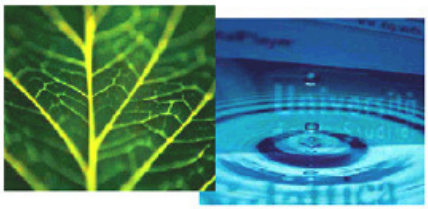
PhD Dissertation



International Doctorate School in Information and Communication Technologies

Department of
Information Engineering and Computer Science
University of Trento, Italy

Application Interference in Multi-Core Architectures: Analysis and Effects

Alexandre Kandalintsev

Advisor:

Prof. Renato Lo Cigno

Università degli Studi di Trento

April 2016

Abstract

Clouds are an irreplaceable part of many business applications. They provide tremendous flexibility and gave birth for many related technologies – Software as a Service (SaaS) and the like. One of the biggest powers of clouds is load redistribution for scaling up and down on demand. This helps dealing with varying loads, increasing resource utilization and cutting down electricity bills while maintaining reasonable performance isolation. The last one is of our particular interest.

Most cloud systems are accounted and billed not by useful throughput, but by resource usage. For example, a cloud provider may charge according to cumulative CPU time and/or average memory footprint. But this does not guarantee that the application realized its full performance potential because CPU and memory are shared resources. As a result, if there are many other applications it could experience frequent execution stalls due to contention on memory bus or cache pressure. The problem is more and more pronounced because modern hardware rapidly increases in density leading to more applications are co-located. The performance degradation caused by co-location of applications is called ***application interference****.*

In this work we study in-depth reasons of interference as well as ways to mitigate it. The first part of the work is devoted to interference analysis and introduces a simple yet powerful empirical model of CPU performance that takes interference into account. The model is based on empirical observations and build up from extrapolation of a two-task (trivial) case.

In the following part we present a method of ranking of virtual machines according to their average interference. The method is based on analysis of performance counters. We first launch a set of very diverse benchmark programs (to be representative for wide range of programs) one-by-one together with all sorts of performance counters. This gives us their "ideal" (isolated) performances. Then we run them in pairs to see the level of interference they create to each other. Once this is done, for each benchmark we calculate average interference. Finally we calculate the correlation between the average interference and performance counters. The counters with the biggest correlation are to be used as interference estimators.

The final part deals with measuring interference in production environment with affordable overhead. The technique is based on short (in the order of milliseconds) freezes of virtual machines to see how they affect other VMs (hence the name of method – Freeze'nSense*). By comparing the performance of the VM when other VMs active and when they frozen it is possible to conclude how much it looses in speed because of sharing hardware with other applications.*

Keywords

Cloud, Resource Management, Application Interference, Datacenters.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

What makes cloud computing so attractive? Deploying applications of almost any size and complexity is easy as never before. Cloud adopters do not need to concern about resources, scalability and reliability: these problems are solved by the cloud provider. Clouds also fostered two business models previously unseen, or rarely used in IT: Pay-As-You-Go (PAYG) and Everything-as-a-Service (XaaS).

Pay-as-you-go frees cloud customers from upfront costs on infrastructure. Before clouds, resource provisioning was a difficult tasks for many services because of variability of the load. For example, the day/night traffic variation can be a factor of 10 [29]. This means that the full computational power is needed only during peak hours. As a result, many systems are underloaded most of the time and unused resources are just wasted. Fig. 1.1 shows the daily variation of traffic on MSK-IX, Moscow Internet Exchange Point; higher traffic corresponds to higher loads of servers. Fig. 1.2 shows backend load (in requests per minute) of `CoDesign.io`; the load changes from almost zero up to 30rpm.

Another issue related to provisioning. An infrastructure without statistical multiplexing cannot scale dynamically; the infrastructure can only sustain a fixed maximum load and it must be planned well in advance. This

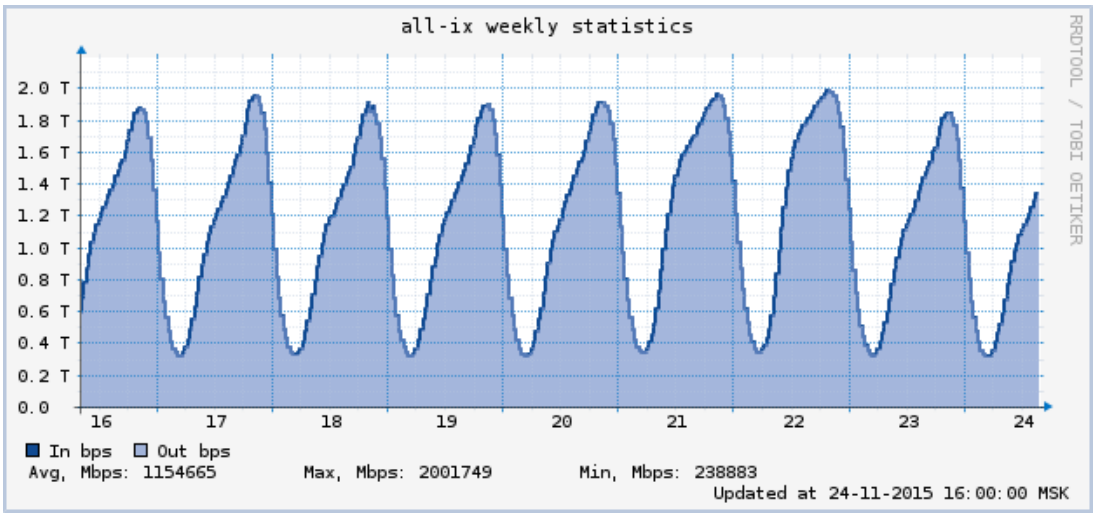


Figure 1.1: Load variation over 24h on Moscow Internet Exchange Point (MSK-IX). The gap between day and night is up to 8x.

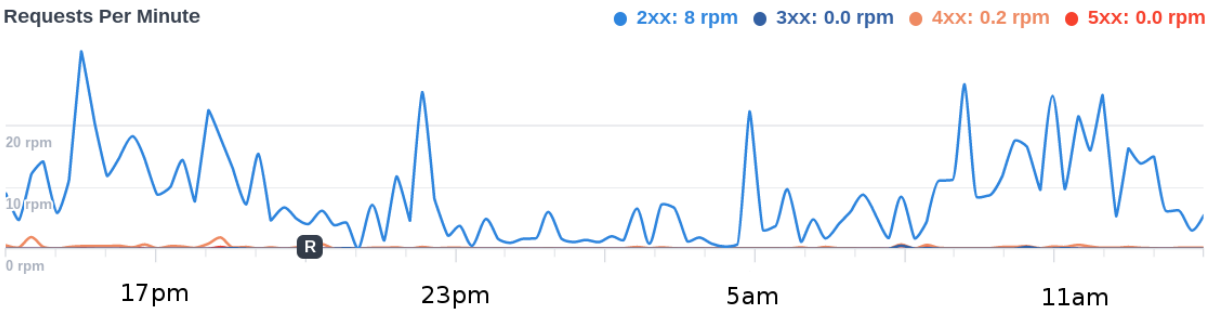


Figure 1.2: Load variation (in requests per minute) over 24h on `CoDesign.io`. "2xx" indicates normal server responses, "3xx" for redirects, the rest are for different types of errors. "R" label on X axis means there was a software update ("release"), it has no special meaning in this context.

makes it hard to deal with unpredictable spike loads. For most Internet services significant load fluctuations is more than normal, and this rendered most resources allocated statically unused at least half of the time. This underutilization is not just bad on initial expenses on equipment. Maintenance costs (energy, cooling, spare parts) are also higher, significantly raising the Total Cost of Ownership (TCO).

XaaS is the further evolution of the cloud concept: not only hardware resources, but software and services can be rent on PAYG principles. It may take many forms and shapes, but, in general, it is something hosted remotely and available through some sort of remote API. Example XaaS: software-as-service – libraries and applications that integrate into other services or to be used alone – Google Translate, Travis CI (continuous integration service), Adobe Creative Cloud (Photoshop and other famous Adobe products) and even YouTube (though it is free for most users). It can also be a storage-, database- and even algorithm-as-a-service. The key aspect of of such services, besides ease of use, is (almost) zero support costs because it is a service provider's responsibility to keep it up and running.

In this Chapter we fist discuss the rationale behind the success of clouds and how they are organized. Then we introduce the problem of resource management in the clouds – application interference, and our motivation. Then we present the structure of the thesis and topics that are covered in this work.

1.1 Clouds under the hood

From an external point of view there is no difference between a cloud and a conventional datacenter: physically a cloud is just a (very big) bunch of interconnected servers with a very good, low-latency connection to talk with the outside world. However, the main difference is not at the physical

level, but deeper inside; it is in the management plane.

Cloud resources must be well tracked and accounted for. The cloud provides customers with dynamic resource allocation: if some resources are not immediately needed they are put back to the resource pool. And vice versa: more resources are readily available if required. This provides customers with dynamic sizing of applications that may rapidly shrunk or expand depending on the demand.

Underneath of almost any cloud is *resource virtualization*. Applications no more run on bare hardware, they run in virtual containers of some kind: Virtual Machines (VMs). A VM mimics a physical node and it is almost indistinguishable from real hardware till it comes to scaling. VMs bring the following properties: *a*) multiple VMs can be collocated on the same node *b*) dynamic resource "sizing" *c*) isolation (problem with one VM does not propagate to others) *d*) relocation (they can be moved from one node to another). The ability to co-locate means denser packing: two or more customers can be put on the same server if resources allow. Dynamic sizing allows for scaling explained earlier. Isolation guaranties that security is not compromised, i.e., the vulnerability of one application cannot be used to gain access to other applications. Finally, relocation means VMs can be moved between the nodes without interruptions, also enabling seamless maintenance. This again helps scaling: applications are distributed to provide hardware footprint adequate to the load.

There are two approaches to provide scaling for applications. The first one relies on dynamic resource provisioning. Every VM is given the minimal portion of resources required to serve the load. Underutilized VMs are shrunk, overloaded VMs are given more resources. If a VM does not fit the node it is relocated to another node with enough resources.

The second approach is to maintain VMs of fixed size. When a single VM is not enough, another one is launched and the load divided. And

vice versa: if the load is not enough to keep all VMs busy, VMs in excess paused or shut down and the load is redistributed. This method requires an external load balancer for load distribution.

In practice, these two approaches are often combined. For example, Amazon micro instances[1] always provide a small baseline performance. In addition to that, micro instances that do not fully use their share are given "CPU credits" that can be used to deal with spike loads, backups or periodic activity. The peak performance can be 5 times higher the baseline.

1.2 Cloud Economics

"There is no Cloud. It's just someone else's computer."
(c) Internet folklore

"I don't understand what we would do differently in the light of Cloud Computing other than change the wording of some of our ads."
(c) Larry Ellison, former Oracle CEO

Depending on the use, clouds can be a project accelerator or a money black hole. On the strong side of clouds are ease of use, scalability, reliability, usage-proportional pricing, (almost) zero initial investment.

The downsides are potential privacy and legal issues, price benefits diminish as the project scales up, vendor lock-in, and the exposure to the cloud provider failures. Here we quickly discuss factors to consider before giving clouds a green light.

1.2.1 Why Clouds

One of the most attractive cloud features is *ease of use and access*. Most cloud providers have nice and simple web interfaces allowing for easy con-

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

figuration of most common deployment scenarios. This straight-forward approach eliminates many risks associated with infrastructure setup: an improperly-configured infrastructure is an easy victim for hackers [68].

Common maintenance burden is also much easier with clouds. This often eliminates the need for dedicated infrastructure workforce, saving headcount for projects.

Scalability. For a rapidly growing company it can be difficult to scale IT infrastructure accordingly. This is less the case with cloud providers who do a number of steps to ensure their scalability. First, big "cloud" datacenters are built in areas where they can be easily expanded or there is enough place for more datacenters. Second, there is a good practice of choosing datacenter places where electricity and thick Internet links are not an issue. As a result, commercial clouds are much better ready for expansion than a typical private infrastructure. And because of their scale, they can sustain enormous spikes of load that would normally kill a typical private cloud.

With on-premises infrastructure it is also easy to mispredict the load. Companies overestimating their growth would waste their money on excessive infrastructure capacity. Underestimating the growth is also dangerous because not every infrastructure can be easily expanded. For example, once the datacenter is full there is no physical place put more hardware. Or the datacenter can be capped by power and cooling capabilities.

Pricing and minimal initial investment. PAYG allows paying only for consumed resources. Although this may not hold true for larger instances (discussed later), it is a money-saving option for projects not requiring lots of resources. But even for larger cloud installments it may worth using clouds because of larger *gross margins*: companies prefer to put money into growth rather than own infrastructure because this is more profitable in the long run [74]. Fig. 1.3 illustrates this.

Ease of Use. Creating new applications and services nowadays as easy

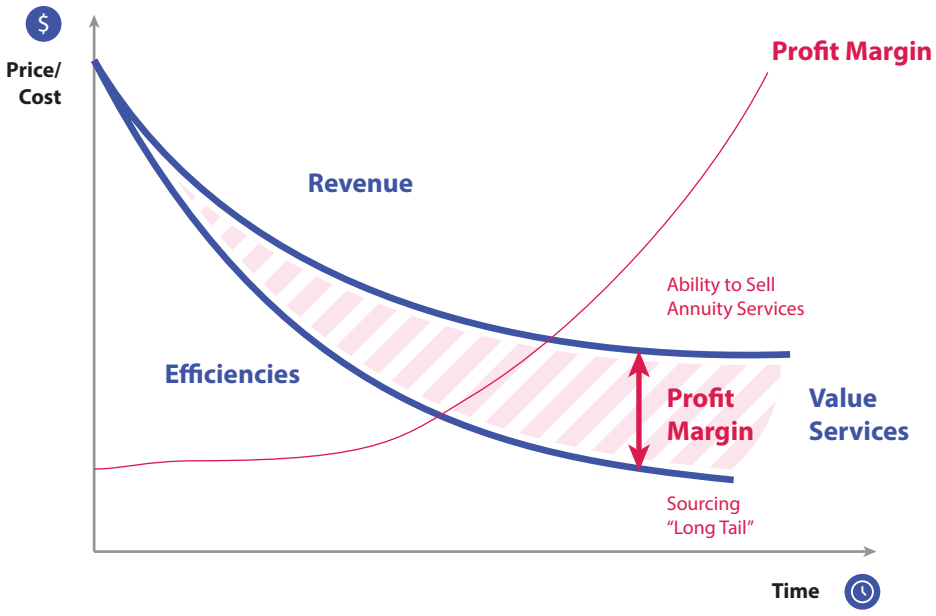


Figure 1.3: Good cloud: money saved on up-front investments helps growing the business. For illustrative purposes only.

as never before, and often can be done with a few clicks in a browser. In fact, many well-known Internet services (like Dropbox, Netflix, Airbnb and many others) are built on top of other services.

Reliability. Comparing to a single-server deployment, a proper cloud has two big advantages. First, there are always spare resources to deal with hardware troubles. Second, storage is often network-enabled, eliminating the need to transfer user data between servers in case of migration. This helps relocating user from one machine into another in case of, e.g., failure or malfunction. Some providers even have live migration.

QoS. Clouds normally have 24x7 support and rapid incident response. This is useful for small companies that cannot afford covering non-working hours with on-call support.

Legal requirements may also put restrictions on security and availability of an IT infrastructure. Building a datacenter satisfying all the needs may

be challenging and expensive. But sometimes it is possible to rent a private cloud that already meets the requirements.

1.2.2 Why not Clouds

Clouds are not always attractive, sometimes they may be undesirable or illegal to use.

Privacy and Trust. Cloud users give full access to their data to cloud providers; they do not have any control of what the provider does with it [46]. If the provider's security is breached it can potentially affect all customers.

Legal issues. Not all data can be put into the cloud. For example, healthcare data is very restricted in Europe by "EU data protection regulation". This makes impossible to use public clouds for many applications dealing with personal and sensitive data.

Pricing for large instances may be less fair. Cloud bills include fees for both hardware and services. As the scale grows the service "overhead" may outweigh PAYG benefits. Sometimes prices for large or dedicated instances are unjustified: for example, Amazon charges $2 per hour for each region of presence ("availability zone") when it comes to dedicated hardware[2]. That is $\$2 * 24h * 365d = \$17.5k/year$ per region and does not include any computational resources.

Latency. Cloud services may not be close to customers and to each other. Building responsible applications in the cloud from elementary building blocks (frontend, backend, database, authentication, file storage, etc) can be a real problem because these blocks may not be in close proximity. The author had once to solve problems with poor application performance. It turned out, round-trip time to the database was too high for applications sending many SQL queries sequentially.

[2]https://aws.amazon.com/ec2/purchasing-options/dedicated-instances/

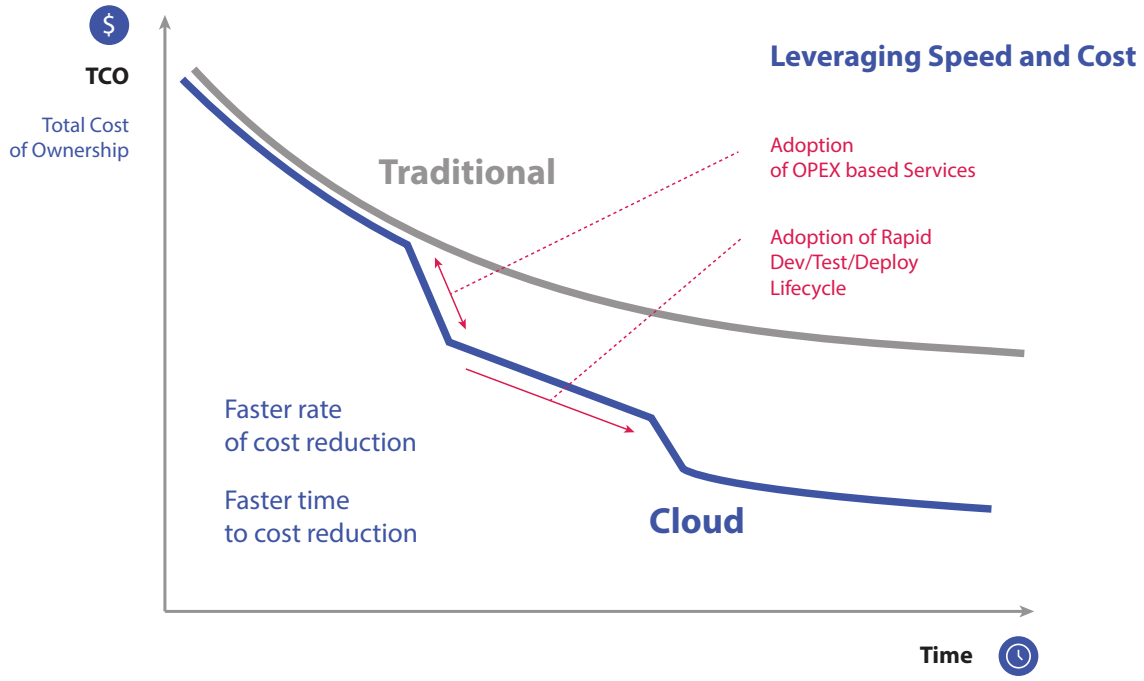
Ownership. Cloud users are normally not in charge of everything. If something goes wrong or there is a lack of functionality customers can only complain and hope to be heard.

Vendor Lock-in. Most cloud providers have competing set of services, but each uses its own API and implementation. As a result, it is hardly possible to migrate from one provider to another without a big headache. This is especially true for storage and databases: different providers have different features and performance. For example, Amazon provides "cloud" databases based on Oracle, PostgreSQL and MySQL, while Google Cloud supports only MySQL. This also complicates the interoperability between different vendors.

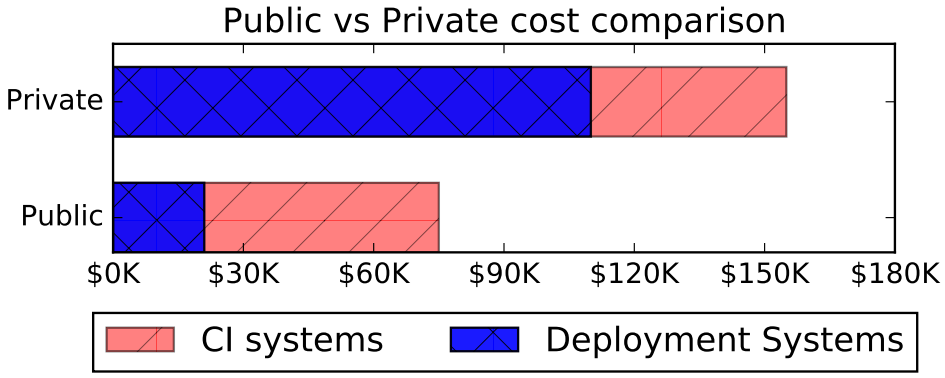
All in all, clouds are a considerable choice for startup companies because they accelerate growth. For mature companies clouds are less attractive because costs savings are less pronounced. These two faces of clouds are on Fig. 1.4. We now move to more technical discussion on one important aspect of life of applications in clouds.

1.3 Application Interference

We tend to think that CPU cores add performance linearly. E.g., the total performance is the performance of one core multiplied by the number of cores in the system. This ideal scenario is not seen in practice on commodity hardware because CPU cores have quite a lot to share. First of all, it is memory. A typical CPU has just one memory bus controller (albeit multi-channel in modern hardware), and it is shared between many, up to tens, of CPU cores. But the gap in speed between a CPU core and the main memory is so huge that memory was not fast enough since many years ago [23, 42]. The last time it could supply CPUs without the need of caches was in 1980: Fig. 1.5. Tab. 1.1 demonstrates that the memory



(a) Clouds help returning on investment by lowering TCO. OPEX (OPerational EXpense) – money spent for keeping business running. Cortesy of The Open Group.



(b) Bad cloud: at some scale and "steady state" of business supporting own infrastructure is cheaper. Source: [60].

Figure 1.4: Two different scenarios: when clouds accelerate business development and when they don't. Good clouds reduce upfront costs on infrastructure and maintenance, allowing to put saved money into business development (upper picture). Bad clouds: consider switching to a private cloud if your cloud provider charges too much (lower picture). For illustrative purposes only.

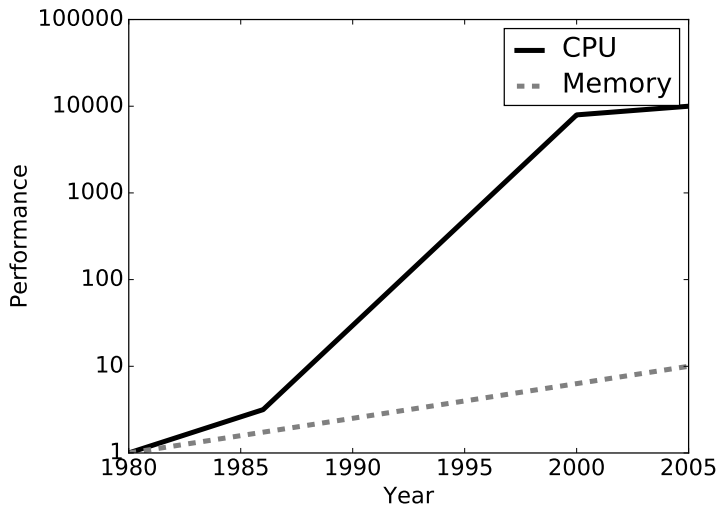


Figure 1.5: How far memory latency lags behind CPU performance. Source: [42].

CPU	**0**	**1**	**2**	**3**
0	136	194	198	201
1	194	135	194	196
2	201	194	135	200
3	202	197	198	135

Table 1.1: Memory access times (in ns) in a four CPU system. Numbers represent how fast a CPU on row n can access memory of another CPU on column m. Source: [87].

latency is far behind typical clock cycles of modern processors.

Moreover, caches are also shared. In a modern CPU every core needs to be supported with a substantial amount of cache or it will stall frequently when main memory cannot deliver data in-time. On the other side, caches occupy almost as much chip space as all other subsystems together, making them very expensive to scale up. Because of this, caches are organized into levels. L1 cache is ultra-fast and directly feeds processor's pipeline. L2 is slower, but substantially larger. Still, it takes enough space to consider it sharing between multiple cores. L3 cache is even larger (and slower) and is shared between all cores of the CPU serving as a last frontier between fast cores and slow memory: Fig. 1.6.

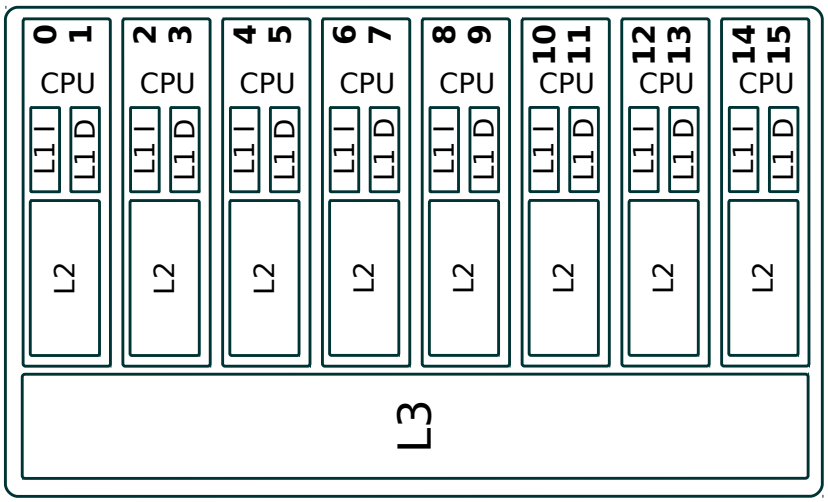


Figure 1.6: Inside Intel Xeon E5-2630 v3: every CPU core has two threads of execution (Hyper-Threading), "private" L1 and L2 caches, and one big L3-cache shared between all cores.

Not only caches are shared, but many other "CPU building blocks" are shared as well. Fig. 1.7 shows the internal architecture of a two-core module of the AMD Bulldozer CPU Family. Apart from caches, module's cores share instruction decoder, branch predictor and Floating Point Unit (FPU) block[3].

Fig. 1.8 highlights the problem. We launched one to eight instances of one machine learning tool [70] that performs comparisons between tree structures. The computing node was based on FX-8120, an eight-core CPU from AMD. If we take the speed of the first instance as 1, two instances show a total performance of 1.8. As the number of instances grows the total performance keeps increasing less than linearly. This means that every next CPU core contributes less and less to the total performance. With all cores active, the per-core performance is just $\sim 60\%$ of what was seen when only one core was active (clearly, adding more cores to this

[3]We do not mention other shared units like instruction fetcher, instruction decoder or resource dispatcher because they designed to sustain throughput of two cores and less likely to cause bottlenecks.

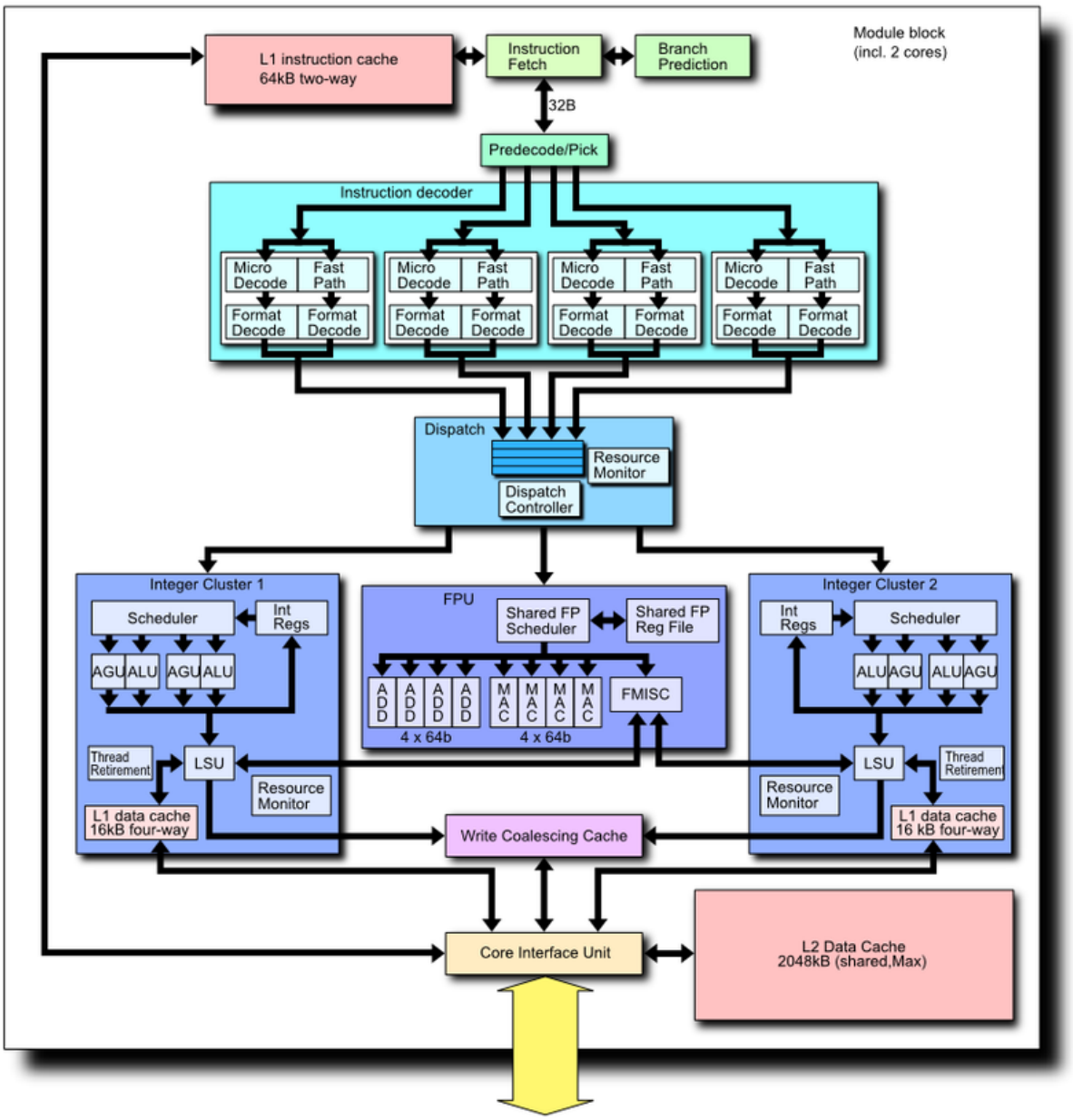


Figure 1.7: An AMD's two-core "bulldozer" module. Picture shows that not only caches, but other CPU units can also be shared: FPU, instruction decoder, branch predictor, and thelike. Shared blocks aim at increasing average block utilization and save some silicon area and power. The picture courtesy of Wikipedia contributor Shigeru23, CC BY 3.0 [1].

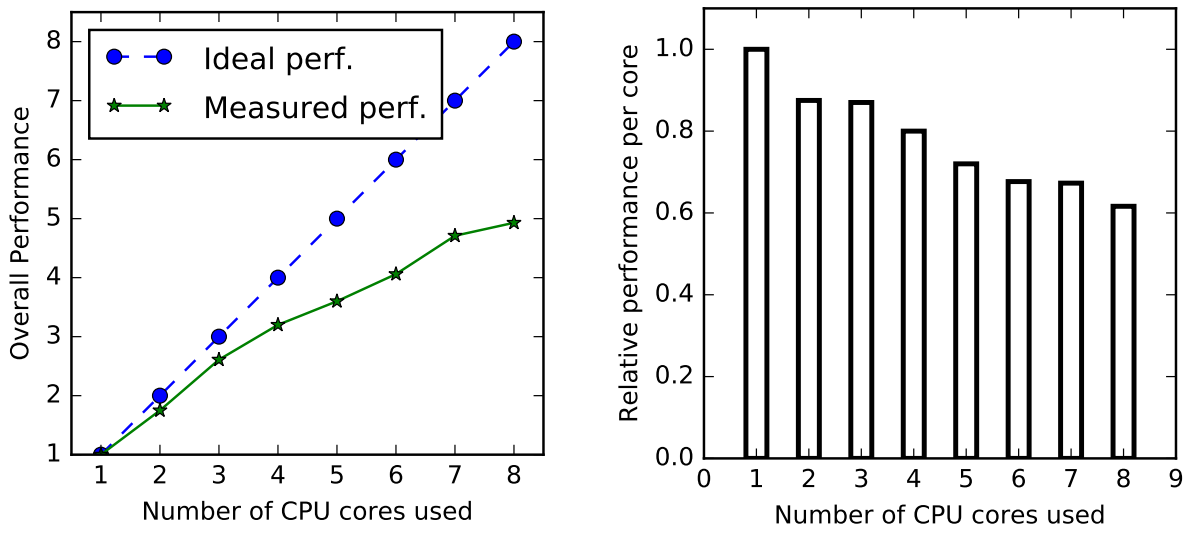


Figure 1.8: Performance scaling of SDAGP on AMDFX-8120 increasing the number of parallel instances; the gap between the two is due to shared hardware resources.

would be just a waste of silicon).

This performance degradation largely depends on the software running in the system (some programs tend to scale worse than others) and it is called *application interference*.

1.4 Motivation

Infrastructure bills can be enormously huge – $20B was spent just by Amazon, Google, Facebook and Microsoft in 2014. And, what is more frightening, the Internet keeps growing (Fig. 1.9). Internetional Data Corporation (IDC) predicts worlwide spendings to reach $107B by 2017 [35]. With such high stakes efficiency plays a major role and every percent counts. The cost, demand, trends and environment – all these have become of great concern. However, despite growing demands, there are economical limits imposed on datacenters, they cannot grow infinitely.

Is there a way around to satisfy the growing demand? The need for

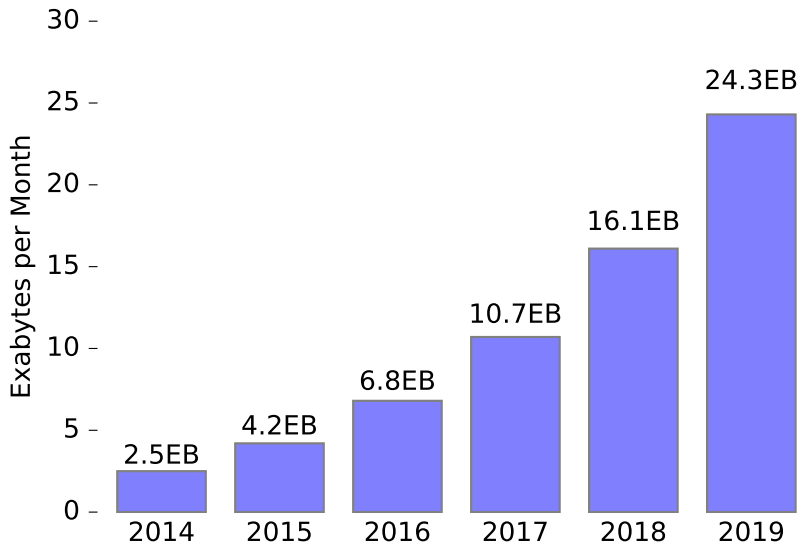


Figure 1.9: Current and future Internet traffic trends as seen by Cisco. Source: Cisco VNI Mobile, 2015 [44].

more computational power caused some drift towards new solutions like many-core systems, General-purpose Computing on Graphics Processing Units (GPGPU) systems, Application-Specific Integrated Circuits (ASICs) and Field-Programmable Gate Array (FPGA) solutions. Many-core and GPGPU systems impose significant restrictions on how a program should be written and work in order to fully utilize the advantages of the architecture. Most real-world programs cannot efficiently scale to tens and hundreds processing units [76].

ASICs are very expensive because it requires designing and manufacturing computer chips specific for a task. It is not just poor flexibility (a chip is normally designed to solve one and only one problem), but also enormous engineering efforts, huge costs of fabrication and long development cycles. This approach has currently no way to mass market.

FPGA has some market potential and may help keeping Moore's law alive for a while. They are quite versatile, but mostly used for signal processing, pattern marching, encryption, search and some other areas where

it could be beneficial to have custom ("soft") computational architecture that can be tuned for the application. Unfortunately, FPGA accelerators are not available for cloud computing to date, although Intel aims at releasing their hybrid CPU+FPGA prototypes [37] in early 2017[4].

For all these reasons traditional datacenters remain here for long, and we need to pay special attention on their efficient usage. And that is what clouds are for – efficiency. Efficiency is achieved by careful tracking and accounting cluster resources. Resource management is at the very heart of every cloud: the quality of management defines performance, efficiency and robustness.

Thus, Cloud Resource Manager (CRM) is a central part of any cloud stack. It performs monitoring, scheduling and accounting of cluster resources and it gives a single point of control for the whole infrastructure. Tasks are no more statically instantiated and assigned to computers, they are dynamically placed according to the load, Service Level Agreement (SLA) and priority. Having different priorities is useful for different computing models. For example, mission-critical applications may co-exist with normal and batch-processing applications. In this case CRM first ensures that there is always enough room for mission-critical applications. The rest is dedicated to normal applications. Any resource leftovers can be given to batch-processing jobs that can tolerate performance variability or even full eviction during peak hours.

But this flexibility is not granted, in large clouds CRM has to handle a lot of resources. Nowadays machines with tens of CPU cores and hundreds of gigabytes of RAM are readily available on the market[5]. The scale of modern datacenters has grown up dramatically, hundreds of thousands of

[4] `http://fortune.com/2015/11/18/intel-xeon-fpga-chips/`, accessed: 2015-12-13

[5] Intel Xeon E7-8800 v3 has 18 cores supporting up to eight sockets, that is 144 cores (or 288 if counting for hyper-threading) in one computing node. IBM has 12-core CPUs with 4 "threads" each, yielding 384 threads of execution in a four-socket server.

servers is not uncommon[6]. That is, a datacenter can have well over million cores, petabytes of memory and exabytes of storage. To this end, it is very important for a CRM to be very efficient because at this scale every percent of performance matters. One thing, however, is often overlooked.

A server is not just a sum of its resources because they are interconnected, particularly memory and CPU cores. This means that tasks can greatly affect each other within the same system. Therefore, a management system that considers task placement problem as a "multidimensional bin packing problem" misses a great optimization opportunity: jointly placing tasks in such a way that their interaction maximizes performance.

Apart from optimizing task placement for better performance there are other questions related to performance and efficiency. Does optimal task placement means optimal performance for every task or we sacrificing performance of some tasks in favor of others? Can we make slow tasks running faster? In a heterogeneous hardware environment, what hardware is better for a specific task and why? What factors affect system performance and hardware efficiency?

1.5 Research Questions and Objectives

In this thesis we focus on the following research questions.

Interference

We discuss why tasks interfere each other, how to measure the interference and what can be done to minimize it. We also discuss how to identify tasks that create the most interference and what placement schemes can help reclaiming lost speed.

[6]http://www.datacenterknowledge.com/inside-microsofts-chicago-data-center/microsoft-chicago-infrastructure/, accessed=2015-12-13

Hardware efficiency

Same programs may show very different performance on different hardware platforms. Moreover, depending on the settings, applications may show quite different performance even on the same platform. We discuss the effects of hardware settings (like prefetching, memory frequency, etc) and even provide clues on choosing hardware platform (Intel, AMD, ARM) and a quick comparison of ARMv7 and AMD Bulldozer platforms.

Task Classification

A cloud may run hundreds of thousands of tasks and measuring interference between all of them can be problematic. Fortunately, many tasks show similarity and we can exploit this to manage them in groups. In this work we will find out how we can identify such groups.

On-the-fly profiling

Tasks running in shared environment are not easy to profile because of interference. It may well happen that the root cause of performance issues is not in the program, but in the environment. We will present a technique of performance sampling that gives accurate estimation of isolated performance of programs in shared environments. We also evaluate accuracy of obtained data, compare how sampling time affects the result and investigate what limits precision.

1.6 Structure of the Thesis

Chapter 2 gives an overview of state-of-the-art. It mainly consists of four sections. The first one is devoted to methods of monitoring and profiling of applications in the cloud. The second section is about performance modeling and interference prediction. Then we discuss modern algorithms for resource allocation and scheduling.

Chapter 3 presents a CPU performance model that takes interference into account. The model is based on the idea that the interference between any two applications is proportional to the time they run together. Thus, the total performance is the sum of isolated performances of all tasks minus the sum of all pairwise interferences.

Chapter 4 shows a simple method for ranking tasks according to average interference they create. The performance model from the previous chapter requires an interference coefficient for each pair of tasks. Measuring them is impractical because for N different tasks we would need to measure N^2 coefficients. But we can infer coefficients by sampling hardware performance counters (performance sampling). The problem is that performance samples do not provide interference characteristics directly, we obtain these by using statistical analysis. There is one drawback: the method requires performance samples not affected by interference because interference greatly affects them in an unpredictable way. This means that while sampling there should be one and only active application running in the system – the application we deriving interference characteristics for. This is fine for research work or profiling a limited set applications, but becomes impractical for anything serious. Fortunately, we effectively addressed this issue in Chapter 5.

Chapter 5 introduces a method of performance sampling that is not affected by interference. In a shared environment the performance degrada-

tion can be very high, making it hard to guess if insufficient performance is due to the interference or problems on the application side. This approach eliminates uncertainty by providing accurate estimation of isolated performance. It works as following. Before taking a performance sample of the application in question, we temporary freeze all other applications in the system (hence why it we call it *Freeze'nSense*). Then we take a sample and unfreeze the system. The frozen phase no need to be long, often $10ms$ or less is enough. Short and fixed duration guaranties small and predictable overhead.

Chapter 6 concludes the thesis with a brief discussion of what is done, what is left to be done and future plans.

1.7 Topics outside the scope of the thesis

The focus of this work is the study and analysis of task interference on CPUs. The following topics are strictly related to cloud management and performance, but they are not addressed in this thesis.

Application Accelerators. We do not address problems with different application accelerators, namely ASIC, FPGA and GPGPU. ASIC is too specific to be used for general computing, although there are clouds providers giving their ASIC facilities for, e.g., bitcoin mining. FPGAs are also too specific, although Intel bought a major FPGA manufacturer (Altera) and we may see wider application of the technology in the future.

Many-core systems. We do not consider many-core systems with many tens of cores because of their marginal market presence. There are only two major players on this market, Intel and Oracle, and their products do not aim at utility computing. Oracle's SPARC T5 is aimed at enterprise database applications [56]. The Intel's initiative with current

name *Xeon Phi*[7] is yet to come. They also, while mimicking the traditional Intel's architecture, quite depart from mainstream CPU computing and require using special programming tools to get full advantages of the architecture [69].

I/O issues. Storage, networking and other I/O are big topics on their own and are not part of the thesis. We do not study interference induced by, e.g, shared storage.

Precise simulation. We also did not aim at precise performance predictions, our work is to steer CRM, preferably in real time, trading accuracy for speed. Therefore, we do not compete with detailed machine- and DC-level simulators.

Placement Algorithms. Full-fledged placement algorithms are also not part of the thesis. There are many well known and "approved" (e.g., multidimensional bin packing) approaches to this. However, we provide a few examples of task placement algorithms for demonstration purposes.

Data locality issues in Non-Uniform Memory Access (NUMA) systems. In large systems, just one memory controller is not enough to serve the needs of CPUs. For this reason every CPU normally has its own controller and local memory, forming a so-called "NUMA domain". From a program perspective there is no "domain boundaries" because hardware takes care and transfers data between CPUs transparently. But inter-domain transfers are expensive because they involve at least two CPUs (or more if it is needed to maintain cache coherency) and cross-domain links [63]. Therefore, the closer data is to the CPU the better the performance. Proper resource allocation place significant role in NUMA systems, but it is to some extent orthogonal to interference analysis.

[7]The project was started in 2009 and changed many names since then: Larrabee (supposed to work as a videocard), Knights Ferry, Knights Corner, Knights Landing and, finally, as Xeon Phi to market it for high-performance server solutions.

Chapter 2

State of the Art

2.1 Monitoring and On-the-Fly Profiling

Most of resource management algorithms consider CPU cores as unified resources adding performance in proportion to their number. That is, a six-core CPU to be three times faster than a two-core CPU running at the same frequency. In reality the performance gain may vary due to the subsystems shared between cores. These subsystems can include CPU caches, memory bus, I/O lines, instruction decoders, branch predictors, computational units and other components. Therefore, under heavy load it is unlikely to see a linear gain in performance when adding more CPU cores. In fact, an increase the number of cores can even degrade VM performance for up to 50% due to the inter-VM interference [96, 9].

Profiling and monitoring applications to evaluate their run-time performance is a multifaceted problem, especially when the goal is to understand and to manage the performance of production environments. Several different techniques have been proposed, but given the complexity of the topic, the variability of the environment, and the difference in final goals, they are rarely comparable one another. It is also surprising that general methodologies are somewhat lacking in the literature. We revise here the works that are comparable to our proposal, even if their final goal is different, or

if their applicability is fare more specific compared to the methodology we propose, which is instead very general.

The most obvious way to profile the interference experienced by an application is to run it alone and compare its performance in isolation with its performance in a shared environment [49, 62]. This approach can be considered a benchmark, but it is not feasible as a production means. First, extra hardware resources are required. Second, given the enormous number of different applications and their specific customization, repeating the measurement for each of them is cumbersome and time consuming. If the spare hardware is limited, then the profiling cycle is also very long as all the different applications must be loaded and measured on the spare hardware. A long cycle may prevent a timely and efficient identification of bottlenecks, because applications can change their behavior in time.

A different perspective is represented by a cluster-wide massive data collection [95]. The idea is to collect performance statistics from different instances of the same application. If the statistical properties of the application are known or can be inferred with some techniques then the methodology can identify instances that are under-performing. The approach is suitable for very large installations running many (stochastically) identical instances of the same application, while it fails for single and unique instances, but also for applications whose instances can be customized so that they are no more stochastically comparable. Applying this technique at the level of VMs is even more difficult as assuming identical configuration of VMs is far fetching.

These methods have in common that they do not rely on measurements taken from single production servers that share resources between many applications. They either try to gather statistics in an isolated environment, or they try to infer the isolated performance out of many data points. It is clear that identifying the performance in isolation measuring the per-

formance only in a shared environment can be difficult, as the "ground truth" is missing.

However, there are techniques that can work in a production environment without requiring many identical instances of the same application, and our work fits into this class.

In [65] Jia Rao et al. studied how pinning the threads of multithreaded applications to different cores influence the performance due to the data locality on NUMA systems [53]. Different thread-to-core mappings lead to different performance, but it is unknown in advance what mapping is the best. Thus, the proposal is based on random relocation of the threads to select the mapping that performs better. The relocation process is done in an initial time window during which the performance data of individual threads is recorded. At the end of the window the best allocation is chosen and the application continues to run steadily. The performance sampling is done with performance counters with sampling interval of 10 ms. The performance measure is based on the score derived from the number of L2-cache misses. The higher the score the more chances the thread is suffering. To accommodate changes in programs' behavior the process of relocation must be repeated regularly. Unfortunately, the overhead of the relocation and measurement process is not negligible, specially when there are many threads in the system.

Intensive I/O operations performed by VMs may be difficult to account when the hypervisor is shared like in Xen [6]. This happens because the VM itself has no rights to perform such activity, so that the I/O operations are actually performed by the hypervisor and accounted to it [36]. Thus, the identification of performance issues due to I/O conflicts becomes very difficult: all the activity is accounted on behalf of the hypervisor. The authors propose a "sidecar" extension to the hypervisor that traces the I/O activities by parsing the log file. The log is periodically (every 100 ms by

default) parsed and the number of I/O operations for each VM is counted. Then, knowing the average CPU cost for each type of operation, it is possible to adjust the CPU consumption of each VM so they fit their performance limits or allocation targets.

Wood et al. proposed and implemented a gray-box monitoring for VMs [89]. In their system VMs are monitored both externally, using, e.g., statistics from the hypervisor, and internally, by running a small monitoring application inside each VM. The application exports performance statistics as it is seen from inside the VM. This helps to understand the way resources are consumed, as well as detect performance issues. The latter are detected by analyzing performance metrics exported by applications and the OS. If performance is not meeting the Quality of Service (QoS) constraints over a "sustained period" then there is a bottleneck and the VM is relocated to another machine with more resources available. The distinctive part of this work is the heuristic used to understand the trends in VMs' behavior. For each resource there are two derived metrics calculated over a sliding time window: distribution of the value and raw time series. The distribution is used by the migration manager "to estimate peak resource requirements and provision accordingly". Time series show if a resource utilization rises, falls, or remains steady and they are used by the so called "hotspot detector" that drives the decisions to move the load from highly utilized servers to servers with lower load.

Another technique is tailored to profile and consolidate databases [27]. Many database are hugely over-provisioned in terms of CPU and memory. Due to the aggressive caching performed by modern database engines it is very hard to estimate the real memory requirements for a database. The performance estimation is done by creating an artificial probing load that gradually increases the memory consumption and reduces the memory available to the real workload. As soon as the database noticeably slows

down we can conclude that the minimum efficient memory footprint is reached and the memory occupied by the artificial workload is the memory the database does not really need to run efficiently. The required CPU timeshare is calculated as the sum of individual timeshares of each database as if they were running alone. The required disk bandwidth is estimated from an empirical model based on multiple synthetic tests. The average probing overhead is claimed to be just 5%. The technique is aimed at large databases running directly on hardware; load balancing is done by moving tables between the databases.

Chopstix [19] is a Performance Monitoring Unit (PMU)-based tool for applications profiling. It has a traditional three-phase architecture (collection → aggregation → analysis) with one distinctive feature: the performance sampling is probabilistic and done in adaptive intervals. That is, code functions that have less samples are more likely to be sampled [54].This greatly reduces the overhead of profiling and the size of collected data by reducing the sampling rate for the functions that already have a lot of samples.

In [31] researchers studied the possibility of predicting performance degradation using regression trees. They first analyzed the correlation between performance counters and the level of interference. This let them to drastically reduce the number of features: from 340 down to 19. Then they used WEKA [38] to build the regression tree. The tree was used for interference prediction of previously unseen workloads. The model was tested on two platforms (Intel and AMD) and showed the absolute error below 20% on the 80th percentile.

The training set needs to resemble the behavior of cluster applications or estimations will not be accurate. The model warns if two or more features are out of the training range, the model yields an error and the application is considered as so that not enough covered by the training set.

In [32] authors proposed a hardware modification suitable for measuring performance degradation in Simultaneous Multi-Threading (SMT) environment. They upgraded PMU of DEC Alpha CPUs to categorize CPU cycles into three types: base (normal execution), miss event (cache misses and branch misprediction) and waiting (execution stalls). They also estimated the increase in cache misses due to resource sharing. The result were evaluated in *SMTSIM* simulator and showed average prediction errors of 7.2% and 11.2% for two- and four-thread SMT respectively. Unfortunately, modification of hardware is expensive, and such proposals rarely have their way into production.

2.2 Performance Modeling

Performance estimation tools for computing systems majorly fall into two categories: high-level models and precise cycle-accurate simulators [55]. The latter category provides nearly exact results at the cost of speed and complexity. Flexible simulators like [18] can not only predict the actual performance of the tasks but also give a hint concerning the bottlenecks. However, simulation speed is a limiting factor. Execution inside fully fledged circuit simulators is too slow, making this technique useless for on-line performance prediction in cloud environments.

High-level models do not try to simulate the behavior of processors. The core of such systems is either an empirical or an abstract model (or a group of loosely coupled models) based on observations. High-level models neither interpret nor execute programs: they operate on traces and performance data obtained trough the normal execution of the programs. They try to predict the overall performance based on the past experience and interaction patterns that are mapped on the model parameters.

As an example of high-level models, the early work [82] proposed a

queueing network model of the memory architecture of CPUs, which was suitable for some classes of tasks.

Another simple performance model dedicated to threading applications and taking into account NUMA topology was proposed in [92]. The model shows a mere 15% error in general, which already enabled predictive management.

The same authors in [91] explored the effects of dynamic page migration and its applicability for Gaussian 3, a multi-threading chemistry computational tool, with a similar model.

A model that tries to predict the performance of threading applications and whose goal is the optimization of both thread and memory allocation is presented in [16], but the model goal is not on-line prediction, rather it is off-line understanding of different NUMA implementations and their threading performances.

The work presented in [11] discusses a hierarchical model used in JavaSymphony, a high-level framework for parallel and distributed systems providing transparent access to remote data as if the data were local. It provides means and tools to build a hierarchical memory model that includes not only local resources (processors, memory bus, interconnections) but remote resources (remote machines and clusters) as well.

In general, all modern task schedulers are aware of data locality, CPU caches, and NUMA domains. For instance, Linux attempts to calculate the performance penalties for task migrations and memory allocations outside the current NUMA domain. This mechanism is usually tunable to accommodate different hardware systems, but does not contain a self-tuning feature.

A novel method of memory allocation is presented in [57] (again for NUMA systems). Usually the memory is allocated during the first access (delayed or lazy allocation). However, the first access is often done from the

initializing thread and the rest of the time the data is accessed from another thread possibly in another domain. The developed algorithm detects such cases and moves the data to the closest possible domain.

Thus, a large body of work exists on performance evaluation and performance impairments due to locality violation, unwanted tasks interaction, etc. However, a systematic study of how these performance impairments can be predicted and how they are influenced by different hardware is missing and this is exactly the kind of tools that are needed for the management of large, heterogeneous facilities supporting cloud computing.

2.3 Task-aware Scheduling

In [73] authors present an interesting approach to optimize scheduling of multithreaded programs that extensively use shared memory. They found that there are cases when Linux cannot handle big shared data structures efficiently. It was proposed to split the load between multiple OS instances so each of them would be in charge of just a small portion of shared memory. The optimization requires multiple OSes, a modified Xen hypervisor, and applications statically linked with a small helper library. The key advantage of the technology is that, thanks to virtualized system calls and memory management, one process may span across multiple OSes. The resulting structure is called SuperProcess, and it is supervised by Virtual Machine Monitor (VMM). VMM manages system calls (including file system that appears to be transparently shared) and memory management so that all processes that form SuperProcess have a consistent view to the resources. The optimization framework was tested on two machines (16 core Intel and 48 cores AMD) with reported average speedups of 1.7x and 4x under full load.

REEact, an execution manager that cooperates with programs, was pre-

sented in [83]. Cooperative programs inform the manager about resource policies they want and the manager dynamically adjusts their execution accordingly. Apart from CPU and memory, tracked resources include temperature, frequency, number of active threads or even memory prefetching. Such a rich set of monitored resources allows the manager to define multiple global goals, such as maintaining the node power or cooling budget or aiming at best power efficiency. Programs may request more threads to be spawned when there are spare resources or adjust CPU limits according to the changing load. It was also shown that disabling prefetching when it does not help can save up to 12% of energy with speedup up to 8% comparing to when it is always enabled. The speedup by prefetching was determined by comparing the system performance with and without it ("start and stop" technique). The manager showed less than 3% overhead, worked on Linux and was evaluated on x86 and SPARC.

In [41] the authors studied two options for application scaling: more threads vs processes. The study performed on Lighttpd webserver and an eight-core system. It was found that, depending on the number of active cores, these approaches produce different results and none is the winner in general. Threads allow for some memory savings because they share common data. But for this reason the performance degrades significantly when they spread over multiple NUMA domains: *a*) processors have to keep shared data in sync, and *b*) access to non-local data is much longer [81]. The best results are achieved by the hybrid approach when every CPU runs its own threaded instance of application.

However, there might be a potential issue with the experimental setup. It was spawned 128 processes per CPU core to fully load the system. Such a big number of active processes may indicate that the bottleneck was not related to CPU or memory. Increased latency could be caused by

task switching, cache displacement and longer scheduler queues[1]. For this reason high-load systems avoid spawning too many processes (and $8 \cdot 128 = 1024$ processes is a bit too much for an eight-core system).

ReSense [28] optimizes tasks placement according to their memory resource consumption. It has two phases, offline and online, and uses performance counters to estimate usage of memory bus and caches, this is done in isolated environment on the target platform (offline phase). Then sensitivity score is derived for each application. During the online phase *ReSense* optimizes task placement according to their sensitivity score. The optimization is to be repeated when the number of active threads is changed. The strong point of this work is the support of multithreaded programs. The proposed method also differentiates two levels (types) of resource contention: between threads of the same application and between different applications.

In [78] authors clustered threads based on their data sharing. If threads frequently access the same memory (at L2 cache-line granularity) and run on different CPUs they are to be placed together. But only if the CPU stalls are above the threshold, otherwise data sharing is considered a non-issue. Access patterns were detected with PMU counting for L1 data misses that were satisfied by remote L2 and L3 caches. The optimization could reduce sharing by up to 70% on the Power5 platform. The performance boost was more moderate – up to 7%, mainly due to the Out-of-Order (OoO) execution and thick inter-chip links.

Zhuravlev *et al.* [97] developed a "contention-aware scheduling" algorithm that balances miss rates among the Last-Level Caches (LLCs). The balancing was done according to estimated mutual interference via the cache. They evaluated six different approaches for interference estimation,

[1]Although the modern Linux scheduler CFS uses red-black tree instead of plain queues ([88]) the reasoning is still valid. The tree does not eliminate latency, but majorly reduces latency spikes

the most promising was based on stack distance profiles [24]. Unfortunately, due to the implementation difficulties this algorithm was not used for online scheduling. They used heuristics based on cache miss rates of individual applications that also showed good estimations. Reported average improvement was 20% with up to 50% for individual applications. The scheduler showed to be good in reducing performance variation between different program executions. The best improvement was observed when memory intensive applications neighbored with non-intensive.

Cache Coloring (Partitioning)

These techniques are a little bit apart from what we are doing, but they serve the same purpose: optimize data access in multicore environments. Therefore, it is worth mentioning them.

Common Instruction Set Architecture (ISA) do not allow for cache control overriding, i.e., a program cannot tell the CPU the importance of different pieces of data. A not-so-uncommon situation is when live data is constantly evicted while access-once data may be held for longer. This is because cache metadata (cache tags and flag bits) may not have enough information to understand complex data access patterns. Notwithstanding that, knowing how the CPU uses caches a programmer may enforce desired cache store policy by placing data in specific addresses.

The problem comes from the fact that continuous addresses in virtual memory are not continuous in the cache [40]. This means that adjacent virtual pages are not adjacent when cached and they may even point to the same place in the cache. This leads to underperformance of the cache because *a*) some cache lines may be less used than others; *b*) some cache lines may be overwritten too frequently; *c*) two applications may fight for the same cache lines. This can be avoided by ensuring that continuous virtual addresses are mapped to continuous virtual addresses. Furthermore, it is

possible to partition the cache between the tasks so they will not overlap in the cache. A few works that tackled this issue.

In [77] authors introduced an OS-level cache partitioning. It was done through a modified process of physical page allocation that required no changes in applications. The algorithm managed L2 cache on IBM Power5 and gave up to 17% speedup. The cache was not divided into equal portions, but dynamically partitioned according to the needs of applications. To achieve this, the performance curves (instruction stalls and L2 miss rates vs partition size) were obtained for every target process. Then each process was given a piece of the cache to minimize total (system-wide) instruction stalls. The cache could be partitioned up to 16 equally-sized blocks because the target CPU uses the last 4 bits of the page address to determine the data location. It was observed that most applications are comfortable with just two blocks (256KB) of L2 cache. This is where the speedup came from: only the applications that could really benefit from larger portions of the cache were given them. The run-time performance statistics was obtained with performance counters.

There are some inherited problems peculiar to these techniques. First, different CPUs have different cache configurations, algorithms must adjust to the running hardware. Second, the CPU's own resource planners may interfere.

Another possible solution would be to override cache control and steer caches from software ([17, 25, 39]). But as of now none of the most common arctitectures (x86, ARM and MIPS) implements this. Therefore we do not mention these works.

Chapter 3

Modeling Tasks Inter-Core Interference

3.1 Introduction

The first step to solve a problem is to recognize it, and devise a first, simple conceptual model that represents it. Inter-core interference has been recognized as a problem in computation since many years [48, 90, 23, 30, 92, 82], but very often either disregarded or dismissed as a minor problem, compared to others [47, 72, 80, 33]. We have discussed the state of the art on data-center management and we have also discussed some works that do represent in some way inter-core interference, but we think a more specific model is needed.

This Chapter introduces a simple, first order model that captures the influence of one task running on a core on another task running on another core. Advanced modeling techniques like [22, 8, 61] can be used in future works to refine the model once the key features of interaction are better understood with simple models.

In contrast to more complex full-featured simulators like [18], our model does not require a complete knowledge about CPU internals and, hence, it is more general and simple to apply. The model answers to two major

questions: *i)* how a given set of tasks performs simultaneously, and *ii)* what tasks are better neighbors one another.

The contribution of this Chapter is thus twofold. First of all, we show and highlight to what extent tasks running on different cores of the same CPU can affect each other performance, casting light in a phenomenon that is qualitatively well known, but quantitatively largely ignored in data center management. Second, we analyse to what extent a simple model that is suitable for on-line training and tuning can be used as a prediction tool to enhance CRM systems.

The remaining is organized as follows. Sect. 3.2 introduces the problem with some initial and simple measures of performance and efficiency and introducing our terminology. Sect. 3.3 contains the major contributions, formalizing the problem and describing the behavioral model. Sect. 3.4 describes the metric we use to evaluate models of performance prediction. Sect. 3.5 presents methodology we use for the model validation. Sect. 3.6 provides a deep analysis and discussion of the results. Sect. 3.7 describes how the parameters of the model can be obtained and Sect. 3.8 concludes the chapter summarizing the contribution.

3.1.1 The Benchmark Programs

Before going further it is worth describing benchmark programs we use for experiments. For the sake of convenience, we describe the whole set of benchmarks present in the thesis. In this Chapter only SDAG, SDAGP, MATRIX and MATH are used (described below).

The choice of benchmarks was not random. To make our studies more comprehensive, we chose programs of different and distinctive classes with different memory footprints, cache-awareness, utilization of memory bandwidth and CPU arithmetic units. Here they are:

1. **MATRIX**: a program performing matrix multiplication of randomly-

generated square matrices. It is based on Basic Linear Algebra Subprograms (BLAS) library – an industry standard for such kind of computation. BLAS takes roots from late 1970-x when first specifications were published.

We used to use a quite trivial Python script with some NUMPY routines (another golden standard for such kind of computations) but it showed great variability in results due to intensive garbage collection, randomised data placement and other factors we could not account for. So we replaced NUMPY with BLAS rewrote program in C.

2. **SDAG**: a machine-learning program from natural language processing domain [70]. The program uses Support Vector Machines (SVM) to build compressed syntactic trees from text. It written in C and uses state-of-the-art processing techniques and manual optimizations allowing it work very fast. Particularly, the memory layout is very cache friendly.
3. **SDAGP**: a machine-learning program from [70]. It mostly resembles the previous one, but with one difference: it does not attempt to split the training set into smaller chunks for parallel processing, the data processed in whole. For the algorithm it is not a big deal, but for the CPU this means much bigger active dataset. As a result, there is much higher pressure on caches and the memory bus.
4. **BLOSC**: a high-performance compression library [12]. Modern CPUs can greatly outperform memory and this is not what can be easily fixed. One potentially good approach is to use data compression. In a multi-core system it is often feasible to dedicate one-two cores for compression and data delivery while other cores access readily available data from CPU caches. This approach increases effective memory bandwidth by the compression ratio. Our benchmark script

is a simple program the sequentially compresses and decompresses $3e6$ numbers (of type float64) evenly distributed between 0 and 100.

5. **FFMPEG**: a set of libraries and programs for decoding, encoding and re-encoding multimedia streams. It is one of two tools (another one is GStreamer) extensively used for video processing. If an application does something with audio or video, chances are one of these libraries is used. As a benchmark we used slightly different scenarios. For this Chapter it was a transcoding of a 1280X720@25Hz video (obtained from a camcoder) to a higher compression (lower quality). For Chapter 4 and Chapter 5 we scaled FullHD trailers down to 720p (Terminator 2 and Avatar trailers correspondingly).
6. **NGINX**: (pronounced as "engine x") a very popular webserver that, as of November 2015, serves 27.9 millions of sites [7] all over the world. It uses asynchronous architecture and heavily tunned for performance. With NGINX a single machine (though a powerful one) can handle hundreds of thousands simultaneous connections with some tens of thousands active. Typically, it is used as a reverse proxy for slow applications or a load balancer.
7. **MATH**: a small ad-hoc C program performing basic integer computations. Ages ago, when floating point computations were expensive, people used to use so-called "fixed-point arithmetic" because it was much faster. Nowadays it less likely to find an application that would do intensive integer computations because floating point is fast enough for most applications (and even if not it is possible to use GPU for acceleration). Yet, to have a comprehensive set of benchmarks, we decided it is worth including a program that would do basic arithmetic with integers and just it, nothing else.
8. **INTEGER**: same as MATH, we renamed this benchmark in later works so the name better describes what it does.

9. **WORDPRESS**: a popular web publishing platform serving more than 60 millions websites [26]. It is not the only of its kind, but it is the most used one. Technically it consists of two big components: the application code and the database. As database we used MySQL because it is the most popular choice for such kind of services. The benchmark is just to access the main page. Nonetheless this creates serious load for the server (mostly by PHP, much less by the database). We did not used any PHP accelerators or any caching, this a pure "dynamic" load.
10. **PGBENCH**: a benchmark for PostgreSQL. The presence of this benchmark is very important for two reasons. First, SQL databases are very popular for data storage and (online-) processing. Having at least one of such solutions is essential for any thorough server performance evaluation. Second, PostgreSQL is the leading open source solution when it comes to performance and reliability of large installations. Any other relational open source database, should it be MySQL (and its derivatives), SQLite or something, just does not have so much all at once. We chose PGBENCH as a very representative benchmark of what people do with databases "on average": reads, updates, deletes, all wrapped in transactions. Sure enough, the TPC-B (the standard this benchmark implements) is quite old (1990), but more sophisticated successors emulating more complex scenarios (like, how a bank works) would not make it any better for our purposes. We run the benchmark in 20 concurrent threads, a more or less typical value for PostgreSQL on loaded servers.

3.2 Problem Statement

One of the most important role of a CRM is to minimize the amount of equipment to be provisioned (i.e., maintained active) in order to maintain the SLA (Service Level Agreement) with customers. This is achieved by dynamically assigning tasks to hardware resources, i.e., consolidating the load into a smaller number of nodes when the system is over-provisioned and increasing the resources when it is approaching overload. In both operations the capability to predict performance is fundamental to achieve efficiency and minimize reconfigurations.

Most CRM algorithms assume that the load of nodes is fully determined by resource requirements of the tasks, and the performance scales linearly with the load; in particular tasks running on different cores are considered independent. If, for instance, two tasks require one CPU core each, the CRM can put them together on any node having two cores free, assuming they will run without interaction. In the next section we show this assumption is wrong.

3.2.1 A Simple Experiment

As discussed in the introduction, most CRMs today consider tasks assigned on different cores and CPUs independent one another: is this assumption reasonable?

let us set up a simple experiment with a single two-core machine: take four tasks, run them in the cluster (two tasks per core) with different allocations, and measure the joint performance. The choice of the four programs (SDAG, SDAGP, MATRIX and MATH) is basically random, but we had some attention to select them with different characteristics, at least to a first heuristic examination. The selected programs are described in Sect. 3.1.1 together with others that we use for the model validation.

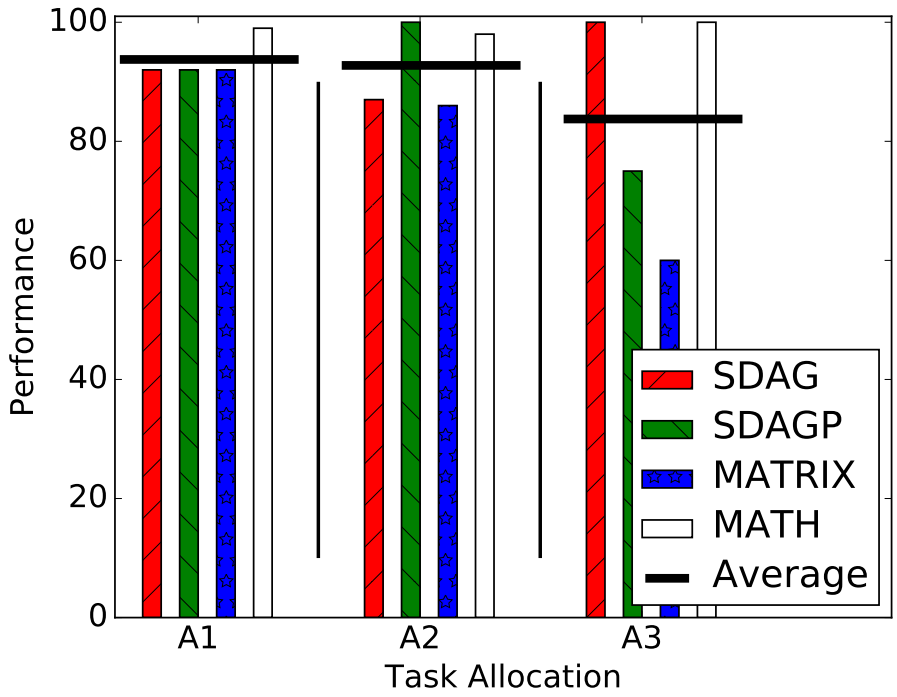


Figure 3.1: Performance of four benchmark programs in three possible allocations A1, A2 and A3.

We consider all the three possible assignments (distributions) of tasks to cores: as cores are identical the other three permutations of four tasks in two cores maps to the first three.

The results are presented on Fig. 3.1. The performance is normalized with respect to the same tasks running alone on the computer. As we see, the performance never reaches the maximum even in the best distribution. Indeed, the task assignment has a huge impact, lowering the average performance from roughly 95% to about 80%. What is worse is the fairness with one program (MATRIX), whose performance is nearly halved in one assignment.

There are many reasons for such a bad behavior, but they can mainly be ascribed to the shared parts of the cores and CPU. For instance, L2 cache is usually shared between each pair of cores; the latest generations of AMD processors have only one FPU per two-cores package; memory bus,

instruction decoders and other circuits are often shared to increase their average utilization and to reduce the silicon area occupied. By assigning two tasks to the neighbor cores we can create a hidden bottleneck due to the shared usage of CPU subsystems (or to the other extreme speed-up the execution if the tasks share some data; more on this in Sect. 3.6).

A discussion on its own would be needed for hyper-threading and the related technologies, but this is beyond our scope.

3.3 A First Order Interaction Model

As proven by our simple experiment, the "load" of a cluster is not simply the sum of the loads of each running task as if it were alone, but it must take into account also the overhead induced by interaction of the tasks. We can treat the overhead as an additional load for the sake of performance prediction. With this simple observation in mind we derive a simple behavioral interaction model with the assumption and notation introduced hereinafter.

Assumptions

We assume that the overhead is present only when tasks are actually running in parallel, and we are interested in the steady state performance, i.e., we consider tasks, as for instance video encoding, that run for long time. We also assume that the dominant interaction is pairwise, i.e., that if multiple tasks run in parallel, their overhead is equivalent to the sum of the overheads of the tasks running in pairs. We expect this assumption to be a non marginal approximation, and to lead to an overestimation of the overhead.

Notation

We define the task load $L_i = L(T_i)$ on a core as the time share required by task i on the core. The per-core load is the sum of loads created by all tasks running on that core and is in the range $[0, 1]$. The number of cores is N. We define the system load L_{sys} as a sum of loads of all CPU cores in the system; it is in the range $[0, N]$. The pipe notation $T_i|T_j$ means that tasks T_i and T_j are running on different cores.

Let's consider the following scenario. We have two tasks A and B running on CPU cores 1 and 2 and exclusively occupying all CPU time. As we noticed, the joint performance in this case is less than what normally expected, and thus we have an overhead that can be expressed as a parasitic load $L^{oh}_{A|B}$, leading to a system load that can be expressed as

$$L_{sys}(A|B) = L_A + L_B + L^{oh}_{A|B} \tag{3.1a}$$

$$L^{oh}_{A|B} = \beta_{A|B} L_A L_B \tag{3.1b}$$

where $\beta_{A|B}$ expresses the level of interference of the tasks A and B as a function of the product of each task load. Remember that loads are represented by time shares, so we can imagine that the product of the time shares of two tasks represents the time during which the tasks actually interfere as they are running together on the cluster.

Instead of introducing β-s we could extend the vector of resource usage of the tasks with more components (like FPU and ALU utilizations) to tackle the problem. There are two reasons for not going this way. The first reason is the fact that the utilization of individual low-level computer components often cannot be measured. The second reason is that taking into account so much data is difficult as we already have a lot of parameters like amount of needed CPU resources, required memory size, storage space, reserved network bandwidth and other parameters. Adding thirty or forty more parameters would greatly complicate the decision process of the CRM

which is already quite complex [80].

The overhead function can be decomposed to represent the performance penalty for individual tasks:

$$L^{oh}_{A|B} = \beta_{A\to B} L_A L_B + \beta_{B\to A} L_B L_A \tag{3.2}$$

This is useful, for instance, for real-time systems when we want to know how high-priority tasks are affected by the other non real-time tasks.

Formula (3.2) can be transformed into a matrix form, which will come very handy in the generalization of the model to multiple tasks and many cores. The tasks running on core i are represented by the vector $\mathbf{T}_i$ of loads they generate. The values of the vector components are in range $[0, 1]$ with 0 meaning that the corresponding task isn't scheduled on this core and 1 meaning it is using the core all the time. These vectors are sub-stochastic, meaning that the sum of their components is less or equal to 1. The matrix form of (3.2) for a two-core architecture two tasks A and B is:

$$\beta_{A\to B} L_A L_B = \begin{vmatrix} L_A & 0 \end{vmatrix} \begin{pmatrix} \beta_{A\to A} & \beta_{A\to B} \\ \beta_{B\to A} & \beta_{B\to B} \end{pmatrix} \begin{vmatrix} 0 \\ L_B \end{vmatrix} \tag{3.3a}$$

$$\beta_{B\to A} L_B L_A = \begin{vmatrix} 0 & L_B \end{vmatrix} \begin{pmatrix} \beta_{A\to A} & \beta_{A\to B} \\ \beta_{B\to A} & \beta_{B\to B} \end{pmatrix} \begin{vmatrix} L_A \\ 0 \end{vmatrix} \tag{3.3b}$$

or in a compact and more general form:

$$L^{oh}_{1\to 2} = \mathbf{T}_1^T \mathbf{B} \mathbf{T}_2 \tag{3.4}$$

where $\mathbf{B}$ is the matrix of β-coefficients, $\mathbf{T}_1$ and $\mathbf{T}_2$ are the vectors of the tasks running on cores 1 and 2 respectively and $L^{oh}_{1\to 2}$ is the overhead that all tasks on core 1 impose the tasks on core 2.

Diagonal elements ($\beta_{A\to A}$, $\beta_{B\to B}$, ...) are of special interest. They tell us how a process affects the same processes running on another core which is a typical situation in many cases like web-servers, databases and other

cases. Knowing the overhead and using Amhdal's law [15] we can estimate the scalability limitations of the system.

In order to generalize the model let's consider the case of two cores and three programs (A, B, C); on the first core one copy of each program is running, while on the second core only the programs B and C are running, hence we have $(\mathbf{T}_1^T = |L_A^1\ L_B^1\ L_C^1|^T)$ and $(\mathbf{T}_2^T = |0\ L_B^2\ L_C^2|^T)$. The program A isn't launched on the second core and therefore its time share is zero in $\mathbf{T}_2$. The overhead created by core 1 tasks on core 2 is:

$$\begin{aligned} L_{c1\to c2}^{oh} &= \beta_{A\to B}L_A^1L_B^2 + \beta_{A\to C}L_A^1L_C^2 \\ &\quad +\beta_{B\to B}L_B^1L_B^2 + \beta_{B\to C}L_B^1L_C^2 \\ &\quad +\beta_{C\to B}L_C^1L_B^2 + \beta_{C\to C}L_C^1L_C^2 \\ &= |L_A^1\ L_B^1\ L_C^1| \begin{pmatrix} \beta_{A\to A} & \beta_{A\to B} & \beta_{A\to C} \\ \beta_{B\to A} & \beta_{B\to B} & \beta_{B\to C} \\ \beta_{C\to A} & \beta_{C\to B} & \beta_{C\to C} \end{pmatrix} \begin{vmatrix} 0 \\ L_B^2 \\ L_C^2 \end{vmatrix} \\ &= \mathbf{T}_1^T\mathbf{B}\mathbf{T}_2 \end{aligned}$$

It is easy to see that (3.4) is a general representation of the overhead imposed by one core on another, regardless of the number of tasks per core.

The description of the total overhead in an N-core CPU is given by the sum of all the overheads imposed by tasks of every core on every other core:

$$\begin{aligned} L_{sys}^{oh} = \quad & L_{c1\to c2}^{oh} + L_{c1\to c3}^{oh} \ldots + L_{c1\to cn}^{oh} \\ & + L_{c2\to c1}^{oh} + L_{c2\to c3}^{oh} \ldots + L_{c2\to cn}^{oh} \\ & + L_{cn\to c1}^{oh} + L_{cn\to c2}^{oh} \ldots + L_{cn\to cn-1}^{oh} \\ = \quad & \sum_{i=1}^{N}\sum_{\substack{j=1 \\ j\neq i}}^{N} \mathbf{T}_i^T\mathbf{B}\mathbf{T}_j \end{aligned}$$

Finally we can estimate the system (CPU) performance as the effective load in terms of time shares by subtracting the overhead from the ideal

system load:

$$L_{sys} = \sum_{i=1}^{N} \mathbf{T}_i - \sum_{i=1}^{N} \sum_{\substack{j=1 \\ j \neq i}}^{N} \mathbf{T}_i^T \mathbf{B} \mathbf{T}_j \tag{3.5}$$

3.4 Performance Measure

The model we are proposing is best suited for persistent tasks like databases or webservers. Users are normally interested in completion or response times, but these metrics are hard to measure for persistent tasks. For this reason we resort to use statistics provided by hardware performance counters (*HPC*). HPC are special registers that collect different kinds of statistics with little to no overhead. They are primary used to ease program debugging [93] and profiling.

3.4.1 The Metric

The metric we use is the *number of instructions executed per clock cycle* I_c. It is important to note that number of instructions does not match the number of cycles. There are several reasons for this. The first two reasons are instruction-level parallelism and pipelining. A core can execute several independent instructions at the same time and pre-fetching enables pipelined execution if the code does not branch.

The third reason is data availability. Comparing to CPU clock rate the memory subsystem has huge delays. A CPU core running at 3Ghz has a 0.3ns clock cycle while a typical random-access delay for memory is 40-60ns. This means that in case of unavailability of data the core would stall and loose a lot of cycles waiting for data to be fetched from the memory.

As resources like memory bus, FPU, ALU, caches are shared between several cores to save silicon space, if a resource is busy it cannot be used

by another core. This leads to resource starvation that lowers the I_c each task can achieve.

3.4.2 Accuracy and Overhead

The most important feature of HPC is that the accuracy does not depend on OS and measuring tools [86]. Unfortunately, as any measure, HPCs are not exact and contain a degree of uncertainty. Furthermore, the actual accuracy can depend on the number of enabled counters [94].

There are many sources of error, most noticeable contributors are hardware interrupts. With each interrupt the CPU core has to restart the execution of the current process from the point it was stopped to serve the interrupt. HPCs do not know exactly what instructions were aborted and this can generate a small overestimation of performances (overcount), as the same instruction is (partially) counted two times: before and after the interrupt.

However, the level of uncertainty is low, usually below 1%, and in many cases overcounts can be compensated [86]; this precision is enough for the most of practical uses. Ad hoc measures we performed before the performance measure campaign confirm that counting both system-wide and per-task statistics for two processes at the rate of 10 samples per second introduces less than 1% of overhead: a level we can consider negligible for the purposes of our research. Furthermore, at least on the Core 2 Duo machine, where we run this additional test, the performance measure for a task is stable regardless of the background core load: the deviation of I_c is less than 0.3% under any background activity.

3.5 Model Validation

The validation of the model consists of two steps. The first step is to train the model, i.e., to obtain the coefficients used in matrix **B** in (3.5). This is done through the multiple execution of each test program. The first time a task A is executed with no background activity. This gives us the *ideal performance* of task A. Then task A is launched together with another task (let's say task B) on another core. The difference in performance gives us the level of overhead that task B creates for task A. The process is to be repeated with all tasks in all combinations. For validation purposes we simply take this brute-force approach, but in Sect. 3.7 we shortly discuss how to reduce the cost of this cumbersome operation.

As long as only two tasks interact one another, we can expect that the model yields exact predictions; however, in real clouds there will be multiple tasks interacting together, thus the goal and second step of the validation is comparing the accuracy of our model with the naive approach that scales performance linearly without accounting for the overheads. To do this we launch randomly generated sets of tasks of specified sizes and compare the naive prediction, our model, and the real performance measured by I_c.

To simplify the evaluation process we perform the measurement campaign running the tasks directly on Ubuntu, but the model can also be used with any kernel-visible task supporting performance counters. This includes widely deployed virtualization technologies like Xen, KVM and linux containers.

3.5.1 Hardware Configurations

The right choice of equipment is very important for model validation: the machines used in experiments have different characteristics to ensure gen-

erality. We used the following hardware configurations from different vendors (Intel and AMD) of different generations with different number of CPU cores:

1. Intel E7600 (2 cores), 8GB RAM
2. Intel W3670 (6 cores), 24GB RAM
3. AMD FX-8120 (8 cores), 16GB RAM

We used *Ubuntu Server 12.04 AMD64* OS with all services and background activity stopped. All tests were performed under the full load of the CPU cores. Test programs were launched at least once before measurements to warm-up the caches. All dynamic performance features like TurboBoost were stopped because they violate the model's assumption of uniform CPU configuration when all cores have the same parameters and equal access to hardware resources; it is possible to extend the model with the coefficients representing the frequency scaling. For the same reason the hyper-threading technology was disabled during the tests, though it is possible to address this issue with hierarchical model like it was done in [92].

The per-core assignment of the tasks were ensured by explicitly set CPU affinity through the *sched_setaffinity* system call.

3.5.2 Measurement Methodology

For each measurement the involved tasks were run 30 seconds for warm-up and then 180 seconds for measurement. In the results reported here we always launched the same number of tasks on each core; however, additional tests with different number of tasks on different cores haven't shown any difference in performance. We also checked the configuration when only six out of eight cores on AMD machine were loaded, the model still show the same good performance as for the tests with all cores loaded.

The performance is measured by launching the tasks with the *perf* utility and averaging results across 10 runs. The system-wide statistics, as recommended by documentation, is measured with the

```
perf stat -a -e <counters> sleep Ns
```

command. On Intel CPUs all available performance events were engaged in measurements for further analysis. On AMD machine, due to the smaller number of available hardware counters [13], only cycles, instructions and cpu-clock events were counted. For machines with different possible cache prefetch settings the measurements were repeated for each setting separately (including the computation of model parameters).

We compared algorithms by the relative root mean square error R_{rmse} on the performance I_c defined as

$$R_{\text{rmse}} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\frac{I_c^{\text{pred}}(i) - I_c^{\text{meas}}(i)}{I_c^{\text{meas}}(i)}\right)^2} \tag{3.6}$$

where n is the number of runs. Lower R_{rmse} means better precision.

3.6 Results and Analysis

First of all, we compare the global results (in terms of R_{rmse}) of our model as compared to the naive linear predictor for the three hardware configurations. Next, we will delve deeper into the experiments and analyze in detail the interaction of different programs.

Table 3.1 reports R_{rmse} for Intel E7600, the two-core machine. Besides power saving features, the machine does not have any extra performance settings that would affect results. Table 3.2 reports results for Intel W3670, the six-core machine; in this case we considered different BIOS settings. The left half of the table refers to BIOS settings that enable the Hardware PreFetcher (HWP) and Adjacent Cache-Line Prefetcher (ACLP); these

Table 3.1: R_{rmse} accuracy of our model compared to the linear prediction in different test scenarios for the two-core E7600 CPU.

Tasks per core	**1**	**2**	**3**	**Overall**
num. of samples	36	210	210	456
Our model	0.016	0.024	0.024	0.021
Lin. pred.	0.233	0.178	0.170	0.194

Table 3.2: R_{rmse} accuracy of our model compared to the linear prediction in different test scenarios for six-core Intel W3670 CPU with enabled and disabled hardware prefetcher (HWP) and Adjacent Cache Line Prefetch (ACLP).

Cache control	HWP and ACLP enabled				HWP and ACLP disabled		
Tasks per core	**1**	**2**	**3**	**Overall**	**2**	**3**	**Overall**
num. of samples	55	50	50	155	175	175	350
Our Model	0.058	0.060	0.050	0.056	0.016	0.013	0.015
Lin. pred.	0.296	0.298	0.290	0.294	0.117	0.117	0.117

technologies pre-populate the CPU caches with data[1]. The right half of the table refers to BIOS settings with HWP and ACLP disabled. Table 3.3 refers to the measurements on the AMD CPU. Like the Intel W3670, the BIOS of the FX-8120 motherboard allows choosing between several values for cache-control parameter: auto, regular and extreme. This setting affects how much data will be additionally prefetched during the memory reads. Results in Table 3.3 refer to all three cases.

As we can see from the tables, the accuracy of our model is more than one order of magnitude better than the simple linear prediction on Intel hardware and roughly three times better on the AMD 8-cores machine, and this independently of the number of concurrent tasks per core, even if the β coefficients have been measured only in the case of a single-task per core.

[1]The difference between them is in the logic: ACLP just blindly fetches two cache lines (128 bytes) instead of one (64 bytes) whenever CPU reads something from memory, but HWP tries to figure out memory access patterns. Both options can be activated and disabled independently.

Table 3.3: R_{rmse} accuracy of our model compared to the linear prediction in different test scenarios for eight-core FX-8120 CPU with different cache control settings.

Cache control	Auto			
Tasks per core	**1**	**2**	**3**	**Overall**
Num. samples	165	165	165	495
Our model	0.108	0.112	0.121	0.114
Lin. pred.	0.295	0.302	0.311	0.303
Cache control	Regular			
Tasks per core	**1**	**2**	**3**	**Overall**
Num. samples	85	85	85	255
Our model	0.139	0.131	0.134	0.135
Lin. pred.	0.292	0.293	0.312	0.299
Cache control	Extreme			
Tasks per core	**1**	**2**	**3**	**Overall**
Num. samples	80	80	80	240
Our model	0.121	0.129	0.132	0.127
Lin. pred.	0.277	0.304	0.313	0.298

3.6.1 Digging Inside the Model

The matrix form we use in the model has the further advantage of easy task interaction analysis. Thus, it provides a handy tool for a classification of tasks empowering performance prediction without the need of painstakingly measuring all possible combination of tasks. The parameters of matrix $\mathbf{B}$ in (3.5) are directly derived from the performance analysis as the one reported in Table 3.4, obtained on the two-core E7600 CPU machine. This matrix shows how tasks affect each other in pair. The numbers are the performance (I_c) reduction in percentage of the two tasks interacting on the two cores.

Using the table we can easily recognize the "conflicting" classes. The tasks hungry for memory bandwidth are the most conflicting, while integer computations and optimized video encoding are highly parallelizable.

Table 3.4: Performance penalty (percentage) for simultaneous task execution on Intel E7600 CPU.

		Task					
		SDAG	SDAGP	MATRIX	MATH	FFMPEG	BLOSC
Background activity	SDAG	0.3	8.6	12.1	0.5	1.6	3.2
	SDAGP	7.5	21.0	37.1	0.5	4.9	12.5
	MATRIX	11.4	25.9	41.8	0.3	7.5	13.0
	MATH	−1.3	−0.6	−2.7	−0.0	−0.1	−0.1
	FFMPEG	1.2	7.1	11.0	0.4	1.4	3.8
	BLOSC	7.6	26.4	43.3	0.4	6.6	23.6

Negative values are not measurement errors, in some cases the parallel execution of multiple processes can even give a small speedup over the performance of the task alone. Even different processes often share some memory pages (shared libraries, disk cache, 'mmaped' areas and other resources). This increases the chances for cache hits. Joint execution of processes also changes the way CPU accesses the memory. Going a bit ahead, due to the different hardware optimizations like cache prefetchers, the memory access pattern can greatly affect the overall performance (more on this in 3.6.4).

Tab. 3.4 exhibits some symmetry, but this is not granted. In Chapter 4 we will see tasks creating more interference are usually more prone to interference. Hence, a pattern can be noted that task A interferes with task B in a similar way as B interferes with A. Yet there is not direct relationship between these two metrics, this best can be seen on BLOSC.

The memory factor dominates in the interference picture; memory subsystem remains the biggest single bottleneck contributor.

3.6.2 The Two-Core Machine

The model is very precise on two cores. This is due to the nature of the model: it is a second order model and, besides, it was trained for the two-cores. The main contributor to the average error is SDAGP test. However, this does not mean that SDAGP makes everything unpredictable, but the level of uncertainty is higher than with other programs. In all cases with more than 10% error the smart model still shows much better results than the naive approach. Both approaches show positive error, i.e., the predicted performance is higher than actual. Higher cache miss ratio leads to less precision in general.

3.6.3 Intel W3670: The Six-Core Case

Compared with the two-core case, the performance of our model in a six-core case is a bit worse as a six-way interaction is more complex. Unlike the two-core machine, the six-core server is equipped with two mechanisms that aimed at pre-populating the cache with data. The first is an automatic cache line prefetch technology. If it is enabled, each time the CPU accesses the memory the next cache line is fetched as well. The second mechanism is a hardware memory prefetcher. It tries to detect sequential memory readings and read the data in advance up to the memory page boundary. Both technologies are aimed at reducing the data access time and introduce some overhead. It is not granted that the overall performance will be higher: the impact depends on the running tasks and how they are distributed as discussed in 3.6.4.

3.6.4 Effects of Prefetching on Intel W3670

One interesting thing is how the BIOS prefetch settings affect the performance. In general it does not hurt, but the maximum effect can be achieved

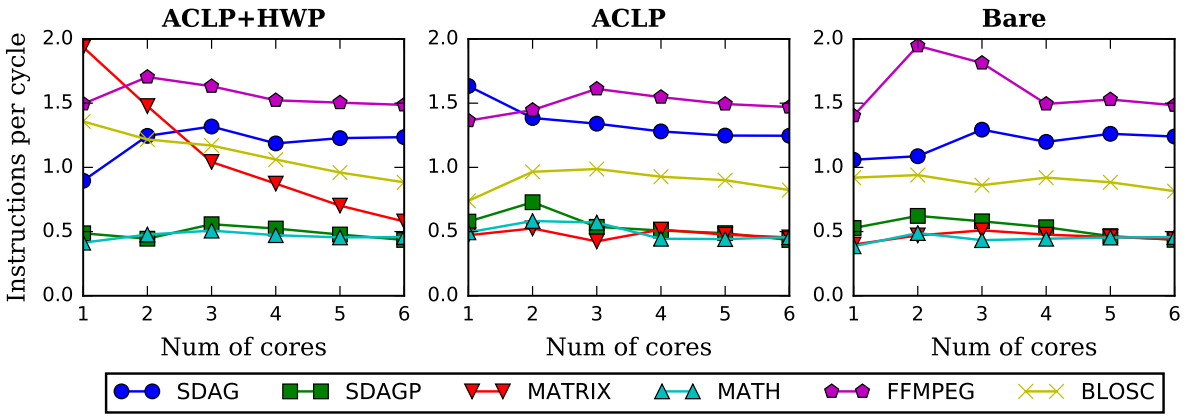


Figure 3.2: The effects of HWP and ACLP on the per-core performance.

only under partial loads. This can be seen on Fig. 3.2. The prefetching can improve performance only if few cores are loaded, while the performance under full load is not affected. Yet, up to five out of six cores prefetching makes positive or mostly-neutral effect on performance. Therefore, it can be safely enabled. It is also seen that ACLP alone is not really useful comparing to HWP+ACLP configuration.

3.6.5 AMD FX-8120: The Eight-Core Case

The eight-core AMD FX-8120 CPU seems to behave differently from the Intel CPUs, but we do not have a clear explanation of whether this is due to the larger number of cores, or to a different hardware architecture. In general, we found that performance of this CPU is far less predictable comparing to six-core Intel W3670.

On this platform our model still performs better than a simple linear predictor, although the advantage is no more a full order of magnitude, but roughly a factor of three. The prediction error is reduced to a few percentage points from more than 15%, thus it is probably still good enough for CRM applications; however, more research and experiments are required to fully understand if this model is accurate enough in general or

not. Moreover, we observed that aggressive prefetch settings make the overhead slightly less predictable. Surprisingly, BIOS cache settings have negligible effect on the performance if there is only one active task, though we expected this to be an ideal showcase for memory prefetchers because bandwidth is not shared between multiple cores and prefetching overhead is unlikely to cause problems.

3.6.6 Improving the Precision

Though the proposed model shows very good results it can be further improved. Modern CPUs often consist of multiple coupled cores. Neighbor cores can share L1 and L2 caches, FPU units, instruction decoders and other circuits. In this case the interaction between two tasks put on the neighbor cores will be different than between the same tasks assigned to the "distant" cores. Thus, taking this into account the accuracy can be improved, obviously at the cost of more complex tuning and of less independence of the model from the specific hardware platform.

Another way to improve the accuracy is in increasing the order of the model. The proposed model considers only first-order interactions, i.e., the impairment $\beta_{A\to B}$ caused by one task running on a core on another task running on another core. It is evident that this can be a good model for 2 tasks on two cores, but as the number of tasks and cores increases higher order models would be more accurate. However, a model must be also simple and usable, so that increasing its order is often not the right direction. In our case, in ultimate analysis, a task always runs 'alone' on the core, due to time-sharing. This means that a proper combination of first-order interactions should be enough to grab the real performance reduction, but this refinement is left for future work.

Another possibility is to resort to stochastic models (e.g., Markovian) which inherently take into account higher orders interaction through the

different moments of the distribution. However, the lack of stochastic models of the software itself (i.e., tasks) makes this direction, so natural in other contexts like networking or hardware modeling, not-so-trivial for the problem at stake.

3.7 Obtaining Model Parameters

Compared to a trivial linear predictor, the drawback of our model is clearly the requirement of obtaining the $\beta-$ parameters. However, there are several ways in which these parameters can be estimated instead of directly measured. Especially the possibility of on-line direct estimation with machine learning techniques is very promising for large-scale CRM management. In the sequel we discuss some techniques to obtain the βs.

3.7.1 Direct Measurement

The level of interaction between tasks can be simply measured. We used this approach to validate the model. It is the most deterministic way and provides the best result, and it is definitely the way to go for a model validation. Unfortunately, it requires a lot of possibly long measurements and in general it does not scale as the number and variety of possible tasks is practically unlimited. Still it can be useful if the expected outcome outweighs the inconveniences, and most of all it can be used to tune other techniques making selected and carefully crafted measurement campaigns.

3.7.2 Task Classification

A direct extension of the direct measurement of each individual β is the reduction of the tasks space by aggregating them into categories characterized by similar interaction levels. In other words, this means answering the

question "Are all tasks so different one another to be accounted separately one by one?" We can introduce broad classes of tasks sharing common features (e.g., numbercrunching, memory intensive and other classes). The level of similarity can be "measured", for example, by looking at the library and resources the tasks use, comparing the hardware performance statistics, tracing the system calls and other methods. Tasks classification in a proper subset of classes will reduce the number of βs to be measured to a few scores, which is well feasible.

3.7.3 Low-level Resource Utilization

A completely different way can be predicting the βs themselves with a theoretical/heuristic analysis. A lot of information can be extracted from performance counters. For instance, modern CPUs expose the level of utilization of FPU, ALU and rich memory-related statistics. Using this information we can understand the process activity and make some guesswork about the level of interference with other tasks.

This road seems more useful for CPU design than for CRM optimization.

3.7.4 On-line Tuning

Finally, a dynamic method can be devised, which is possibly the most promising. Starting from a coarsely populated $\mathbf{B}$ matrix, e.g., with βs coming from tasks classification, a CRM can use machine-learning techniques to gradually refine the coefficients for each and every task it happens to handle repeatedly. For newer programs a library-match approach can be used as initial assumption that is to be corrected later, but even starting from a null interaction coefficient may work.

3.8 Conclusion

In this Chapter we presented an empirical CPU performance model. Devised with real hardware and simplicity in mind, it suites cloud management much better than the linear prediction, a naive approach used in most CRMs today. Thus, the usage of this model would greatly increase the efficiency of clouds. Besides higher precision, this model has two more advantages. It was made trying to be backward compatible with existing solutions, so that it can be used as a drop-in replacement for other load models. The second advantage is its tolerance to the parameters needed to run. In case some parameters are missing it still will be able to run at the cost of reduced accuracy: in the worst case it will behave like a naive linear predictor having all parameters set to zero.

But we can do better than just falling back to linear prediction. It turned out the interference characteristics of a task or a VM can be "guessed" by observing its performance counters. This is discussed in the following Chapter.

Chapter 4

Ranking VMs by their interference

4.1 Introduction

The level of interference between VMs depends on many factors. OS, libraries, programs and their versions, compilers used to build the system, hardware, BIOS settings are some of the factors that affect the performance results. It is difficult to predict how a particular VM will co-exist with other VMs in a given scenario. The most precise way to understand the loss in performance caused by a VM is to measure it. A good precision of this method is on the expense of its complexity – each VM should be executed with each other VM at least once during the measurement phase. It is possible for systems with limited number of executed VMs, but becomes impractical when the number of VMs is large.

Another factor contributing to complexity is the VM diversity: there is a virtually infinite number of different virtual machines. But are they really that different in terms of interference? What if we find a simple way to rank arbitrary VMs according to the interference they cause? Such ranking may be imperfect, but it will certainly be useful for cloud management.

In this Chapter we present a novel methodology to classify and rank VMs based on the analysis of *Hardware Performance Counters* (HPCs). HPCs accumulate resource access statistics such as the number of time a

VM accessed CPU caches or the success rate of the branch predictor. The main contributions are:

- Development of a methodology for profiling and ranking of VMs to estimate the level of their mutual interference;
- Evaluation of the methodology on x86-based and ARM-based hardware platforms;
- Analysis of the obtained profiling data to understand which shared resources are the main contributors to the VM interference.

4.2 Methodology

Fig. 4.1 presents a high-level overview of the ranking process. It consists of two major phases: learning phase and working (ranking) phase. During the learning phase the system figures out which hardware counters are the most important for application profiling and how they are related to system performance. The obtained knowledge is stored in a database. The ranking of a previously unseen application is performed by comparing the profile of a new task with classes from the database.

If it is known which VMs co-exist well and which do not, we can perform correct placement and schedule their execution properly. But each VM behaves differently in the presence of other VMs competing for computing, memory, or network resources: it can run unaffected, degrade or even increase performance. *How to assess and predict VM interference?* A straightforward way would be to launch all the VMs together in pairs and measure their mutual interference. However, this would take too much time. A better way would be to profile the VMs individually and then, based on their profiles, reason how they will interact with each other.

Our goal is to rank VMs based on two parameters of interest – *sensitivity* and *interference*. These parameters could then be used to, e.g., guide

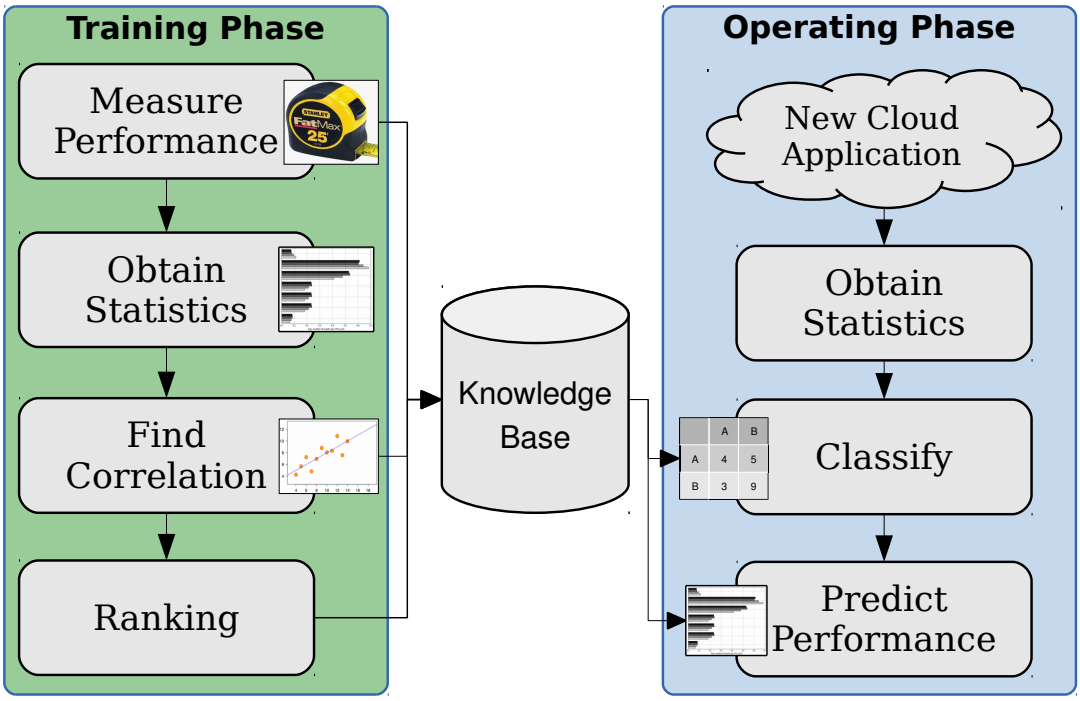


Figure 4.1: Ranking process at a glance.

resource allocation and scheduling. *Sensitivity* is a measure of how the performance of a given VM is affected by the activity of other VMs. On the contrary, *interference* describes how the behavior of a given VM affects operation of neighboring VMs. As both the sensitivity and interference cannot be measured directly, we derive their values from the analysis of HPCs.

4.2.1 Hardware Performance Counters

HPCs are a mechanism for application profiling. HPCs are built-in CPU circuits designed to collect runtime low-level execution statistics. HPCs consist of two parts: event detectors and 64-bit registers (counters). Each time an event occurs the register associated with this kind of event is incremented.

HPC statistics include the frequency of access to instruction decoders,

caches and Floating Point Unit (FPU).

4.2.2 Virtual Machines Profiling

The mapping between interference/sensitivity and the values of HPCs can be measured through correlation analysis. For this, we first calculate interference and sensitivity for a small set of VMs. This can be done by launching all pairs of the VMs and measuring their execution performances. Then we compute linear correlation by calculating *Pearson's product-moment correlation coefficient* ("Pearson's r" [59]) between the interference/sensitivity and each of the hardware counters. The Pearson's correlation was chosen because it does not require data to have specific distribution (e.g., normal) or any kind of dependency. The HPCs with strong correlation are selected to predict interference/sensitivity values of an arbitrary VM. This prediction can be done, for example, with regression analysis.

4.3 Experimental Study

Our experiments are executed on a small scale heterogeneous testbed accounting for different architectures, using collection of different benchmark applications.

4.3.1 Testbed

We use the following equipment:

ARM Exynos – "Odroid-U2" board based on Samsung Exynos-4412 system-on-chip with ARM Cortex-A9 four-core CPU clocked at 1.7GHz. ARM Exynos has 2 GB of RAM and 8 GB eMMC storage.

AMD FX – board based on an eight-core AMD FX-8120 CPU. The CPU consists of four two-core blocks, each equipped with 2 MB dedicated

L2 cache. In addition, all two-core blocks share the same 8 MB L3 cache. In order to obtain stable and repeatable results the dynamic overclocking is disabled in BIOS. AMD FX is supplied with 16 GB DDR3-1600 RAM. A Crucial^{TM}M4 Solid State Drive (SSD) with 64 GB is used as a storage.

All measurements are done by "perf stat" command using all relevant counters reported by "perf list" command.

4.3.2 Benchmarks

Benchmarks are selected to provide a comprehensive comparison of cloud workloads. The emphasis is given to the real-world programs (*FFMPEG*, *NGINX*, *PGBENCH*, *SDAG*, *SDAGP*, *WORDPRESS*), although a few synthetic benchmarks (*MATRIX*, *BLOSC* and *INTEGER*) are present as well. The complete description of these programs is given in Sec. 3.1.1, here we briefly describe them for the sake of clarity.

1. ***BLOSC:*** a high performance compression library that optimizes data transfers between CPU and memory;
2. ***FFMPEG:*** transcoding a H264 FullHD video into 720p format;
3. ***INTEGER:*** integer computations with the four operations;
4. ***MATRIX:*** matrix multiplication benchmark based on gsl/blas library with the size of matrices of 2048*2048; data type is 64 bit float;
5. ***NGINX:*** web server benchmark focused on static files [4];
6. ***PGBENCH:*** PostgreSQL database stress test [5];
7. ***SDAG:*** benchmark with machine learning [70];
8. ***SDAGP:*** same as SDAG, but with a different memory layout;
9. ***WORDPRESS:*** default installation of a popular blogging and publishing platform.

4.3.3 Software Architecture

Fig. 4.2 presents the software architecture of our experiments. The core component is the *Experiment Controller*. It sets-up VMs, checks OS settings, and launches the benchmarks.

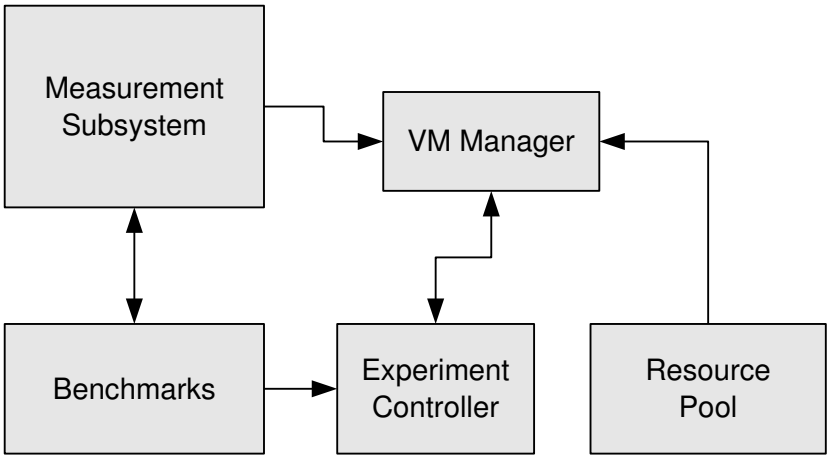


Figure 4.2: Software architecture of the experiments.

We implemented a specific *VM Manager*, a subsystem which provides virtualization method appropriate for the platform – QEMU for AMD FX and Linux Containers (LXC [3]) for ARM Exynos, as these platforms do not allow for standard VM management. The *Resource Pool* provides an abstraction layer for VMs to hardware resources. In our experiments each virtual machine was allocated 1 Gb of RAM and one core of the CPU. The *measurement subsystem* serves two different purposes. The first one is to collect the HPC statistics. The second purpose is to ensure that there is no activity in the system left unaccounted.

4.4 Performance Results and Analysis

Tables 4.1 and 4.2 show how the VMs affect the performance of each other during their concurrent execution. Columns specify the names of the

Table 4.1: Performance degradation for concurrent execution of VMs running the benchmarks on ARM Exynos reported in percents.

		Benchmark (foreground)								
		BLOSC	FFMPEG	INTEGER	MATRIX	NGINX	PGBENCH	SDAG	SDAGP	WORDPRESS
Benchmark (background)	BLOSC	0.9	4.7	0.3	13.3	8.7	11.4	10.7	9.8	6.9
	FFMPEG	1.1	2.3	0.2	9.2	2.8	8.3	4.3	7.4	3.1
	INTEGER	-0.7	0.0	-0.1	-0.1	-1.8	-0.8	0.1	0.7	-0.6
	MATRIX	9.6	8.4	0.3	15.4	21.9	24.7	22.8	41.3	14.0
	NGINX	3.4	7.5	0.5	18.0	8.4	15.2	14.4	16.2	10.5
	PGBENCH	5.2	6.5	0.4	16.8	19.4	8.3	12.4	12.6	10.9
	SDAG	0.1	2.6	0.2	8.5	4.9	10.8	5.3	9.0	3.7
	SDAGP	4.2	10.0	0.3	15.4	20.1	27.8	24.0	50.3	14.8
	WORDPRESS	3.1	4.8	0.2	9.8	9.3	15.5	11.1	12.9	6.8

benchmarks currently being measured (*foreground* VMs), while rows are associated with the benchmarks executed at the same time on the neighboring core (*background*). The numeric values reported show the performance degradation of the foreground benchmarks with respect to their standalone execution. The dark grey cells correspond to performance degradation of more than 15%, while the light grey cells show the degradation between 10% and 15%. The values reported in square boxes signal the performance increase. The latter can be achieved when the concurrent benchmark execution makes the use of the shared hardware resources (e.g., caches) more efficient than during standalone runs. For AMD FX we report interference results for both sibling cores that share the local cache and distant cores that share less resources.

These synoptic tables give several interesting insights. The first one is clear: running VMs on dedicated CPU cores does not ensure performance

Table 4.2: Performance degradation for concurrent execution of VMs running the benchmarks on AMD FX reported in percents.

(a) Sibling cores

		Benchmark (foreground)								
		BLOSC	FFMPEG	INTEGER	MATRIX	NGINX	PGBENCH	SDAG	SDAGP	WORDPRESS
Benchmark (background)	BLOSC	5.2	0.8	0.4	8.4	3.3	7.9	1.5	4.3	2.1
	FFMPEG	3.7	-0.5	0.2	3.8	1.6	7.4	1.0	3.6	0.4
	INTEGER	1.4	0.9	0.1	-0.7	0.5	5.8	-1.1	3.2	-0.8
	MATRIX	5.2	3.3	0.2	19.1	11.1	15.0	4.2	14.3	6.6
	NGINX	4.3	1.1	0.4	12.9	5.8	11.9	3.0	12.3	3.5
	PGBENCH	2.6	1.6	0.1	5.7	2.6	9.3	-0.1	2.6	1.2
	SDAG	1.1	-0.2	0.1	1.3	0.3	4.6	-1.2	-0.7	-0.5
	SDAGP	2.8	-1.0	0.3	5.6	2.7	6.0	1.6	2.7	1.2
	WORDPRESS	2.2	-0.1	-0.3	3.9	1.0	8.2	1.1	3.4	1.1

(b) Distant Cores

		Benchmark (foreground)								
		BLOSC	FFMPEG	INTEGER	MATRIX	NGINX	PGBENCH	SDAG	SDAGP	WORDPRESS
Benchmark (background)	BLOSC	3.5	1.4	0.2	7.7	3.3	12.5	2.1	4.7	1.5
	FFMPEG	2.3	0.4	0.4	4.6	1.0	7.7	0.9	3.5	0.2
	INTEGER	2.3	0.7	0.0	0.0	0.3	10.0	-0.6	3.5	-0.5
	MATRIX	5.2	3.8	1.1	19.0	11.7	15.1	4.9	17.6	6.1
	NGINX	4.5	1.7	0.1	7.5	4.3	12.1	2.5	12.7	3.1
	PGBENCH	2.9	0.6	0.4	4.8	2.9	9.2	2.3	10.4	1.2
	SDAG	1.4	1.1	0.0	2.1	0.5	10.5	-0.4	9.2	0.0
	SDAGP	2.5	-0.5	0.3	4.2	1.9	8.9	1.9	3.8	1.1
	WORDPRESS	3.3	0.2	0.5	4.8	1.1	12.1	0.4	11.5	0.4

Table 4.3: Interference and sensitivity of benchmarks on ARM Exynos.

Interference		Sensitivity	
SDAGP	18.6%	SDAGP	17.8%
MATRIX	17.6%	PGBENCH	13.5%
NGINX	10.4%	MATRIX	11.8%
PGBENCH	10.3%	SDAG	11.7%
WORDPRESS	8.2%	NGINX	10.4%
BLOSC	7.4%	WORDPRESS	7.8%
SDAG	5.0%	FFMPEG	5.2%
FFMPEG	4.3%	BLOSC	3.0%
INTEGER	-0.4%	INTEGER	0.3%

isolation. The degradation of performance is in some cases definitely high and can easily affect even the perceived QoS. Another interesting observation is that in the AMD FX architecture the interference is largely independent from the cores' distance. Finally, the performance improvement, which is at first sight counter-intuitive. First of all, the gain is usually small and in some cases it can well be just a measure noise, even if the measures are the average of many runs. Second, e.g., for caches, the algorithm that manages them is based on a very complex heuristic. Thus, the scenarios and setups in which the heuristic works better than others are not so surprising, specially taking into account that sharing resources is far more common than running in isolation, thus, the heuristic has been studied and tuned for these cases.

Tables 4.3 and 4.4 present the values of VM interference (how much the background affects the foreground) and sensitivity (how much a foreground is sensitive to have some other concurrent VM) calculated based on the performance degradation values reported in Tables 4.1 and 4.2. The sensitivity is obtained as an average from the values each column, while the interference is an average on the rows. According to [49], the performance

Table 4.4: Interference and sensitivity of benchmarks on AMD FX.

Interference		Sensitivity	
MATRIX	8.8%	PGBENCH	8.4%
NGINX	6.1%	MATRIX	6.7%
BLOSC	3.8%	SDAGP	5.1%
PGBENCH	2.8%	NGINX	3.2%
SDAGP	2.4%	BLOSC	3.1%
FFMPEG	2.4%	WORDPRESS	1.7%
WORDPRESS	2.3%	SDAG	1.1%
INTEGER	1.0%	FFMPEG	0.6%
SDAG	0.5%	INTEGER	0.2%

(a) Sibling Cores

Interference		Sensitivity	
MATRIX	9.4%	PGBENCH	10.9%
NGINX	5.4%	SDAGP	8.6%
BLOSC	4.1%	MATRIX	6.1%
PGBENCH	3.9%	BLOSC	3.1%
WORDPRESS	3.8%	NGINX	3.0%
SDAG	2.7%	SDAG	1.6%
SDAGP	2.7%	WORDPRESS	1.5%
FFMPEG	2.3%	FFMPEG	1.0%
INTEGER	1.7%	INTEGER	0.3%

(b) Distant Cores

degradation increases following a power law with the number of CPU cores: 5% interference between each two cores leads to ∼18.5% and ∼33.6% of overhead for four and eight cores respectively, but we cannot draw strict conclusions on this yet.

4.4.1 Analysis of different HPCs

Events can be different in nature, but all of them can be assigned a performance cost. For example, each LLC cache miss costs around 30-60 cycles of additional CPU time [71]. However, to understand the impact of the event on a system performance it is necessary to analyze the rate of the event occurrence in addition to the cost of the event. Low-frequency events do not contribute much to the VM interference. Therefore, we exclude low-frequency events from the analysis, even if they are costly. We operate with normalized frequency of events to avoid bias from the CPU clock rates.

Tables 4.3 and 4.4 give a high-level perception of the sensitivity and interference "properties" of the benchmarks. A quick investigation, to no surprise, indicates that all the top interfering benchmarks heavily use mem-

ory subsystem. SDAGP operates over a large set of scattered data. This requires a lot of memory access requests that cannot be served efficiently. MATRIX is optimized for efficient memory access, but uses all available caches and constantly displaces other cached data. BLOSC was designed to compress scientific data on-the-fly at extreme rates. It is capable of fully occupy the memory bus, which heavily impacts all other applications requesting bus access.

The high demand for memory resources that makes a benchmark an interferer, also makes it sensitive to the same resources. Therefore, there is a clear correlation between interference and sensitivity figures.

So far for the pure empirical observations. Now we proceed to identify what HPCs are the most representative of the interference/sensitivity properties. We compute the correlation between each performance counter and the values of interference and sensitivity and focus further investigation on counters with high correlation.

Fig. 4.3 presents HPC measurements for different levels of task interference: no interference (NGINX alone), low interference (NGINX + INTEGER), medium interference (NGINX + WORDPRESS), and high interference (NGINX + MATRIX). Fig. 4.4 shows the measured HPCs counters for all the benchmarks on the Exynos, corresponding to the interference values of Table 4.3. As expected, for low interference there are no significant changes in HPCs values. This means benchmarks execute as if they were alone. However, when interference becomes significant the HPCs values reflect task competition for the resources.

Surprisingly, there is no increase in any memory-related counters, which indicates that the main memory is not a primary bottleneck: the bottleneck arise inside the CPU before the main memory is accessed. The CPU cannot dedicate enough internal resources for all active cores. The cores compete for the resources, and this race creates a lot of pipeline stalls. This is

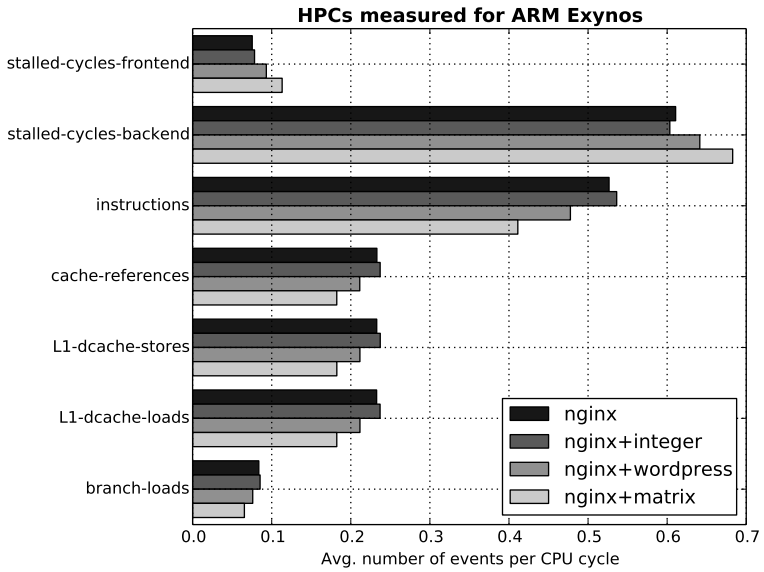


Figure 4.3: Four cases of interference for ARM Exynos: no interference (only *NGINX* is running), negative interference (*NGINX* runs with *INTEGER*), medium interference (*NGINX* with *WORDPRESS*) and strong (*NGINX* with *MATRIX*).

reflected by "stalled-cycles-backend" parameter.

Referring again to Table 4.2 we now interpret results based on the HPC analysis. For sibling cores (Table 4.2a), there are many cases of performance improvement (represented with negative values of interference). This is especially evident for FFMPEG benchmark. The reason for performance improvements becomes evident from the analysis of two HPC counters: TLB and L1 caches. These parameters indicate that some data (probably kernel code) is shared between VMs. Shared data may speedup the simultaneous execution because if one core accesses it, there is a chance that another core already fetched it and stored in shared cache. Another possible reason is that the overhead for keeping cache lines coherent is lower for the processes running on sibling cores [96]. The average per-task interference is around 3%.

For distant cores (Table 4.2b) the average per-task interference is equal to 4%. It is higher than for the sibling cores which is due to the fact

Table 4.5: Correlation between interference, sensitivity and HPC. P-value is the probability that results are statistically insignificant (null hypotesis), less is better.

(a) ARM Exynos

Interference		
Parameter	Correlation	P-value
stalled-cycles-backend	0.887	0.1%
cache-misses	0.712	3.1%
L1-dcache-stores	-0.808	0.8%
branch-loads	-0.810	0.8%
instructions	-0.851	0.4%
Sensitivity		
Parameter	*Correlation*	P-value
stalled-cycles-backend	0.804	0.9%
cache-references	-0.830	0.6%
L1-dcache-loads	-0.831	0.6%
branch-instructions	-0.832	0.5%
instructions	-0.851	0.4%

(b) AMD FX

	Parameter	Correlation	P-value
Sibbling cores	*Interference*		
	LLC-stores	0.915	0.1%
	L1-dcache-stores	0.732	2.5%
	Sensitivity		
	stalled-cycles-frontend	0.753	1.9%
	LLC-stores	0.694	3.8%
	cycles	-0.743	2.2%
Distant cores	*Interference*		
	LLC-stores	0.881	0.2%
	L1-dcache-stores	0.736	2.4%
	L1-dcache-prefetches	0.720	2.9%
	Sensitivity		
	L1-dcache-prefetch-misses	0.691	3.9%
	iTLB-load-misses	0.683	4.3%
	cycles	-0.775	1.4%

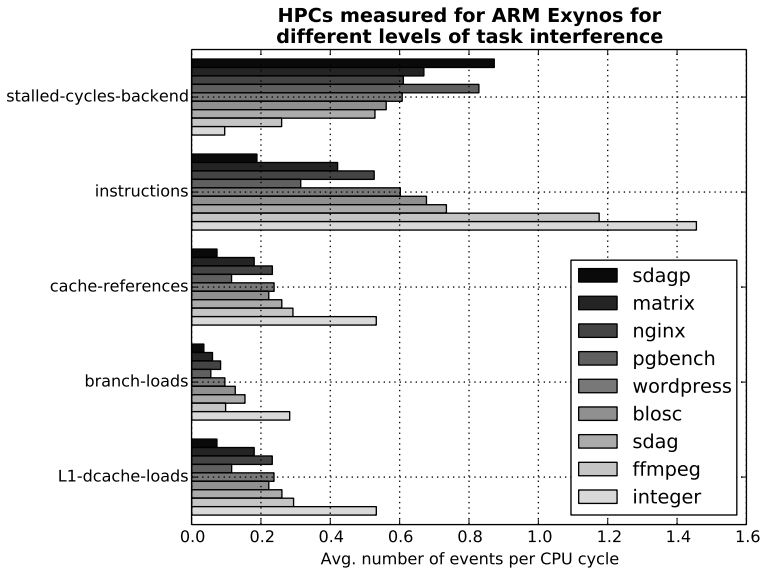


Figure 4.4: Execution profiles of benchmarks running on ARM Exynos. Benchmarks are arranged according to their *interference* factors.

that sibling cores can share data more efficiently. The picture is quite similar to the case with sibling cores. There is no single largest contributing counter to the interference. The average per-VM interference is 4%. This is slightly higher than the previous case and might be due to cache coherency protocol.

One question frequently arising is "how ARM compares to x86?". We could not help ourselves from comparing the performance of these two. Table 4.6 presents performance comparison in terms of Instructions per Cycle (IPC) and execution time. The ARM Exynos has substantially smaller IPCs. Interestingly, higher IPC does not necessarily lead to a higher performance per MHz. This is due to the differences in hardware architectures and optimization of compilers. We also spotted a few performance issues on ARM with FFMPEG and PGBENCH benchmarks. FFMPEG benchmark does a lot of data crunching. For this to work faster, it includes some hand-optimized assembly code. But this is done only for certain types of hardware. For ARM platform it still uses generic C functions that are more

Table 4.6: Performance comparison of AMD FX and ARM Exynos platforms[1].

	IPC			Performance		
Bench	ARM	AMD	Ratio	ARM	AMD	Ratio (Norm.)
BLOSC	0.68	1.10	1.6	49.53s	12.7s	3.9 (2.1)
FFMPEG	1.18	1.36	1.2	689s	48.2s	14.3 (7.8)
INTEGER	1.46	0.57	0.4	16.8s	15.3s	1.1 (0.6)
MATRIX	0.42	1.16	2.7	84s	16.8s	5.0 (2.8)
NGINX	0.52	0.77	1.5	525MB/s	631MB/s	1.2 (0.7)
PGBENCH	0.31	0.52	1.7	155 tr/s	1293 tr/s	8.3 (4.6)
SDAG	0.73	0.99	1.4	24.9s	8.3s	3.0 (1.7)
SDAGP	0.19	0.44	2.3	132s	30s	4.4 (2.4)
WORDPRESS	0.60	0.82	1.4	8.45r/s	7.19r/s	0.85 (0.5)

[1] Values in parentheses show performance ratios normalized to CPU frequencies.

than ten times slower [64]. As for PGBENCH, it measures the number of transactions per second. Every transaction is first written into transaction log and only then to the actual database. To ensure reliable updates, each write is accomplished with buffer flush which expensive on this platform because flash memory (its main storage) should be erased first (at the block granularity). Because of this and because the IO throughput of its storage slow, PGBENCH is tend to be limited more by IO, rather than by CPU.

The experimental results presented in this section reveal clear differences between the analyzed hardware platforms. In general, ARM cores have less optimization features than traditional x86 CPUs. They do "less job" per CPU cycle. For both platforms the most interfering tasks are the tasks that do heavy memory use (MATRIX, NGINX, SDAGP and BLOSC). This proves that the memory-related subsystems are the biggest bottleneck of general-purpose CPUs [34]. The bottleneck makes them sensitive as well because their performance almost entirely depends on data availability.

We can conclude that ARM Exynos performs well integer operations

and web-servicing duties (WORDPRESS and NGINX benchmarks). Heavy memory-intensive applications (SDAG,SDAGP, MATRIX and BLOSC benchmarks) perform better on AMD FX.

4.4.2 Lessons Learned

During the experiments we faced a number of technical problems. In the following we list the most relevant of them.

1. The HPC implementations vary across platforms. Not only the number of available events differs across platforms, but also their meaning. We checked OS Linux sources and developer manuals to ensure that our interpretation is correct.
2. We observed that VMs may shortly migrate to another CPU even if they are "pinned" to specific CPU cores. These cases are rare and do not change the overall picture.
3. Care should be taken when a large number of events is enabled. The number of available events exceeds the number of counting registers by a factor of 5 to 10. If too many events are enabled simultaneously, then the operating system has to do time multiplexing which leads to loss of precision.
4. Drivers and I/O can significantly affect the performance. We observed up to 40% deviation in instructions per second on heavy benchmarks if the system flushes disk caches. This does not affect the long-term average performance, but becomes critical for periodic measurements.

4.4.3 Conclusion

In this Chapter we presented a methodology for ranking VMs according to their level of mutual interference. The methodology was evaluated on

two platforms – x86 (AMD Bulldozer) and ARMv7 (Samsung Exynos-4412) – and showed there is a good correlation between between average interference of VMs and certain performance counters.

There is one inherited disadvantage of the proposed technique: for ranking to work every application needs to be profiled in the first place. And profiling must be done in an isolated environment so they are not affected by interference because it affects measurements in an unpredictable way. Another issue – depending on the input some programs may change their behavior. For this reason one-time profiling may not be enough. Ideally, all programs should be monitored for their behavior in run-time. The next Chapter describes an easy and elegant way of obtaining performance profiles in production environments on-the-fly and with small overhead.

Chapter 5

Freeze'nSense: Isolated Performance Sampling in a Shared Environment

5.1 Introduction

One good thing about clouds is that they "just work" – customers do not need to care about monitoring or backups – all these are done by the cloud provider. Now it is the provider's responsibility to provide satisfactory service, that is what they are paid for. The quality of the service and liability are defined in so-called SLA – a document that formally defines services and guaranties specific performance in ways that can be measured.

An SLA can be defined in terms of equivalent hardware or absolute computing and communication capabilities. In these cases meeting the SLA is relatively easy – just by throwing enough dedicated resources so the application would never suffer from underperformance – but very little statistical sharing can be achieved, thus, the infrastructure costs remain high. In many other cases the SLA is better defined in terms of application response latency, or equivalent performance: the service (platform, infrastructure, VM, ...) is provided guaranteeing that its performance is

as good as the one it would achieve in isolation on a given "bare iron," i.e., on a specific hardware platform.

The second definition of SLA allows a much higher level of statistical multiplexing, and, hence, reduced costs, but it introduces the problem of measuring that the SLA is indeed met. Verifying if SLA is met can be done in two steps: run the service in isolation and compare its performance with the performance of the service executed on the operational facility. In most of the cases this is not feasible for commercial services, but it is useful to understand if other techniques are reliable and dependable or not.

The high-level integration and parallelism of modern high-end computing nodes for data centers complicate the problem even more. Even if a node has N seem-to-be-independent CPU cores, it does not mean the node is capable of executing N tasks independently. Server resources, such as memory bus, CPU caches, network and storage, are shared between multiple cores, and NUMA architectures further complicate the scenario (see for instance [58] for an experimental analysis of the behavior of NUMA memory controller on a specific hardware architecture). As a result, the system performance does not scale linearly with the number of cores as one would hope. Fig. 5.1 reports the performance of SDAGP, a state of the art machine learning tool using a Directed Acyclic Graphs (DAG) data representation taken from [70]. The performance is measured on an 8-core AMD FX-8120 processor progressively increasing the number of cores used. When all 8 cores are used, the overall performance is only 65% of the one resulting from linear scaling (blue dotted line with crosses). This is significant degradation and definitely an SLA violation if the SLA stated that the performance is to be equivalent to the task running in isolation.

Running other software tools (from video transcoding to matrix multiplications) shows different scaling factors ranging from 72% down to a stunning 28% on the same hardware. These experiments show that scaling

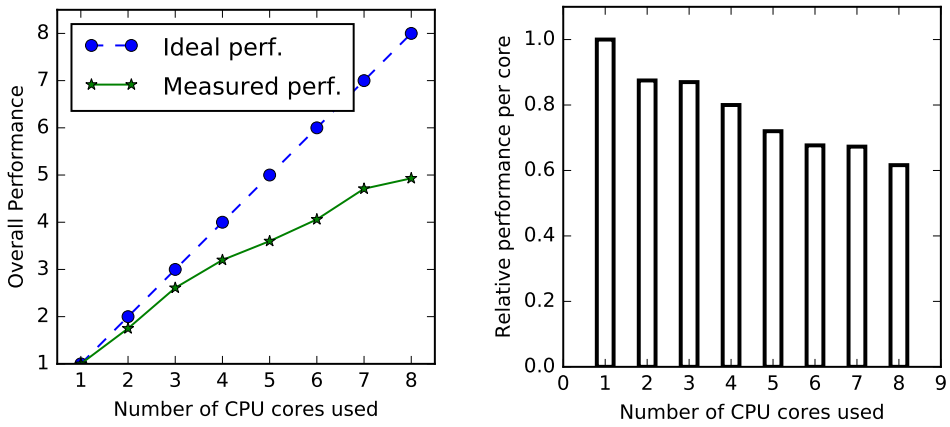


Figure 5.1: Performance scaling of SDAGP on AMDFX-8120 increasing the number of parallel instances; the gap between the two is due to shared hardware resources.

efficiency depends a lot on the workload, but also that the more cores a machine has the more interference on common subsystems of the CPU is possible, reducing the overall performance.

Chapter 3 explored the possibility of modeling interference between tasks executed in parallel on different CPU cores. The approach yielded an interesting and simple model; however, the difficulty of task classification (i.e., identification of tasks with similar characteristics), and the fact that the interference depends also on the CPU architecture and on the type of common subsystems shared by the cores makes this approach too limited for a widespread implementation.

Runtime performance monitoring is an effective way to measure the gap between "theoretical" and "actual" performances, knowing which helps to make decisions on the state of the system, assess whether it behaves healthy or whether it is overloaded and requires intervention (load balancing, powering on additional resources, VM relocation, etc.).

Measuring actual performance of applications is relatively straightforward. But to assess the level of interference from other tasks concurrently running in the system, the actual performance should be compared with

the performance of this application executed in isolation on the same hardware resources.

In this Chapter we present a novel methodology to estimate the theoretical performance at run time, by measuring IPCs [51, 10] with millisecond-level temporary performance isolation of the application under analysis. To estimate performance of a task in isolation, all the concurrent tasks are temporary frozen and put on hold for the duration of measurement interval, which is in the order of milliseconds. The short duration of the phase ensures low overhead on the task execution in the examined system. Knowing the ratio between actual performance and application performance in isolation allows estimating how sensitive a given task to the presence of other tasks in the system.

The remainder of this Chapter organized as follows. The notation and terminology we use is described in Sec. 5.2. Sec. 5.3 introduces the notions of performance isolation and performance monitoring. Sec. 5.4 describes the methodology we propose for run-time monitoring. Sec. 5.5 describes the equipment we use and Sec. 5.6 finally presents the results. In Sec. 5.7 we demonstrate one of many possible uses of this research work. Conclusion and future works are in Sec. 5.8.

This Chapter is based on the article we submitted to "Software: Practice and Experience" journal [50].

5.2 Notation and Terminology

The terminology used in computer performance is very vast and often ambiguous, due to the mix of strictly scientific terms together with commercial-driven keywords used by vendors to advertise their products. For the purpose of clarity, Tab. 5.1 specifies the notation adopted in math and algorithms used in this Chapter..

Symbol	***Meaning***
N	Number of cores in CPU
K	Number of tasks executed on CPU
K_s	Number of sensitive tasks ($\zeta_b(\cdot)$) to be estimated)
O_h	Overhead introduced by measurements as fraction of time
$\zeta(i)$	Actual IPC measured for task i in the shared environment
$\zeta_b(i)$	Target (in isolation) IPC for task i
t_m	Time required to obtain a point estimate of $\zeta_b(\cdot)$
t_{sleep}	Time between measurements

Table 5.1: Notation specific to Chapter 5.

5.3 Performance Isolation and Monitoring

Performance isolation means that an application is running exploiting all the resources it is entitled to. It is strictly related to the SLA between the customer and the computing facility: failing to provide it not only represents a breech of the contract, but it may result in unstable performance leading to unpredictable freezes, spike overloads, monitoring alerts and degraded user experience. In the latter term we include direct human experiences as in video streaming fruition, as well as indirect ones like delayed termination of a scientific computing task.

Processor sharing with preemptive multitasking is the earliest form of performance isolation, it requires careful crafting of the machine load and complex monitoring of the actual fraction of CPU time dedicated to tasks, and of the system freezes caused, for instance, by intensive IO. With the advent of multicore systems it became possible to dedicate one (or more) cores to a single task. Apparently, this solves the problem of isolation, at least for the applications and tasks that require enough resources to make it convenient to fully dedicate them one or more cores so they do not need to share CPU time with others.

Modern operating systems support both methods and their combination. For example, normal tasks can migrate from one core to another, and specific interrupt handlers can be pinned in a way to evenly spread the interrupt load across the cores [67].

5.3.1 Symmetric Multiprocessing System (SMP) Open Issues

Unfortunately, SMP cannot provide complete performance isolation. Hypervisors and operating systems see CPU and its cores as separate entities, which is not the case in practice. CPUs and cores share some components that are either very expensive (e.g., cache memory), or are deemed to be less used than the CPUs/cores themselves (FPU or command decoders), or simply are inherently unique to the board, as the main memory or the peripheral bus. As a result, a shared component is available to a specific core only if it is not used by other cores. If the resource is busy the task has to wait for it, and this creates stalls in the program execution [79], wasting computing resources and violating performance isolation. Predicting the components usage and classifying tasks to understand their mutual interaction is very difficult [49], thus, a methodology is needed to measure and estimate performance isolation at run time.

Fig. 5.2 shows how bad the violation of performance isolation can be. It shows eight different applications (see Sec. 5.5 for details of selected applications) evaluated in isolation, i.e., running alone on the CPU, and in a shared environment, i.e., running on a dedicated core, but with the other cores busy as well. The performance measure is IPC, a metric strictly correlated with the application throughput and efficiently derived from Hardware Performance Counters (HPC) [84, 51]. For all the applications considered the difference is striking, on average halving the performance, and also increasing the spread of the performance (see BLOSC and SDAG). The boxplots in Fig. 5.2 show the average value of the measures (red dots

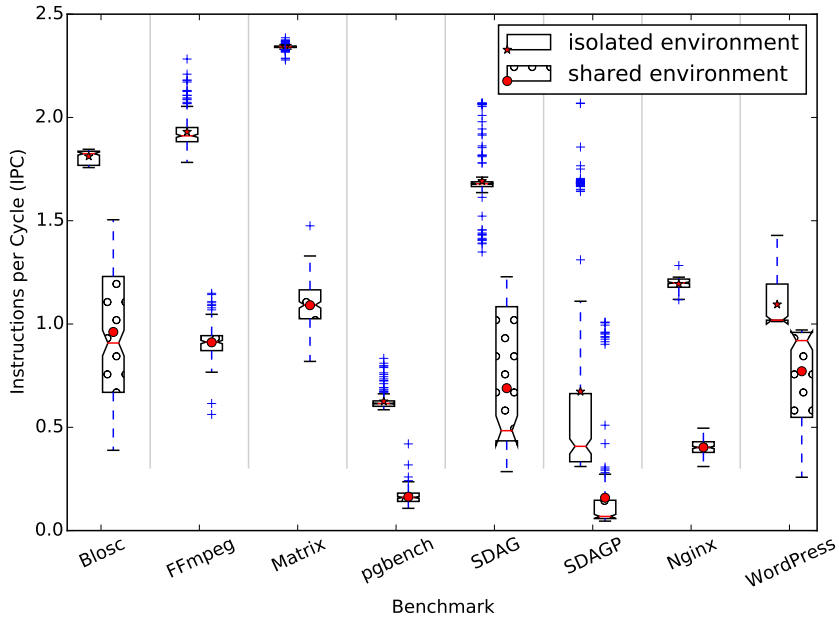


Figure 5.2: Applications profiled in isolated (no other core is loaded) and shared environments (the other cores are used too): the gap shows how large the difference can be (CPU: Xeon E3-1245 V2, 100 ms sampling).

and crosses), median (notches connected by red line), the 25th and 75th percentiles (the boxes), the 10th and 90th percentiles as whiskers, and all the outliers as isolated blue crosses. Interestingly, the measures are neatly separated and statistically meaningful, as boxes are always completely separated and in many cases outliers do not overlap.

The problem is in obtaining the isolated performance of applications in the shared environment. For some very well known and profiled applications running on well known hardware the self-profiling [75] can be an option, but it is difficult to tell in general if the performance is low due to the application problems or due to the poor hardware multitasking.

Notice that the guarantee of performance isolation is a *relative measure* problem: it is about being able to tell if the application is running as it would in isolation, or instead it is suffering from interference coming from applications running on resources that are ideally separated (other cores),

but indeed share components with the considered core.

5.4 Profiling Methodology: *Freeze'nSense*

As shown in [49, 95] the use of HPC, and IPC in particular, is a very efficient technique to estimate relative performance of a task. The key issue is deriving the target (isolated) performance of the task to compare it with the current performance. Let $\zeta_b(i)$ and $\zeta(i)$ be the target performance of a generic task i and its performance measured in the shared environment. Measuring $\zeta(i)$ is relatively easy (see [49, 95]); estimating $\zeta_b(i)$ is instead a challenge. $\zeta_b(i)$ can be time-varying and it is in general a stochastic process, so that a single sample bears little insight on $\zeta_b(i)$ itself. The authors of [95] base their estimation on the long-term observation of $\zeta(i)$, assuming that task i is persistent and runs most of the time without substantial interference. This assumption allows collecting the Empirical Probability Density Function (e-pdf) of the marginal distribution of the $\zeta_b(i)$ process, so that measures on $\zeta(i)$ deviating substantially from the e-pdf can be used to identify a task suffering from interference.

The approach we propose is radically different. Its goal is to obtain samples of $\zeta(i)$ and $\zeta_b(i)$ that are time-correlated, i.e., they can be compared nearly in real-time without a priori knowledge of the characteristics of task i.

To achieve this goal we estimate $\zeta_b(i)$ by freezing all applications except task i for a very short time period t_m, so that task i is effectively running "alone" on the computing node. The challenge is keeping the overhead of this operation at an acceptable level, while having a reliable estimate of $\zeta_b(i)$. The operation can be repeated for any task that is deemed "sensitive" or at risk. Furthermore, since the procedure is task-agnostic, it can also be directly applied to an entire VM, where it is difficult to know what

applications are running and the hypervisor is unable to stop single tasks inside the VM. We stress that the tasks (or threads) to be frozen are only those that are running on the same computing node of task i, not the entire data-center! Moreover, the inter-measure time $t_{sleep} >> t_m$, and t_m is just a few ms, so that the overhead is low and the other tasks barely notice the interruption. Only the estimation of $\zeta_b(\cdot)$ requires freezing other applications; $\zeta(\cdot)$ of all applications can be measured at run time with negligible overhead. Alg. 1 summarizes the measure procedure running on each node.

Algorithm 1: Loop used to estimate the ground-truth performance target for sensitive tasks on a computing node.

```
1 while True do
2     for app ∈ sensitive applications do
3         rest ← applications \ app
4         stop(rest)
5         after t_m   ζ_b(app) ← ζ(app)
6         start(rest)
7     sleep(t_sleep)
```

The overhead of *Freeze'nSense* is expressed by Eq. (5.1), where K_s is the number of sensitive applications for which $\zeta_b(\cdot)$ has to be evaluated, t_m is the time needed for the estimation and t_{sleep} is the time between measures.

$$O_h = \frac{K_s t_m}{K_s t_m + t_{sleep}} \tag{5.1}$$

Values of t_m as low as 10 ms can be used, while t_{sleep} can be 10 s or more. With these numbers and assuming that the number of critical tasks on a single CPU does not exceed 10, the overhead is $O_h \leq 10^{-2}$. If instead we assume that the infrastructure supports exactly one computational task per core, and all of them are critical, then the overhead is simply $O_h = \frac{K_s N}{K_s N + t_{sleep}}$ where N is the number of cores.

Table 5.2: Main characteristics of our test platforms.

Platform name	**Xeon**	**AMD FX**	**ARM Exynos**
CPU	Xeon E3-1245 V2	FX-8120	Exynos-4412
Cores	8	8	4
Frequency	3.4GHz	3.1Ghz	1.7GHz
Memory	4x4Gb DDR3-1333	2x8Gb DDR3-1600	2Gb
Storage	3Tb RAID-1	64Gb SSD	8Gb eMMC
OS	Arch Linux		
Kernel	3.19.3	3.19.2	3.8.13
Virtualization	qemu-kvm		bare metal

5.5 Implementation

The proposed methodology is general and can be applied to almost any hardware and does not require large facilities to be implemented, although it naturally scales well to any number of computing nodes, as the measures and the overhead are strictly local to one CPU. We use three different platforms (Intel, AMD and ARM) described in Tab. 5.2 to test the functionality and performance of *Freeze'nSense*. The Xeon platform has 4 cores with Hyper-Threading (HT), claiming 8 independent processing units, the AMD FX presents four 2-core modules for a total of 8 cores, and the Exynos platform has 4 symmetric cores. Thus, we are considering three quite different architectures, that can be claimed to cover reasonably well the spectrum of computing devices architectures available today.

5.5.1 Benchmarks and Workload

The experiments with *Freeze'nSense* are run using eight different programs that are our benchmarks. To make the scenario more realistic, each program is run into a VM implemented as a Kernel Virtual Machine (KVM) Linux container. These benchmarks are selected to provide a comprehen-

sive blend of cloud workloads, ranging from a web Content Management System (CMS) to multimedia encoding, to scientific calculus. The emphasis is given to standard applications (*FFMPEG*, *PGBENCH*, *SDAG*, *SDAGP*, *NGINX*, *WORDPRESS*), although two synthetic, ad-hoc, benchmarks (namely *MATRIX* and *BLOSC*) are present as well. The benchmarks are described in Sec. 3.1.1.

During experiments each core of the platform is loaded with a VM running one of the benchmarks. The Xeon and FX platforms has exactly 8 cores, so all benchmarks are run at the same time on these platforms.

5.5.2 Performance Sampling Issues

Unless otherwise stated, performance sampling is done using `perf` [84] (the former PCL – Performance Counters for Linux – toolbox) to exploit its stability and support from the community. The main limitation of `perf` is that it is mostly designed to serve as a standalone tool, not as a library interfaced from a program. As a result, interacting with `perf` may cause noticeable overhead when the sampling time is very short ($< 50\,\text{ms}$). To overcome this (and other) limitations we developed an additional performance sampling library that we used in these cases. The remainder of this Section describes faced technical challenges and lists the solutions adopted to overcame them.

Limited Sampling Frequency.

`perf` supports hardware and software (tracepoints) events, system-wide, per-process, per-core, per-thread monitoring and many other functions. Unfortunately, the standard release does not support sampling periods smaller than 100 ms. We patched this tool to support sampling periods t_m

as small as 1 ms[1].

Freezing VMs.

Even idle VMs consume some noticeable CPU and memory resources. In our measurements each KVM instance caused roughly 200 context switches and $5 * 10^6$ instructions per second. Hence, the estimation of $\zeta_b(\cdot)$ is more accurate when freezing the whole VM and not just the job inside them. Unfortunately, unlike normal processes, KVMs ignore the SIGSTOP signal; additional interaction with the hypervisor is required to stop VMs. This adds some overhead and makes the freezing phase before the measure of $\zeta_b(i)$ a bit longer ($\sim 0.25ms$ per VM).

Performance sampling for VMs is ragged.

Current implementations do not allow for random access of counters of virtual CPUs[2]. The measurements are done by a virtual device called virtual PMU. At some points in time the real PMU is synchronized with the virtual one, but between synchronization points the virtual PMU does not reflect properly the situation on the VM. Though the synchronization is done quite frequently, for high-frequency sampling we can miss some data points. The more counters activated the bigger chances to hit the problem. The behavior of PMU is OS- and hardware-dependent; currently nothing can be done here to improve the situation, but monitoring PMU to discard incorrect samples (e.g., returning too big or too small values).

[1]The patch is available at `https://github.com/kopchik/limit/blob/master/perf-small-interval.patch`

[2]Avi Kivity, "Performance monitoring for KVM guests." Invited talk at KVM Forum, Vancouver, Canada, August 16, 2011. Available as RedHat Video at `https://www.youtube.com/watch?v=skQrYiME-N4`

Opening counters may suddenly fail.

Sometimes Linux kernel fails to initialize performance counters. This is seen as an error returned by $sys\_perf\_event\_open()$ system call. We overcome this by repeating the syscall till it succeeds. Fortunately, this error is quite rare ($< 0.1\%$ of total calls).

Measurements may create interference.

`perf` is a powerful and flexible tool, but in adopting it some precautions must be taken. As discussed in Sec. 5.5, the overall overhead is in the order of 1% or less, yet the tool may affect measurements, because the `perf` tool consumes CPU cycles right when the measure is performed. Under full load when no spare CPU time available it has to share the CPU time with a task. If the task is profiled it may show lower performance.

To overcome this potential problem we designed a small performance sampling tool that generates as little interference as possible, following the steps below.

1. At bootstrap it initializes ("opens") the needed performance counters and keeps them open. Any subsequent measurements do not need any initialization.
2. Each measurement is done with just three system calls. The first is to reset the counters to zero, the second is a sleep function. The final system call reads the performance measure. With the current design the overhead of one measurement can be as low as a fraction of a microsecond [85].
3. As we assume a VM per core, there is no need to account performance on a task level; we gather statistics at the core granularity. This prevents the system from saving and restoring the counters on every context switch, making the measure faster and lighter.

5.6 Results

During the evaluation and measurement campaign we collected a very large amount of data, all confirming the results we present here and supporting the insight gained. Since the results on different platforms are very similar, we often present results only for one platform (normally Xeon) implying, without repeating it continuously, that the result is valid for all platforms. If not otherwise stated, all the boxplots presented in figures are based on 1000 samples measured with `perf` during tens of minutes of operations, thus, all results are statistically very meaningful, and we can safely state that percentiles and outlier values would not change significantly with longer measures.

5.6.1 Freezing Validation

The first step in the evaluation is the validation that the performance measure of task i freezing all other VMs in the CPU is a good estimator of $\zeta_b(i)$, and that it can be done with the low overhead and for all platforms.

Fig. 5.3 reports boxplots of the estimate of $\zeta_b(i)$ for all the benchmarks obtained on the Xeon platform both in total isolation (i.e., the application is running inside a VM and the VM is the only active task in the entire CPU, apart from the hypervisor and other compulsory management tasks, only one core out of N is used), or in freezing the environment (i.e., each core is assigned a VM and a different benchmark runs in every VM, all N cores are used, but in turn, $N-1$ VMs are frozen and the remaining one is measured and an estimate of $\zeta_b(i)$ is taken). The distribution of $\zeta_b(i)$ samples is remarkably similar in all cases, strongly supporting our idea of temporary freezing to estimate the isolated performance. Clearly, the distribution changes slightly with changing t_m; only BLOSC displays a remarkable change with t_m. BLOSC is hand-written, optimized and

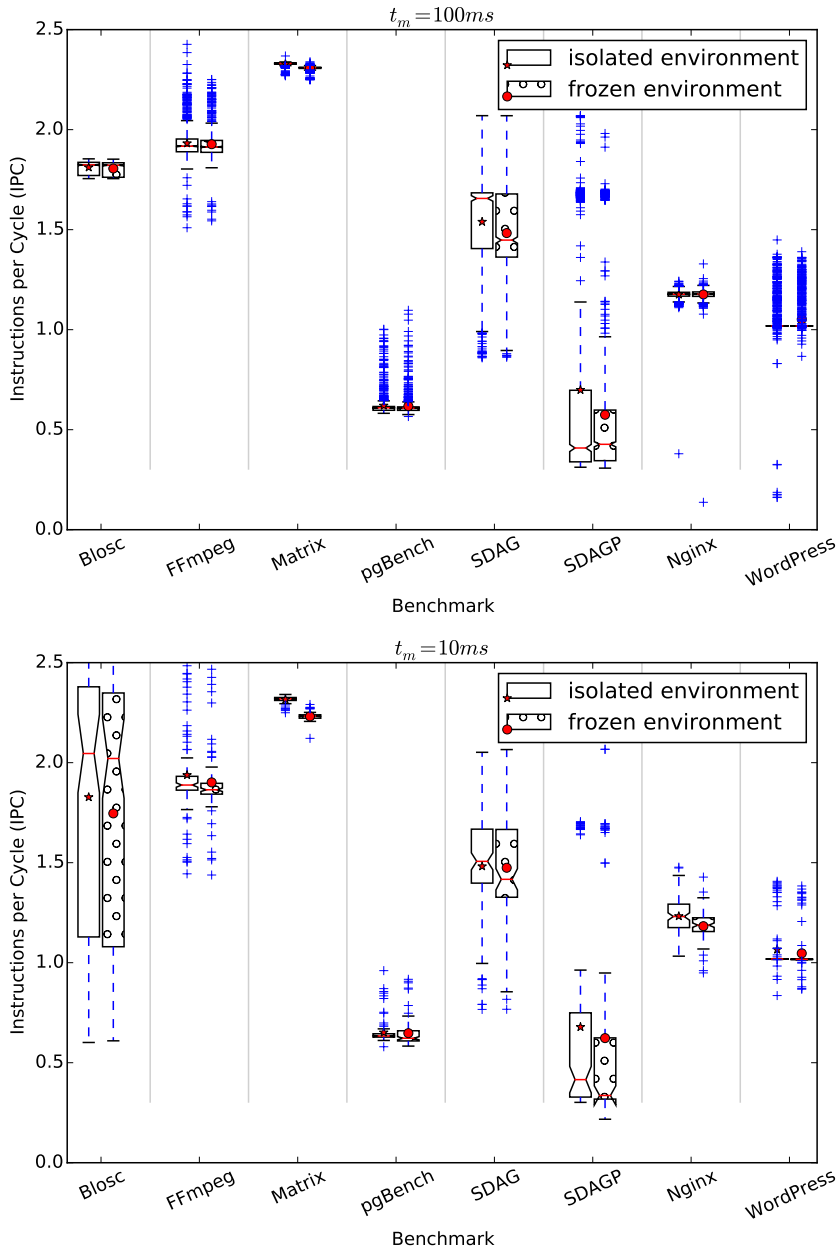


Figure 5.3: Intel Xeon: estimate of $\zeta_b(i)$ when tasks runs alone in the CPU and when the environment is frozen; t_m =100 ms in the upper plot, t_m = 10 ms in the lower plot.

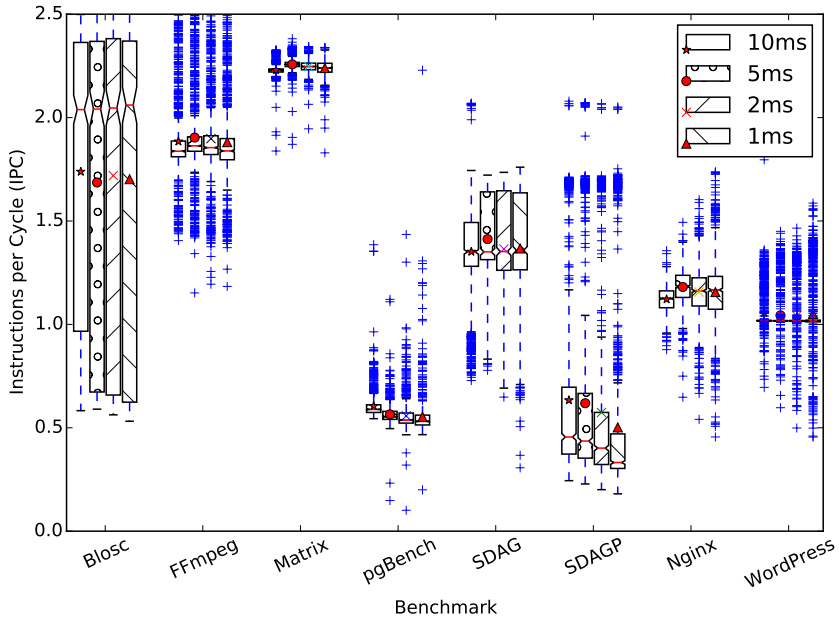


Figure 5.4: Intel Xeon: Reducing t_m to the limit: estimate of $\zeta_b(i)$ for $t_m = 10$, 5, 2, and 1 ms; t_{sleep} is reduced to 50 ms.

uses inline assembler expansions; it performs extremely monotonous operations (fast compression of big memory regions). This allows a mostly linear access to memory and near-100% branch prediction hit rate, thus its performance in the long run is almost constant, while in the short run is dominated by interrupts and memory accesses. Remarkably, this very peculiar behavior is perfectly captured by *Freeze'nSense*.

Fig. 5.4 confirms that freezing the environment to estimate $\zeta_b(i)$ is extremely reliable and performing: in practice the sample distribution does not change even if t_m is reduced down to 1 ms and samples are separated by a mere 50 ms. This observation opens the possibility of using adaptive values for t_m and t_{sleep} depending on the scenario and application. For instance, taking fairly long measures with very long t_{sleep} for long-lasting applications that show unpredictable short-term performance, while taking shorter, more frequent samples for very intensive but stable applications.

Coming to the AMD FX platform, we have to note that some of the

counters were hard-wired to the Linux watchdog subsystem, making measures unreliable. To replicate our results the watchdog must be disabled with the command `sysctl kernel.nmi_watchdog = 0`. More details and information on this behavior can be found in [14, 45]. With this issue solved *Freeze'nSense* behaves as predicted on this platform too, yielding consistent and correct estimates of $\zeta_b(i)$. For this platform we report the comparison of $\zeta_b(i)$ estimation reducing t_m in the freezing scenario in Fig. 5.5 where it is clear that also in this case t_m and t_{sleep} can be configured to adapt to different needs of cloud computing. Boxplots give a favor of the measure distribution, but a proper inspection of the empirical pdf bears more insight. Fig. 5.6 reports the empirical pdf of $\zeta_b(i)$ with $t_m = 2\,\text{ms}$ measured in isolation and with *Freeze'nSense* for two applications that display very different behavior: NGINX and BLOSC. The figure shows the reason of the different behavior: a single-mode distribution with a relatively low variance for NGINX and a bi-modal, very skewed distribution for BLOSC. The most interesting observation, however, is that *Freeze'nSense* correctly estimate not only averages, but the entire distribution, so that in a long-term measurement campaign a data center or cloud manager (human or autonomic [52]) can even take decisions based on the performance percentiles.

The ARM Exynos platform has some limitations (lack of hardware virtualization, limited RAM, and slow storage) that makes the application of *Freeze'nSense* challenging. So far only single-threaded benchmarks are supported. Therefore we present only a proof of concept with four benchmarks and t_m =100 ms. Fig. 5.7 shows the estimation of $\zeta_b(i)$ for four benchmarks both in isolation and freezing the environment for t_m. Once more *Freeze'nSense* is reliable in predicting the performance in isolation.

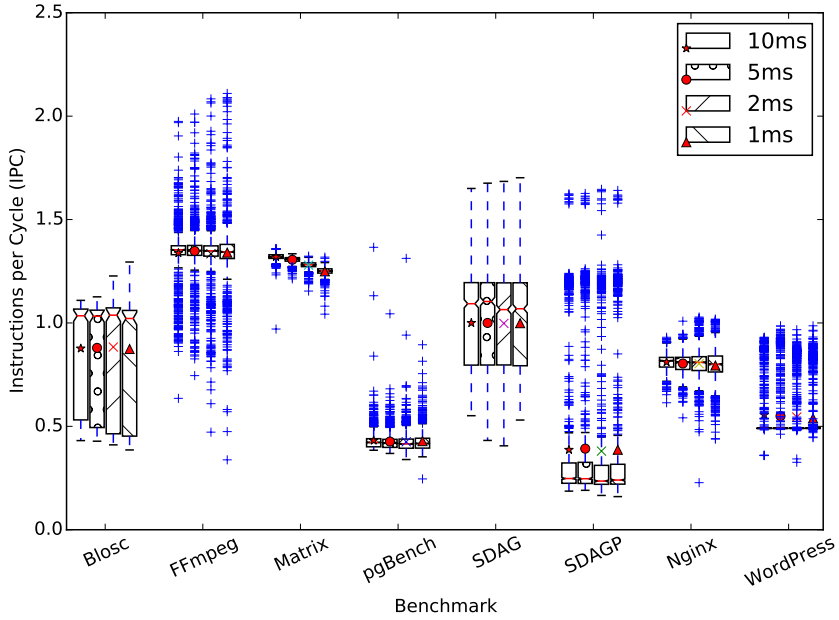


Figure 5.5: AMD FX: Reducing t_m to the limit: estimate of $\zeta_b(i)$ for $t_m = 10$, 5, 2, and 1 ms; t_{sleep} is reduced to 50 ms.

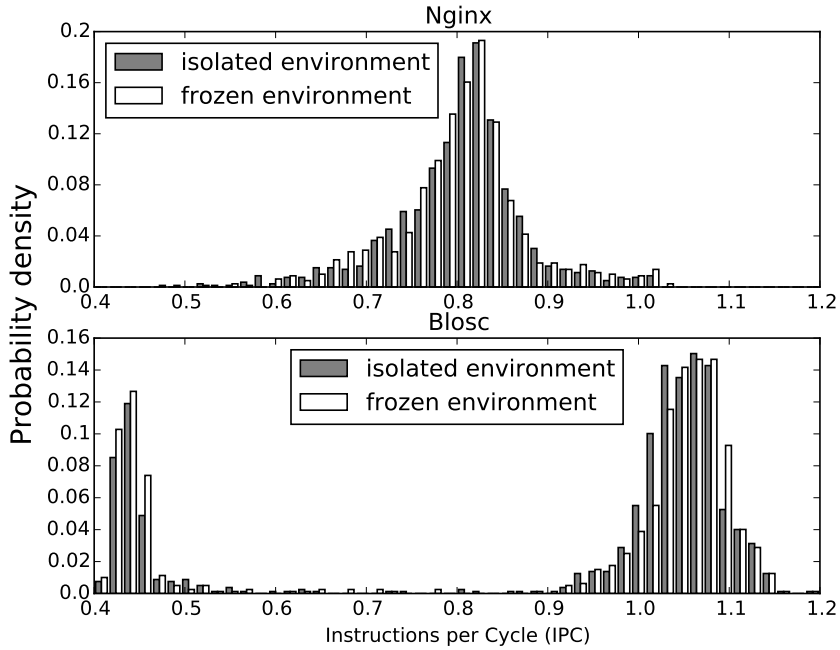


Figure 5.6: AMD FX: empirical pdf of $\zeta_b(i)$ estimates in isolation and with *Freeze'nSense* for NGINX and BLOSC for $t_m = 2$ ms.

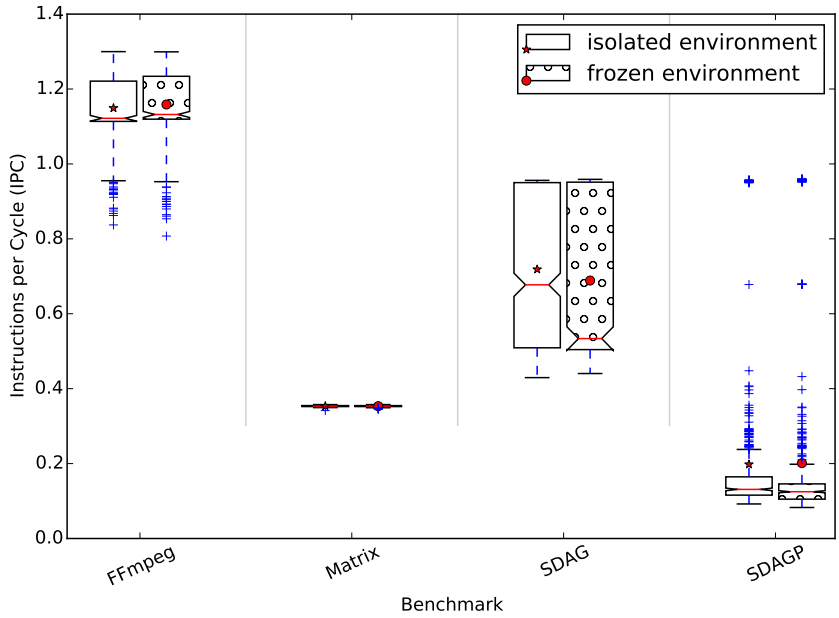


Figure 5.7: ARM Exynos: estimate of $\zeta_b(i)$ when tasks runs alone in the CPU and when the environment is frozen; t_m =100 ms.

5.7 Applying *Freeze'nSense* for CPU Balancing

Results of Sec. 5.6 are very encouraging, but can they be used in practice? Given the impossibility to access a large scale computing infrastructure for a full-scale evaluation campaign, we devised a simple proof-of-concept experiments on the Xeon and AMD FX platforms we have. We disregarded the Exynos for this experiment due to its problems and a lack of maintenance.

The idea is to fully load half of the CPU running benchmarks on 4 sibling cores and measure their overall performance. To do this we loop the benchmarks so that results are consistent and stable across different experiments. Next, we redo the experiments enabling *Freeze'nSense* to estimate if some of them are underperforming, select two most underperforming and remove them to distant cores, those are most probably less subject to interference, and measure the overall performance again to compare it

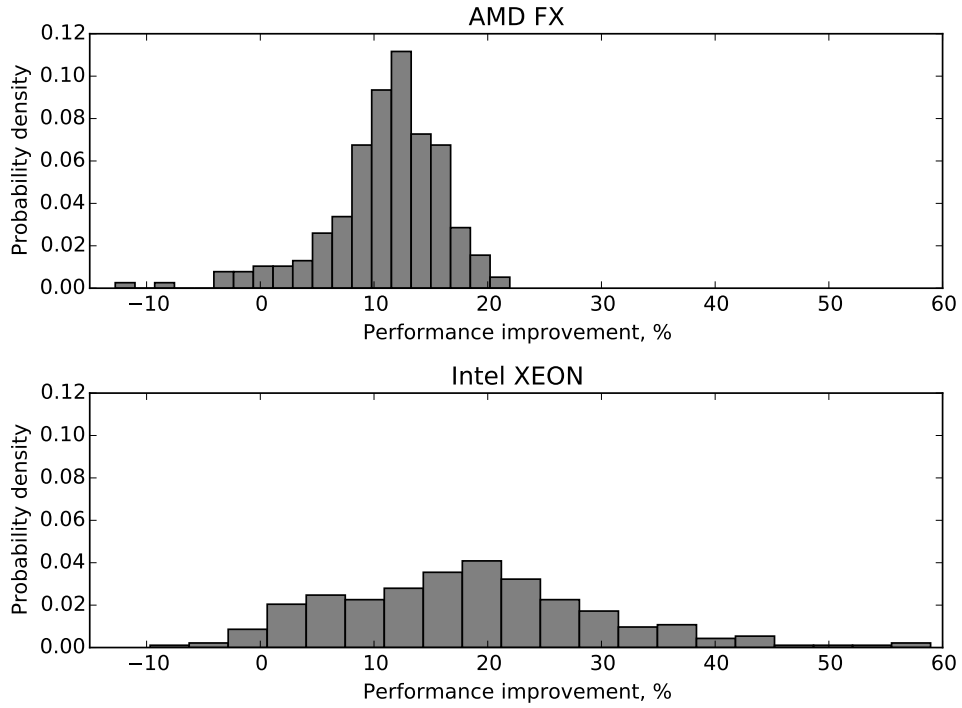


Figure 5.8: Distribution of performance improvement using *Freeze'nSense* to decide VM relocation.

with the one measured before. Recall that all our benchmarks run inside KVM VMs, so that entire VMs are relocated. In a full scale infrastructure this procedure would normally run on the entire computing node (or CPU) and relocation would probably involve other computing cores, possibly also switching on and off entire nodes and CPUs to optimize power consumption, but this is beyond the scope of this Chapter. Finally, the cost of relocation should also be taken into account for a global optimization. Once more this issue goes beyond the scope of this work, and its evaluation requires the knowledge of the architecture of the data center (to estimate the cost of relocation), so that a larger scale infrastructure is needed.

Fig. 5.8 reports the experimental pdf of the performance gain after relocation for both the Intel Xeon and AMD FX platforms. The results are very encouraging, showing an average gain in performance around 12% for

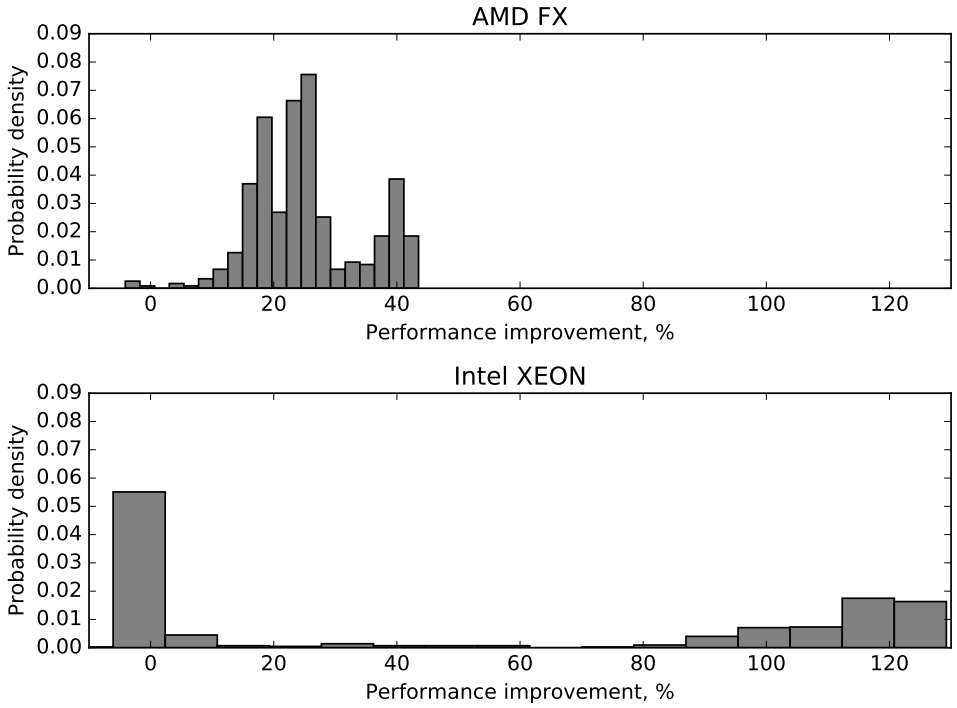


Figure 5.9: Distribution of performance improvement of VMs relocated by *Freeze'nSense*.

Xeon and 20% for AMD FX. The histogram shows performance change after letting *Freeze'nSense* optimize the placement.

The starting point is the worst (but not impossible) scenario: 4 random benchmarks launched on sibling cores, their performance is measured for $30s$ to average out any unexpected activities or change in programs behavior. Then for another $180s$ *Freeze'nSense* determines the interference between VMs. As interference data is ready we relocate two most starving VMs to separate (distant) cores. The new performance is measured for another $30s$. There is $90s$ warm-up before each round to populate caches. The histograms represent 250 launches of the experiment.

While the overall performance improvement is not that high, the relocated tasks have much higher gain: Fig. 5.9 shows performance improvement of tasks relocated by *Freeze'nSense*. Comparing two platforms it can be seen that Intel Xeon benefits much more from the relocation. This is

because Intel's Hyper-Threading technology relies on heavy resource sharing [66]. The AMD sibling cores share much less, hence, less competition of resources and much less interference [21].

The timings no need to be so long. For most of our benchmarks just a few seconds is enough for warm-up and measurements. However, there is a notable exception: SDAGP takes up to 1 minute to complete one iteration. Because of this short measurements may produce inconsistent results.

Interestingly, under full load both platforms show quite similar performance. The Intel CPU has significantly better peak performance per core, but hyper-threading does not help too much. AMD cores are more mild, but they all contribute to full speed. Both CPUs have comparable transistors budgets (1.4B for the former and 1.2B for the latter).

5.8 Conclusions and Discussion

On-the-fly application profiling is vital for efficient management of data centers and cloud computing facilities. Performing measurements and assessing performance of applications is challenging in virtualized environments. Performance degradation often comes from interference of hardware subsystems shared between cores including caches, FPU, etc.

This Chapter presented *Freeze'nSense* a general purpose, transparent and minimally invasive technique for estimation of the intended performance of a task, i.e., the performance it would have if it were running in isolation. To estimate performance in isolation, the execution of all the concurrent tasks is temporarily stopped for the duration of the measurement period, which is in the order of milliseconds. IPC is used as the main performance indicator. It is generic and can be applied to most of the tasks and application as well as VMs.

Extensive evaluation on three different platforms confirms *Freeze'nSense*

ability to estimate task performance reliably. The obtained results inspired a simple proof-of-concept task relocation methodology that has been implemented and proved *Freeze'nSense*'s ability to automatically detect VMs that suffer the most from interference and relocate them.

Freeze'nSense can naturally be used in heterogeneous infrastructures that are common in most of data centers due to gradual hardware upgrades. This raises questions how to distribute applications between old and new servers and what hardware to buy next. With our technique these problems are easy to solve by analyzing (potentially without human intervention) performance profiles.

Chapter 6

Conclusion and the Road Ahead

Cloud computing has irreversibly changed the computing market and unleashed great power for good. Despite all the hype around it, the outcome is difficult to overestimate: fine-grained resource management, application life cycle management, scalability and reliability are all affected and improved by clouds. Clouds also open new business opportunities that would be impossible without the Pay-As-You-Go model.

This thesis contributed to the cloud well-being in several ways. Our first contribution is a simple yet powerful application interference model that helps predicting the interference between programs. The model predicts the performance of tasks running on the same machine by estimating the interference they create to each other. This can be used for, e.g., estimating and comparing performances of different task placements.

The second contribution is a task ranking technique that allows to sort running applications according to the average level of interference they create. In a cloud with many applications it is very likely to see applications competing for resources. With the advent of high-density server equipment the situation gets ever worse. For this reason it is very important to identify (and possibly relocate) applications making the strongest impact on others. We have proposed a simple technique to identify such programs. The

technique is based on the empirical observation that program's average interference has good correlation with specific performance counters. Once performance counters representative for interference are found they can be used to estimate interference of programs to steer resource management.

Our final contribution is a technique for estimating performance isolation of tasks and virtual machines running in shared environment. The tightening efficiency constraints and raising environmental concerns require cloud providers to increase cloud density more and more. But the density cannot be increased infinitely, at some point interference and resource starvation will make it impractical. Where to draw the line? Our technique provides an accurate estimate of the performance of an application as if it would be running alone on the machine. This is done by freezing other applications for a few milliseconds and taking performance samples that are representative for isolated performance. These samples can be compared with performance samples obtained without freezing. The difference will show how badly the application is affected by interference. Given this information, the CRM can, e.g., estimate SLA or adjust placement to reduce interference.

6.1 Future Research

So far we run our experiments on a budget and on a very small scale – just one machine of each hardware architecture. It would be interesting to see the proposed techniques at scale of tens machines or more. That would bring new challenges as well as new optimization opportunities. For example, one of the challenges is to align applications' resource requirements with cloud resource management. Some applications are very dynamic in nature and can change their behavior over time. The CRM should detect such applications and treat them specially because, depending on the rate

of changes, redistributing the load may or may not be an option.

Another topic to dig into could be aligning resource management with new application architectures. In this work we dealt only with small applications without dependent services running outside their containers. However, a typical service consists of many components (frontend, backend, database, message bus, etc) potentially scattered across multiple VMs and machines. Or it can be an application based on *microservices*, it is when most of the processing is delegated to external services (like a file storage, database, authentication, load balancing, video encoding, logging and others). In such environments it is not granted that increasing efficiency of one component we make noticeable impact to the whole software stack. Furthermore, networking distance may also affect the performance. This requires identification of hot spots, dependencies between services and even co-location/co-scheduling of them to reduce network distance or increase data sharing. All that are potential topics for our future research.

Appendix A

Vocabulary

Term	*Meaning*
Counter	Register counting events of a given type.
Event	Low-level hardware event, e.g., a cache miss or successfully executed (retired) instruction.
Freeze'nSense	Name of the performance isolation measuring technique proposed in this thesis.
Hyper-Threading (HT)	technology of presenting a single physical CPU core as multiple logical CPU cores that share the same computational resources (for their better utilization).
Kernel Virtual Machine (KVM)	Linux virtualization technology.
Isolated Environment	Servers dedicated to run a given application exclusively, without sharing resources with other applications.

Interference	Degradation of performance caused by task A running on one CPU core on task B running on another core. The source of interference is in the resources shared between CPU cores (cache memory, Algebric and Logic Unit (ALU), etc.), which in most of the cases do not allow simultaneous access: whenever one CPU core accesses a resource, the other must wait until it is released (instruction stalls).
Performance Monitoring Unit (PMU)	A built-in CPU circuit that gathers low-level performance statistics, such as number of executed instructions or cache misses.
Precise Event Sampling	Mode of PMU operation when there is a dedicated counter for each enabled event.
Shared Environment	Hardware running many applications simultaneously (e.g., shared hosting, clouds).
Symmetric Multiprocessing System (SMP)	Computing system with multiple semi-independent and identical processing units (cores).
Simultaneous Multithreading (SMT)	Technology to allow a CPU core to execute multiple (usually two) tasks simultaneously if resources allow. Unlike SMP most resources are shared. SMT provides modest speedup, but cheap in terms of silicon area and power consumption.
CPU Affinity	A list of CPU cores a task or a thread can be scheduled on. Proper task pinning may well improve the performance due to, e.g., better caches utilization.

Out-of-Order (OoO)	Modern CPUs execute instructions not in program order to minimize pipeline stalls. This is only possible for instructions that can be executed independently and not depend on each other. The profit comes when the current instruction waits for data, but later instructions do not have to.
Virtual Machine (VM)	Virtual machine or container.
Total Cost of Ownership (TCO)	Total cost of ownership, an amount to be paid during the lifecycle of a facility, product or service.
Queries Per Second (QPS)	Queries per second.
(CPU) Affinity	CPU cores on what a task is allowed to run.
KVM	Kernel-based Virtual Machine, virtualization technology used in Linux kernel. Requires hardware support.
Resource Pool	Free cluster resources that do not belong to any of running tasks.
Cloud Resource Manager (CRM)	software suite that manages resources, clouds, and tasks in the entire cluster in order to maintain its efficiency.
Non-Uniform Memory Access (NUMA)	Non-Uniform Memory access is a memory design when every CPU has its own local memory.

Appendix B

Research Hiccups and Dead-ends

While the author enjoyed almost every moment of this research, it was not always pure fun. Sometimes the hardware did not work as expected, sometimes software was a bit buggy... Occasionally the research was troublesome and we spent quite a bit on dealing with obscure problems. We think it is worth sharing some of our experience so other researchers could learn from it and avoid these traps for young players.

B.1 Importance of Storage

Our original intention was to prepare a live-usb drive allowing us to conduct the experiments on any platform that could boot from it. And the first platform was this one. However, the first uses uncovered many hidden problems of such solution.

We bought a seemed-to-be-decent USB3 Stick (Kingston DataTraveler R3.0). But the experiments didn't go well crashing in random places due to timeouts. A quick investigation showed that during timeouts the system were spending most of the time in "waiting for IO". This was strange considering our tests are not IO bound. So we dug further and found that our VM images are quite large and updated quite frequently. But why?

We do not make a dedicated image for each VM instance. Instead we

have a template (a master image) that we clone. The clones are copy-on-write (CoW): unmodified blocks are referenced from the template. If a VM modifies something it gets its local copy of the block and the original templates remains intact. This saves a lot of space. Yet images are not air light because CoW works on a block level. Even if one changed byte makes whole block copied. This becomes extremely inefficient on random small writes. In our case such writes are logs and updates of file timestamps. As we have 4-8 VM instances this overhead is multiplied by a factor of 4-8 as well. The usb stick can only approach the advertised speeds (up to 30Mb/s writes) only on sequential access. The actual speed was about 3-5Mb/s because of randomized access and write amplification [43] (to write a couple of bytes a flash drive has to erase a whole *erase block* of size of up to several megabytes and only then write the modified data back).

Another issue was that the stick tended to sleep after a very short time of inactivity; the wake up time was rather noticeable leading to frequent "microfreezes" of applications.

We solved all our storage problems by replacing the usb stick with a good SSD (Crucial M4-CT064M4SSD2). We also tuned the filesystem not to update access timestamps (*atime*).

So just small bunch of VMs was enough to put a cheap storage on the knees. And the point of the story is "never underestimate even small VMs if they are in quantity".

B.2 Looping programs

Our ARM testbed is based on Exynos 4412 SoC that has very limited support in Linux. This imposed limitations on testing we had to overcome. One of the major problem was that performance counters could not be inherited by child processes. As a result, we could not run benchmarks

with short execution time. Normally, we use a parent process (supervisor) to spawn new (child) prosses upon their termination. But because children did not inherit performance counters our software could not measure their performance. So we were limited with just one process (possibly with multiple threads though) per container. We decided to modify sources and execute main function of our benchmarks in loop. But it turned out many programs do not properly release resources after use and rely on OS functions to free memory and other resources after they finish execution.

Hence, we tried to add garbage collector – BoehmGC ([20]) – to suppress memory leaks. Otherwise a couple of seconds was enough to fully clutter all RAM. The garbage collections supposed to work by replacing memory allocation functions (malloc/realloc/free) with their GC variant (in this case free is no-op). This would be done without touching the sources by setting an environment variable (LD_PRELOAD) to tell the linker to pre-load our library ([2]) that does the trick. Unfortunately, some benchmarks still leaked. This could be caused by low-level libraries that do not use system malloc. So we went another way.

There is a family of exec* system calls that does exactly what we need: it spawns a new process inside the old one (the old ceases to exist). The usage is very simple, a complete listing of am example program is shown in Listing B.1.

The exec* calls do not close file descriptors. It might happen that with each iteration the program opens more and more file descriptors. Unfortunately, there is no efficient platform-independent way to do this. For the sake of simplicity, we just iterate over the first 1024 descriptors (skipping stdio, stdout and stderr) and unconditionally close them.

Listing B.1: How to loop a program.

```
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
  // close any open file (except stdin/stdout/stderr)
  for(int i = 3; i < 1024; i++) {
    close(i); }
  }

  // spawn new process in-place
  execvp(argv[0], argv);

  // never reached but required by the function signature
  return 0;
}
```

B.3 Unexpected Load Variation

When we were collecting the data for the thesis we asked our friends to provides us with data from their projects. And we got quite interesting results from one project: it showed min/max load ratio of was just 1.3 at most. How so?

We contacted the project owner and he kindly agreed to record performance history for 24 hours. The data showed that there was unusually low (2x) day-to-day variation. A quick investigation confirmed that night visitors and search bots create significant load even off-peak hours. But there was more. A big portion of resources was consumed by a backup/mirroring tool (duplicity[1]) that constantly synchronizes data between servers. And because of the way it works (it performs full scan on every execution)

[1] duplicity.nongnu.org/

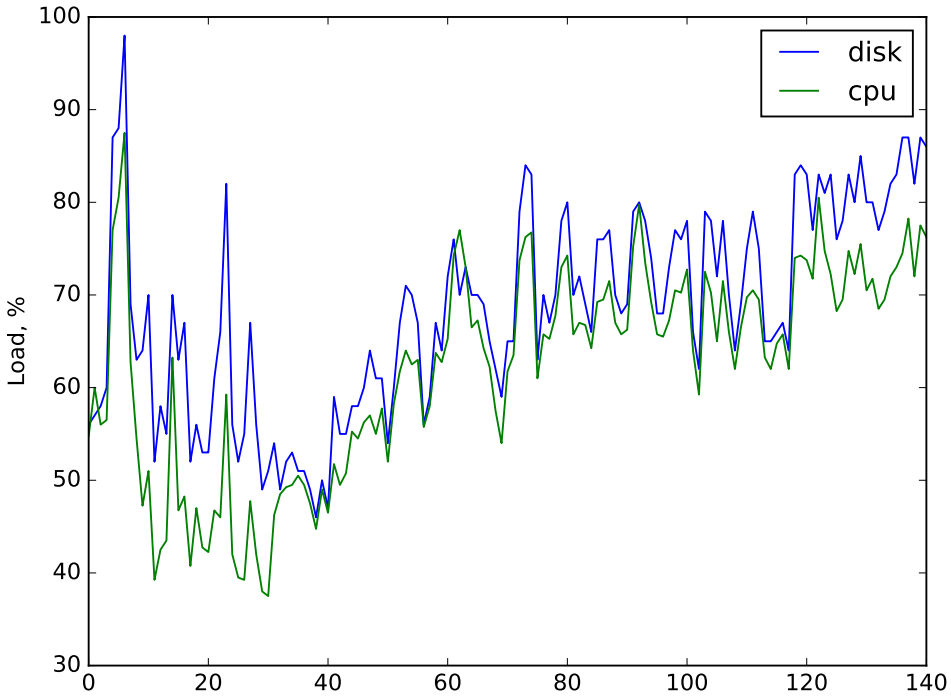


Figure B.1: CPU and DISK load variation over 24hours for `linux.org.ru` server.

it creates noticeable permanent load that barely correlates with visitors traffic.

Bibliography

[1] Creative commons attribution license. https://creativecommons.org/licenses/by/3.0/.

[2] GCPreload, a library that replaces malloc/realloc with their GC equivalents. https://github.com/kopchik/gcpreload.

[3] Linux containers – chroot on steroids. http://lxc.sourceforge.net/.

[4] Nginx web server. http://nginx.org/en/.

[5] pgBench, a benchmark for PostgreSQL. http://www.postgresql.org/docs/9.4/static/pgbench.html. Accessed: 2015-06-08.

[6] The Xen project. http://www.xenproject.org/.

[7] November 2015 web server survey. http://news.netcraft.com/archives/2015/11/16/november-2015-web-server-survey.html, 2015.

[8] Marco Ajelli, Renato Lo Cigno, and Alberto Montresor. Modeling botnets and epidemic malware. In *Proceedings of the IEEE International Conference on Communications (ICC)*, pages 1–5, 2010.

[9] Mayez A. Al-Mouhamed and Khaled A. Daud. Experimental analysis of SMP scalability in the presence of coherence traffic and snoop filtering. In *Proceedings of the 14th IEEE International Conference on*

High Performance Computing and Communication & 9th IEEE International Conference on Embedded Software and Systems (HPCC-ICESS), pages 81–88, 2012.

[10] Alaa R. Alameldeen and David A. Wood. IPC considered harmful for multiprocessor workloads. *IEEE Micro*, (4):8–17, 2006.

[11] Muhammad Aleem, Radu Prodan, and Thomas Fahringer. On the evaluation of JavaSymphony for heterogeneous multi-core clusters. In *Proceedings of the 16th International European Conference on Parallel and Distributed Computing (Euro-Par)*, pages 23–30. Springer, 2010.

[12] Francesc Alted. Why modern CPUs are starving and what can be done about it. *Computing in Science & Engineering*, 12(2):68–71, 2010.

[13] AMD. BIOS and kernel developers guide (BKDG) for AMD family 15h models 00h-0fh processors. `http://support.amd.com/us/Processor_TechDocs/42301_15h_Mod_00h-0Fh_BKDG.pdf`, 2012.

[14] AMD. BIOS and kernel developers guide (BKDG). `http://developer.amd.com/wordpress/media/2012/10/31116.pdf`, 2013.

[15] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS Spring Joint Computing Conference*, pages 483–485. ACM, 1967.

[16] Joseph Antony, Pete P. Janes, and Alistair P. Rendell. Exploring thread and memory placement on NUMA architectures: Solaris and Linux, UltraSPARC/FirePlane and opteron/hypertransport. In *Proceedings of the 13th annual IEEE International Conference on High Performance Computing (HiPC)*, pages 338–352. Springer-Verlag, 2006.

[17] Nathan Beckmann and Daniel Sanchez. Jigsaw: scalable software-defined caches. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 213–224. IEEE, 2013.

[18] Reinaldo Bergamaschi, Indira Nair, Gero Dittmann, Hiren Patel, Geert Janssen, Nagu Dhanwada, Alper Buyuktosunoglu, Emrah Acar, Gi-Joon Nam, Guoling Han, et al. Performance modeling for early analysis of multi-core systems. In *Proceedings of the 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 209–214. IEEE, 2007.

[19] Sapan Bhatia, Abhishek Kumar, Marc E. Fiuczynski, and Larry L. Peterson. Lightweight, high-resolution monitoring for troubleshooting production systems. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 103–116, 2008.

[20] Hans-J. Boehm. Dynamic memory allocation and garbage collection. *Computers in Physics*, 9(3):297–303, 1995.

[21] Michael Butler, Leslie Barnes, Debjit Das Sarma, and Bob Gelinas. Bulldozer: An approach to multithreaded compute performance. *IEEE Micro*, (2):6–15, 2011.

[22] Damiano Carra, Renato Lo Cigno, and Ernst W. Biersack. Stochastic graph processes for performance evaluation of content delivery applications in overlay networks. *IEEE Transactions on Parallel and Distributed Systems*, 19(2):247–261, 2008.

[23] Carlos Carvalho. The gap between processor and memory speeds. In *Proceedings of the IEEE International Conference on Control and Automation (ICCA)*, 2002.

[24] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA)*, pages 340–351. IEEE, 2005.

[25] Jichuan Chang and Gurindar S Sohi. *Cooperative caching for chip multiprocessors*, volume 34. IEEE Computer Society, 2006.

[26] J.J. Colao. With 60 million websites, wordpress rules the web. so where's the money. `www.forbes.com/sites/jjcolao/2012/09/05/the-internets-mother-tongue/`. Last accessed: March 2016, 2012.

[27] Carlo Curino, Evan P.C. Jones, Samuel Madden, and Hari Balakrishnan. Workload-aware database monitoring and consolidation. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 313–324, 2011.

[28] Tanima Dey, Wei Wang, Jack W Davidson, and Mary Lou Soffa. ReSense: Mapping dynamic workloads of colocated multithreaded applications using resource sensitivity. *ACM Transactions on Architecture and Code Optimization*, 10(4):41, 2013.

[29] Idilio Drago, Marco Mellia, Maurizio M. Munafo, Anna Sperotto, Ramin Sadre, and Aiko Pras. Inside Dropbox: understanding personal cloud storage services. In *Proceedings of the ACM Conference on Internet Measurement Conference (ICM)*, pages 481–494, 2012.

[30] Ulrich Drepper. What every programmer should know about memory. *Red Hat, Inc.*, 2007.

[31] Tyler Dwyer, Alexandra Fedorova, Sergey Blagodurov, Mark Roth, Fabien Gaud, and Jian Pei. A practical method for estimating performance degradation on multicore processors, and its application to hpc workloads. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 83. IEEE, 2012.

[32] Stijn Eyerman and Lieven Eeckhout. Per-thread cycle accounting in SMT processors. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 133–144. ACM, 2009.

[33] Michal Feldman, Kevin Lai, and Li Zhang. The proportional-share allocation market for computational resources. *IEEE Transactions on Parallel and Distributed Systems*, 20(8):1075–1088, 2009.

[34] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, volume 47, pages 37–48. ACM, 2012.

[35] Frank Gens, Margaret Adam, David Bradshaw, Christian A Christiansen, and Laura DuBois. Worldwide and regional public IT cloud services 2013-2017 forecast. *International Data Corporation Market Analysis*, 38, 2013.

[36] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing performance isolation across virtual machines in Xen. In *Proceedings of the 7th ACM/IFIP/USENIX International Middleware Conference*, pages 342–362. Springer, 2006.

[37] Prabhat K. Gupta. Xeon+ FPGA platform for the data center. In *Proceedings of the 4th Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*, volume 119, 2015.

[38] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: an update. *ACM SIGKDD Explorations*, 11(1):10–18, 2009.

[39] Erik G. Hallnor and Steven K. Reinhardt. A fully associative software-managed cache design. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, pages 107–116. ACM, 2000.

[40] Amir H Hashemi, David R Kaeli, and Brad Calder. Efficient procedure mapping using cache line coloring. In *ACM SIGPLAN Notices*, volume 32, pages 171–182, 1997.

[41] Raoufehsadat Hashemian, Diwakar Krishnamurthy, Martin Arlitt, and Niklas Carlsson. Characterizing the scalability of a web application on a multi–core server. *Concurrency and Computation: Practice and Experience*, 26(12):2027–2052, 2014.

[42] John L. Hennessy and David A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., 5th edition, 2011.

[43] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. Write amplification analysis in flash-based solid state

drives. In *Proceedings of the SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–9. ACM, 2009.

[44] Cisco Visual Networking Index. Global mobile data traffic forecast update, 2015-2020 white paper. `http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html`, 2015.

[45] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual. `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf`, 2014.

[46] Sushil Jajodia, Krishna Kant, Pierangela Samarati, Anoop Singhal, Vipin Swarup, and Cliff Wang. *Secure Cloud Computing*. Springer, 2014.

[47] Joe Wenjie Jiang, Tian Lan, Sangtae Ha, Minghua Chen, and Mung Chiang. Joint VM placement and routing for data center traffic engineering. In *Proceedings of the 31st IEEE International Conference on Computer Communications (INFOCOM)*, 2012.

[48] Alain Kägi, James R Goodman, and Doug Burger. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA)*, pages 78–78. IEEE, 1996.

[49] Alexandre Kandalintsev and Renato Lo Cigno. A behavioral first order CPU performance model for clouds' management. In *Proceedings of 4th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*, pages 40–48. IEEE, 2012.

[50] Alexandre Kandalintsev, Renato Lo Cigno, and Dzmitry Kliazovich. *Freeze'nSense*: Estimation of performance isolation in cloud environments (submit). *Software: Practice and Experience*, 2016.

[51] Alexandre Kandalintsev, Renato Lo Cigno, Dzmitry Kliazovich, and Pascal Bouvry. Profiling cloud applications with hardware performance counters. In *Proceedings of the 28th International Conference on Information Networking (ICOIN)*, pages 52–57. IEEE, 2014.

[52] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[53] Andi Kleen. An NUMA API for linux. Technical report, 2005. Accessed: 2015-06-08.

[54] Abhishek Kumar and Jun (Jim) Xu. Sketch guided sampling - using on-line estimates of flow size for adaptive data collection. In *Proceedings of the 25th IEEE International Conference on Computer Communications (INFOCOM)*, pages 1–11, 2006.

[55] Steven R. Kunkel, Richard J. Eickemeyer, Mikko H. Lipasti, Timothy J. Mullins, B.O. Krafka, Harold Rosenberg, Steven P. Vanderwiel, Philip L. Vitale, and Larry D. Whitley. A performance methodology for commercial servers. *IBM Journal of Research and Development*, 44(6):851–872, 2000.

[56] Tirthankar Lahiri, Marie-Anne Neimat, and Steve Folkman. Oracle TimesTen: An in-memory database for enterprise applications. *IEEE Data Engineering Bulletin*, 36(2):6–13, 2013.

[57] Henrik Löf and Sverker Holmgren. affinity-on-next-touch: increasing the performance of an industrial PDE solver on a cc-NUMA system.

In *Proceedings of the 19th ACM Annual International Conference on Supercomputing (ICS)*, pages 387–392, 2005.

[58] Zoltan Majo and Thomas R. Gross. Memory system performance in a NUMA multicore multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage (SYSTOR)*, pages 1–10. ACM, 2011.

[59] Steve McKillup. Statistics explained. *An Introductory Guide for Life Scientists, Cambridge*, 2006.

[60] Eric Melski. Public versus private clouds for dev/test. `https://blog.melski.net/2010/10/27/public-versus-private-clouds-for-devtest/`, Last accessed: April 2016, 2010.

[61] Kashif Munir, Renato Lo Cigno, Pascale Primet Vicat-Blanc, and Michael Welzl. Planning data transfers in grids: a multi-service queueing approach. *Concurrency and Computation: Practice and Experience*, pages 407–422, 2011.

[62] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. DeepDive: Transparently identifying and managing performance interference in virtualized environments. In *Proceedings of the USENIX Annual Technical Conference*, pages 219–230. USENIX, 2013.

[63] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out numa. *ACM SIGARCH Computer Architecture News*, 42(1):3–18, 2014.

[64] Zhonghong Ou, Bo Pang, Yang Deng, Jukka K Nurminen, Antti Ylä-Jääski, and Pan Hui. Energy- and cost-efficiency analysis of ARM-

based clusters. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 115–123. IEEE, 2012.

[65] Jia Rao, Kun Wang, Xiaobo Zhou, and Cheng-Zhong Xu. Optimizing virtual machine scheduling in NUMA multicore systems. In *Proceedings of the 19th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 306–317, 2013.

[66] Subhash Saini, Haoqiang Jin, Robert Hood, David Barker, Piyush Mehrotra, and Rupak Biswas. The impact of hyper-threading on processor resource utilization in production applications. In *Proceedings of the 18th International Conference on High Performance Computing (HiPC)*, pages 1–10. IEEE, 2011.

[67] Alexander Sandler. SMP affinity and proper interrupt handling in linux, 2008. Accessed: 2015-06-08.

[68] D. Sarno and S. Rodriguez. Hacker attacks show vulnerability of cloud computing. *Los Angeles Times*, 17, June 2011. Accessed: 2016-03-25.

[69] Erik Saule, Kamer Kaya, and Ümit V Çatalyürek. Performance evaluation of sparse matrix multiplication kernels on Intel Xeon Phi. In *Parallel Processing and Applied Mathematics*, pages 559–570. Springer, 2014.

[70] Aliaksei Severyn and Alessandro Moschitti. Fast support vector machines for structural kernels. In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases (ECML-PKDD)*, pages 175–190. Springer, 2011.

[71] Rami Sheikh and Mazen Kharbutli. Improving cache performance by combining cost-sensitivity and locality principles in cache replacement

algorithms. In *Proceedings of the 28th IEEE International Conference on Computer Design (ICCD)*, pages 76–83, 2010.

[72] Vivek Shrivastava, Petros Zerfos, Kang-Won Lee, Hani Jamjoom, Yew-Huey Liu, and Suman Banerjee. Application-aware virtual machine migration in data centers. In *Proceedings of the 30th IEEE International Conference on Computer Communications (INFOCOM)*, pages 66–70, 2011.

[73] Xiang Song, Haibo Chen, Rong Chen, Yuanxuan Wang, and Binyu Zang. A case for scaling applications to many-core with os clustering. In *Proceedings of the 6th Conference on Computer Systems (EuroSys)*, pages 61–76. ACM, 2011.

[74] William H. Starbuck. Learning by knowledge-intensive firms. *Journal of Management Studies*, 29(6):713–740, 1992.

[75] Dylan Stark, Gabrielle Allen, Tom Goodale, Thomas Radke, and Erik Schnetter. An extensible timing infrastructure for adaptive large-scale applications. In *Proceedings of the 7th International Conference on Parallel Processing and Applied Mathematics (PPAM)*, pages 1170–1179. Springer, 2007.

[76] M. Aater Suleman, Yale N. Patt, Eric Sprangle, Anwar Rohillah, Anwar Ghuloum, and Doug Carmean. Asymmetric chip multiprocessors: Balancing hardware efficiency and programmer efficiency. Technical report, University Texas (TR-HPS-2007-001), 2007.

[77] David Tam, Reza Azimi, Livio Soares, and Michael Stumm. Managing shared l2 caches on multicore systems in software. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*, pages 26–33. Citeseer, 2007.

[78] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *Proceedings of the 2nd Conference on Computer Systems (EuroSys)*, volume 41, pages 47–58. ACM, 2007.

[79] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: Sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *Proceedings of the 2nd Conference on Computer Systems (EuroSys)*, pages 47–58. ACM, 2007.

[80] Chunqiang Tang, Malgorzata Steinder, Michael Spreitzer, and Giovanni Pacifici. A scalable application placement controller for enterprise data centers. In *Proceedings of the 16th International Conference on World Wide Web*, pages 331–340. ACM, 2007.

[81] Lingjia Tang, Jason Mars, Xiao Zhang, Robert Hagmann, Robert Hundt, and Eric Tune. Optimizing Google's warehouse scale computers: The NUMA experience. In *Proceedings of the 19th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 188–197, 2013.

[82] Joseph Torrellas, John Hennessy, and Thierry Weil. Analysis of critical architectural and programming parameters in a hierarchical. *ACM SIGMETRICS Performance Evaluation Review*, 18(1):163–172, 1990.

[83] Wei Wang, Tanima Dey, Ryan W Moore, Mahmut Aktasoglu, Bruce R Childers, Jack W Davidson, Mary Jane Irwin, Mahmut Kandemir, and Mary Lou Soffa. REEact: a customizable virtual execution manager for multicore platforms. *ACM SIGPLAN Notices*, 47(7):27–38, 2012.

[84] Vincent M. Weaver. Linux perf_event features and overhead. In *Proceedings of the 2nd International Workshop on Performance Analysis of Workload Optimized Systems (FastPath)*, pages 342–362, 2013.

[85] Vincent M. Weaver. Self-monitoring overhead of the linux perf_event performance counter interface. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 102–111, 2015.

[86] Vincent M. Weaver and Jack Dongarra. Can hardware performance counters produce expected, deterministic results? In *Proceedings of the 3rd Workshop on Functionality of Hardware Performance Monitoring*, 2010.

[87] Thomas Willhalm. Memory latencies on intel xeon processor e5-4600 and e7-4800 product families. `https://software.intel.com/en-us/blogs/2014/01/28/memory-latencies-on-intel-xeon-processor-e5-4600-and-e7-4800-pro` Last accessed: April 2016.

[88] C.S. Wong, I.K.T. Tan, R.D. Kumari, J.W. Lam, and W. Fun. Fairness and interactive performance of O(1) and CFS Linux kernel schedulers. In *Proceedings of the 3rd International Symposium on Information Technology (ITSim)*, volume 4, pages 1–8. IEEE, 2008.

[89] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the 4th USENIX Conference on Networked Systems Design Implementation (NSDI)*, pages 229–242. USENIX, 2007.

[90] Di Xu, Chenggang Wu, Pen-Chung Yew, Jianjun Li, and Zhenjiang Wang. Providing fairness on shared-memory multiprocessors via process scheduling. In *Proceedings of the 12th ACM SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems*, volume 40, pages 295–306, 2012.

[91] Rui Yang, Joseph Antony, and Alistair Rendell. Effective use of dynamic page migration on numa platforms: The gaussian chemistry code on the sunfire x4600m2 system. In *Proceedings of the 10th International Symposium on Pervasive Systems, Algorithms, and Networks (ISPAN)*, pages 63–68. IEEE, 2009.

[92] Rui Yang, Joseph Antony, and Alistair P Rendell. A simple performance model for multithreaded applications executing on non-uniform memory access computers. In *, 2009. HPCC'09.*, pages 79–86. IEEE, 2009.

[93] Cemal Yilmaz. Using hardware performance counters for fault localization. In *Proceedings of the 2nd International Conference on Advances in System Testing and Validation Lifecycle (VALID)*, pages 87–92. IEEE, 2010.

[94] Dmitrijs Zaparanuks, Milan Jovic, and Matthias Hauswirth. Accuracy of performance counter measurements. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 23–32, 2009.

[95] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI 2: CPU performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, pages 379–391, 2013.

[96] Qin Zhao, David Koh, Syed Raza, Derek Bruening, Weng-Fai Wong, and Saman Amarasinghe. Dynamic cache contention detection in multi-threaded applications. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)*, volume 46, pages 27–38, 2011.

[97] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, volume 38, pages 129–142. ACM, 2010.