Partitioning Deep Cache Hierarchies in Software for Predictable Performance

BY

ALBERTO SCOLARI
B.S., Politecnico di Milano, Milan, Italy, September 2011

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2014

Chicago, Illinois

Defense Committee:

Jon A. Solworth, Chair and Advisor
Jakob Eriksson
Marco D. Santambrogio, Politecnico di Milano

ACKNOWLEDGEMENTS

Several people made this thesis possible: they gave me help, support, suggestions, guidance, inspiration, patience, precious time, confidence and everything I needed to do this work.
The first people who gave me all this are Filippo Sironi and Davide Basilio Bartolini: their knowledge and experience guided my work, their patience and suggestions gave my work a form. With Filippo and Davide, I want to thank my advisor, Marco Domenico Santambrogio, for what he gave to me: his support, his trust in me foster my passion in our work.

I want to thank my family, whose patience helps me concentrating on my work.

A thank you also to Prof. Jon Solworth, my UIC advisor, whose lectures and comments stimulated and convinced him and me on my own passion for this field of research.

Thank you also to all the people in the NECSTLab at Politecnico di Milano, with whom I share the working time in a funny and stimulating atmosphere: helping us each other and sharing our knowledge are really a pleasure.

A thank you to the PoliMi-UIC students of Fall 2012, who shared with me such a new experience.

Finally, I want to thank Alessandro Rosina and Matteo Turri, who shared with me innumerable hours of study during these years of "Poli".

AS

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API Application Programming Interface. vii, 1

ARM Advanced RISC Machines. vii, 26

CMP Chip-MultiProcessor. vii, x, xi, 2, 3, 5–8, 15–19, 28–31, 37, 40, 42–45, 47, 48, 51, 87

CPU Central Processing Unit. vii, 4, 8–10, 13, 15, 17, 18, 20, 26, 28, 32, 36, 38, 40, 43, 45, 51, 65, 71–74, 76, 78, 80, 88, 90

DMA Direct Memory Access. vii, 20, 21, 69

DRAM Dynamic Random Access Memory. vii, 9

EAF Evicted Address Filter. vii, 34

eDRAM embedded Dynamic Random Access Memory. vii, 15

FIFO First-In, First-Out. vii, 11, 34

GNU GPLv2 GNU General Public License (version 2). vii, 44

I/O Input/Output. vii, 4, 88, 90

IPC Instructions Per Second count. vii, 8, 33, 39

IT Information Technology. vii, 1, 4

L1 Level 1. vii, 16, 40, 49, 76, 77, 80, 89

L2 Level 2. vi, vii, 16, 17, 47, 48, 64, 76, 77, 80, 82, 87

L3 Level 3. vii, 17, 64, 76

LIFO Last-In, First-Out. vii, 11

LIST OF ABBREVIATIONS (Continued)

LLC Last Level Cache. vi, vii, x, xi, 3, 6, 7, 17–19, 26, 28–30, 33, 35–40, 42–44, 46–51, 60, 63–65, 68, 74, 76–80, 82–85, 87, 89, 90

LRU Least Recently Used. vii, 11, 12, 18, 19, 31–34

LTS Long Term Support. vii, 78

MRC Miss Rate Curve. vii, 39

MTBF Mean Time Between Failures. vii, 4

NoC Network on Chip. vii

OS Operating System. vii, 13, 20, 27, 28, 31, 37, 38, 51, 89

PDF Parallel Depth First. vii, 30

PFN Page Frame Number. vii, 64

PIPT Phisically Indexed, Physically Tagged. vii, 13, 16, 20

PIVT Phisically Indexed, Virtually Tagged. vii, 14

POSIX Portable Operating System Interface for Unix. vii

QoS Quality of Service. vii, x, 2–5, 29–31, 37, 41–43

RAM Random Access Memory. vii, 8, 13, 76, 82

ROCS Run-time Operating system Cache-filtering Service. vii, 39

SLA Service Level Agreement. vii, 2–4

SMT Simultaneous MultiThreading. vii, 48, 89

SPEC Standard Performance Evaluation Corporation. vii, 78

SRAM Static Random Access Memory. vii, 9

SRC Stall Rate Curve. vii, 39, 40

LIST OF ABBREVIATIONS (Continued)

SRM Selected Region Mapping. vii, 38, 39

TLB Translation Lookaside Buffer. vii, 13, 14, 16, 20, 26, 27, 47

TLP Thread Level Parallelism. vii, 2

UCP Utility-based Cache Partitioning. vii, 33, 34

UMON Utility MONitoring. vii, 33

VIPT Virtually Indexed, Physically Tagged. vii, 14, 16

VIVT Virtually Indexed, Virtually Tagged. vii, 14

VM Virtual Machine. vii, 29, 30, 41, 45, 87–89

SUMMARY

With the advent of Chip-MultiProcessor (CMP) architectures and the spreading of cloud services, computational resources experience increasing phenomena of contention. The applications running on the different cores may interfere with each other in the usage of several hardware resources. Such conflicts exacerbate in cloud platforms, where the workload varies over time in an unpredictable way. Therefore, isolation of applications becomes a key aspect to ensure performance and Quality of Service (QoS) in cloud environments. The Last Level Cache (LLC) is one of the resources where contention is experienced most. This resource is fundamental in determining the applications performance, which is negatively affected by contention.

Page coloring is a technique that allows partitioning the LLC of current commodity processors. This technique exploits the position of data in main memory to control how they are mapped into the LLC. The evolution of CMP architectures has increased the number of cache levels, but how page coloring works on this hierarchy and on modern architectures has not been studied. This work aims at investigating the possible advantages, limitations and trade-offs that derive from the usage of page coloring on such architectures.

Cloud workloads can benefit from isolation in LLC and thus from page coloring. Moreover, cloud platforms leverage the power of modern CMPs to run cloud workloads. Therefore, our work can bring a contribution to guarantee isolation in LLC on these platforms. To implement our findings, we realized *Rainbow*, an implementation of page coloring in the Linux kernel that partitions the LLC among a user-defined subset of processes. We evaluated *Rainbow* with a

set of computational-intensive benchmarks to show its effectiveness. Finally, we discuss the achievements of this work, its limitations and possible future research.

This work is organized as follows:

- in Chapter 1 we provide a high-level view of cloud platforms and of the main issues they face to fulfill the users' needs and to manage the computational resources
- in Chapter 2 we present a more detailed description of the cache functioning and of cache hierarchies in modern CMPs. This chapter provides the useful details to proceed with the next chapters
- Chapter 3 shows the most important techniques to obtain isolation in the LLC
- Chapter 4 discusses the design of *Rainbow* in the context of the deep cache hierarchies of modern CMP architectures
- in Chapter 5 we explain how *Rainbow* is implemented, based on the findings in the previous chapter
- Chapter 6 reports the results obtained running *Rainbow* with a set of benchmarks
- in Chapter 7 we discuss the results together with the limits of our solution and with other research possibly deriving from this work.

CHAPTER 1

INTRODUCTION AND MOTIVATIONS

In the present chapter we explain the context of this research and we present the objective of our work. In particular, in Section 1.1 outlines the problem with respect to the current scenario of Information Technology (IT), which is actually experiencing a shift towards different computational models than those adopted so far. Then, Section 1.2 focuses on cloud platforms and gives an overview of its applications and of the goals that drive the management of their computational resources.

1.1 Problem description

The current scenario of computing services is becoming increasingly heterogeneous and complex. The main trend visible in the current years is the shift of the computing paradigms towards a so-called *cloud scenario* [1,2], where IT services are offered by external providers and final users access the machines only via a network interface, usually through a simplified Application Programming Interface (API) [3,4]. Heterogeneity regards on one side the diverse offer of services for multiple categories of users, from single individuals to big enterprises, and on the other side the hardware and software architectures designed to provide these services. Complexity, in turn, affects the design and the management of these architectures and naturally arises from the diversification and from the multiple goals of these platforms (security, scalability, etc.). As cloud applications differentiate, the demand and the offer of computing services

introduce the need for Service Level Agreement (SLA) between providers and customers [4,5]. These SLAs must be translated into Quality of Service (Qos) requirements. Measuring the parameters stated in these requirements is fundamental for accounting and for the management of the infrastructure itself [6].

On the other side, platform managers attempt to obtain other goals such as maximum platform utilization and low energy consumption [7]. Hence, providers' and users' goals often conflict, thus making resource provisioning for the various cloud services a trade-off between the two stakeholders. In particular, Chip-MultiProcessors (CMPs) allow resource sharing and became the standard computational resources for servers and workstations in the past years. These architectures rely on Thread Level Parallelism (TLP) to increase the performance of the whole processor, in accordance with Moore's law [8]. Since CMP architectures are the actaul and future trend [9], co-locating different tasks on the same CMP is a natural choice platform designers must take into account from the early design phases. Yet, hardware resource sharing poses several challenges that researchers and designers are currently facing to guarantee the scalability of the platform as the number of computational cores increases. In more details, actual commodity CMP architectures contain some resources which can be explicitly managed (like cores) and others whose control is exclusively left to the hardware. Furthermore, the hardware is unaware of the "big picture" of the whole machine workload and its Quality of Service (QoS) requirements, and cannot assign resources based on such requirements. Contention on shared computational resources, such as the memory controller, the on-chip interconnection and the caches limits the utilization of the infrastructure and also their performance. Contention

phenomena exacerbate as the diversity of services increases, because how the variable workload of a cloud infrastructure uses the shared resources is hardly predictable [10]. Hence, isolation of shared resources has become a fundamental aspect for efficient computing infrastructures.
This thesis focuses on the Last Level Cache (LLC), a shared resource whose effectiveness is fundamental for the performance of the whole system. Many research works have investigated contention on the LLC by solutions requiring either hardware or software modifications, so that a wide literature is currently available. But these solutions remained confined in the research environment and have not been adopted by CMP designers. On one side, hardware solutions require big investments and actually do not provide a definitive, effective solution to contention. On the other side, software solutions are often ineffective or limited to specific scenarios, and can require deep modifications in the software infrastructure.

1.2 Cloud platforms: applications and QoS

This section presents various aspects of cloud platforms that are of interest for this work. Section 1.2.1 shows the main parameters of cloud services seen from users; Section 1.2.2 shows a simple classification of cloud applications with respect to their requirements and finally Section 1.2.3 introduces the main issues due to resource sharing.

1.2.1 The user's view: SLA and QoS

SLAs define the service parameters to be guaranteed and are a part of the contract between the final user and the service provider that ensures the quality of the services provided and their cost [11,12]; these agreements specify sets of QoS parameters aimed at measuring the effective quality of the service. These parameters must be taken under control and can be general

enough to be understood and measured by non-technical users like non-IT enterprises, but can also reach a high level of detail. SLAs also prescribe the interval these parameters must be within and the penalties in case of violation. Typical examples of QoS parameters are Mean Time Between Failures (MTBF), response time, throughput. Among these parameters, some are related to the availability of the service, while others are related to the performance of the service. Performance parameters are based on the many layers and devices needed to build a cloud infrastructure: they can be related to the latency inside the provider's network, to the disks Input/Output (I/O), to the Central Processing Unit (CPU), etc.

1.2.2 Basic classification of cloud applications

With respect to QoS, applications can be classified in two broad categories: *batch* applications and *latency-sensitive* applications[13].

Batch applications do not have a continuous interaction with users and do not have strict requirements in terms of latency; their typical QoS requirement is the completion time, which must be below a threshold. An exception is Spark, a framework for machine learning[14] that can be assigned latency constraints because it must respond to interactive queries. A notorious example of batch application is MapReduce[15], but more and more applications with batch characteristics are being implemented for cloud platforms, also from the scientific world[12,16,17].

Conversely, latency-sensitive applications define other QoS parameters to be satisfied, often expressed in terms of maximum latency allowed; this latency is often related only to the cloud platform of the provider. Typical examples of applications with these requirements are user

facing applications like web search, e-mail and online gaming platforms, which can be granted very low latency if the resources usage is not too high [18].

1.2.3 The provider's view: energy cost and utilization

Other parameters characterize a cloud infrastructure from the perspective of the cloud provider, like energy consumption and utilization. These constraints conflict with QoS requirements, which are becoming increasingly important and drive the choices of big cloud providers. Barroso et al. [7] show that, for example, the utilization of data centers computing resources seldom goes above 20%. Over-provisioning of resources is a way to increase the performance: decreasing this provisioning means sharing hardware resources, but consolidation of more tasks on a single CMP can cause a slowdown of up to 40% [19]. Such a value can be intolerable for some applications and thus forces the cloud providers to avoid co-location [20].

1.3 Cloud platforms and computational resource sharing

How to share computational resources is one of the main research field in the cloud computing area. This research is motivated by different goals, such as energy, efficiency or performance.

Energy efficiency is playing an increasingly important role in the computing scenario, so that it is emerging as a novel area of active work [19,21,22]. Performance, however, has still a central role in the design and management of cloud platforms. Given the ubiquitous spreading of CMP architectures in commodity machines, sharing hardware resources through co-location is a hot research topic in the cloud community. There are several research directions to enhance the usage of cloud platforms while fulfilling QoS requirements; two in particular must be cited, which are somehow inter-related: *scheduling techniques* and *isolation techniques*.

Scheduling techniques attempt to improve resources usage by changing the schedule of applications to cores and to different sockets, so that contention phenomena are ideally minimized, but do not change the way the single applications exploit the resources.

Isolation techniques attempt instead to minimize contention by changing the usage of resources of single applications; therefore, they imply architectural modifications which are often tailored on the hardware they are deployed on.

Despite the inherent differences of these techniques, they can benefit from each other as a proper co-location can improve the isolation of applications. The design of *Rainbow*, explained in Chapter 4 is based on isolation techniques, and is specifically tailored to isolate the LLC space of applications scheduled on the same CMP.

1.4 Our proposal

Scheduling techniques have limited benefits in improving the LLC usage, because they do not avoid contention, but they basically search for an execution scenario where contention is the least possible. On the contrary, isolation techniques try to solve the problem "at the root" preventing contention.

Our proposal derives from this last family of techniques and attempts to isolate the LLC portion devoted to applications. Unlike existing works, we strongly contextualize our design in the field of the modern CMP architectures by exploring the benefits, the drawbacks and the possible trade-offs to obtain isolation in LLC . These architectures have deep layers of caches, where the LLC is usually shared among the cores and the other layers are per-core. We will explore how this affects partitioning the LLC and how this can be performed. Moreover,

modern CMPs also offer several facilities in terms of memory management that can conflict with the well-known technique we employ for partitioning (explained in Section 3.3). From these considerations, discissed in Chapter 4, we will implement *Rainbow*, a modification to the Linux kernel to isolate applications in the LLC.

CHAPTER 2

BACKGROUND

This chapter presents the background of this work, with a quick review of the CPU cache. This review starts from the reasons that caused its adoption (Section 2.1), then shows its structure (Section 2.2) and explains how more caches are organized into a hierarchy inside a CPU (Section 2.3). Then, we present a basic explanation of contention phenomena in Section 2.4.

We also present the main algorithm for page allocation in Section 2.5.

Finally, in Section 2.6 we explain how modern CMPs can manage physical memory areas with a different granularities.

2.1 Motivations of caching

From the 80's, the architectural and technological evolutions of CPUs and Random Access Memorys (RAMs) caused a diverging growth of the speed of these two components. Their speeds grew up at an exponential pace, but the coefficient of growth is different in favor of the CPU speed; therefore, the difference between the two speeds also grew up exponentially, thus creating over the years the so-called *processor-memory gap* [23]. This gap forces the CPU to wait for the completion of the memory operations and may dramatically decrease the effective performance of the CPU, causing its Instructions Per Second count (IPC) to be much less than the expected one. This happens because the CPU can spend most of its time waiting for the data. Hence, the need of overcoming this bottleneck arose and a faster memory was needed.

To build such memories, designers leverage a key feature of applications that is the principle of *access locality*, which is distinguished in *spatial locality* and *temporal locality*.

Definition 2.1. (Spatial locality) *if a (virtual) memory address n is referenced, it is likely that the address $n + 1$ will be referenced in the near future.*

Definition 2.2. (Temporal locality) *if a memory address is referenced at cycle c, it is likely that it will be referenced again at cycle $c + T$ (with T small).*

By combining these two principles, it turns out that if a block of words is referenced by the CPU, it is likely to be referenced again and multiple times in the near future. This intuition, proved by experimental evidence from the execution traces of many programs, suggested the introduction of a *CPU cache*, i.e. a fast memory that keeps the most used data "close" to the CPU.

Typically, caches are implemented with Static Random Access Memory (SRAM) technology. This technology has a greater cost in terms of area and energy with respect to the cost of main memory (typically implemented with Dynamic Random Access Memory (DRAM) technology), but permits to build memories whose access time is much smaller, thus ensuring more performance. For these reasons, the size of cache memories is a careful trade-off that takes into account many parameters.

2.2 Structure of cache

The basic granularity for data management inside the cache is the *line* (also called *block*, with possible confusion with the granularity of main memory). A cache loads (or discards) the

data in units of lines from main memory, where a cache line has a size that is bigger than a word and always a power of 2: in symbols the size is $L = 2^l$. For example, the line size in the x86 architecture is 64 B (with word of 32 or 64 bits), while in the Power architecture it is 128 B.

A line always stores contiguous words from main memory. Using cache lines bigger than the CPU word permits to leverage spatial locality, because the cache stores also the words that are adjacent to the requested one. Instead, a more complex mechanism leverages temporal locality; this mechanism determines most of the features of the cache, and is explained in Section 2.2.1. Since CPUs operate in virtual mode, the choice of which address space to use to address data for caching is a fundamental design choice: Section 2.2.2 explains the advantages and drawbacks of each choice.

Finally, Section 2.2.3 shows other types of caches that can be found on the market.

2.2.1 Design, operations and types of caches

Based on the trade-off between latency and performance, several cache models exist to leverage temporal locality. The most general one is the *n-way set-associative cache*, where n is usually an even number. The whole cache is divided into *sets*, each set containing n lines. The number of sets in a cache is 2^s, where s is the number of bits used to address the cache set. Using both line size and set number in powers of two makes it possible to use subsets of the data address bits to address the cache. In particular, the less significant l bits determine the offset inside the line of the referenced byte (they are called *line offset* bits), while the s bits after the line offset are the *set number* bits and determine the cache set. This hint avoids

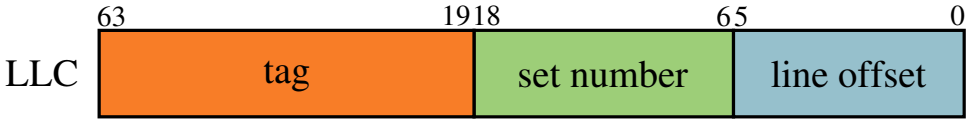


Figure 1: Bit fields of a memory address to access a cache
The physical address is divided into three fields for cache access: line offset, set number and tag.

expensive computations of cache locations and is realizable simply by re-routing the address wires.

If we assume a set to be initially empty, the first n lines are loaded in the free positions. To check whether a referenced line is already present in cache, the most significant bits of the data address are used: these bits are the *tag bits* (see Figure 1), and are used as a label to mark each line in the set. When a word is referenced, its set is obtained from the address and its tag is compared to the tags of all the lines inside the set. This comparison is done in parallel (thus justifying the name of "associative cache") and is the most expensive operation in terms of latency; the other major component in latency is the time to access the set. When a line is found, a *cache hit* happens, otherwise a *cache miss* happens.

How sets are managed when they are full is they key mechanism to leverage temporal locality. In fact, when the $n+1$-th line is inserted, the cache must discard (or *evict*) a line and load the incoming one in its place. The line to be evicted is chosen usually according to the Least Recently Used (LRU) policy. Other policies have been employed, like random, First-In, First-Out (FIFO) and Last-In, First-Out (LIFO), but are uncommon in modern architectures.

The LRU policy prescribes to evict the line that is the least recently used one. Thus, after the eviction, the set contains the new line and the $n - 1$ most used lines.

The LRU policy is implemented by adding to each cache line a field with a counter storing the number of accesses: when a line is loaded its counter is set to 0, when the line is referenced its counter is increased. If a counter overflows, it is typically left to the maximum value and all the others counters inside the set are decremented.

The choice of the parameters of the cache is a trade-off between different objectives: in principle, the number n of ways per set (the so-called *associativity*) must be the maximum possible, in order to leverage temporal locality. But a higher associativity implies a longer time for line lookup during each cache access (and this operation is on the critical path) and more area to implement the comparators. Therefore, the associativity is in practice limited to a small set of value, usually ranging from 2 to 16.

At the extreme positions of this model, there are *fully associative* caches and *direct-mapped* caches. A fully associative cache can be viewed as a particular case of an n-way set-associative cache where n is the maximum possible number given a cache size S, i.e. $n = \frac{S}{L}$: in this model the whole cache has only one set and the highest possible associativity. This model thus applies the LRU policy to the whole cache, and exploits temporal locality at best; but, because of the limitations mentioned above, this type is rare in practice. Instead, in direct-mapped caches each set contains only one line ($n = 1$) and the number of sets is the maximum possible, i.e. $\frac{S}{L}$. This model does not provide the benefits of temporal locality, but is easy to implement as the cache line coincides with the set and can be determined just by the set bits, without any

comparator or counter, thus with a minimum latency. Caches of this type were used in the past years for small, fast caches at the lowest levels of the memory hierarchy (see Section 2.3).

2.2.2 Cache addressing

During the cache access, two fields are used for the line look up: the set number and the line tag. Both these fields are taken from the memory address of the referenced word. Modern CPUs provide two address spaces for data, the virtual address and the physical address. The physical address is the one used by the CPU to load data from RAM. The physical address space is unique and is usually managed by the Operating System (OS) only. Instead, applications use virtual addresses to reference memory, and cannot use physical addresses. Each application has a separate virtual address space, which is mapped to the physical address by the OS in a way that is transparent to applications. Therefore, the same virtual address in the context of two applications can reference different physical locations in RAM. The translation from a referenced virtual address to the corresponding physical one is performed at runtime by the Translation Lookaside Buffer (TLB); this component resides in the CPU and its performance is fundamental.

Both address spaces can be used for cache addressing, with advantages and drawbacks. Moreover, as the set number and the tag are needed for different purposes, one address space can be used for the set and the other for the tag.

A cache can be Phisically Indexed, Physically Tagged (PIPT), with the advantage that both the set number and the tag are unique for all the addresses referenced, as all the addresses come from the same address space. The drawback of this model is that the TLB translation is needed

before accessing the cache, thus increasing the latency. However, most of the modern caches are implemented in this way.

Instead, a Virtually Indexed, Virtually Tagged (VIVT) cache uses the virtual address for both the set number and the tag, with no latency penalty. Caches of this type suffer from two main problems, collectively called *aliasing*. In case a used virtual address is mapped to another physical address, the data in cache become inconsistent (*homonyms problem*). Mapping the same virtual address to different physical ones happens very frequently, as each application has its own virtual address space. Conversely, different virtual addresses can refer to the same physical locations (for example to share data, as with inter-process communication mechanisms), thus creating potentially non-coherent copies of the same data with also wasting space. Possible solutions to overcome these problems are flushing the whole cache at each context switch, which is extremely expensive with modern big caches, or augmenting the tag with an address space identifier that is unique to each virtual address space. But these solutions usually have a high cost, and VIVT caches are rare.

Phisically Indexed, Virtually Tagged (PIVT) caches suffer from the same problems of VIVT caches and in addition require the TLB translation to access the set, and are not used in practice.

Finally, Virtually Indexed, Physically Tagged (VIPT) caches can access the set in parallel with the TLB translation and do not suffer from the homonyms problem, but can still have synonyms, which is an undesirable situation. Due to their small latency, VIPT caches are

currently used for the lowest level of the cache hierarchy, where access speed is fundamental (see Section 2.3).

2.2.3 Other types of cache

Other types of cache exist. A first type is the *victim cache*, which holds the lines evicted from the main cache; since this cache is accessed only after looking up the data in other caches, it can have a higher penalty and thus a higher associativity. This cache serves as an added layer between the normal cache and the main memory, can have a huge size (from 32 to 128 MB) and can be implemented with a technology having a smaller cost in terms of area and energy, like embedded Dynamic Random Access Memory (eDRAM).

Caches can be specialized to store only some type of data, for example those needed in particular operations. *Trace caches*, for example, store small traces of instructions that are executed often. Similarly, a *micro-operation cache* stores instructions that have already been partially decoded: these caches are typical of the modern x86 architectures, where the variable-length instructions are pre-decoded into fixed-length micro-instructions before entering the CPU pipeline. To speedup the slow decoding phase of variable-length instructions, groups of corresponding micro-instructions are stored inside such a cache and looked up before proceeding with a new translation, thus reducing the bottleneck of decoding.

2.3 Cache hierarchy

Designing a cache is always a trade-off between latency, area and performance, indicated in terms of number of hits over the number of accesses (*hit ratio*). With the increment of the CPU frequency that occurred in the early 2000s and later with the advent of CMPs, the

role of the cache has become increasingly important in leveraging the power provided by the computational resources. Therefore the cache had to scale-up with the rest of the architecture, without causing bottleneck effects when the cores load data from memory. Since the latency of a single cache is determined by the number of sets and the associativity (see Section 2.2.1), even with the last technological enhancements it is not possible to build a unique cache with high size, high associativity and small latency. Thus, designers decided to organize different caches into a *hierarchy* of layers, where each layer has size and latency greater than the previous one.

Level 1 (L1) caches are the first ones to be accessed for lookup and are very small, typically 32 or 64 KB, and usually divided into a *data cache* and an *instruction cache* (realizing a so-called *Harvard architecture*). The first one contains only the applications data, which often exhibit higher locality than instructions; to leverage this fact, data caches can have higher associativity than instruction caches (for example, 8 against 4). In order to limit latency (around 2 cycles), the L1 cache has small size and employs a VIPT addressing scheme, so that no homonyms issue occurs and the set access is performed immediately. In CMP architectures, L1 caches are per-core, in order to provide to the core datapath a dedicated memory and avoid contention with other cores.

In case of cache miss in the L1 layer, the subsequent Level 2 (L2) cache is accessed, which usually is bigger (from 256 KB to 8 MB) and has higher associativity (around 8), but also higher latency (from 5 to 12 cycles). This layer of the hierarchy employs the PIPT scheme as the TLB translation is performed in parallel with the L1 access, so that no overhead is required to handle aliasing. If this cache is the last layer in the hierarchy, it is shared among more cores

(typically all, but sometimes only between two, so that two separate slices of L2 cache may exist), otherwise it is per-core.

Modern CMPs have an upper Level 3 (L3) cache with even greater associativity (from 12 to 16), size (from 8 to 32 MB) and latency (from 10 to 20 cycles) and present thus a high hit ratio. This level is usually the LLC, and is connected to the main memory via a controller (with 100 cycles or more of latency); it is usually shared by all the cores, but may rarely be divided into different, independent slices shared between couples of cores.

Because of the increasing number of cores and of the increasing LLC size, this last layer is split into more slices (or tiles) that are interconnected with the cores. Therefore, the cores may experience contention on the intercommunication while loading data from the LLC.

Finally, CPUs can have other specialized caches to enhance some stages of the datapath (like a micro-instruction cache) or an upper victim cache.

2.4 Contention on a Shared Cache

To understand the nature of this work and of those presented in the next chapter, we quickly review how contention can arise on shared caches. Given the considerations on modern CMPs presented in the previous section, we assume that only the LLC is shared and we focus on this level; but the same considerations apply as well to other shared layers. Since the LLC is shared by multiple cores (usually all), the different applications executing simultaneously may be slowed down when conflicting for this resource. Lower layers indeed are more isolated as they are per-core, and do not cause bottlenecks.

Measuring and modelling contention on shared caches is the objective of previous work [24], which models the sensitivity of an application to cache sharing and its impact on other applications.

Focusing on the LLC, contention is a phenomenon well known in the literature. It is due to several factors, both hardware and software.

On the hardware side, the main factor is the actual design of caches, where the mapping of the data from main memory depends on the memory address. Assuming a physically mapped cache, some of the bits of the memory address determine the cache set the referenced data are mapped to; if two applications are mapped to memory addresses with these bits being equal, their data will be mapped to the same cache set and contention may occur.

Commodity CMP architectures lack interfaces to control the cache mapping. Instead, other architectures provide simple interfaces for this purpose [25], but on CMPs currently found in servers and workstations such interfaces are not provided nor planned for the future (an exception is the CPU used by Cook et al. [26], but this modified processor is not available on the market).

Another hardware source of LLC contention is the LRU policy employed to choose the cache lines to be evicted. The LRU policy is based on the past behavior of the application: thus, with a single application, it is effective in leveraging temporal locality. With the advent of CMP architectures, a shared cache is accessed by multiple applications: hence, a set may respond to requests coming from several applications, thus mixing lines referenced with different access patterns. To optimize the data placement, it would be necessary to predict the locality of the lines. However, the LRU policy is unable to predict the future usage of lines already

in cache and thus may replace them with others having lower locality; in this case, the policy goes against the final objective of the cache itself. Non-optimal data fetching is exacerbated by cache prefetchers, whose aggressive behavior can load useless data evicting other applications lines. Furthermore, some applications have a working set bigger than the LLC: even with very high locality, two applications may conflict because of the limited LLC size.

On the software side, the basic factor that exacerbates LLC conflicts is the poor locality of co-located applications, which causes the LRU policy to be ineffective. In addition, different execution phases of the same application may change the temporal locality of the data accesses and determine cache conflicts that change in time. Another factor is the OS layer, which schedules applications without taking into account contention.

Apart from its cause, cache contention belongs to two categories: *thrashing* and *pollution*. Thrashing[27] refers to lines with high locality being evicted by other lines with high locality, because both sets of lines are mapped to the same cache set, which experiences a high memory pressure that causes the continuous eviction and reloading of data. These frequent transfers to and from memory stress the memory controller, which becomes the performance bottleneck. Thrashing is typical of co-scheduled (i.e., running at the same time on cores of the same CMP) memory-intensive applications with good locality and it is hardly predictable as it depends on the physical location of data in main memory, in turn depending on the OS, the workload, etc. Pollution refers to the eviction of lines with low locality due to the insertion of other lines with low locality: since the LRU policy is unable to predict the future usage of inserted lines, it must evict more useful ones to make room for the formers. The evicted line, assuming they

have higher locality, will be reloaded to the cache, causing contention on cache sets and on the memory controller.

2.5 Buddy memory allocator

Since modern shared caches are PIPT, the cache set where data are placed depends on their physical address. In turn, physical addresses depend on a component of the OS called *physical memory allocator*, which manages the physical memory of the machine. When physical memory is needed, various subsystems of the kernel (the page fault handler, Direct Memory Access (DMA) drivers, etc.) issue a request to the physical memory allocator, which allocates a contiguous physical area of at least the requested size.

Physical areas must be managed at a minimum granularity that is determined by the CPU architecture. Physical areas of this minimal size are called *physical pages*, while the areas with the same size in virtual memory are called *virtual pages*. Both physical and virtual pages are memory-aligned, so that the address of a byte inside a page can be split into two bit fields: the most significant bits refer to *page address*, which identifies the page, and the less significant bits refer to the *page offset*, which identifies the referenced byte.

Mapping the addresses of virtual pages to those of physical pages is the task of the TLB. Because of the memory-alignment of pages, the page address to be translated is found by re-routing the most important bits; once this virtual address is translated into a physical address, it is re-padded with the offset bits. Commodity CPUs have a page size of 4 KB, thus with 12 bits of page offset and 52 bits of page address (in 64 bit architectures). Modern CPUs also support other page sizes, as discussed in Section 2.6.

A physical memory allocator is designed with several goals in mind:

- the first goal is *efficiency*: when a page is requested, it must be returned as fast as possible
- *scalability* is also important, as machines contain very different amounts of memory (from few MB to hundreds of GB) that the allocator must manage efficiently
- another goal is keeping *internal fragmentation* low: internal fragmentation is the waste of memory because of the granularity of allocation (depending on the page size)
- also *external fragmentation* must be limited: external fragmentation refers to the unavailability of large contiguous memory areas due to the interleaving between free and used areas

Internal fragmentation is high when the allocated areas have size very different from the requested amount. Allocators try to keep this phenomenon low by reserving a space that is the closest possible to the one requested. However, the granularity of the page imposed by the hardware inherently causes some internal fragmentation, which, for requests smaller than the page size, is handled by the upper layers (like the applications libraries).

External fragmentation plays a more important role in modern computers, where a large amount of memory is often available. This fragmentation prevents the allocation of large memory areas to sub-systems that need it (like DMA drivers). External fragmentation may cause performance degradation of the allocator because it must manage many small memory fragments in its own data structures.

In Linux, the heart of the physical memory allocator is the Buddy algorithm [28]. It is known in the literature from almost 40 years and is widely used because of its features. This algorithm is very efficient, and is effective in limiting external fragmentation. Moreover, it has shown to scale well on a wide range of machines equipped with very different memory amounts [29]. The following sections show the functioning of this algorithm with reference to the Linux kernel: in particular, Section 2.5.2 explains the data structure the algorithm is based on, while Section 2.5.2 explains how this data structure is leveraged to perform the typical operations of the Buddy algorithm.

2.5.1 Buddy data structure

The main data structure of the Buddy algorithm is an array of doubly-linked lists, as depicted in Figure 2; each list links groups of physically contiguous pages. A single group is called *buddy*, and has a certain size that is defined by a parameter called *order*: a buddy of order i is composed of 2^i physically contiguous pages. To represent a buddy of order i, the simplest way is using the page address of its first page as identifier, called *buddy address*; like memory pages, also the buddies are memory-aligned, so that the less significant i bits of the buddy address are set to 0. The Linux kernel uses this representation, which makes it easy to get the buddy address from the address of any of its pages, given the order.

The order of the buddy determines the list it is stored in, so that the proper list can be accessed in constant time via an addition to the array base pointer. Inside each list, the buddies are doubly linked in order to iterate easily among them. When a page is released, it is added to

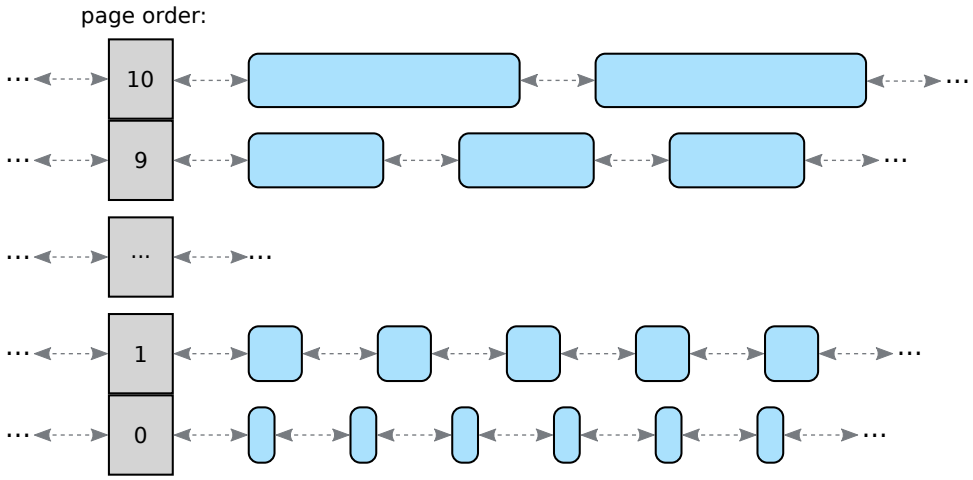


Figure 2: Lists of free areas of the Buddy allocator
Different groups of contiguous physical pages (*buddies*) are stored in different lists on the basis of their *order*.

the head of the list, in order for a page insertion to be performed in constant time. Similarly, a page removal can be done in constant time by picking the head of the list.

2.5.2 Buddy algorithm

The Buddy allocator responds to memory requests by allocating buddies of a certain order. To fulfill a memory request, the Buddy allocator allocates the smallest buddy whose size is greater than or equal to the requested amount. Allocating the least amount of physical space (compatible with the buddies sizes) is a way to limit internal fragmentation, though this objective is partially left to higher levels (e.g. kernel/userspace allocation calls like `malloc()`).

Since the size of a buddy is a power of 2 and is determined by its order n, the Buddy algorithm uses n to access the list of all buddies of order n, and picks the buddy on top of the list. If no buddy is present, the allocator splits a buddy at order $n + 1$ in two buddies of order

n, stores one of the two halves in the free buddies list of order n, and finally returns the other. To split a higher order buddy, the allocator first checks the list of order $n+1$ not to be empty. If no buddy is present at order $n+1$, the list of order $n+2$ is accessed, and so on until a buddy is found. The algorithm splits this buddy in two halves, adds one of them to the free list of the lower order and further splits the other half; this splitting process is iterated until a buddy of size n is obtained.

Each buddy of order n is uniquely identified by the physical address of its first page, whose n less significant bits are 0 because of the memory-alignment of buddies. Hence, to find the two halves of a buddy it sufficient to consider only the n-th less significant bit of its address: if this bit remains 0, the address identifies the first half, and if it is set to 1 the obtained address identifies the upper half. Therefore, splitting a buddy can be done in constant time.

In order to limit external fragmentation and maintain scalability, in case of buddy insertion the buddy algorithm attempts to re-group buddies into bigger buddies. When a buddy of order n and address a_n is freed, the algorithm checks whether the "twin buddy" is free too, i.e. whether the buddy with the n-th less significant bit inverted ($a_n \bigoplus 2^n$) is free. If this last buddy is not free, the freed buddy is inserted into the list of order n and the algorithm terminates. In the other case, depicted in Figure 3, the two twin buddies are removed from the free list of order n and coalesced into a single one of order $n+1$ and address $a_{n+1} = a_n \wedge (a_n \bigoplus 2^n)$; this bigger buddy must be inserted in the list of order $n+1$. Before the insertion, the algorithm

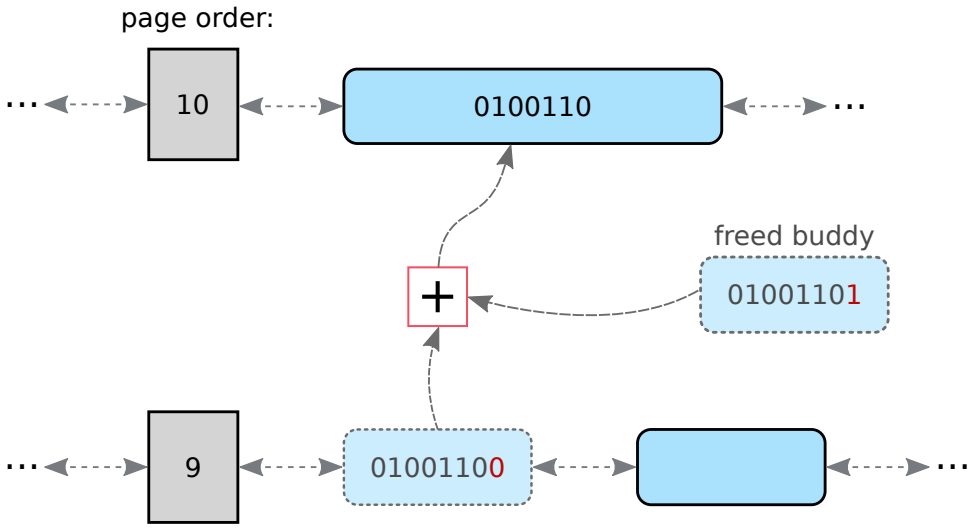


Figure 3: Coalescing of two buddies
After a buddy of order n is freed, it is coalesced with the "twin" buddy and the resulting $n+1$ buddy is placed in the proper list; the smaller buddies differ only in the least significant bit of their address; the coalesced buddy does not consider this bit.

again checks whether the twin buddy of order $n+1$ is free too, and the same procedure is applied.

In Linux, to look for the state of a buddy (free or allocated) there is no need to scan a list as the data structures to manage physical pages are indeed in an array in kernel space. All the physical pages are indeed described by the elements of a long array; these elements have a special data structure that describes the main features of a page (virtual address it is mapped to, swap information, usage, etc.), including whether it is the first page of a buddy: in case it is, also the order of the buddy is stored. Therefore, all the operations of accessing a buddy take constant time: in fact, to access the first page of a buddy of order n and address a_n, it

is sufficient to add $S \times a_n$ to the base pointer of the pages array, where S is the size of the data structure describing a physical page. Hence, the operations of splitting and coalescing described above take constant time.

To limit external fragmentation by leveraging the coalescing of buddies, Linux applies a small optimization: after a buddy split, the allocator always returns the lower half, so that its counterpart is stored in the free list and available for a subsequent allocation, without the need of splitting another buddy. Always using the same half has been shown to be more efficient and to decrease fragmentation, particularly with kernel drivers of fast devices that typically need large contiguous buffers[30].

2.6 Hugepages

Section 2.5 shows how the kernel allocator manages physical areas of different sizes; yet, the mapping between physical areas and virtual areas is then managed by the TLB at the granularity of the page, whose size is usually 4 KB. Therefore, a large memory area, even if physically and virtually contiguous, results in many mappings stored into the TLB. This increases the probability of TLB miss, penalizing applications with a big memory footprint. To overcome this limitation, modern TLBs can manage memory at a higher granularity. This feature can conflict with the technique we employ to partition the LLC, as Section 4.4.2 shows. In this section, we give a quick overview of this feature of modern CPUs in order to present the context of Section 4.4.2.

As said, modern CPUs allow bigger page sizes: for example, the x86 architecture allows page sizes of 2 MB and of 4 MB, Advanced RISC Machines (ARM) allows pages of 2 MB

and 16 MB and the IBM Power architecture allows pages of 64 KB, 16 MB and even 16 GB. Pages of those sizes are called *hugepages* in the Linux terminology and *large pages* in Windows terminology; in this work, we will refer to them as hugepages.

To exploit hugepages, the kernel must allocate a contiguous physical area of the same size, which is mapped into a corresponding virtual area. Also these pages must be memory-aligned, for the same reasons explained for 4KB pages.

Because of the size of hugepages, their offset field is longer than the that of 4 KB pages: while 4 KB pages have 12 bits of offset field, 2 MB pages have 21 bits of offset, 4 MB pages have 22 offset bits and 16 GB pages have 34 offset bits. This longer size of the offset field will introduce some problems, as anticipated above.

Hugepages support is actually limited both in Linux and Windows OSs, as it requires deep modifications to the subsystems that manage physical memory, physical-to-virtual mapping and the TLB. In Linux, for example, the use of hugepages for memory allocation must be explicitly configured by the system administrator. Nonetheless, the research community is investigating the benefits and drawbacks of this possibility, and some results have already been presented [31,32].

CHAPTER 3

STATE OF THE ART

In this chapter, the work forming the state of the art for this thesis is presented, with an emphasis on cloud platforms. The objective of the techniques presented here is enhancing the usage of computational resources, in particular of the LLC.

Section 3.1 quickly reviews the main scheduling techniques that improve the usage of the CPU. The following sections are more specifically focused on the techniques aimed at reducing contention over the LLC. These works span from new hardware design models for caches to modifications of the OS layer and explore a wide variety of solutions.

Several techniques address contention on the LLC. These techniques can be divided into *hardware* and *software* techniques. Section 3.2 gives a quick overview of the main hardware techniques in order to show the main research lines of the community, whose efforts show the importance of the problem in current and future CMP architectures. Section 3.3 reviews several software techniques, which have a smaller impact on existing architectures and are thus more easily implementable. Our proposal, discussed in the next chapter, is indeed based on software techniques, with a focus on modern CMP architectures.

3.1 Performance-aware scheduling techniques

The aim of scheduling techniques is trying to minimize contention by "re-arranging" the running applications on the cores; to guide re-scheduling, these techniques need a measure of

LLC conflicts or of applications performance, which allows them to predict how applications will behave when co-located.

The impact of co-scheduling different applications on the same CMP has been studied in prior work[20,24,33]. The basic idea behind these works is to characterize how applications behave when co-located: typically, batch applications have a big working set that stresses the LLC and benefits from more cache space, with a good locality. In contrast, latency-sensitive applications have a smaller working set and run with very low resource usage for most of the time. These applications have sudden bursts in resource usage caused by the interaction with the users; these applications, which may have QoS requirements, usually suffer from co-location with other applications, of both types. Batch applications, in fact, tend to have a continuous usage of resources that constrains the remaining amount available to latency-sensitive applications. These, in turn, cannot be co-located with others of the same type because they create a high contention over resources in case of simultaneous bursts. Data centers actually solve this problem by scheduling batch applications in co-location and latency-sensitive applications on dedicated CMPs, thus potentially under-utilizing resources. Given this situation, different policies and techniques are employed to overcome these limitations. In particular, cloud platforms present specific scenarios, like Virtual Machines (VMs) or multi-thread workloads, whose peculiarities affect the scheduling policy.

Tang et al.[10] perform an in-depth study over the performance of some cloud-typical applications, showing the importance of co-scheduling based on resources usage. This work measures

the key parameters (like LLC miss rate) of the running applications and the proposed heuristics attempt to maximize the benefit from sharing, for example by co-locating threads that share a lot of memory or by avoiding the co-location of applications with high LLC footprint or with high memory bus usage.

Mars et al.[20] attempt to predict performance degradation by adding a controllable pressure to the memory subsystem in order to measure the sensitivity of each application to the LLC and the sharing of memory controller . The results found are then used to devise a co-location scheme for latency-sensitive applications, whose QoS requirements are satisfied in almost any case.

Similarly, Kang et al.[34] exploit both hardware information and application-specific performance metrics to optimize the co-location of threads on a many-core architecture (the Tilera64 processor[35]).

Gong et al.[36] do a high level modeling of resource usage through PAC, a monitor of running applications that tracks the way applications use resources. Its design is based on the existence of repeated patterns in resources usage. Implemented in a cloud environment, PAC monitors the performance of VMs and achieves a good prediction of the future resource demand of VMs with respect to prior approaches. Based on this information, the hypervisor can choose a schedule of VMs that minimizes resource contention and improves performance.

Finally, Chen et al.[37] propose to leverage multi-thread workloads to enhance the sharing of cache space among threads, thus increasing the usage of the CMP. They evaluate a scheduling policy named Parallel Depth First (PDF) that tries to mimic how a sequential application would

execute its own tasks, which have been parallelized: these tasks, when scheduled on the same core, tend to access the same data, which are already inside the cache. Therefore, more cores and cache space remain available to other, independent tasks.

3.2 Hardware techniques

The work to improve the cache usage and decrease conflicts due to sharing proposes different solutions, which investigate two aspects: the *partitioning technique* employed to partition the cache, which is implemented in hardware, and the *partitioning policy*, which can be implemented in hardware or software. Section 3.2.1 explains the main partitioning techniques, which refer to the way the partitioning mechanism is designed and implemented. Section 3.2.2 shows the policies, i.e. the metrics that are used to guide the partitioning mechanism and decide the size of each partition.

Implementing a partitioning technique in hardware reduces the overhead of the mechanism, which can be easily exploited by the cache controller (in case of a hardware partitioning policy) of by the OS. Instead, whether to implement the policy in hardware or software is a more disputable choice: a hardware-only implementation has lower latency and is transparent to the software layer, but a software implementation (usually inside the OS) can consider also high level goals like QoS requirements to determine the partitions.

3.2.1 Partitioning techniques

Partitioning techniques have very different design models: some change the actual implementation of the LRU algorithm, which has been identified as one of the sources of conflicts in CMP architectures (see Section 2.4), while others have a more disruptive approach, and

re-design the cache structure.

A technique for partitioning is called *way partitioning*, in which each core can access a configurable subset of cache lines for each set (also overlapping with the sets of other cores). In case of eviction, the LRU policy works only on the cache lines assigned to the core, so each set is effectively partitioned among the cores. This technique is used in some special-purpose architectures like Octeon[25], or is implemented in special prototypes of commodity CPUs[26]. The main drawback is the decrease in associativity: if a non-overlapping way-partitioning is performed, each core accesses only a subset of the lines, thus operating with a smaller associativity. For example, if a core is given only one line, that core "sees" a direct-mapped cache. Since associativity is a key feature to leverage temporal locality, decreasing it can be a counter-productive solution.

Instead, other techniques make the LRU policy work on per-set subsets of cache lines. To perform this, they associate each line to the core that loads it by storing this information too; when an insertion occurs, the set is determined from the incoming line as usual, a core is selected according to a certain policy and the LRU policy works only on the set lines that are associated to that core. In this way, it is possible to "move" cache lines from a core to another core, according to a certain metric.

Another technique consists in changing the LRU value of the line to be inserted. As the future evictions from the same cache set are performed according to this value, initialize the line with a higher value than usual potentially increases its persistence inside the set, i.e. it makes that line more unlikely to be evicted. This technique must be combined with a policy

able to predict the future of the line, so that temporal locality is exploited at best over all the running applications, even with different access patterns.

3.2.2 Partitioning policies

Partitioning policies state which parameters to use and how to compute them in order to decide the partitioning. These policies prescribe how to measure the performance of each configuration of partitions (i.e. the cache space devoted to each core) and try to maximize this performance, with an exact or heuristic approach.

Sharifi at al.[38] considers the scenario of a multi-thread application, whose threads are assumed to have similar characteristics: to balance the performance of the threads, the work penalizes the core with highest IPC (the *victim* core) in favor of the others. This is done by evicting some of the cache lines the victim loaded previously. When a thread loads a line into a full set, the IPC of all the cores that loaded lines inside that set is compared, and the core with maximum IPC becomes the victim. Then, the LRU policy is performed only on the victim's lines, and the new line is allocated.

Instead, Utility-based Cache Partitioning (UCP)[39] explicitly partitions the cache space by *way-partitioning*. UCP computes how many lines each core needs in order to maximize the global performance. Therefore, UCP computes the *utility* of each configuration, i.e. how much an application benefits in terms of LLC misses going from the previous to the actual partition size. To collect this measure, it introduces a novel circuit called Utility MONitoring (UMON) which monitors the utility of each application. Then, based on the recent measurements, UCP

minimizes the total number of misses assigning to each core a certain number of cache ways (at least one way to each core).

Seshadri et al. explicitly address thrashing and pollution [40]. They employ a FIFO structure called Evicted Address Filter (EAF), that is added to the cache to capture the frequency of recently evicted lines. The aim of this structure is to predict whether evicted cache lines are likely to be referenced again in the near future. The EAF helps overcoming the limitation of the LRU policy to predict the future usage of lines, above all with mixed access patterns. The EAF indeed stores the addresses of the last evicted lines and, in case one of them is re-inserted, its LRU priority is increased; this decreases pollution, because data with good enough locality is likely to be "captured" by the EAF, and thus to be kept in cache. To decrease the area overhead of the EAF, it is implemented with a Bloom filter [41], a hash-based memory structure that needs to be periodically cleared; thanks to this periodic clearing, in case of thrashing only few lines are predicted to have high re-use, thus being kept in cache; this fact increases the hit rate and limits thrashing.

With a completely different vision, *Vantage* [42] is based on the z-cache model [43], which maps the incoming line by a tunable hash function whose parameters can be changed to control the mapping. Vantage explicitly addresses the problem of partitioning by the novel structure it is based on. By exploiting the concept of utility of UCP, Vantage selects at runtime a new partition size for each core and resizes the existing partitions accordingly. The novel idea behind Vantage is to use an unmanaged area (i.e. not assigned to any core) of the cache for resizing partitions. When a partition is modified, the lines belonging to the old core are swapped

to the unmanaged area as soon as the new core holding the partition inserts its new data. Moreover, Vantage lets partitions outgrow into the unmanaged area to avoid that, in case of many partitions, conflicts among them arise in the unmanaged area (which is shared). The advantage of this approach is that it is able to react to applications changing phases faster than actual caches. Yet, these advantages come at the cost of a complete redesign of the CPU cache.

3.3 Page coloring and software techniques

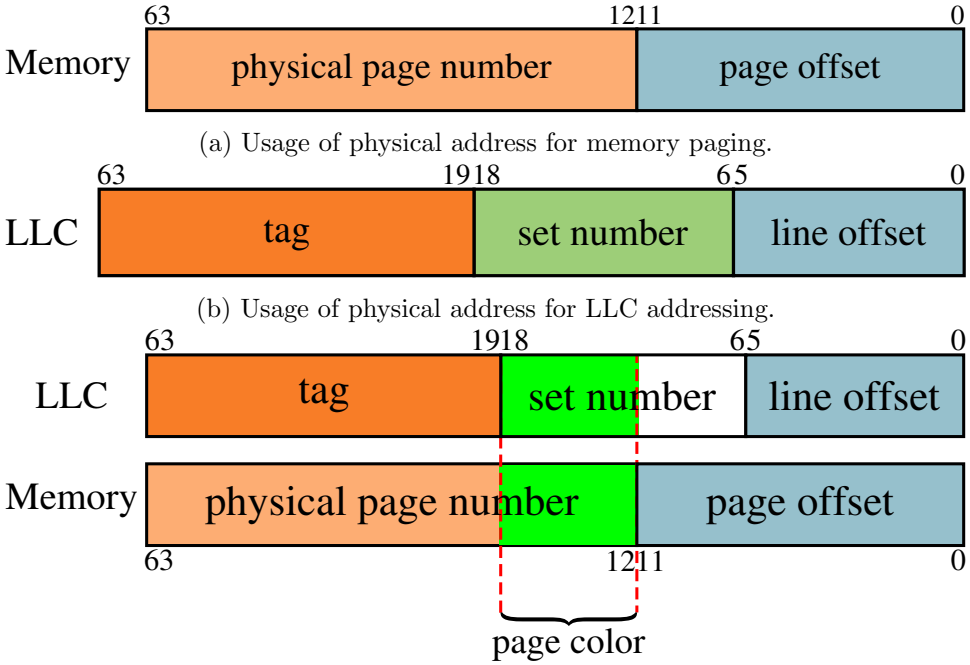
Software techniques are of high interest as they usually involve only the OS layer and can thus be deployed on real machines without changing the application layer or the hardware infrastructure.

The mechanism at the basis of software techniques is *page coloring*, which depends on the way modern caches are addressed.

Modern architectures use the physical memory address to map data into the LLC. Figure 4 shows the usage of the physical address in main memory and in the LLC: the parameters depicted in this figure are related to a real CPU, namely an Intel® Xeon™ W3540, where the LLC is the third layer of cache.

Figure 4a shows the two bit fields to manage memory pages: the lowest p bits are the page offset, while the highest bits are the page address. Similarly, Figure 4b shows how the physical address of data is used to determine the cache location: the l less significant bits contain the line offset, the upper s bits contain the number of set to look in and the highest t bits contain the tag.

Looking at Figure 4c, we assume a page size of $P = 2^p$ bytes and line size of $L = 2^l$ bytes. To



(a) Usage of physical address for memory paging.

(b) Usage of physical address for LLC addressing.

(c) Overlap of bit fields and page coloring

Figure 4: Various bit fields of a physical memory address.

control where data are stored inside the LLC, the key idea is to control their physical address; between its two fields, only the page number can be chosen by software, because the offset refers to the single referenced byte inside the page, and its length is due to the page size. Therefore, the minimum granularity at which the cache can be controlled is $\max_{(L,P)}$. This value equals P in modern CPUs.

Yet, not all of the bits of the page address determine the LLC placement. Among the page address bits, only the ones that overlap with the cache set bits really affect the placement of data; in fact, the higher bits are used as cache tag. Looking at Figure 4c, this limited overlap

restricts the number of bits available to control the LLC mapping: the number of these bits is in fact $l + s - \max_{(l,p)} = l + s - p = c$. In the example architecture of Figure 4, $c = 7$. These bits are called *color bits*, while a configuration of them is called *page color* or simply *color*.

It is important to note that a color may correspond to multiple cache sets because of the minimal granularity of the page size. For example, in the configuration shown in Figure 4c a color spans $2^{s-c} = 2^{p-l}$ sets. This is the minimum amount of cache sets that can be allocated: always referring to in Figure 4c, these sets correspond to $2^{13-7} \times 16 \times 64B = 64KB$, being 16 the associativity and 64 B the cache line size.

Page coloring consists essentially in leveraging the c color bits of the page address to control the LLC data mapping. Since the OS has complete control over the applications data in physical memory, it can obtain a LLC partitioning via page coloring.

This technique and its benefits are well-known in the literature from years [44,45]. Yet, the advent of CMPs architectures exacerbated the cache contention explained in Section 2.4, thus enforcing the need of applications isolation. Given the lack of definitive hardware solutions, this need makes page coloring a always-hot research topic. Moreover, in the recent years the spreading of cloud services with QoS requirements further increase the interest in it.

The main drawback of page coloring is its static nature, as the partitions must be determined a priori. Re-partitioning indeed is very expensive, because it consists in changing the physical address of a page. The only way to perform such operation is to copy the original page into the new physical location (*re-coloring*). This operation, even with modern hardware, takes a time in the order of magnitude of a microsecond, which is a very long interval compared to the

CPU cycle time. During this time interval, the new page is not accessible, and the application that uses it could be locked waiting for the copy operation (for example if it tries to write inside the page).

However, some solutions exist that allow changing the partitioning. These solutions often exploit hardware performance counters available in modern processors to measure the applications performance and devise a better partitioning according to a certain metric; but they need careful design to limit the overhead and in particular to minimize re-coloring.

Some of the work based on page coloring is reviewed in the next sections. In particular, Section 3.3.1 shows the usage of page coloring to limit the occurrence of cache pollution, while Section 3.3.2 presents an extensive usage of page coloring to realize LLC partitioning in software.

3.3.1 Anti-pollution techniques

One main source of LLC contention are OS data buffers: these memory areas contain data loaded from devices and exhibit a sequential access with an inherently low locality. These memory areas can occupy a considerable portion of the main memory (around 60%[46]) and can pollute the cache space of applications. Limiting the colors assigned to buffer pages is the key idea behind the techniques addressing this issue.

JongWon et al.[47], for example, adopt a static policy using a fixed number of colors.

With a more flexible approach, Xiaoning at al.[46] propose the usage of a Selected Region Mapping (SRM)-buffer. SRM-buffer maps buffer pages which are likely to be accessed sequentially to a set of physical pages of the same color (called *sequence*). To predict a sequential access pattern, SRM-buffer employs two heuristics: one marks as sequential the pages mapping

the same file in memory, while the other tracks the applications' accesses. To keep pollution of buffers low even when they increase in size, on a page miss due to a buffer growth SRM-buffer detects the color of the sequence which was being accessed and chooses pages of the same color, so that new data are mapped only into the LLC sets of previous data.

Instead, Run-time Operating system Cache-filtering Service (ROCS) [48] addresses the problem of pollution focusing on single applications. ROCS tracks the usage of pages and maps to a small area named *pollute buffer* those which exhibit a bad cache behavior, i.e. those which are more likely to compete for cache space without any benefit. Thus, the address space of each application is "re-organized" to prevent its buffer from polluting the cache. To identify the polluting pages, ROCS exploits hardware monitoring interfaces to sample the miss rate of the application when accessing each page; then, it uses this value to classify pages and decide which one to re-map to the pollute buffer. Re-mapping is done by re-coloring, but to avoid excessive data moving ROCS tracks the application IPC and saves the best configuration found.

3.3.2 Partitioning techniques

In this section, we present an overview of the main policies to partition the LLC through page coloring. These policies perform an effective partitioning of the LLC, and may employ both static approaches and dynamic ones.

Tam et al. [49] use a static approach to partition the LLC cache of a dual-core processor in order to increase the global performance of the workload. They identify the effects of contention and build application cache profiles to guide the partitioning mechanism. To do this, they exploit two curves, the Miss Rate Curve (MRC) and the Stall Rate Curve (SRC). The MRC

curve shows the miss rate with respect to the cache size devoted to the application, and is a measure of how the applications exploits the LLC. The SRC curve shows the stall rate due to the instructions retired to the L1 cache with respect to the cache size, thus taking into account also the latency of memory operations. This last curve in particular is proved to be more effective in guiding the partitioning because it takes into account also the memory latency, which varies above all in presence of a victim cache (like in the CPU used for the work). Based on the measured curves, a partitioning is determined that reduces the slowdown of co-located applications. Furthermore, the work provides interesting hints on the behavior of applications, like the different patterns of LLC usage.

Other approaches attempt instead to vary the number of colors assigned to applications. The key idea behind the work by Xiao et al.[50] is to employ partitioning only on the "hottest" subset of pages of each application in order to color only the areas which are most used. Limiting the number of colored pages could theoretically make it simpler to change the partitioning at runtime, because only a small subset of pages would be re-colored. However, to implement such a technique it is necessary to measure the "hotness" of each page. In order to do this with the hardware support modern CMP provide, the proposed solution periodically traverses the list of memory pages to check whether they have been accessed, with a huge overhead. In order to decrease this overhead, the solution exploits spatial locality of applications to check only the pages which are close to accessed pages, and hence the more likely to be "hot". Another hint employed is *lazy recoloring* of pages: a pages to be recolored is not immediately copied to the new physical location, but marked as inaccessible (by a bit in the virtual page table) and

copied only when really accessed, i.e. when an exception is raised because accessing the page. However, despite the optimizations implemented, the authors reach the conclusion that such an approach is not viable in practice.

A work that summarizes different contributions on the field was done by Jiang et al.[51]. They test several static and dynamic policies to fulfill different requirements like performance, fairness and QoS. Several metrics of performance and fairness are evaluated, and are used to guide the dynamic partitioning policies. These policies outperform the static ones in almost all cases. The improvement with respect to each objective (performance, fairness, QoS) measured with dynamic policies is due to the fact that such policies are able to react to the different phases an application has during the execution.

Coming closer to the cloud environment, Xinxin et al.[52] apply page coloring to the Xen hypervisor to show that also VMs may benefit from LLC partitioning, although virtualization already provides some isolation. This work is continued by adding a dynamic re-coloring mechanism[53], with the hint that the VM is not stopped during the page copy; when the copying operation is done, the hypervisor checks whether the original page has been modified, and in such case it retries the copy for a limited number of times. Otherwise it changes the entry in the VM page table pointing to the old page to point to the new page, and finally releases the old physical page.

CHAPTER 4

DESIGN

This chapter shows the design of our proposal, which is a modification of the Linux kernel to implement page coloring. Our solution is aimed at guaranteeing to applications isolation in the LLC. Its design is based on the architecture of modern CMPs, with a careful study of which constraints these architectures pose to our proposal. Furthermore, we also study the limitations our modification poses on the applications, in order to show the main trade-offs needed to employ page coloring on modern architectures.

In particular, Section 4.1 will explain the overall vision behind our work, while Section 4.2 reports more details about the design with respect to the Linux kernel. Section 4.3 discusses the assumptions on the LLC at the base of our proposal and the limitations they impose on our work, while Section 4.4 shows how page coloring impacts on applications. Finally, the modifications to the Buddy data structure and to the allocation algorithm are explained in Section 4.5.

4.1 Vision

The overall vision behind this thesis is to let cloud platforms provide better QoS via isolation of running applications. A way to ensure QoS is partitioning the hardware resources shared by co-running applications. Some techniques like time sharing mechanisms already exist from a long time, but cannot avoid the effects of contention exposed in Section 2.4. Moreover,

partitioning is actually limited to a coarse granularity, because some resources are partitioned and others are not. Unpartitioned resources make performance unstable, increasing the variance with respect to an "ideal" partitioning of resources. Therefore, it is necessary to partition those hardware resources where most of the conflicts arise, taking into account the various layers of the architecture, to achieve a better QoS with modern CMP.

One of these resources is the cache, where contention occurs (as in Section 2.4), thus affecting application performance. In particular, the LLC is shared among the cores, leading to conflicts among applications that force reloading data from main memory; this in turn increases the latency of each process in an unpredictable way, as the conflicts depend on the mix of co-running applications. What we want to achieve is a complete isolation of applications in the LLC, in order to make their performance predictable. This would allow us to assign to an application a LLC partition of a suitable size, in order to achieve a certain QoS. This goal must be achieved when the application is co-located with others on the same CPU, which is the scenario we consider in this work.

Ideally, we would like to achieve predictable performance for mixes of applications sharing a single commodity CMP, like those powering data centers, without any modification to either runtimes (e.g., the Java virtual machine) or applications. The implementation of page coloring within an OS makes a good fit, since it just requires updating the OS and profile applications to understand their resource requirements.

4.2 Approach

Our goal is to partition the LLC having low impact on existing platforms and taking into account the cache hierarchy of CMPs architectures. Given this goal, a software technique is a mandatory choice; supporting page coloring is the only solution to LLC partitioning in software. Page coloring requires deep modifications to the physical memory allocator, a core part of any operating system that is not configurable nor "pluggable" at runtime.

Performing such a deep change requires the modification of the kernel parts that manage the physical allocation and the processes. Furthermore, our focus on cloud infrastructures leads us to use an hypervisor for this work. Thus, the available choices for our target kernel are Linux and Xen, while other solutions like VMware ESX and Microsoft Hyper-V are impractical since the source code is not freely available. We choose Linux, released under GNU General Public License (version 2) (GNU GPLv2) license, because it allows us to make prototypes without the need of a dedicated development environment.

Because of this choice, our design is affected by the architecture of the Linux kernel. The system we designed is called *Rainbow* and accounts for the following objectives. Introducing page coloring into the Linux kernel in order to *isolate* the LLC partitions of co-running applications requires changes to the Buddy data structures, storing free pages, and to the algorithm that performs allocation and de-allocation (Section 4.5). These components are at the core of physical memory management, and any modification to them must be carefully studied.
In fact, physical memory management is a frequently accessed subsystem, involved in the page

fault mechanism that allows Linux to grow and shrink the physical memory of a task (i.e., process, thread or VM).

Despite the low level implementation of *Rainbow*, our aim is to provide an easy interface to processes wishing to exploit its novel functionality.

Given these guidelines, the present chapter explores the main design decisions behind *Rainbow*. Section 4.3 explains the basic assumptions *Rainbow* relies on, in particular those about the cache hierarchy, and the constraints imposed by the architecture; Section 4.4 investigates the limitations of page coloring on CMP architectures and finally Section 4.5 explains the modifications to the Buddy algorithm made to implement page coloring.

4.3 Cache hierarchy model and consequences

This section explains the major assumptions at the base of *Rainbow* and what are their consequences. In particular, Section 4.3.1 goes through these assumptions in the context of a generic CMP architecture with multiple layers of cache. Instead, Section 4.3.2 shows the consequences of page coloring on the assumed caches hierarchy, with the possible drawbacks.

4.3.1 Assumptions on the cache hierarchy

Devising a model of our target cache hierarchy is an essential task to go through the design of *Rainbow*. On one side, this model reflects the cache hierarchies available in commodity CPUs, such as the layering of caches and their interconnection with the cores. On the other side, it is general enough to be applied to a wide class of commodity CMP architectures.

First, we assume an architecture with multiple cores and multiple layers of cache. The generic model of each layer of the cache hierarchy is a *set associative* cache; each layer of cache

has specific values of associativity and number of sets, but the line size is assumed to be constant across all the layers and to be equal to L. We immediately note that modern architectures have a first cache layer designed according to a Harvard architecture, with two separate caches for code and data. This detail can be considered in our model, but will not impact on the design of *Rainbow*.

Two assumptions are at the base of *Rainbow*: the LLC is physically indexed and unique, and is shared among the cores. These two assumptions are important to control the placement of data in the cache and thus to obtain an effective partitioning. In particular, the first assumption cannot be relaxed: denying it means using a virtually indexed cache, which can be controlled only through the virtual memory address space of the application. Yet, this virtual space is under the control of the applications loader (or of the runtime environment, like the Java virtual machine) and only partially under the control of the OS; this would cause some of the memory regions of the process (like the text segment) not to be manageable and so not to be partitionable. Therefore, the physical indexing of the cache is a fundamental feature. The other key requirement is that the LLC is shared among the cores through the lower levels of per-core caches. This assumption simplifies the management of the physical memory as the system deals with a unique cache, and makes partitioning the LLC necessary to guarantee isolation.

These assumptions fit a wide variety of commodity architectures, like those if Intel, ARM and IBM. Some architectures like IBM Power6 and Intel Crystalwell (a variant of Haswell) also have an higher layer of cache used as a victim cache, which is outside our model. Because of

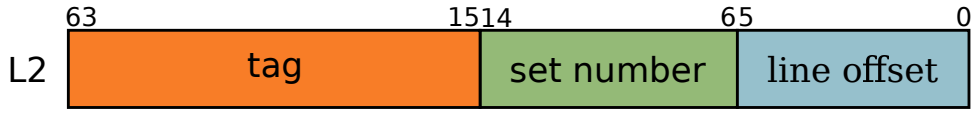


Figure 5: Physical address bit fields for L2 cache access

the different nature of these memories, which act solely as a victim buffer for the lower levels of the hierarchy, modeling them is not needed for the design we propose. Hence, the lower level to which they are connected is assumed to be the LLC and to be directly connected to the main memory controller.

It is important to stress that the physical indexing of the LLC is a feature found in all modern CMPs because of the small latency of the TLB. For the same reason, also the lower level cache (usually a per-core L2 cache) is often physically indexed. The consequences of this fact are discussed in the following section.

4.3.2 Partitioning the hierarchy

Page coloring is the only software technique available to partition the LLC; this restricts the address bits that can be used for partitioning, limited to those in common between the LLC cache set bits and the page address bits (Figure 4c). Yet, cache allocation is subject to another constraint. Assuming three layers of caches, partitioning the LLC could impact also the use of the L2 cache. Figure 5 shows the use of the physical address to access the L2 cache of our target platform, which differs from Figure 1 in the number of bits of set number: here the set number consists of 9 bits.

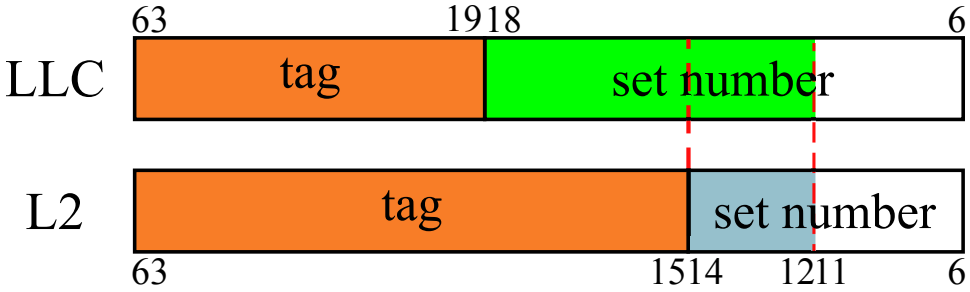


Figure 6: Overlap of color bits and L2 set bits

Among these bits, some may overlap with the LLC color bits, as in Figure 6; in this figure, the 2 most significant bits of the L2 set number overlap with the LLC color bits. Let the LLC color bits be c_{LLC} bits (in Figure 6, they are the 7 color bits identified in Figure 4c) and the overlapping bits be v.

This overlap causes also the lower level of cache to be partitionable through the subset of v bits just identified. If we assume that only one thread runs on each core, partitioning this level of cache can penalize the running thread as it would restrict the data available in this layer; this, in turn, increases the accesses to the LLC and hence the latency of data fetching operations. Instead, partitioning the L2 cache may be reasonable with Simultaneous MultiThreading (SMT) architectures, but we discuss the partitioning of the L2 cache in SMT-enabled cores in Section 7.2. In this work, we chose not to partition this layer and thus not to use the v bits. This decision is another key feature of *Rainbow*, that ensures the full exploitation of the lower cache levels capacity in CMP architectures. The remaining color bits are thus $c = c_{LLC} - v$, and are those effectively usable for LLC partitioning. The same check should be

done for all the cache layers, but the lowest level (the L1 cache) is usually virtually indexed, so that partitioning the upper layers does not influence the data placement inside this layer. But even if this layer was physically indexed, the set number bits of both the L1 code and data caches are much less than those of the LLC and do not overlap with the color bits.

4.4 Consequences of page coloring on applications

The tight relation between LLC mapping and physical page address identified in Section 4.3.2 poses some limitations to the applications to be isolated. These limitations are discussed in this section.

4.4.1 Cache-Memory constraint

The first limitation is that the physical memory available to the application is constrained by the percentage of cache allocated to it. If we assume the amount N of physical memory pages to be an exact multiple of the number of colors $N_c = 2^c$, then each color corresponds to $\frac{N}{N_c}$ pages and a corresponding amount of memory of $\frac{N}{N_c} \times P$ bytes (P beig the page size). Hence, if an application is reserved a percentage k of LLC and thus $n = k \times N_c$ page colors, it can allocate at most $n \times \frac{N}{N_c}$ pages corresponding to $n \times \frac{N}{N_c} \times P$ bytes of main memory. Relaxing this constraint by using pages of the same color for different applications is not considered, as it would force applications to share LLC sets and would prevent perfect isolation.

This memory constraint can theoretically be a limitation for applications with a big memory working set and a small cache working set; to fit the cache usage of these applications, it would be natural to devote them a small partition and so a small number of colors, thus restricting

the physical memory available to the application. In such case the Linux kernel would swap to the disk many memory pages of the application, causing an intolerable performance penalty. Furthermore, if the available memory is much less than the requested memory the kernel could even fail in allocating pages, leading to crashes. Yet, applications with a big memory footprint and small LLC usage are rare in practice, as the LLC footprint is roughly proportional to the memory occupation.

A second drawback is that if the amount of memory is not an exact multiple of N_c, some colors are associated to a smaller amount of pages while others to a greater amount; the LLC sets corresponding to these last colors would thus experience a higher pressure in terms of accesses and of evictions. This situation is however unlikely to happen because of the huge amount of main memory installed in current machines, which gives applications a lot of memory and so a huge number of memory pages to "spread" their data on, thus avoiding pressure on small areas. For example, with 12 GB of main memory installed and $k = \frac{1}{8}$, 1.5 GB is available to the application.

4.4.2 Limits on physical page size

The second limitation is the impossibility of using hugepages (see Section 2.6), available in some modern architectures.

As from Figure 7, the very different granularities of hugepages and of the LLC management cause the color bits to overlap entirely with the page offset; therefore, a single hugepage covers all the possible colors of the platform and is mapped to all the cache sets, thus denying isolation

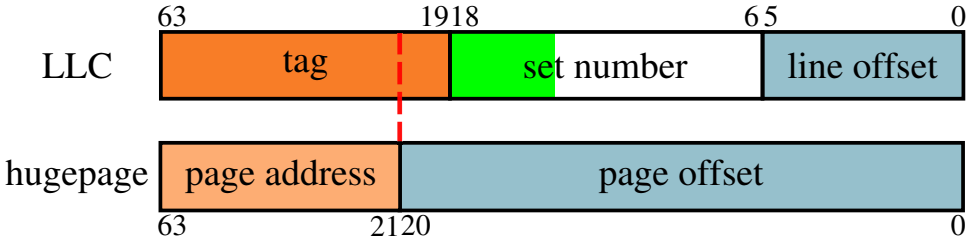


Figure 7: Bit fields for hugepages and LLC.
The color bits are highlighted and do not overlap with the page address, thus making page coloring unfeasible.

of different applications.

This is the main limitation of *Rainbow* and cannot be relaxed because of the strict correspondence between physical memory and LLC mapping. Given the size of modern LLCs, the only way to overcome this limitation would be adding hardware interfaces to control the LLC mapping. Otherwise, architectural modifications like a bigger line offset, more LLC sets or more granular page dimensions could make hugepages available with *Rainbow*. But all these features would have a big impact both on CMP and on OS architectures and have not been planned by CPU designers.

4.5 Rainbow Buddy

Implementing page coloring requires a modification of the Buddy algorithm and data structure, which have been explained in Section 2.5. The aim of these modifications is to permit the allocation of a page of a specified color. The modifications are based on the LLC parameters identified in Section 4.3 and have a high impact on the Buddy allocator: in more details, Sec-

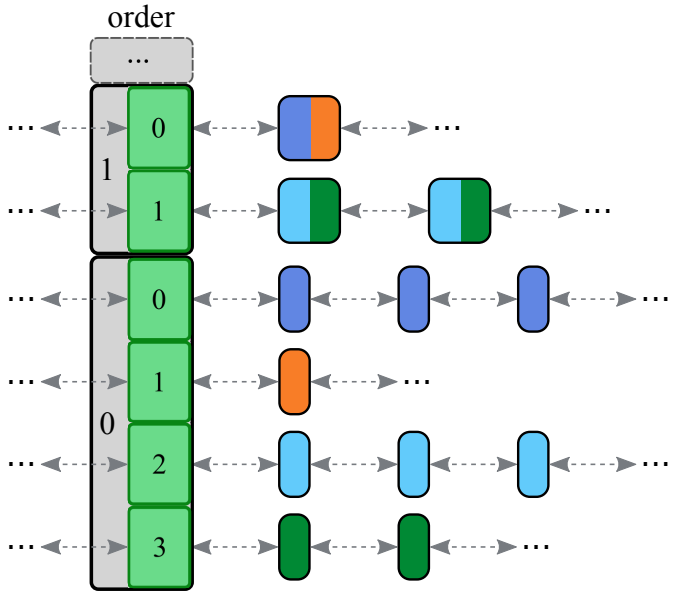


Figure 8: Rainbow Buddy data structure.
The designed data structure with colored lists per-order; in the higher levels multiple colors are aggregated.

tion 4.5.1 explains how the data structures are modified while Section 4.5.2 explains how the basic algorithms of insertion and removal are modified.

4.5.1 Data structure modifications

The data structure of the Buddy system must be modified to allow insertion and removal of pages of a specific color. The color is specified in the request: thus, it is natural to divide each order list into several sub-lists, one per color. Therefore, the *Rainbow* Buddy allocator uses an array of lists, which can be accessed in constant time by adding the color number to the base pointer.

More colors can be specified in the request, for example when the application to be isolated

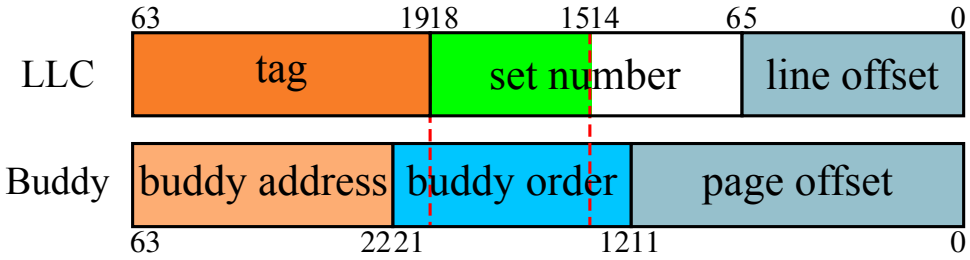


Figure 9: Overlap of color bits and buddy bits
The complete overlap determines the division into arrays of different size of the Buddy order lists.

is given more colors and any one of them can be used. In this case, a drawback of the new data structure is the need of iterating through multiple lists to search for an available buddy. To avoid this, we also decide to store the number of buddies for each order, so that in case no buddy is present the algorithm immediately looks for a buddy of higher order to be split.

Figure 9 shows a possible configuration of the bit fields used to identify the buddy, with the page color highlighted; the color bits depend on the parameters identified in Section 4.3. Because of the limitations discussed in Section 4.3.2 on the color bits, these bits may completely overlap with the "buddy order" bits. These bits (also highlighted in Figure 9) are the ones that must be 0 in the buddy address, and their number depends on the buddy order, as Section 2.5 explains; Figure 9 highlights all the possible buddy bits, assuming that the maximum allowed order is 10.

The overlap of buddy bits and color bits affects the structure of *Rainbow* Buddy. If we assume that no bits overlap, which happens for buddies of low orders (in Section 4.3.2 from order 0 to order 2, included), than the buddy address contains a specific color; in this case the

buddy spans only that single color. Since all the N_c possible colors are available under this hypothesis, the lists at these orders must be split into N_c sub-lists, one per-color.

Instead, if we assume that only 1 bit overlaps with buddies of a certain order i (order 3 in Figure 9), the possible colors allowed are $\frac{N_c}{2}$, and thus the per-color lists at i-th order must be only $\frac{N_c}{2}$. Similarly, if n colors overlap for a certain order, its sub-lists must be $\frac{N_c}{2^n}$. Finally, if all the bits overlap, the corresponding orders have only one sub-list (this happens in Figure 9 for orders from 7, included, to 10). The different lists are exemplified in Figure 8.

4.5.2 Algorithm modification

To insert a buddy, its color is found from its address and is used to access the proper color list of the buddy order; then, as before, the buddy is inserted on the head of the list. To get the color of a buddy it is sufficient to extract the color of its first page (which identifies the buddy itself) and right-shift this color of the order of the buddy. The *Rainbow* buddy allocator uses the same procedure for coalescing buddies as the original buddy allocator, since it depends on the physical contiguity of the two buddies. The insertion algorithm is explained in the following pseudo-code snippet.

```
1 global data: list_head buddies[NR_ORDERS][NR_COLORS]
2
3 procedure INSERT_BUDDY
4
5 inputs: buddy_address addr
6     buddy_order ord
7
8 output: none
```

```
9
10 code:
11   order_color = PAGE_COLOR(addr) >>  ord
12
13   twin_addr = addr + (1 XOR ord)
14
15   if( BUDDY_IS_FREE(twin_addr) )
16     new_buddy = COALESCE_BUDDIES(addr, twin_addr, ord)
17     INSERT_BUDDY(new_buddy, ord + 1)
18     return
19   else
20     HEAD_INSERT_INTO_LIST(buddies[ord][order_color],addr)
21
22 end procedure
```

The removal of a buddy of order n and color c is done by checking the list of the desired color; if no buddy is present, it is necessary to split a higher order buddy. This higher order buddy must be chosen carefully as it must contain a buddy of the desired color c. If the number of colors of order $n+1$ is the same as that of order n, then the same color list is accessed. If the number of colors is smaller, then each color of order $n+1$ spans 2 colors of order n: in this case the color to be accessed at order $n+1$ is thus $c/2$, and after the splitting the proper buddy must be returned and the other stored into its own color list. Buddy splitting and removal procedures are exemplified with the following pseudo-code snippet. The main procedure is `REMOVE_BUDDY()`, which looks for a buddy in the global list; if a buddy of a requested color (colors are stored in a set, as an application can receive several colors) is not found, the procedure `SPLIT_BUDDY()` in invoked, which splits a buddy of the higher order and returns one half. Before this call, the colors set is translated into another colors set containing the colors for the higher order (different orders can have different colors, as from Section 4.5.1).

The procedure `SPLIT_BUDDY()` recursively calls itself in two cases: in the first case when the

entire order has no buddies, and in the second case after no buddy is found among those reserved to the application (through its color). In particular, if `SPLIT_BUDDY()` founds a suitable buddy it splits it into two halves which are represented by their initial page and computes their addresses assuming that the pages are all stored inside an array (as in Linux).

```
1  global data: list_head buddies[NR_ORDERS][NR_COLORS]
2           integer nr_colors[NR_ORDERS]
3
4
5  procedure MAP_COLOR_SET
6
7  inputs: buddy_color_set c_set
8      buddy_order old_ord
9      buddy_order new_ord
10
11 output: buddy_color_set
12
13 code:
14   if nr_colors[new_ord] == nr_colors[old_ord]
15     return c_set
16   else
17     buddy_color_set new_c_set = EMPTY_SET()
18     shift = log2(nr_colors[old_ord] - nr_colors[new_ord])
19     for color in c_set
20       new_color = color >> shift
21       INSERT_UNIQUE_ELEMENT(new_color,new_c_set)
22
23     return new_c_set
24
25 end procedure
26
27
28 procedure SPLIT_BUDDY
29
30 inputs: buddy_color_set c_set
31     buddy_color_set target_c_set
32     buddy_order ord
33
```

```
34 output: buddy_addr
35
36 code:
37   if ord > MAX_ORDER
38     return Null
39
40   if ORDER_HAS_NO_BUDDIES(ord)
41     new_c_set = MAP_COLOR_SET(c_set, ord, ord + 1)
42     return SPLIT_BUDDY(new_c_set, c_set, ord + 1)
43
44   for color in c_set
45     if buddies[ord][color] is empty
46       continue
47     else
48       buddy = REMOVE_LIST_HEAD(buddies[ord][order_color])
49       lower_half = buddy
50       upper_half = buddy + (1 << ord)
51       lower_color = PAGE_COLOR(lower_half) >>  (ord - 1)
52       upper_color = PAGE_COLOR(upper_half) >>  (ord - 1)
53       if lower_color in target_c_set
54         INSERT_BUDDY(upper_buddy, ord - 1)
55         return lower_buddy
56       else
57         INSERT_BUDDY(lower_buddy, ord - 1)
58         return upper_buddy
59
60   new_c_set = MAP_COLOR_SET(c_set, ord, ord + 1)
61   return SPLIT_BUDDY(new_c_set, c_set, ord + 1)
62
63 end procedure
64
65
66 procedure REMOVE_BUDDY
67
68 inputs: buddy_color_set c_set
69     buddy_order ord
70
71 output: buddy_addr
72
73 code:
74   for color in c_set
75     if c_set(color) is empty
76       continue
77     else
78       new_c_set = hal
79       return REMOVE_LIST_HEAD(buddies[ord][order_color])
```

```
80
81    new_c_set = MAP_COLOR_SET(c_set, ord, ord + 1)
82    return SPLIT_BUDDY(new_c_set, c_set, ord + 1)
83
84  end procedure
```

CHAPTER 5

IMPLEMENTATION

This chapter discusses the details of the implementation of *Rainbow* on the basis of the design choices of the previous chapter. Section 5.1 shows the way *Rainbow* is implemented inside the Linux kernel, while Section 5.2 gives an overview of the various components added with *Rainbow*.

Section 5.3 explains how *Rainbow* represents page colors and uses reads the color of a memory page or of a buddy; this representation is used to associate a set of colors to an application, as Section 5.4 shows. Finally, Section 5.5 discusses how the modifications to the Linux Buddy allocator are implemented.

5.1 Implementation approach

The design proposed for *Rainbow* in Chapter 4 is implemented on the Linux kernel version 3.9.4.

Because of the fundamental role of the physical memory allocator, this subsystem cannot be plugged or unplugged at runtime, and must be statically linked inside the Linux kernel binary. Hence, the only way to implement *Rainbow* in Linux without disrupting the default codebase was to add the new source code to the kernel working tree and to switch between the default allocator and *Rainbow* at compile time. Our first care is not to change the higher levels of the Linux memory management system, in order to have the smallest possible impact on the

existing infrastructure and above all on the applications running in Linux. Normal applications are therefore not isolated with respect to the LLC, while applications wishing this feature must request it explicitly.

5.2 Overall mechanism

As *Rainbow* is basically a modification of the Buddy allocator, it cannot be developed as a per se component, but must be tightly integrated into the Linux kernel. Several calls have been added to the Linux kernel code, whose task is explained in the next sections.
When the kernel boots, it is fundamental to initialize the new data structures correctly; hence a call for boot initialization is added in order to collect the cache parameters and initialize the data structures properly. The following sections explain how the data structure is initialized.

After boot, each colored application (i.e., each application whose LLC portion is isolated) must explicitly ask the kernel for a dedicated LLC partition through a system call *Rainbow* introduces. Applications that have requested an LLC partitions are called *colored applications*.

During runtime the Rainbow *allocator* must determine which physical color to allocate from depending on the running application and on the allocation origin (userspace or kernel). The final call performs the *color reservation removal*, to make the page colors reserved to an application newly available when this application terminates.

5.3 Page colors representation in *Rainbow*

Here we explain how page colors are represented and managed in *Rainbow*. The data structure to do this is discussed in Section 5.3.1 and how colors are read from pages and buddies is explained in Section 5.3.2.

The overall objective behind the implementation we propose to represent page colors is efficiency: since the color information associated to a colored application is accessed frequently during allocation (see Section 4.5), it is fundamental to get the assigned colors in an easy and fast way.

5.3.1 Color sets representation

A colored process is assigned a set of page colors, which must be represented in a compact and efficient way. The physical allocator uses this information to search a buddy of the requested file through the per-color lists (Section 4.5). Therefore, *Rainbow* needs to associate this information to the process that is colored. The data structure that contains the information of a colored process is shown in Listing 5.1:

Listing 5.1: Colors set representation

```
1 struct color_info {
2   color_mask cmask;
3   unsigned int last_color;
4   unsigned int count;
5   atomic_t thread_count;
6 };
```

This data structure is associated to the process it refers to via a new field of the process handler; in fact, in Linux each thread is represented by a structure `task_struct`, to which we added a field `struct color_info *cinfo` that holds the coloring information. If this field is `NULL` (as by default), the process is not colored, otherwise this field points to the process coloring information.

The first field `color_mask cmask` is a bit mask used to hold the colors associated to a process; the *Rainbow* allocator must iterate over this bit mask in order to find a free buddy to allocate. This bit mask must contain at least one bit per color, where the number of colors of the architectures is indicated in the code as `NR_COLORS`. The third field, `cmask`, stores the number of colors in the bit mask.

The second field is a hint to decrease the number of iterations over the per-color lists: in fact, if buddy of a certain color has been accessed recently, it is very likely that its "twin" buddy is stored inside the next list, thanks to the splitting procedure. This may save time when the allocator must iterate upon many colors.

Finally, the last field contains a counter of the process threads that are running. All these threads receive physical memory from the "color pool" (i.e., the set of reserved colors) of the process. This counter is incremented whenever a thread is created and is decremented whenever a child process or a thread exits. The type of this counter is `atomic_t`: this is an architectural-independent data type defined in the kernel that guarantees the atomicity of accesses. Atomicity avoids concurrency problems (like lost updates) due to the simultaneous execution of multiple threads, which may be created or may terminate at the same time.

Rainbow uses two global variables of type `struct color_info`: `cinfo_global` and `cinfo_process`. The first one contains a bit mask with all the colors in the system and is used for kernel allocations (as explained later). The second one contains a bit mask with all the colors not yet associated to any process and is used for allocating memory to non-colored processes. Whenever a process is colored, the colors assigned to the process are removed from the `cmask` field of

`cinfo_process`, where they are re-set when the process terminates. This guarantees that the non-colored processes cannot allocate memory pages mapped into the LLC partitions of colored processes, thus guaranteeing isolation.

5.3.2 Reading the color

To re-insert a buddy into the data structures of the allocator, it is fundamental to read its color. This information can be derived from its address and its order. Recalling Figure 9 in Section 4.5.1, the buddy color is made by the lower bits of the buddy address. Because of the memory alignment of buddies, some the lower bits of the buddy address are 0 (depending on the buddy order). Hence, this zeroed bits are useless and the buddy color must be eventually shifted rightward to eliminate these bits. The number of positions to shift depends on the buddy order.

To find the color of the buddy, a set of C macros has been implemented. This macros are shown in Listing 5.2.

Listing 5.2: Macros to get page and buddy color

```
1 #define page_mcolor(page, order)                                    \
2   ((((unsigned long)(page_to_pfn(page))) & color_bitmask) >> (
      cshifts[order]))
3
4 #define page_vcolor(page) page_mcolor(page,0)
5
6 #define mcolor_from_vcolor(vcolor, order) (vcolor >> (cshifts[
    order] - cshifts[0]))
```

To briefly explain the meaning of these macros, throughout the code *mcolor* indicates the color of a buddy of any order, while *vcolor* the color of a single physical page. The first macro

receives in input the physical address of a buddy and its order (parameters `page` and `order`, respectively) and computes its color. To do this, the first step is to get the physical address of the page, called Page Frame Number (PFN) in the Linux terminology. The built-in function `page_to_pfn()` solves this task. Then, the macro performs a bitwise AND of the PFN with a constant bit mask to extract the bit of the LLC set number and finally discards the lower bits according to the buddy order. The number of the bits to be discarded for each order is stored in the array `cshift`, which is initialized at boot time on the basis of the L2 and L3 parameters and of the buddy order. The last macro, used throughout the allocator code, is used to find the color of a buddy from that of a page in case of higher order buddy search: it consists in a right bit shift, where the number of bits to shift is determined by the buddy order.

5.4 Coloring an application

Applications must request *Rainbow* to be "colored" in order to get a dedicated partition of the LLC. The major steps to mark an application as colored are three: determining how many colors to be devoted, which colors and finally associating the set of colors to the application. Section 5.4.1 shows how to decide the number of colors to be assigned to a process, while Section 5.4.2 explains the mechanism to choose which colors to assign and to associate the colors set to the requesting process.

5.4.1 Determining the number of colors

Applications wishing to be colored must explicitly notify this wish to *Rainbow* in order to be associated a set of page colors. How many colors each application needs to be isolated

in LLC is a careful decision which must not penalize the application nor wasting LLC space that can be devoted to other applications. The possible penalization is due both to a limited LLC space which increases the LLC miss rate for the application and to the memory constraint discussed in Section 4.4.1. This is a major issue with the static partitioning policy implemented in *Rainbow*, but is unavoidable with the current design; on the other side, leaving the choice of the colors number to the user allows to profile each application with a varying number of colors; this makes it possible to characterize the applications behavior with respect to the LLC. In addition, the memory footprint of an application is a hardly predictable parameter. For these reasons, we chose to leave to the users the choice of the number of colors. This number must be communicated to *Rainbow* via the interface described in the following section.

5.4.2 Rainbow interface with application

The interface between the application and *Rainbow* is a Linux system call. This mechanism is a natural choice for this task as it is synchronous from the point of view of the application: when the application is assigned the CPU after the system call, it is guaranteed to be colored. This is important for applications eventually relying on the benefits provided by *Rainbow*. Another possible implementation is adding a configuration knob to the cgroup subsystem of the Linux kernel[54], in order to give a unified interface with other options; this choice has been discarded as it is beyond the scope of this work, because a clearer interface does not give any benefit in terms of effectiveness and a system call is an easier way to implement our prototype. Therefore, we implemented the userspace interface of *Rainbow* as a system call which takes as input the number of colors to be allocated to the calling process. Its code is shown in Listing 5.3.

Listing 5.3: System call to color an application

```
1  asmlinkage long sys_use_cache_colors(const unsigned int count)
2  {
3    unsigned long mask=0;
4    unsigned int i,j;
5    struct color_info *global, *ci;
6    struct task_struct *p = current,*c;
7
8    ci = (struct color_info*)kmalloc(sizeof(struct color_info),
       GFP_KERNEL);
9    if(ci == NULL) {
10     return -1;
11   }
12   for(i=0; i < count; i++) {
13     mask |= 1UL << i;
14   }
15   spin_lock(&cinfo_lock);
16   global = &cinfo_process;
17   if((count > global->count) || p->cinfo != NULL) {
18     goto fail;
19   }
20   for(i = 0; (i <= NR_COLORS - count) && (mask & reserved); i
       ++) {
21     mask <<= 1;
22   }
23   if(mask & reserved) {
24     mask = 0;
25     j=0;
26     for(i = 0; (i < NR_COLORS - count) && (j < count); i++) {
27       if(!color_is_set(i,&reserved)) {
28         set_color(i,&mask);
29         j++;
30       }
31     }
32   }
33   if(mask & reserved) {
34     goto fail;
35   }
36   reserved |= mask;
37   global->count -= count;
38   global->cmask = ((1UL << NR_COLORS) - 1) &  reserved;
39   spin_unlock(&cinfo_lock);
40   ci->cmask = mask;
41   ci->count = count;
42   atomic_set(&ci->thread_count,1);
```

```
43    c = p->group_leader;
44    p = c;
45    c->cinfo = ci;
46    while_each_thread(p, c) {
47      c->cinfo = ci;
48      atomic_inc(&ci->thread_count);
49    }
50    return 0;
51  fail:
52    spin_unlock(&cinfo_lock);
53    kfree(ci);
54    return -1;
55  }
```

The first part (lines 12-32) determines the colors to be reserved to the requesting application. The colors are determined with a best effort mechanism. Initially the system call attempts to allocate a set of contiguous colors (lines 20-22), in order to increase the probability of buddy coalescing when freeing memory and limit external fragmentation. If this attempt fails, the call allocates non-contiguous colors throughout the memory (lines 26-32), until the specified number of colors is allocated. Then, the new configuration is stored into the variables which control the memory allocation: the allocated colors are set in a mask called `reserved` which stores all the colors reserved for all the colored applications (line 37); then the variable `cinfo_process` is modified in order to contain only the colors still available (line 38).

These operations are protected by the spinlock `cinfo_lock`, which is acquired at the beginning of the procedure and is released only after the changes have been saved. Even if its use would theoretically decrease the scalability of the system, this code section is not a "hot" one since it is invoked only in the application initialization, and contention on the spinlock is thus very unlikely.

In the final part of the call, the color reservation is stored into a variable of type `color_info` , which is associated to the group leader of the running process and to all its threads (lines 40-49) via the field `cinfo` added to the structure `task_struct` (that represents the processes in Linux). This choice is fundamental to confine the whole applications into the LLC partition and prevent the other threads from mixing their data with those of other processes: Linux in fact represents every thread as a different *task* via a structure of type `task_struct`. Therefore, the color set must be associated with all the threads composing the application.

The final part of the call counts the number of process threads and stores it into the field `thread_count` atomically, for the reasons explained in Section 5.3.1.

Whenever a process calls `fork()` to create a child process or a new thread, its color reservation is copied from the parent process: if it is NULL, the parent process has not been colored, otherwise the child process receives the address of the same color set and the `thread_count` field is incremented. Instead, whenever a thread terminates via the `exit()` system call, this counter is decremented to keep track of only the processes effectively using the color reservation: when the counter reaches 0, the colors reservation is removed by updating the variables `reserved` and `cinfo_process` and the memory area containing the colors set is freed.

5.5 Rainbow Buddy

In this section we show the major modifications to the Buddy algorithm, according to the design proposed in Section 4.5. Section 5.5.1 shows how the Buddy data structures have been modified, while Section 5.5.2 shows the changes of the algorithm itself.

5.5.1 The modified Buddy structure

In the Linux kernel, the physical memory of a machine is divided into zones, represented by the array `struct zone node_zones[MAX_NR_ZONES]`. The zones are non overlapping memory areas used for different types of allocation; they are defined at compile time and depend on the architecture of the machine. For example, in a x86-64 system three zone usually exist: the *DMA* zone consists in the first 16 MB of memory and is for old DMA controllers, which use only 24 bits to store the address of the physical memory area where they copy the data read from the device; the *DMA32* zone extends from 16 MB to 4 GB and is used for DMA controllers using 32 bits for the physical address; the *Normal* zone is used to control the rest of the memory. Applications are preferably given memory areas from the normal zone, and the allocator falls back to the other areas if no memory is present in this zone; similarly DMA controllers with 32 bits address fall back to the DMA area if no memory is left.

The structure describing a zone contains many fields, most of them used to store information; instead, the field `struct free_area free_area[MAX_ORDER]` is the main data structure described in Figure 2, where the constant `MAX_ORDER` is the number of buddy orders allowed for the current architecture (10 by default).

Listing 5.4 shows the re-definition of the `struct free_area` data structure containing all the changes described in Section 4.5.1

Listing 5.4: The new Buddy data structure

```
1 struct free_area {
2 #ifdef CONFIG_LLC_ISOLATION
```

```
3   struct list_head  free_list[NR_COLORS];
4   unsigned int last_color;
5   unsigned int count;
6 #else
7   struct list_head  free_list;
8 #endif
9   unsigned long   nr_free;
10 };
```

where the macro `CONFIG_LLC_ISOLATION` enables the *Rainbow* subsystem at compile time. By comparing the two definitions of `free_list`, it is evident that with *Rainbow* each page list is split into `NR_COLORS` sublists, where `NR_COLORS` is the number of colors in the system. This is done for all the orders, although the higher orders have a smaller number of colors for the considerations of Section 4.5.1.

This decision simplifies the prototyping of *Rainbow*: a per-order number of sublists would have required the allocation of the needed memory space at boot time in order to initialize the memory allocator itself. This chicken-egg dilemma would be solved by using the boot allocator provided by Linux, but the modifications required to implement such a feature would have required large modifications of the data structure initialization and access which are beyond the scope of this work.

In addition to the free lists, two other fields are added. The field `unsigned int count` counts the number of free buddies and equals the sum of all buddies listed inside the `free_list [NR_COLORS]` lists; this condensed information is useful when allocating memory for a non-isolated application: in order to check whether the current order and migrate type contains buddies it is sufficient to check this count without iterating through the colored lists. The field `unsigned int last_color` provides instead a hint for the allocation algorithm to decrease the

number of iterations required to allocate a buddy: in particular, because of the splitting procedure, if a buddy is present in color i it is likely that its "twin" buddy is also present in color $i+1$, where the algorithm will restart its search at the next allocation.

Related to physical page allocation, the field `struct per_cpu_pageset __per_cpu *pageset` inside `struct zone` implements an optimization of Linux for multiprocessors systems: in fact, it is defined for every CPU of the machine (as the annotation `__per_cpu` suggests). This structure in turn contains a field `struct per_cpu_pages pcp`, which holds the most recent freed pages. Its definition, which is very similar to that of `free_list[NR_COLORS]`, is shown in Listing 5.5.

Listing 5.5: The per-cpu-pages structure

```
1  struct per_cpu_pages {
2    int count;     /* number of pages in the list */
3    int high;    /* high watermark, emptying needed */
4    int batch;     /* chunk size for buddy add/remove */
5  #ifdef CONFIG_LLC_ISOLATION
6    struct list_head lists[NR_COLORS];
7    unsigned int last_color;
8    unsigned int count_migtype;
9  #else
10   struct list_head lists;
11 #endif
12 };
```

Holding the most recent freed pages is an optimization of Linux to speed up the allocation: since single pages are the most requested buddies (because user space applications are given physical pages only after a page fault event), it is useful to store a certain number of them in

special per-CPU caches, so that most allocations can be served immediately without searching in the main Buddy data structure and acquiring locks (defined for each zone) which limit scalability. The number of pages inside each cache is kept in a range defined at boot time by specific procedures, executed whenever the number of pages is inside the range: in case too few pages are present, they are reclaimed from the Buddy data structure, while in case of excess they are released into the same structure and eventually coalesced into bigger buddies. In the code listing above, we can see the same modifications introduced in the definition of `struct free_area`, with the same purpose.

The initialization of these two data structure is done at boot time and is similar to the initialization of the normal Buddy data structures: the per-CPU pageset of the running CPU is initially linked to a "generic" boot pageset, all the CPU pagesets are later allocated at runtime and the zones free area ares initially set to 0 and then populated. Populating the free area proceeds as in the normal Buddy allocator: all the memory pages are freed one by one, added to the corresponding free area and time by time coalesced into bigger buddies; the only difference here is the per-color distribution into the different lists, shown in the following section. Similarly, the per-CPU pagesets are initialized by claiming pages from the free area for each color.

5.5.2 The modified Buddy algorithm

In order to explain the modifications to the Buddy algorithm, we provide here an example from one of the main functions to access the free area and the per-CPU pagesets called

`buffered_rmqueue`. At the beginning, it is fundamental to determine the type of allocation to be performed: this type can be isolated, non-isolated or kernel allocation. Isolated allocations are performed for processes to which a `struct color_info` is associated and allocate memory pages only from the reserved colors. Non-isolated applications allocate pages from non-reserved colors. Kernel allocations are served with memory pages of every color, in order to preserve the functionality of the kernel, avoid swapping of pages to disk due to high pressure on little areas of memory and allow the allocation of buffers for serving device interrupts. The origin of the allocation is identified at the beginning of the allocation, where the per- CPU pagesets are accessed.

Listing 5.6: Determining the color set for an allocation

```
1 if((gfp_flags & (GFP_DMA | GFP_DMA32 | GFP_ATOMIC)) || (order >
      0) || !ccount(get_allowed_cinfo())) {
2   ci = get_global_cinfo();
3 } else if(!colored_task(current)) {
4   ci = get_allowed_cinfo();
5 } else {
6   ci = get_task_cinfo(current);
7 }
```

The first branch checks if a kernel allocation is happening or if all the memory colors are being used: the first condition happens if the allocation comes from the DMA subsystem or is an atomic allocation (usually coming from device drivers) or a buddy of order greater than 0 is requested (only the kernel can allocate more than a page at once). The second condition is instead given by the number (function `ccount()` of colors in the variable `cinfo_process`, accessed via its interface `get_allowed_cinfo()`): if it is 0, all the colors have been associated

to processes and any color can be used. The second branch checks instead whether the task currently running is LLC-isolated: if not, the general mask for non-isolated processes is used for allocation, otherwise (last branch) the information associated to the process is used for allocation. Going further, the code checks the order of the allocation: if it is 0, it checks the presence of pages in the per-CPU pageset of the current core, otherwise it goes through the free areas of the Buddy allocator. In the first case it is necessary to find an allowed color for the allocation, i.e. to iterate through the set bit of the color bit mask. Listing 5.7 shows the initialization of the `color` variable (line 1) and the search for a non-empty per-CPU pageset of the desired color (lines 3-5). As soon as one is found, the page is removed from the list and returned.

Listing 5.7: Non-empty pageset search

```
1 color = find_color_round(cmask(ci),get_last_color(ci));
2 list = &pcp->lists[migratetype][color];
3 while(list_empty(list)) {
4   color = find_color_round(cmask(ci),color + 1);
5   list = &pcp->lists[migratetype][color];
6 }
```

If instead the allocation order is greater than 0, the Buddy allocator is invoked and the coloring information, stored in the variable `ci`, is passed to it.

Similarly, the other allocations throughout the Buddy code initially search for a free list of an allowed color which is non-empty. To take in consideration the order of the current list, the macro `mcolor_from_vcolor` is used as array displacement, with the fourth line of the code example above becoming

Listing 5.8: Per-color list selection

```
1 list = &area->free_list[migratetype][mcolor_from_vcolor(color,
    current_order)];
```

The last example we provide of the modifications to the Buddy allocator is the split of a buddy into two smaller ones, performed in the function `expand()`. Here, the desired page color must be taken into account for the splitting: this parameter, called `vcolor`, is used to determine the mcolor of the buddy to be further split (or returned). Listing 5.8 shows a code snippet that is inserted inside a loop which iteratively decrements the buddy order (variable `high`) and splits the buddy stored into the variable `page`.

Listing 5.9: Splitting of a free buddy

```
1 twin = &page[size];
2 if(mcolor_from_vcolor(vcolor,high) >= page_mcolor(twin,high) &&
      high < single_list_min_index) {
3   tmp = twin;
4   twin = page;
5   page = tmp;
6 }
7 list_add(&pag->lru, &area->free_list[migratetype][page_mcolor(
    twin,high)]);
```

In the first line the variable `twin` is initialized to the first page of the second half of the buddy to be split, while the passed parameter `page` contains the first page (recalling that the pages structures are stored inside a unique array). The branch checks whether `twin` contains the buddy of the desired mcolor and if the current buddy order spans more than one mcolor: in this case the variables `twin` and `page` are swapped for `page` to finally point to the buddy to be split again or to be returned. At the end, the `twin` buddy is stored into the free list.

CHAPTER 6

EXPERIMENTAL RESULTS

In this chapter we present the results obtained by experimenting with *Rainbow*. Section 6.1 explains the experimental environment used for testing *Rainbow*, with a brief description of the hardware and software components and of the experimental methodology used to perform the tests. Section 6.2 explains how applications are characterized and chosen in order to obtain the final evaluation of *Rainbow*, which shown in Section 6.3.

6.1 Experimental environment

This section explains the setup of the testing environment. To show the effectiveness of *Rainbow*, we needed to run CPU-intensive applications with the least possible disturbance, thus carefully setting the hardware and software of the platform.

6.1.1 Testbed and coloring parameters

The machine used as testbed is equipped with a single socket holding a quad-core Intel® Xeon™ W3540CPU with a clock frequency of 2.93 GHz and 12 GB of RAM. These parameters fairly represent the ones of current server systems, for which *Rainbow* is meant. The CPU has three layers of cache, with a fixed line size of 64 B:

- a shared L3 (LLC) cache of 8 MB, with 8192 sets and 16-way associativity
- a per-core L2 cache of 256 KB, with 512 sets and 8-way associativity
- a per-core L1 instruction cache of 32 KB, with 128 sets and 4-way associativity

- a per-core L1 data cache of 32 KB, with 64 sets and 8-way associativity

To measure the real effects of *Rainbow*, we disabled the cache prefetchers, the Hyper-Threading support [55] and the Turbo Boost support [56]. These features could indeed influence the applications performance and are not controllable in software nor measurable. In particular, cache prefetchers aggressively load data according to speculative policies, thus affecting the performance in an unpredictable way.

The physical page size of the architecture is the default one for x86 architectures, i.e. 4 KB. The parameters of interest for *Rainbow* are the following:

- 12 bits of physical page offset
- 6 bits of cache line offset
- 13 bits of LLC set number
- 9 bits of L2 set number
- 7 bits of L1 set number (considering only the instruction cache, the one with more sets)

from which we compute the fundamental parameters for page coloring:

- $13 + 6 - 12 = 7$ bits of LLC color
- $9 + 6 - 12 = 3$ bits of L2 color, i.e. the bits of L2 set number overlapping with the LLC color bits
- $7 - 3 = 4$ bits of color, hence with the possibility of partitioning the LLC cache in $2^4 = 16$ slices of 512 KB each one while avoiding L2 cache partitioning (the L1 set number bits don't overlap with these final bits); the color bits are those from the 15th to the 18th

On the software side, the Linux distribution installed is Ubuntu version 12.04 Long Term Support (LTS), and the kernel used is our custom *Rainbow* version with the default parameters of Ubuntu distribution. In particular, the number of buddy orders (Section 2.5) is the default one, i.e. 10. This implies that at most the 10 lower bits of the physical page address are 0 to indicate the buddy's first page (and thus the buddy itself). These bits, ranging from the 13th to the 22nd, completely overlap with the color bits, so that:

- the orders from 0 to 2 (included) have complete lists of 16 colors
- the orders from 3 to 5 (included) have half of the colors of the previous order (from 8 to 2)
- the orders from 6 on have only one color

6.1.2 Test applications

It is fundamental for showing *Rainbow* features to stress the CPU and memory subsystems with computational-intensive applications, possibly having different characteristics of cache-friendliness to shape a well-mixed workload and stress the different parts (LLC, memory controller, etc.). Moreover, it would be useful to use a well-known set of applications for a comparison with other works. For these reasons we choose the Standard Performance Evaluation Corporation (SPEC) CPU2006 benchmark suite[57], which is a reference suite for many research works[26]. This suite has a large set of applications from multiple fields having different characteristics with respect to the usage of the CPU resources.

Since many applications use more input sets sequentially to perform the tests, they can present

TABLE I: SELECTED SPEC CPU2006 TESTS.

Test	**Input**
462.libquantum	control
429.mcf	inp
471.omnetpp	omnetpp
437.leslie3d	leslie3d
450.soplex	pds-50
483.xalancbmk	t5
482.sphinx	ctlfile
473.astar	rivers
401.bzip2	text

different phases that would make the measurements collections of different, heterogeneous and not equally distributed data. Therefore, we chose to split all the possible inputs of the applications and to evaluate each one separately. In particular, we selected 9 benchmarks with different usage patterns in order to have a reasonable but representative set of applications with respect to their cache usage. For each application we chose the input set causing the longest run. The selected applications and inputs are reported in Table I

6.1.3 Experimental methodology and tools

Once the applications are selected, we illustrate the steps performed for testing the effectiveness of *Rainbow* and how they are performed.

To launch an application and give it a LLC partition, a small launcher application has been developed called `rlaunch`, which takes the number of colors to be used and the application to run. `rlaunch` executes the *Rainbow* system call `sys_use_cache_colors()` passing the color

number and then calls the `exec()` system call to run the target applications, which "inherits" the cache partition from the caller.

The application profile is built by recording data via the `perf stat` tool provided by Linux[58], which exploits the hardware performance counters available in the x86 architecture to measure the per-thread performance. For our experiments, we measured the IPC, the number of LLC accesses and the number of LLC misses. It is important to remark that, as our CPU has four hardware performance counters, the count of events has not been sub-sampled and the reported measures are real measures, not obtained by multiplexing.

All the run applications have been forced to run on a fixed core through the `taskset` command to avoid any overhead due to moving the application and reloading the data into the L2 and L1 caches. Finally, any power saving mechanism has been disabled using the `cpufreq -set` command and setting the `performance` CPU governor, which sets the maximum allowed frequency for all the cores.

6.2 Application characterization

The first step is to build an accurate profile of the applications behavior with respect to varying size of caches. These profiles will indicate the sensitivity of each application to the assigned cache space and will guide the choice of suitable workloads to show the effectiveness of the performance isolation *Rainbow* provides.

Therefore, we built the cache usage profile of the target applications by assigning them LLC partitions of different cache size. Each application was run separately, without any significant application running at the same time. For the various set points, 10 measurements were col-

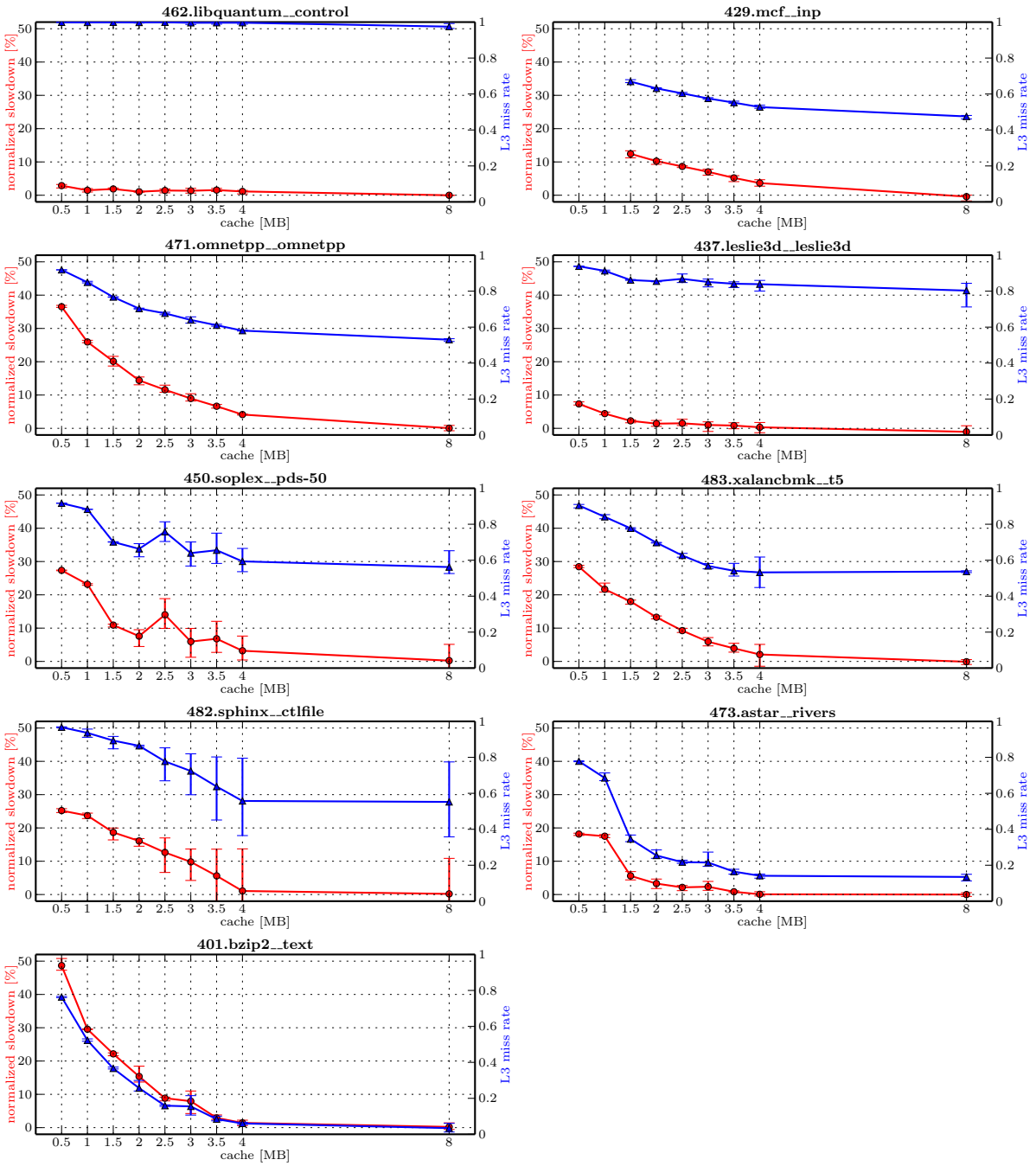


Figure 10: Applications profiles with different cache partitions.

lected, which are summarized in Figure 10. This figure shows the cache profiles with cache partitions sizing from 0.5 MB (1 color) to the whole cache (8 MB - 16 colors) with step of 0.5 MB. A particular case is 429.mcf, which needs at least 1.8 GB of RAM memory on 64 bits architectures to run. Because of the limitations found in Section 4.4.1, 429.mcf must receive at least 1.5 MB of cache, corresponding to 2.25 GB of memory.

Figure 10 plots the averaged data points with the interval bars, with time slowdown percentage (red line), LLC miss rate (blue line) and L2 miss rate (green line). What immediately strikes are the plots of 450.soplex and 482.sphinx, whose interval bars indicate a huge variability of the collected data. As these phenomenon is not seen in the other profiles, it is likely to be caused by a random behavior of the two applications, whose duration is about one order of magnitude bigger than the others (this fact could worsen the effects of a random pattern in data access). In particular, 450.soplex works with a high-dimensional sparse matrix and 482.sphinx emplys randomization to sample the input set repeatedly [59]. However, we are actually investigating these behaviors, in order to gain a clearer insight on the LLC usage.

At a general glance, the collected profiles confirm our good choice of a variegated set of tests, where some like 401.bzip and 471.omnetpp have great benefits from more cache space, while others have almost none like 462.libquantum and 437.leslie3d.

Furthermore, we measured the L2 miss rate, which is represented in Figure 10 with a green line. The fact that this measure remains almost constant in all the measurements of each application ensures the L2 cache is used completely, thus avoiding performance bottlenecks due to a bad partitioning.

TABLE II: CLASSIFICATION OF APPLICATIONS.

Category (symbol)	**Sensitivity range**	**Applications**
Red (R)	$\geq 25\%$	401.bzip2, 471.omnetpp, 483.xalancbmk
Yellow (Y)	$\geq 10\%$	429.mcf, 450.soplex, 473.astar, 482.sphinx
Green (G)	$< 10\%$	437.leslie3d, 462.libquantum

6.3 Isolation of co-running applications

With the cache profiles collected previously, we can choose a number of workloads with mixed applications characteristics. The aim of this choice is to run the applications simultaneously on different cores, isolating one of them in LLC and measuring it. Therefore, to exploit the potential benefits provided by *Rainbow* the target application must be sensitive to cache space, while the others should be varied in order to see how *Rainbow* effectively partitions the LLC. To choose the workloads, we classified the applications into three categories according to their "sensitivity" to the LLC partition size. This feature can be indicated by the reduction of slowdown with various LLC partitions, as previous work does [26]. In more details, we looked at the difference in slowdown between the run with 0.5MB of LLC and that with the whole cache. According to this value being in certain ranges, we categorized the tests as *red*, *yellow* or *green*, with decreasing cache sensitivity. The classification is shown in Table II, with the thresholds that define the category.

With respect to the values chosen for the ranges, 450.soplex and 482.sphinx appear as corner cases, but have been classified as Yellow because of their high variability, which sometimes prevents them from exploiting the LLC as "true" Red applications (if we consider the worst

TABLE III: TESTING WORKLOADS.

Workload	**Target**	**Polluters**
3R_1Y	471.omnetpp	483.xalancbmk, 401.bzip2, 450.soplex
3R_1G	401.bzip2	471.omnetpp, 483.xalancbmk, 437.leslie3d
2R_2Y	483.xalancbmk	471.omnetpp, 429.mcf, 473.astar
2R_1Y_1G	483.xalancbmk	401.bzip2, 437.leslie3d, 482.sphinx
2R_2G	401.bzip2	471.omnetpp, 437.leslie3d, 462.libquantum
1R_3Y	471.omnetpp	450.soplex, 429.mcf, 482.sphinx
1R_2Y_1G	483.xalancbmk	473.astar, 482.sphinx, 462.libquantum
1R_1Y_2G	401.bzip2	473.astar, 437.leslie3d, 462.libquantum
3Y_1G	429.mcf	473.astar, 482.sphinx, 462.libquantum
2Y_2G	450.soplex	482.sphinx, 437.leslie3d, 462.libquantum

case, the slowdown difference is in fact below the threshold for the Red category).

From these categories, 10 workloads have been chosen to be tested. Red applications, being the most sensitive ones, are granted an LLC partition and their performance is measured. The primary criterion to choose the applications' mix for each workload is mixing cache-sensitive applications, so that the competition among the applications, which would heavily penalize the target application, can result effectively prevented by *Rainbow*. After all the Red applications have been chosen, then the Yellow ones are chosen as "polluters" and finally the Green ones are used.

The experimental methodology consists in running all the applications of each workload simultaneously, each application being kept on a specific core and the target application being colored. In case a polluter terminates before the target application, it is immediately restarted.

The measurements with the whole cache assigned to the target applications have been skipped, because in such situation *Rainbow* would allocate pages for all the applications from all the colors, thus denying isolation. The whole experiment is repeated 10 times, and the results are shown in Figure 11, where the dashed lines report the profile of the target application when running alone (those in Figure 10). As visible from experiments, the curves of the co-located applications roughly follow those of the solo execution and in some cases there is a good overlapping between the LLC miss rate curves. This fact indicates that *Rainbow* is effectively able to reserve an LLC partition and thus make the application benefit from isolation. The distance between the slowdown curves is instead due to the non-partitionable resources, in particular the on-chip bandwidth and the memory controller.

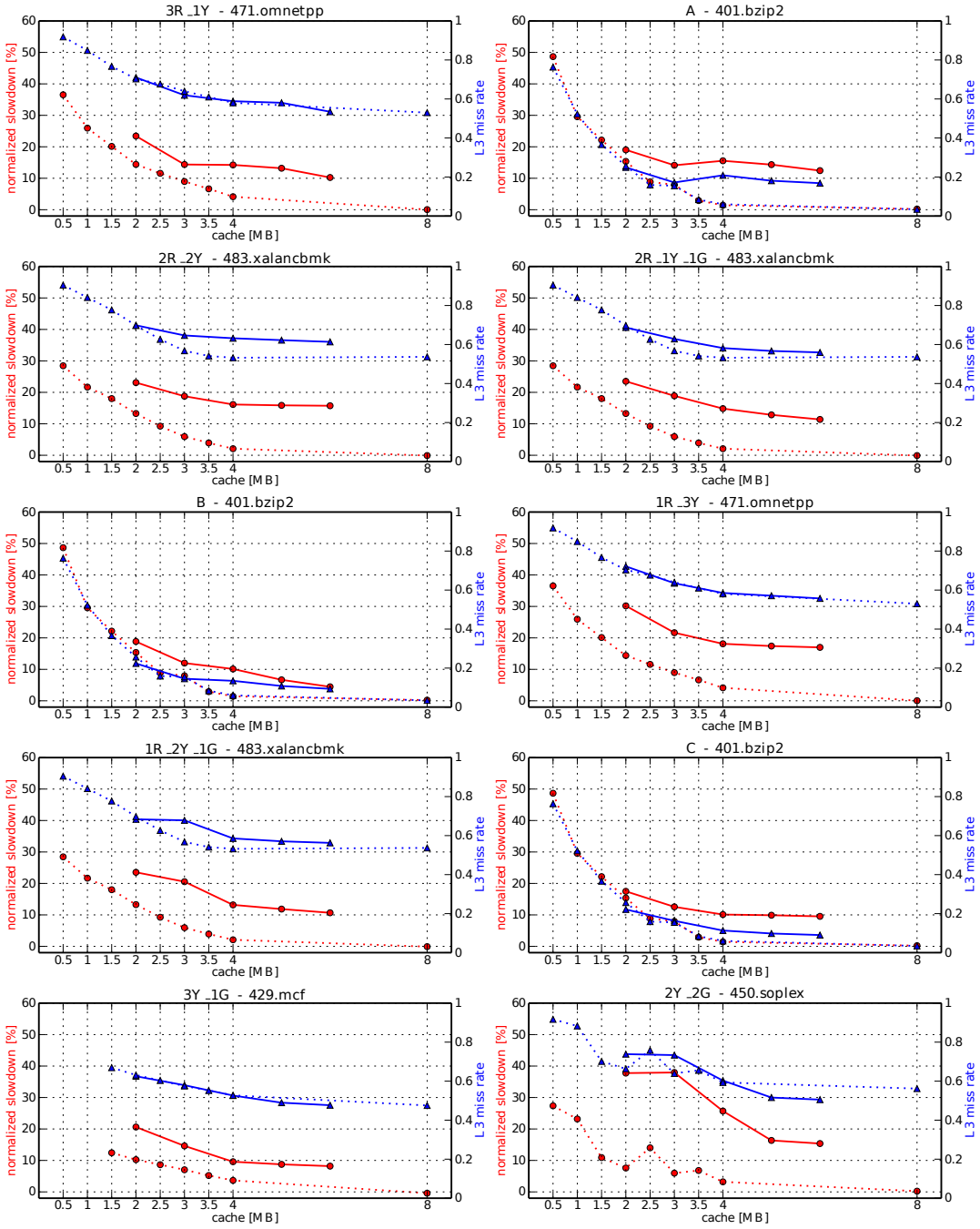


Figure 11: Workloads profiles with different cache partitions.
For each workload, the target application is shown. The continuous line indicates the workload, the dashed line the target application profile from Figure 10

CHAPTER 7

CONCLUSIONS

Here we present the final considerations on the work done with this first version of *Rainbow*, a modification of the Linux kernel to implement page coloring in a transparent and flexible way. *Rainbow* has effectively succeeded in controlling the execution of co-located applications, even in an environment with high resources contention.

In Section 7.1, we discuss the contribution of this work together with its limits, while in Section 7.2, we discuss some possible future works which could derive from *Rainbow*.

7.1 Contributions and limits

The main, novel contribution of this work is the careful identification of the constraints posed by page coloring and by applying it to the commodity CMPs, in particular of those explained in Section 4.4 and Section 4.3.2. At the best of our knowledge, the possible overlap between the LLC color bits and the set number bits of the other cache layers has been ignored so far, as the previous works partitioned an L2 cache and neglected this issue.

The contextualization of this work inside the emerging area of cloud platforms drove us to the identification of other consequences of page coloring which could have deep implications for cloud computing architectures. On one side, the benefits of page coloring for isolating VMs are currently being investigated by the research community; this work shows that this technique can be effective also with a deeper cache hierarchy. On the other side, we have identified limitations

of page coloring which could potentially harm its effectiveness in the cloud environment, like a limitation in the availability of physical memory for colored applications and the renounce to hugepages, which have been shown to bring benefits with VMs.

A major limit of this work is the evaluation done with one benchmark suite only; although it was fundamental to prove *Rainbow* effectiveness with a CPU-intensive workload, investigating its possible benefits with a more varied set of tests is a fundamental step towards the a "real world usage"; in particular, cloud-like applications involving I/O and multi-threaded applications would be good steps to try *Rainbow* "in the cloud".

Another limit is that the impact of renouncing to hugepages has not been studied; this lack is due to the fact that our benchmark suite does not use this feature. Measuring the impact of hugepages requires indeed applications that specifically benefit from this feature, which are rare in practice.

7.2 Future work

The most immediate step is a wider testing with a large spectrum of applications, ranging batch (e.g., Hadoop) to latency-sensitive applications (e.g., Web search). In particular, VMs are a very interesting target for future experiments, where *Rainbow* could add one more knob to enhance the isolation they already provide to applications. It would be particularly interesting to test VMs running applications that are typical of a cloud platform, as this scenario would closely simulate the way cloud platforms are actually managed. Here too, a reasonable mix of applications would be needed, with a huge number of possible scenarios, from single-threaded batch-like applications to multi-threaded web services. An extreme research scenario in this

area would be performing 2-levels LLC partitioning: the first level runs in the hypervisor and partitions the LLC among the VMs, while the second level runs inside each VM and sub-partitions the LLC space devoted to the each guest.

Focusing on more specific scenarios, the cost of renouncing to hugepages can bring a deeper insight on how applications contend on resources, in this case the LLC against the TLB. Another interesting research in this area is investigating how the guest OS impacts on the applications performance, for example by investigating the miss rate with and without a guest OS.

Pushing forward the employment of page coloring, another possible work is overcoming the limitations posed by the static LLC partition assignment currently implemented by using page re-coloring. This research requires an investigation of the cost of re-coloring and is open to intuitions and optimizations aimed at decreasing the huge overhead of this operation. At the same time, investigating the policies which drive re-coloring and their possible uses of the performance monitoring facilities current architectures offer is another challenging task.

Another suggestion of this work is the possibility of partitioning the lower layers of caches through the LLC, which could be a novel idea for SMT architectures and is currently a completely unexplored area. Due to the sharing of the L1 cache by the threads running on the same physical core, contention on this layer is likely to occur, and should be evaluated.

Finally, the implementation of *Rainbow* starting from the Linux kernel suggests a tighter integration with the Linux components and philosophy. *Rainbow* could be ported and tested on other architectures like ARM and IBM Power, which are becoming increasingly important

in the fields of cloud computing and of High Performance Computing. Moreover, it can be managed through a clearer interface like that offered by cgroups [54], by which users could define constraints over a hierarchical set of processes in terms of cores, CPU time, memory, I/O etc.; adding another option like a "reserved LLC cache pool" could be an appreciable feature. This possibility would integrate LLC partitioning with resource scheduling, thus making scheduling itself even more effective and resulting particularly suited for the careful management of many resources at once, an appealing aspect for cloud environments.

CITED LITERATURE

1. Wang, L., Laszewski, G., Younge, A., He, X., Kunze, M., Tao, J., and Fu, C.: Cloud computing: a perspective study. New Generation Computing, 28(2):137–146, 2010.
2. Pastaki Rad, M., Sajedi Badashian, A., Meydanipour, G., Ashurzad Delcheh, M., Alipour, M., and Afzali, H.: A survey of cloud platforms and their future. In Computational Science and Its Applications ICCSA 2009, eds. O. Gervasi, D. Taniar, B. Murgante, A. Lagan, Y. Mun, and M. Gavrilova, volume 5592 of Lecture Notes in Computer Science, pages 788–796. Springer Berlin Heidelberg, 2009.
3. Cusumano, M.: Cloud computing and saas as new computing platforms. Commun. ACM, 53(4):27–29, April 2010.
4. Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J., and Brandic, I.: Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Gener. Comput. Syst., 25(6):599–616, June 2009.
5. Patel, P., Ranabahu, A., and Sheth, A.: Service level agreement in cloud computing. 2009.
6. Goiri, I. n., Julià, F., Fitó, J. O., Macías, M., and Guitart, J.: Resource-level qos metric for cpu-based guarantees in cloud providers. In Proceedings of the 7th International Conference on Economics of Grids, Clouds, Systems, and Services, GECON'10, pages 34–47, Berlin, Heidelberg, 2010. Springer-Verlag.
7. Barroso, L. A. and Hölzle, U.: The case for energy-proportional computing. Computer, 40(12):33–37, December 2007.
8. Moore, G. E.: Cramming More Components Onto Integrated Circuits. Electronics, 1965.
9. Hemani, A., Jantsch, A., Kumar, S., Postula, A., Oberg, J., Millberg, M., and Lindqvist, D.: Network on chip: An architecture for billion transistor era. In Proceeding of the IEEE NorChip Conference, volume 31, 2000.
10. Tang, L., Mars, J., Vachharajani, N., Hundt, R., and Soffa, M. L.: The impact of memory subsystem resource sharing on datacenter applications. SIGARCH Comput. Archit. News, 39(3):283–294, June 2011.
11. Alhamad, M., Dillon, T., and Chang, E.: Conceptual sla framework for cloud computing. In Digital Ecosystems and Technologies (DEST), 2010 4th IEEE International Conference on, pages 606–610, 2010.
12. Vecchiola, C., Pandey, S., and Buyya, R.: High-performance cloud computing: A view of scientific applications. In Pervasive Systems, Algorithms, and Networks (ISPAN), 2009 10th International Symposium on, pages 4–16. IEEE, 2009.

13. Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E.: Bigtable: A distributed storage system for structured data. ACM Trans. Comput. Syst., 26(2):4:1–4:26, June 2008.

14. Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I.: Spark: cluster computing with working sets. In Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, pages 10–10, 2010.

15. Dean, J. and Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. Commun. ACM, 51(1):107–113, January 2008.

16. Evangelinos, C. and Hill, C. N.: Cloud computing for parallel scientific hpc applications: Feasibility of running coupled atmosphere-ocean climate models on amazons ec2. In In The 1st Workshop on Cloud Computing and its Applications (CCA, 2008.

17. Ostermann, S., Iosup, A., Yigitbasi, N., Prodan, R., Fahringer, T., and Epema, D.: A performance analysis of ec2 cloud computing services for scientific computing. In Cloud Computing, pages 115–131. Springer, 2010.

18. Wan, Z.: Sub-millisecond level latency sensitive cloud computing infrastructure. In Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT), 2010 International Congress on, pages 1194–1197, 2010.

19. Srikantaiah, S., Kansal, A., and Zhao, F.: Energy aware consolidation for cloud computing. In Proceedings of the 2008 Conference on Power Aware Computing and Systems, HotPower'08, pages 10–10, Berkeley, CA, USA, 2008. USENIX Association.

20. Mars, J., Tang, L., Hundt, R., Skadron, K., and Soffa, M. L.: Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-44 '11, pages 248–259, New York, NY, USA, 2011. ACM.

21. Berl, A., Gelenbe, E., Di Girolamo, M., Giuliani, G., De Meer, H., Dang, M. Q., and Pentikousis, K.: Energy-efficient cloud computing. The Computer Journal, 53(7):1045–1051, 2010.

22. Jung, G., Hiltunen, M., Joshi, K., Schlichting, R., and Pu, C.: Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures. In Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on, pages 62–73, 2010.

23. Carvalho, C.: The gap between processor and memory speeds. In Proc. of IEEE International Conference on Control and Automation, 2002.

24. Fedorova, A., Blagodurov, S., and Zhuravlev, S.: Managing contention for shared resources on multicore processors. Commun. ACM, 53(2):49–57, February 2010.

25. Octeon processors family by cavium networks, September 2004. `http://www.cavium.com/newsevents_octeon_cavium.html`.

26. Cook, H., Moreto, M., Bird, S., Dao, K., Patterson, D. A., and Asanovic, K.: A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In Proceedings of the 40th Annual International Symposium on Computer Architecture, ISCA '13, pages 308–319, New York, NY, USA, 2013. ACM.

27. Denning, P. J.: Thrashing: Its causes and prevention. In Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I, AFIPS '68 (Fall, part I), pages 915–922, New York, NY, USA, 1968. ACM.

28. Knowlton, K. C.: A Fast Storage Allocator. CACM, 1965.

29. Korn, D. G. and Vo, K.-P.: In search of a better malloc. In USENIX Summer, pages 490–506, Portland, Oregon, USA, June 1985.

30. First buddy half optimisation. `https://www.kernel.org/pub/linux/kernel/people/akpm/patches/2.6/2.6.7/2.6.7-mm1/broken-out/buddy-reordering.patch`.

31. Lu, H., Seth, R., Doshi, K., and Tran, J.: Using hugetlbfs for mapping application text regions. In Proceedings of the Linux Symposium, volume 2, pages 75–82, 2006.

32. Zhang, P., Li, B., Huo, Z., and Meng, D.: Evaluating the effect of huge page on large scale applications. In Networking, Architecture, and Storage, 2009. NAS 2009. IEEE International Conference on, pages 74–81, 2009.

33. Tang, L., Mars, J., and Soffa, M. L.: Contentiousness vs. sensitivity: Improving contention aware runtime systems on multicore architectures. In Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era, EXADAPT '11, pages 12–21, New York, NY, USA, 2011. ACM.

34. Kang, M., Kang, D.-I., Crago, S. P., Park, G.-L., and Lee, J.: Design and development of a run-time monitor for multi-core architectures in cloud computing. Sensors, 11(4):3595–3610, 2011.

35. Bell, S., Edwards, B., Amann, J., Conlin, R., Joyce, K., Leung, V., MacKay, J., Reif, M., Bao, L., Brown, J., et al.: Tile64-processor: A 64-core soc with mesh interconnect. In Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International, pages 88–598. IEEE, 2008.

36. Gong, Z. and Gu, X.: Pac: Pattern-driven application consolidation for efficient cloud computing. In Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on, pages 24–33. IEEE, 2010.

37. Chen, S., Gibbons, P. B., Kozuch, M., Liaskovitis, V., Ailamaki, A., Blelloch, G. E., Falsafi, B., Fix, L., Hardavellas, N., Mowry, T. C., and Wilkerson, C.: Scheduling threads for constructive cache sharing on cmps. In Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '07, pages 105–115, New York, NY, USA, 2007. ACM.

38. Sharifi, A., Srikantaiah, S., Kandemir, M., and Irwin, M. J.: Courteous Cache Sharing: Being Nice to Others in Capacity Management. In Proceedings of the 49th Annual Design Automation Conference, 2012.

39. Qureshi, M. K. and Patt, Y. N.: Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In Proc. of MICRO, 2006.

40. Seshadri, V., Mutlu, O., Kozuch, M. A., and Mowry, T. C.: The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing. In Proc. of PACT, 2012.

41. Mitzenmacher, M.: Compressed bloom filters. IEEE/ACM Trans. Netw., 10(5):604–612, October 2002.

42. Sanchez, D. and Kozyrakis, C.: Vantage: Scalable and Efficient Fine-Grain Cache Partitioning. In Proc. of ISCA, 2011.

43. Sanchez, D. and Kozyrakis, C.: The ZCache: Decoupling Ways and Associativity. In Proc. of MICRO, 2010.

44. Bray, B. K., Lunch, W. L., and Flynn, M. J.: Page allocation to reduce access time of physical caches. Technical report, 1990.

45. Kessler, R. E. and Hill, M. D.: Page placement algorithms for large real-indexed caches. ACM Trans. Comput. Syst., 10(4):338–359, November 1992.

46. Ding, X., Wang, K., and Zhang, X.: SRM-Buffer: An OS Buffer Management Technique to Prevent Last Level Cache from Thrashing in Multicores. In Proc. of EuroSys, 2011.

47. Kim, J., Jeong, J., Kim, H., and Lee, J.: Explicit Non-reusable Page Cache Management to Minimize Last Level Cache Pollution. In Proc. of ICCIT, 2011.

48. Soares, L., Tam, D., and Stumm, M.: Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer. In Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture, pages 258–269. IEEE Computer Society, 2008.

49. Tam, D., Azimi, R., Soares, L., and Stumm, M.: Managing Shared L2 Caches on Multicore Systems in Software. In Proc. of WIOSCA, 2007.

50. Zhang, X., Dwarkadas, S., and Shen, K.: Towards Practical Page Coloring-based Multi-core Cache Management. In Proc. of EuroSys, 2009.

51. Lin, J., Lu, Q., Ding, X., Zhang, Z., Zhang, X., and Sadayappan, P.: Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems. In Proc. of HPCA, 2008.

52. Jin, X., Chen, H., Wang, X., Wang, Z., Wen, X., Luo, Y., and Li, X.: A Simple Cache Partitioning Approach in a Virtualized Environment. In Proc. of ISPA, 2009.

53. Wang, X., Wen, X., Li, Y., Luo, Y., Li, X., and Wang, Z.: A dynamic cache partitioning mechanism under virtualization environment. In Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on, pages 1907–1911. IEEE, 2012.

54. Control group. https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.

55. Intel® hyper-threading technology. http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html.

56. Intel® turbo boost technology. http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html.

57. Spec cpu2006. http://www.spec.org/cpu2006.

58. Performance analysis tools for linux. http://lxr.free-electrons.com/source/tools/perf/Documentation/perf.txt?v=3.9.

59. Henning, J. L.: Spec cpu2006 benchmark descriptions. SIGARCH Comput. Archit. News, 34(4):1–17, September 2006.

VITA

Alberto Scolari

Education

B.S., Engineering of Computing Systems
Politecnico di Milano
2011

M.S., Computer Science (current)
University of Illinois at Chicago, Chicago, IL
May 2014

Publications

Alberto Scolari, Filippo Sironi, Davide B. Bartolini, Donatella Sciuto, Marco D. Santambrogio. **Coloring the Cloud for Predictable Performance**. In *Accepted Posters of the 2013 ACM Symposium on Cloud Computing* , Santa Clara, California, USA, October 1-3, 2013