the NECSTLab at Politecnico di Milano, with whom I spend the working time in a stimulating atmosphere sharing useful suggestions and knowledge. A thank you to the PoliMi-UIC students of Fall 2016 both from Computer Science and Bioengineering programs, who shared with me such a great new experience. Last but not least, I would like to express my gratitude to my family for the encouragement, support and perseverance they gave me.

AM
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>INTRODUCTION AND MOTIVATIONS</td>
</tr>
<tr>
<td>2</td>
<td>BACKGROUND</td>
</tr>
<tr>
<td>2.1</td>
<td>Xen project</td>
</tr>
<tr>
<td>2.2</td>
<td>XeMPower</td>
</tr>
<tr>
<td>2.3</td>
<td>Intel Running Average Power Limit (RAPL) interface</td>
</tr>
<tr>
<td>2.3.1</td>
<td>MSR Power Unit</td>
</tr>
<tr>
<td>2.3.2</td>
<td>MSR Package Power Limit</td>
</tr>
<tr>
<td>2.3.3</td>
<td>MSR Package Energy Status</td>
</tr>
<tr>
<td>3</td>
<td>STATE OF THE ART</td>
</tr>
<tr>
<td>3.1</td>
<td>Classification criteria</td>
</tr>
<tr>
<td>3.2</td>
<td>Hardware approaches</td>
</tr>
<tr>
<td>3.3</td>
<td>Software approaches</td>
</tr>
<tr>
<td>3.4</td>
<td>Hybrid approaches</td>
</tr>
<tr>
<td>3.4.1</td>
<td>PUPiL</td>
</tr>
<tr>
<td>4</td>
<td>METHODOLOGY</td>
</tr>
<tr>
<td>4.1</td>
<td>XeMPUPiL: a bird’s eye view</td>
</tr>
<tr>
<td>4.2</td>
<td>Observe Decide Act (ODA) as a gradient ascending algorithm</td>
</tr>
<tr>
<td>4.3</td>
<td>XeMPUPiL ODA control loop</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Observe</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Decide</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Act</td>
</tr>
<tr>
<td>4.3.3.1</td>
<td>Hardware Power Cap</td>
</tr>
<tr>
<td>4.3.3.2</td>
<td>Software resource management</td>
</tr>
<tr>
<td>5</td>
<td>IMPLEMENTATION</td>
</tr>
<tr>
<td>5.1</td>
<td>Architecture design</td>
</tr>
<tr>
<td>5.2</td>
<td>RAPL command line interface</td>
</tr>
<tr>
<td>5.3</td>
<td>XeMPUPiL orchestrator</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Act</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Observe</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Decide</td>
</tr>
<tr>
<td>6</td>
<td>EXPERIMENTAL EVALUATION</td>
</tr>
<tr>
<td>6.1</td>
<td>Experimental setup and benchmarking</td>
</tr>
<tr>
<td>CHAPTER</td>
<td>PAGE</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6.2 Baseline definition</td>
<td>57</td>
</tr>
<tr>
<td>6.3 XeMPUPiL methodology evaluation</td>
<td>60</td>
</tr>
<tr>
<td>7 CONCLUSIONS AND FUTURE WORKS</td>
<td>64</td>
</tr>
<tr>
<td>APPENDICES</td>
<td>67</td>
</tr>
<tr>
<td>Appendix A</td>
<td>68</td>
</tr>
<tr>
<td>Appendix B</td>
<td>70</td>
</tr>
<tr>
<td>CITED LITERATURE</td>
<td>72</td>
</tr>
<tr>
<td>VITA</td>
<td>76</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>32</td>
</tr>
<tr>
<td>7</td>
<td>33</td>
</tr>
<tr>
<td>8</td>
<td>35</td>
</tr>
<tr>
<td>9</td>
<td>42</td>
</tr>
<tr>
<td>10</td>
<td>59</td>
</tr>
<tr>
<td>11</td>
<td>60</td>
</tr>
<tr>
<td>12</td>
<td>61</td>
</tr>
<tr>
<td>13</td>
<td>62</td>
</tr>
<tr>
<td>14</td>
<td>63</td>
</tr>
<tr>
<td>15</td>
<td>69</td>
</tr>
</tbody>
</table>
**List of Algorithms**

<table>
<thead>
<tr>
<th></th>
<th>Pseudocode for a gradient ascending algorithm</th>
<th>28</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Pseudocode for a generic ODA control loop</td>
<td>28</td>
</tr>
<tr>
<td>3</td>
<td>Pseudocode for the xe_xempower_setRAPL tool</td>
<td>50</td>
</tr>
<tr>
<td>4</td>
<td>Pseudocode for the xe_xempower_monitorRAPL tool</td>
<td>50</td>
</tr>
</tbody>
</table>
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
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>NASA</td>
<td>National Aeronautics and Space Administration</td>
</tr>
<tr>
<td>NPB</td>
<td>NAS Parallel Benchmarks</td>
</tr>
<tr>
<td>EP</td>
<td>Embarassingly parallel</td>
</tr>
<tr>
<td>BT</td>
<td>Block Tri-diagonal solver</td>
</tr>
<tr>
<td>MC</td>
<td>Memory Controller</td>
</tr>
<tr>
<td>SoA</td>
<td>State of the Art</td>
</tr>
<tr>
<td>CPU</td>
<td>central processing unit</td>
</tr>
<tr>
<td>PMC</td>
<td>performance monitoring counter</td>
</tr>
<tr>
<td>TDP</td>
<td>thermal design power</td>
</tr>
<tr>
<td>EDA</td>
<td>electronic design automation</td>
</tr>
</tbody>
</table>
SUMMARY

In the era of Cloud Computing, services 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 to the as a Service paradigm: this, together with virtualization techniques, allows 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. This 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
SUMMARY (continued)

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 adopter 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 XeMPUPiL is able to achieve higher performance under different power caps for almost all the different classes of benchmarks analyzed (e.g., CPU-, memory- and IO-bound).
CHAPTER 1

INTRODUCTION AND MOTIVATIONS

Computing infrastructures changed considerably in the last few decades [1]. 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. In this context, “virtualization” enables cloud providers to run multiple applications on the same physical resources, still ensuring strong isolation to each of them [2; 3].

Unfortunately, this can only lead to a better utilization of the hardware platforms 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 [4; 5]. 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 [6]: 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 [7].

RAPL uses Dynamic Voltage and Frequency Scaling (DVFS) techniques to guarantee the desired power cap but is not aware of the impacts that these 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 its 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, based on the PUPiL ODA control loop [8], that aims at maximizing the performance of a workload under a power cap. The main contributions of this work 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 workloads;

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 developed 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 describes the state of the art;
- Chapter 4 details the methodology behind *XeMPUPiL*;
- Chapter 5 digs into the implementation details of the orchestrator;
- Chapter 6 presents the experimental results that validates our approach;
- Finally, Chapter 7 draws the conclusions and presents the future directions of this work.
CHAPTER 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 [9] and Huawei FusionSphere [10]. In Figure 1 [11] is introduced a schema of the Xen Project architecture.
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 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 Virtual Machines (VMs), are virtualized spaces, each of them can run different Operating System (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 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 systems 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 of the running VMs.

---

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

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

Figure 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 Model Specific Registers (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.

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.

![Figure 3: Representation of the MSR_RAPL_POWER_UNIT Register.](image-url)
2.3.1 MSR Power Unit

As shown in Figure 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 Equation 2.1 [13], where $PU$ stands for Power Unit.

\[
\text{unitInWatts[WATTS]} = \frac{1}{2^{PU}}
\] (2.1)

Equation 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 Equation 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 Equation 2.2, where $ESU$ stands for Energy Status Unit.

\[
\text{unitInJoule[JOULE]} = \frac{1}{2^{ESU}}
\] (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 Figure 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:
**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

---

**Figure 4:** Representation of the MSR_PKGPOWERLIMIT Register
In order to allow runtime modification at the power limit. In order to define a power limit the relation expressed in Equation 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, Equation 2.3 must be applied.

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

Equation 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 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 Figure 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 Equation 2.4.

\[
\text{energyConsumed[Joule]} = \text{TOTAL ENERGY UNITS} \times \frac{1}{E_{\text{ESU}}} \tag{2.4}
\]

Equation 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 Equation 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 Equation 2.5.

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

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 Equation 2.4 and Equation 2.5 the power consumption is retrieved. \(\Delta t\) 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.
CHAPTER 3

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 3.1 a brief introduction about the methodology adopted in order to classify the State of the Art (SoA) is presented. Then, in Section 3.2 we are going to introduce in detail the hardware approaches and the concept of timeliness. Instead in Section 3.3 the software approaches and the concept of efficiency are presented. Finally in Section 3.4 a novel approach exploiting at the same time both hardware and software techniques, the so called hybrid approach, is analysed in detail.

3.1 Classification criteria

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

3.2 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 technique exploits socket’s frequency and voltage controllers. The main idea is to reduce dynamic power consumed per time unit via frequency and voltage decrement using Equation 3.1,

\[
DynamicPower = C \times V^2 \times A \times f
\]  

(3.1)

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 Equation 3.1 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 [14] 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.[15], 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 [16] 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 [17] provides useful information about the new features introduced in the sandy-bridge processors family. The Intel developer manual [13], volume 3B, section 14.9, shows that RAPL can be defined as an interface providing mechanisms to enforce power consumption
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 Section 2.3.

### 3.3 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 approach 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 [18] 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) classifier 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 [19] 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**[20; 21; 19; 22]. 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*[19]. 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*[19]. 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 [15; 23; 24] or in increasing the battery life of embedded systems [25; 26; 27]. 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.

### 3.4 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.

3.4.1 \textit{PUPiL}

Among all the proposed works in literature, the one proposed by Zhang and Hoffmann [8]
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. \textit{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 [13]
its possible to impose a strict power cap for the entire socket. The \textit{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 [28] 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* [28; 29], 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* [30], a bare-metal type-1 hypervisor widely adopted in real production environments [31], 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.
CHAPTER 4

METHODOLOGY

In this chapter we are going to introduce the methodologies adopted inside XeMPUPiL and the goals of the proposed approach. In Section 4.1 a brief introduction of the proposed approach is presented. In Section 4.2 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 4.3 the ODA methodology is explained in detail.

4.1 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 [8] and detailed in Section 3.4, 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 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 configuration in order to find
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 ← P;
result ← $f(tmpPoint)$;
repeat
    tmpResult ← result;
    tmpPoint ← tmpPoint + $\gamma \nabla f(a)$;
    tmpResult ← $f(tmpPoint)$;
until tmpResult - result > 0;

Algorithm 1: Pseudocode for a gradient ascending algorithm

the one providing the best performance. These tasks are pursued while exploiting the Intel
RAPL interface ensuring that the power constraint over the system is respected.

4.2 ODA as a gradient ascending algorithm

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
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 problem, hence finding the global maximum of a target function. The objective function to maximize is the overall performance in our case. 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;

- 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. 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 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.
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.

4.3 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 as a maximization problem. Given a power cap as a constraint, the software approach maximizes the performance of the running application. 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 Figure 6. 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.

4.3.1 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
Start with minimal config features

Set RAPL power cap

exists a resource that has not been tried yet?

turn it on at full throttle

Monitor performance

if performing better keep on else turn off

binary search for accurate tuning

Figure 7: Workflow diagram leading the ODA cycle
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 Instruction Retired (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 [32] . The challenges here are three:

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.

All the challenges are addressed by XeMPower, a tool designed at Politecnico di Milano in collaboration with the SwarmLab at Berkeley. XeMPower provides a lightweight infrastructure to monitor power consumption of the running domains in Xen. To obtain these results it
implements a set of mechanism that allows to access to low level registers, retrieving those information and divide the data with a per domain policy, exposing to the user the power consumption of each domain. We slightly modified this tool in order to retrieve IR per domain.

4.3.2 Decide

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
performance 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 two meaningful assumptions in order to adopt this technique. The performance function is a concave 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 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 Figure 7, 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:

$$vCPU_s(i) = \left\lfloor \frac{M - \sum_{j=0}^{i-1} vCPU_s(j)}{N - i} \right\rfloor$$

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 Equation 4.1 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 Figure 8.
4.3.3 **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 performance 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.

**4.3.3.1 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_rdmser" 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 be-
between different hierarchical protection domains [33] (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.

4.3.3.2 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.
CHAPTER 5

IMPLEMENTATION

In this chapter we will present the current implementation of \textit{XeMPUPiL}. In Section 5.1 we will describe the architecture of the proposed solution. In Section 5.2 we will detail the implementation of the CLI needed to exploit the RAPL interface in the context of virtualized environment, while in Section 5.3 we will present the integration of RAPL with the ODA control loop.

5.1 Architecture design

In this section we will describe the architecture on which the \textit{XeMPUPiL} approach is based on. From Figure 9 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 \textit{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 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 5.2. 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 version of the XeMPower monitoring tool, a tool working among the three different layers, as explained in Section 2.2.

5.2 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 5.1,

Listing 5.1: Code needed in order to declare a new hypercall

```
ENTRY(hypercall_table)
.quad do_set_trap_table /* 0 */
.quad do_mmu_update
.quad do_set_gdt
.quad do_stack_switch
```
.quad do_set_callbacks
...........

.quad do_kexec_op

.quad do_tmem_op

.quad do_xempower_rdmsr  /* XeMPower MSR reading hypercall */

.quad do_xempower_wrmsr  /* XeMPower MSR writing hypercall */

.rept __HYPERVISOR_arch_0-((.-hypercall_table)/8)

.quad do_ni_hypercall

.endr

...........

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 5.2.

Listing 5.2: Code needed in order to declare the arguments passed to the hypercall management routine

...........

ENTRY(hypercall_args_table)

\(^1\)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.[34]
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 chosen must be sequential with respect to the pre-existing hypercall definition in the file, as shown in Listing 5.3.
Listing 5.3: Code needed in order to define the hypercall in the hypervisor interface

\[
\begin{align*}
\texttt{#define \_\_HYPERVISOR\_kexec\_op} & \quad 37 \\
\texttt{#define \_\_HYPERVISOR\_tmem\_op} & \quad 38 \\
\texttt{#define \_\_HYPERVISOR\_xempower\_rdmsr} & \quad 39 \\
\texttt{#define \_\_HYPERVISOR\_xempower\_wrmsr} & \quad 40 \\
\texttt{#define \_\_HYPERVISOR\_xc\_reserved\_op} & \quad 41 */ reserved for XenClient */ \\
\end{align*}
\]

Then it is possible to declare the prototypes of our routines inside \texttt{xen/include/xen/hypercall.h}, the file containing all the prototypes for the system hypercalls as shown in Listing 5.4.

Listing 5.4: Code needed in order to define the prototypes for the routines that will manage the hypercalls

\[
\begin{align*}
\texttt{extern long} & \quad \texttt{do\_tmem\_op(} \\
& \quad \texttt{XEN\_GUEST\_HANDLE\_PARAM(tmem\_op\_t) uops);} \\
\texttt{extern long} & \quad \texttt{do\_xenoprof\_op(} \texttt{int op,} \texttt{XEN\_GUEST\_HANDLE\_PARAM(void) arg);} \\
\texttt{extern long} & \quad \texttt{do\_xc\_reserved\_op}; \\
\end{align*}
\]
As shown in Listing 5.4, 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 concern 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.

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.

5.3 **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,
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

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

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 4.3. 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.

5.3.1 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 \textit{xc} tool described in Section 5.2. 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:

- \texttt{xl cpupool-create} in order to create a new pool for each running workloads;
- \texttt{xl cpupool-cpu-remove} in order to remove a pCPU from a given pool;
- \texttt{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 \textit{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 \textit{AdjustConfig}. This function receives the results coming from the decision phase and puts them in practice. To do so, it exploit the \texttt{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.

5.3.2 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. This is possible thanks to XeMPower [12]. This tool was slightly modified in order to retrieve IR per domain and to save them in a log file in dom0. This tool already uses IR and HPC metrics in order to retrieve information about the power consumption for each domain. The patch consisted into retrieving this information and saving it in a file accessible from dom0. This metric is gathered over a time window that can be defined prior to launching this monitoring tool. This log file is accessed by the GetFeedBack method, the new entries 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 in the log file 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.
5.3.3 Decide

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 performance 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 PerfBsearch 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.
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 6.1 will present the benchmarks specification and also the experimental setup on which the experiments were conducted, including a description of the system. Section 6.2 will introduce the set of experiments conducted in order to define and validate the baseline. Finally in Section 6.3 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.

6.1 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 hard-
ware 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 NAS Parallel Benchmarks (NPB) version 3.3 [35]. 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 [36], 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 [37], 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.

6.2 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 XeMPUPl. 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 \(\approx 43\) 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 \(\approx 18\) 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 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 Figure 10.

From Figure 10 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.

### 6.3 XeMPUPiL methodology evaluation

![Graphs showing performance results](image)

Figure 11: Results obtained for the four benchmarks under a power cap enforced through the proposed hybrid approach

In this second 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 ex-
exploited for these tests are the four presented in Section 6.1. In 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 11a the results obtained for the EP are shown. For what concern the IOzone benchmark the results are shown in Figure 11c, instead the behaviour of the memory-bound workload is represented in Figure 11b. Finally, the
pseudo application BT is presented in 11d. All the results proposed are normalized with respect to the NO_RAPL configuration. In Figure 12, Figure 13 and Figure 14 we compare the results obtained in the case of pure RAPL with the ones retrieved in the case of the hybrid 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 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.
CHAPTER 7

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 [8] 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 performances avoiding any instrumentation, resulting in less overhead pending on the workload developer to understand and use third parties APIs. In Chapter 3 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 4 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 to find the best resource configuration under the power cap that maximize the workload performance. In Chapter 4 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 5 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 environment, 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 6 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).

**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 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.
APPENDICES
Appendix A

COPYRIGHT PERMISSIONS 1

This appendix includes information regarding Figure 2. In detail it includes a copy of the letter in which Dr. Matteo Ferroni, PhD. (retainer of the copyright) gave his permission to use his material in this thesis.
Appendix A (continued)

April 19, 2017
15, via Pasquale Paoli
Como, Italy 22100
Phone: 0039 - 368 8036971
Email: mr.arnaboldi@gmail.com

Dr. Matteo Ferroni, PhD
Copyright retainer

Dear Mr. Ferroni,

I am writing to request permission to use the following material from your publication (Ferroni, Matteo, et al. “Enabling power-awareness for the Xen Hypervisor.”, EWiLi ’16, October 2016 ) in my thesis. This material will appear as originally published. Unless you request otherwise, I will use the conventional style of Graduate College of University of Illinois at Chicago as acknowledgement.

A copy of this letter is included for your records. Thank you for your kind consideration of this request.

Sincerely yours,

Marco Arnaboldi

The above request is: approved
Reques approved by: MATTEO FERRONI Date: 04/20/17

Figure 15: Matteo Ferroni’s copyright consensus
Appendix B

COPYRIGHT PERMISSION 2

This second appendix include the terms under which is copyrighted Figure 1. Follows a brief extract from the Xen logos and images copyright page [38].

Xen mascots, templates and other images available through this page and downloads.xen.org/Branding (excluding the Xen logo) are made available under CC BY-NC 3.0. Feel free to use images on your blog, as wallpaper, for fun and the like, but you must link to www.xen.org.

Creative Commons CC BY-NC 3.0 allows you to:

**To Share:** Copy, Distribute, and Transmit the material,

**To Remix:** Create Derivative works of the material.

To Remix: Create Derivative works of the material Under these conditions: **Attribution:**

You can use the images and material without attribution when used in a context that refers to the Xen hypervisor, xen community, xen projects or xen.org (e.g. in a blog post or article about Xen, a Xen hackathon, A Xen user group meeting, etc.). Where the context is not clear, attribution must be made to xen.org or images used on merchandise, such as a sticker must contain the xen.org logo. Examples of clear context are:

- A training course entitled "How to use the Xen hypervisor to build a cloud";
- A website "xen_hall_of_fame.com" with pictures of famous Xen contributors;
Appendix B (continued)

- A monthly journal called "Xen User journals" describing how you use Xen;
- A blog post or article about the Xen Hypervisor;
- As avatar for a social media account with a reference to Xen in the profile name or in the profile description;
- Give-away of a Xen Fu Panda button, sticker, mug, etc. at a Xen user group meeting or Hackathon.

Examples where the context is not clear:

- A freely distributed Xen wallpaper or theme;
- Give-away of a Xen Fu Panda button, sticker, mug, etc. at a kids event.

**Noncommercial:** The image must not be used for commercial purposes, except for usage that complies with the The Xen trademark policy. In practice that means that you can use the images without concerns for commercial usage within the Xen and open source community.
CITED LITERATURE


CITED LITERATURE (continued)


# VITA

<table>
<thead>
<tr>
<th>NAME</th>
<th>Marco Arnaboldi</th>
</tr>
</thead>
</table>

## EDUCATION

<table>
<thead>
<tr>
<th>Date</th>
<th>Degree and Institution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spring 2016 - On date</td>
<td>Master of Science in Electrical and Computer Engineering, University of Illinois at Chicago, USA</td>
</tr>
<tr>
<td>Fall 2014 - On date</td>
<td>Master Degree in Computer Science and Engineering, Polytechnic of Milan, Italy</td>
</tr>
<tr>
<td>Fall 2010 - Spring 2014</td>
<td>Bachelor’s Degree in Biomedical Engineering (BE), Feb 2014, Polytechnic of Milan, Italy</td>
</tr>
</tbody>
</table>

## LANGUAGE SKILLS

<table>
<thead>
<tr>
<th>Language</th>
<th>Proficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Italian</td>
<td>Native speaker</td>
</tr>
<tr>
<td>English</td>
<td>Full working proficiency</td>
</tr>
<tr>
<td></td>
<td>2015 - IELTS examination (7.5/9)</td>
</tr>
<tr>
<td></td>
<td>A.Y. 2016/17 One Year of study abroad in Chicago, Illinois</td>
</tr>
<tr>
<td></td>
<td>A.Y. 2014/17. Lessons and exams attended exclusively in English</td>
</tr>
</tbody>
</table>

## SCHOLARSHIPS

<table>
<thead>
<tr>
<th>Scholarship</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N/A</td>
</tr>
</tbody>
</table>

## TECHNICAL SKILLS

<table>
<thead>
<tr>
<th>Level</th>
<th>Skill</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advanced</td>
<td>Programming (C, C++, Scala, Java, Python)</td>
</tr>
<tr>
<td>Advanced</td>
<td>Problem Solving</td>
</tr>
<tr>
<td>Advanced</td>
<td>Computer Architecture</td>
</tr>
<tr>
<td>Advanced</td>
<td>Distributed Systems</td>
</tr>
</tbody>
</table>

## WORK EXPERIENCE

<table>
<thead>
<tr>
<th>Date</th>
<th>Position and Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>May 2017 - On date</td>
<td>Backend software developer, Bottega52 S.R.L., Milan, Italy</td>
</tr>
<tr>
<td>Sep 2015 - Oct 2015</td>
<td>Backend development of webservice based Spring framework for Java, consultant position, NetPolaris S.R.L., Turate, Italy</td>
</tr>
</tbody>
</table>

## PUBLICATIONS

76
**VITA (continued)**

|---|---|

**PROJECTS**

**Ongoing**

- **SafeTree.** SafeTree is a monitoring system aims at evaluating trees stability status by exploiting a given set of sensor data. It is composed of a three-axis gyroscope, an accelerometer, an anemometer and a humidity sensor. The sensors send the data to a Control Center consisting of a server equipped with a PYNQ board that is used both as a receiver of the sensors measurements, and as compute device for signals analysis and manipulation.

- **Halstead complexity measure.** A tool written in Java8 that measures the Halstead complexity of a given Java project. Developed during the course "Object oriented Languages and Environment" at UIC.

- **Google API.** A client that will listen to your Gmail account and will check to see if new messages arrived. When the client detects new messages, it extracts the time of their delivery and the subjects of the newly delivered message and it inserts this information into the corresponding cells in the worksheet for the given day/month/year.

- **Understand™ API.** A client that will listen to your Gmail account and will check to see if new messages arrived. When the client detects new messages, it extracts the time of their delivery and the subjects of the newly delivered message and it inserts this information into the corresponding cells in the worksheet for the given day/month/year.

- **MapReduce™ IEEE crawler.** A distributed software application for automatically analyzing the content of research papers in computer science using a map/reduce model.

- **Akka™ GitHub crawler.** A tool developed in Scala exploiting the Akka actor framework, which crawls different projects from GitHub repositories through the provided APIs and analyze those projects through Understand APIs and return a graph visualization.
ElasticSearch™ code analyzer. A code search engine based on ElasticSearch. You will deploy your code search engine in the Google Cloud using your provided Google Cloud accounts. Your client program will take key words from users and then it will make a REST call to your web service that will use the key words to retrieve software projects where these key words are located. As parameters to your web service, clients can specify in what attributes they want to search the key words (e.g., language, committers, issues, or code) and how many results they want to retrieve.

Lambda calculus interpreter. A tool, based on a Read-Evaluate-Print Loop, developed in Scala that interpretes Lambda expressions and solve them.

Chord simulator. A tool developed in Scala and using the Akka actor framework, that simulate an overlay network ruled by the Chord algorithm.

Spring 2016

Scalable and High performance betweenness centrality. Analysis, implementation and scalability study of the algorithm proposed by McLaughlin et al. for Betweenness Centrality computation over graphs. For the Advanced Algorithms and Parallel Programming class we implemented a parallel CPU implementation of the algorithm with OpenMP and C++, and we tested its performance and scalability with respect to the graphs proposed in the paper.

Fall 2015

Key Twister. A videogame developed during the course of Videogame Design and Implementation. This is a party game enjoyable with one up to three friends using keyboard or joysticks. Single player modes are available too. The main feature of this game is non-determinism: the levels are randomly generated every time you play so that you can always have a new experience. The challenge is simple, but the path to the glory will not. Reach the bottom of the dice tower before your friends using your die, but be careful! During your fall to the finish line you have to avoid the hateful hands that will try to reroll your die, changing in this way your skills and your reflexes. Yes, we said ability, because depending on the one out of six faces of your die, you will be able to use a unique ability to take advantages over the other players.
Spring 2015  

MeteoCal. A web calendar developed using JavaEE during the class of Software Engineering 2. Users, once registered, should be able to create, delete and update events. An event should contain information about when and where the event will take place, whether the event will be indoor or outdoor. During event creation, any number of registered users can be invited. Only the organizer will be able to update or delete the event. Invited users can only accept or decline the invitation. Whenever an event is saved, the system should enrich the event with weather forecast information (if available). Moreover, it should notify all event participants one day before the event in case of bad weather conditions for outdoor events. Notifications are received by the users when they log into the system.