Politecnico di Milano

Facoltà di Ingegneria Industriale e dell'Informazione

LAUREA SPECIALISTICA IN INGEGNERIA INFORMATICA

An efficient software cache hierarchies partitioning system for predictable performance

Master of Science thesis of:
Alberto Scolari
Matricola 783795

Advisor:
Prof. Marco D. Santambrogio

Co-advisor:
Dott. Davide B. Bartolini

08/04/2015

Acknowledgments

hic illum vidi iuvenem, Meliboee, quotannis
bis senos cui nostra dies altaria fumant,
hic mihi responsum primus dedit ille petenti:
"pascite ut ante boves, pueri, submittite tauros".

Virgil, Bucolics I

In addition to the friends and the relatives who supported me during these latest years, some people gave a major contribution to this work: these few words are only a part of my thankfulness to them. My first thanks are for Filippo Sironi and Davide Bartolini, with whom I had the privilege of working at the beginning of this thesis. In addition to fundamental suggestions and guidance, they gave me confidence, support and a precious example of competence and devotion to the work.

Another thank is for Marco Rabozzi, whose talent for modeling and rationality helped me to organize ideas and measurements.

Last but not least, I am pleased to thank my advisor, Marco Santambrogio, for his guidance, his continuous incitements and his example: to push the best out of us, he always demands the best out of himself.

Abstract

The advent of Chip-MultiProcessor (CMP) architectures in computing platforms allows the co-location of applications, which run simultaneously on the same chip. These architectures are at the base of distributed computing platforms, and in particular of today's cloud environments. These environments manage a diverse and hardly predictable workload, which causes computational resources to experience increasing phenomena of contention, as applications running on the different cores may interfere with each other in using several hardware resources. Therefore, isolation of applications becomes a key aspect to ensure performance and Quality of Service (QoS) in such environments. Among various components, the Last Level Cache (LLC) is one of the resources where contention is experienced most, and is fundamental to ensure applications' performance. Although contention is a well known phenomenon in the research, even the most recent commodity CMPs do not provide effective mechanisms to alleviate it, and no widely accepted solution yet exists.

Page coloring is a technique well known in the literature that allows partitioning the LLC of 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 and introduced important modifications, while how page coloring works on these cache hierarchies has not been studied in detail. This work aims to investigate the possible advantages, limitations and trade-offs that derive from the usage of page coloring on such architectures. In particular, recent CMPs by Intel, namely those of the Sandy Bridge family, adopt a hash-based LLC addressing scheme. This addressing scheme changes how data are mapped to the LLC and, consequently, the effectiveness of page coloring. Considering such changes, this work aims to adapt page coloring to the latest Intel's architecture, which power cloud platforms.

In our vision, cloud workloads can benefit from page coloring, leveraging isolation of applications in LLC to fulfill QoS requirements. To prove our vision, we realized *Rainbow*, an implementation of page coloring in the Linux kernel that partitions modern LLCs for user-defined sets of applications. We evaluated *Rainbow* with a set of computational-intensive benchmarks to show its ef-

fectiveness, and finally discussed the achievements of this work, its limitations and possible future research.

This work is organized as follows:

- chapter 1 provides a high-level view of cloud platforms and of the main issues they face to fulfill users' requirements, while, at the same time, optimizing the utilization of their computational resources
- chapter 2 provides the necessary background, explaining the architecture of modern LLCs; it also shows how controlling physical memory allows controlling the LLC and, consequently, how the software layer manages physical memory
- chapter 3 shows the techniques to obtain isolation in the LLC, offering a general view of the state of the art and of actual unsolved issues
- chapter 4 discusses the design of *Rainbow* in the context of the modern cache hierarchies, focusing on the the recent Intel's architectures
- chapter 5 investigates Intel's Sandy Bridge architecture by reconstructing the hash function of these CMPs
- chapter 6 explains how *Rainbow* is implemented within the Linux kernel, based on the design in chapter 4 and on the findings of chapter 5
- chapter 7 reports the results obtained running *Rainbow* with a set of benchmarks, showing how it can improve isolation and provide strict guarantees on applications' performance
- chapter 8 discusses the results, the limits and possible work deriving from the proposed solution, in the context of cloud platforms.

Sommario

L'avvento delle architetture Chip-MultiProcessor (CMP) nelle piattaforme di computazione permette la co-locazione di applicazioni, che eseguono contemporaneamente sullo stesso chip. Queste architetture sono alla base delle piattaforme di calcolo distribuito, in particolare dei recenti ambienti di cloud-computing. Questi ambienti gestiscono un insieme di applicazioni variegato e difficilmente predicibile, che causa crescenti fenomeni di contesa sulle risorse condivise, dal momento che le applicazioni che eseguono su cores differenti possono interferire fra di loro nell'utilizzo di diverse risorse hardware. Perciò, l'isolamento delle applicazioni diventa un aspetto fondamentale per assicurare le prestazioni e la qualità del servizio, o Quality of Service (QoS), in tali ambienti. Fra vari componenti, la cache di ultimo livello, o Last Level Cache (LLC), è una delle risorse più contese, ed è fondamentale per assicurarne le prestazioni delle applicazioni. Nonostante tale fenomeno sia ben noto nella ricerca, anche i più recenti CMP comunemente reperibili sul mercato server non offrono meccanismi efficaci per alleviarla, e nessuna soluzione con largo consenso esiste ancora.

Page coloring è una tecnica ben nota nella letteratura che permette di partizionare ls LLC dei comuni processori server. Questa tecnica sfrutta il posizionamento dei dati nella memoria centrale per controllare la loro posizione nella LLC. L'evoluzione delle architetture CMP ha accresciuto il numero di livelli di cache e introdotto importanti modifiche, mentre il funzionamento del page coloring su tali gerarchie di cache non è stato studiato dettagliatamente. Questo lavoro si prefigge di investigare i possibili vantaggi, limitazioni e compromessi che derivano dall'utilizzo del page coloring su queste architetture. In particolare, i recenti CMPs di Intel, ossia quelli della famiglia Sandy Bridge, adottano un indirizzamento della LLC basato su hash. Questo schema di indirizzamento cambia come i dati vengono mappati nella LLC e, di conseguenza, l'efficacia del page coloring. Considerando tali cambiamenti, questo lavoro si prefigge di adattare il page coloring alle più recenti architetture Intel, che equipaggiano le piattaforme cloud.

Secondo la nostra visione, le applicazioni cloud possono trarre beneficio dal page coloring, sfruttando l'isolamento delle applicazione nella LLC per soddisfare requisiti di QoS. Per dimostrare la nostra intuizione, abbiamo realizzato *Rainbow*, un'implementazione del page coloring per il kernel Linux che partiziona le moderne

LLCs per insiemi di applicazioni definiti dall'utente. Abbiamo valutato *Rainbow* con un insieme di applicazioni di riferimento computazionalmente intensive per mostrare la sua efficacia, per discutere infine i risultati di questo lavoro, i limiti e possibilità di ricerca futura.

Questo lavoro è organizzato come segue:

- il capitolo 1 dà una visuale di alto livello delle piattaforme cloud e dei maggiori problemi che queste devono affrontare per soddisfare le richieste degli utenti, ottimizzando contemporaneamente l'utilizzo delle risorse computazionali
- il capitolo 2 fornisce le conoscenze necessarie, spiegando l'architettura delle moderne LLCs; inoltre, mostra come controllare la memoria fisica permette di controllare la LLC e di conseguenza, come lo strato software gestisce la memoria fisica
- il capitolo 3 mostra le tecniche per ottenere isolamento nella LLC, offrendo una visione generale dello stato dell'arte e delle attuali problematiche irrisolte
- il capitolo 4 discute le scelte progettuali alla base di *Rainbow* nel contesto delle moderne gerarchie di cache, con particolare riferimento alle recenti architetture Intel
- il capitolo 5 approfondisce l'architettura Intel Sandy Bridge ricostruendo la funzione di hash di tali CMPs
- il capitolo 6 spiega come *Rainbow* viene implementato nel kernel Linux, basandosi sulle linee guida progettuali esposte in nel capitolo 4 e sui risultati del capitolo 5
- il capitolo 7 riporta i risultati ottenuti eseguendo *Rainbow* con un insieme di applicazioni di riferimento, mostrando come esso può migliorare l'isolamento e fornire rigorose garanzie sulle prestazioni delle applicazioni
- infine, il capitolo 8 discute i risultati, i limiti e possibili lavori futuri che derivano dalla soluzione proposta, nel contesto delle piattaforme cloud

Contents

List of Tables

List of Figures

Introduction and motivations 1

S'io credesse che mia risposta fosse
a persona che mai tornasse al mondo,
questa fiamma staria senza piu scosse.
Ma perciocché giammai di questo fondo
non tornò vivo alcun, s'i'odo il vero,
senza tema d'infamia ti rispondo.

Dante Alighieri, Divina Commedia, Inferno XXVI

This chapter explains the context of this research and presents the objective of our work. In particular, section 1.1 outlines the problem in the current scenario of Information Technology (IT), where the cloud computing paradigm is actually causing a shift towards different computational and economical models than those adopted so far. Then, section 1.2 focuses on cloud platforms and gives an overview of the goals that drive the management of computational resources and of the challenges the providers face, highlighting the issues that are still unsolved with in the context of Chip-MultiProcessor (CMP) architectures. Following on, section 1.3 focuses on the occurrence of contention in the Last Level Cache (LLC) even with the latest commodity CMPs and its important consequences for providers and users. Finally, section 1.4 outlines the solution we propose, its main guidelines and its evaluation.

1.1 The context of cloud platforms

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* [92, 65, 58], where external providers offer IT services to final users, who have access to computing resources only via a network. On the software side, cloud platforms offer a wide variety of services, from direct access to a physical machine to high-level Application Programming Interfaces (APIs) [23, 10]. In cloud scenarios, the hardware and software deployments are heterogeneous, as well as the various services these platforms offer to multiple categories of users, from single individuals to big companies [70]. Therefore, these platforms are inherently complex, and must fulfill multiple goals such as security, scalability, cost-effectiveness, etc. With the differentiation of cloud applications, the market of computing services introduced Service Level Agreements (SLAs) between providers and users [10, 66], which formalize requirements and guarantees about a certain cloud service into a contract between the parties. SLAs describe the technical parameters the service will provide in terms of Quality of Service (QoS) requirements: for example, a service can guarantee an upper bound on the Mean Time Between Failures (MTBF) or on the Turn-Around Time (TAT), an average throughput or other high-level metrics [31]. On the provider's side, the management of cloud platforms pursues different goals, such as platform utilization and low energy consumption [4]. Thus, providers' and users' goals are often in conflict, and resource provisioning for cloud services is an open field of research [12].

The major computational resources in typical cloud environments are Central Processing Units (CPUs), in particular CMPs, which became the standard computational resources for servers and workstations over the past years. These architectures rely mainly on Thread Level Parallelism (TLP) to increase the performance, in accordance with Moore's law [60], and are the actual and future trend [35]. These architectures allow co-location of different tasks on the same chip, achieving a higher computational density than previous single-threaded CPUs. Therefore, cloud environments largely employ CMPs in order to keep the utilization of the infrastructure high.

Yet, in actual commodity CMP architectures some resources can be explicitly managed from software (like cores), while others cannot, their control being exclusively up to the hardware. Thus, some of these resources are implicitly shared by running applications, with potentially detrimental effects. Contention on shared computational resources, such as the memory controller, the on-chip interconnection and the caches, limits the performance applications can exploit [49]. 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 [85]. Hence, isolation of shared resources has become a fundamental aspect for efficient computing infrastructures.

This thesis focuses on the LLC, a shared resource that is fundamental for the performance of compute-intensive applications. Many research works have investigated contention on the LLC and proposed 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 nor by production-ready software environments. 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 the aspects of cloud platforms that are of interest for this work. Section 1.2.1 shows the main parameters to evaluate the services cloud environments provide and shows a simple classification of cloud applications with respect to their requirements. Then, section 1.2.2 introduces the main issues due to resource sharing.

1.2.1 QoS for cloud applications

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 [2, 89, 88]; these agreements specify QoS parameters that measure the quality of the service. These parameters often are of interest for both technical and non-technical users, and 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 MTBF, throughput and response time. Some parameters are related to the availability of the service, while others are related to its performance.

In particular, performance requirements are experiencing an increasing demand [96, 81] and bring novel challenges to light. For example, latency is becoming a key requisite for user-facing applications, and providers often employ this metric to characterize the QoS of their service.

With respect to performance-related QoS, applications can be classified in two broad categories: *batch* applications and *latency-sensitive* applications. 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. A notorious example of batch application is MapReduce [24], but more and more applications with batch characteristics are being implemented for cloud platforms, also from the scientific world [27, 63, 89].

Instead, latency-sensitive applications have different QoS parameters to be satisfied. For example, Spark, a framework for machine learning [99], can be assigned latency constraints because it must respond to interactive queries. Typically, cloud platforms measure the time needed to service a request, from the instant it reaches the datacenter to the instant it exits. To measure and control the service QoS, previous work [3, 42] has shown that the 95-th (or 99-th) percentile of the latency distribution is a good proxy for its QoS, while the mean can be misleading. Using the percentile ensures high confidence in evaluating the QoS of a service, imposing strict requirements to the provider. Instead, the mean does not take in account the statistical distribution of the latency, which can have long "tails". Latency requirements are currently spreading also on common services like web search, e-mail and online gaming[56]. Previous research showed that, in general, a platform can guarantee very low latency if the usage is low [91].

1.2.2 The provider's view: resource provisioning

Unlike users, a cloud provider attempt to optimize different parameters, with goals that conflict with the user's objectives. In particular, parameters like energy consumption and utilization are at the base of the management of cloud infrastructures and drive the design choices of providers [6, 44, 79]. Nonetheless, keeping utilization high while meeting sufficient performance levels is still a big challenge, and the research is very live on this topic. For example, Barroso and Hölzle [4] show that the utilization of data centers computing resources seldom goes above 20%.

A first solution to maintain a sufficient level of performance is over-provisioning of resources, a solution that has, yet, a very high infrastructural cost (server purchase, energy, etc.). On the contrary, decreasing the provisioning diminishes infrastructural cost, but implies the consolidation of more tasks onto a limited number of machines, sharing hardware resources. This sharing, in turn, leads to contention on those hardware resources that are not partitionable. Among all the shared resources (like disk bandwidth, network bandwidth, etc.), computational resources are the key of many applications. At the heart of them, CMPs architectures are a critical component where contention occurs, despite the continuous progresses in manufacturing technology and design. Indeed, while the computational power can be partitioned by core assignment and by time sharing (a long-living, well-known mechanism), other resources like the memory bandwidth and the LLC are not partitionable by design. Therefore, co-location of more tasks on a single CMP can cause considerable slowdowns [79], which can be intolerable for applications with strict requirements. This forces cloud providers to limit or even to avoid co-location at all [56].

1.3 Contention on the Last Level Cache of modern architectures

The employment of CMPs in cloud platforms, although it allows the consolidation of more applications and thus a potentially higher utilization, forces the sharing of several resources, posing novel challenges the research is tackling. In particular, the LLC has a central role in ensuring applications' performance, but is transparent in today's commodity CMPs and is subject to unpredictable contention.

A wide literature already discusses the occurrence of contention on the LLC [28, 56], also within multi-tenants cloud environments [34, 61, 85, 33]. The occurrence of contention is particularly detrimental in these environments since it affects two aspects that are of fundamental importance. The first aspect is the accounting policy: cloud platforms pro-

vide pay-per-use services, where users typically pay for the amount of resources they rent over time (CMP cores, memory, bandwidth, etc.), expecting a certain, proportional performance. Nevertheless, “transparent” resources such al the LLC have an important role with respect to final performance, and contention may cause the final performance to be different from the expected one, leading to unfair pricing policies. This contention, which depends on how the provider consolidates the workloads, requires novel prediction models [33] and accounting policies.
The second aspect pertains QoS requirements, and has already been introduced in section 1.1. In presence of contention, a provider cannot guarantee a priori a certain QoS without resorting upon over-provisioning [56], thus with additional costs.

Despite the vast literature upon contention on the LLC, researchers and manufacturers did not found a definitive way to solve it, nor an effective and widely acknowledged way to alleviate it. This reflects to CMP architectures, which still allow the occurrence of noticeable contention phenomena. Moreover, economical and legacy reasons prevent deep changes in CMPs, so that these issue are likely to remain unsolved for the upcoming years.
In particular, the latest commodity CMPs by Intel, namely the Nehalem and Sandy Bridge families, which power most of the cloud infrastructures, do not offer any feature to tackle this issue. Therefore, the research has been actively working on a software technique called *page coloring*, which exploits the Phisically Indexed, Physically Tagged (PIPT) data mapping of modern LLCs [8] to partition the LLC and ensure isolation, as section 3.3 explains. Nonetheless, Intel’s Sandy Bridge family introduces a hash-based addressing scheme [52], which changes the PIPT mapping of data to the cache and makes “classical” page coloring infeasible.

1.4 Proposed solution

Addressing this lack, this work proposes to make page coloring viable also on the Sandy Bridge architecture and onto a modern hardware/-software stack, so that modern cloud infrastructures can leverage the benefits of isolation.

Recent CMPs have deep memory hierarchies to hide the main memory latency and offer new functionalities that conflict with page coloring. Therefore, this work deeply studies the consequences of these features with respect to page coloring and which restrictions derive from modern architectures, and provides the enabling technology to obtain LLC partitioning also on recent architectures.
Since we aim to design a software LLC-partitioning capability, we will integrate it within the physical memory management mechanism of recent Linux kernels. These kernels are the base of cloud platforms deployments, and manage physical memory via the *buddy algorithm*. Taking in account the main goals of this algorithm and its implementation details, we will show how our extension, called *Rainbow*, enhances it with page coloring capabilities in an efficient and non-disruptive way. This implementation will require the knowledge of the Sandy Bridge hash function, whose form will affect our design. Nonetheless, we will discuss how to generalize our findings. Since also older architectures, with a “classical” PIPT-based LLC addressing, are still used, we will design *Rainbow* so that its basic mechanisms and its implementation can fit both Sandy Bridge and Nehalem architectures. This choice generalizes page coloring to a wide variety of recent and legacy architectures, the formers represented by Sandy Bridge and the latters by Nehalem. Moreover, Nehalem’s addressing scheme is similar to that of many different architectures like SPARC or ARM, potentially extending *Rainbow*’s viability also on these platforms, which are nowadays appearing to the market of cloud platforms.

To provide an easy interface to isolate applications in LLC, we will also add a suitable interface, following the recent guidelines of Linux design and the best use practices of large computing environments. Based on these guidelines, we will implement a cgroup interface [15] to expose *Rainbow*’s capabilities to userspace, allowing the platform manager to handle the co-location of applications to different LLC partitions.

Finally, we will thoroughly evaluate *Rainbow* with reference benchmarking applications from the Standard Performance Evaluation Corporation (SPEC) suite [78], showing how *Rainbow* allows the platform manager to give an application a suitable LLC partition based on high-level policy and on QoS requirements. With *Rainbow*’s technology, users will be able to alleviate contention on the LLC for any set of applications of their choice, enhancing performance isolation and predictability.

To explain the relevant work in the field and our own work, this thesis is organized as follows.

- chapter 2 explains the necessary background knowledge to understand the architecture of modern CMPs, focusing on the LLC, and the functioning of the buddy algorithm, where *Rainbow* will integrate
- chapter 3 provides an overview of the state of the art, showing diverse solutions and highlighting unsolved issues with respect to LLC contention
- chapter 4 explains the design of *Rainbow*, showing how it specifically take in account modern cache hierarchies, in particular those of recent Intel's architectures Nehalem and Sandy Bridge, and how its capabilities are exposed
- chapter 5 shows a repeatable methodology to reconstruct the hash function of a Sandy Bridge CMP, providing insights on its physical layout
- chapter 6 explains the main implementation details of *Rainbow*, following the concepts of chapter 4 and the findings of chapter 5
- chapter 7 evaluates *Rainbow*'s effectiveness and highlights its achievements and limitations in practice
- chapter 8 provides a more general view over the results of this work and over its limitation, highlighting possible future work and application scenarios.

Background 2

In this chapter, we introduce the hardware and software components this thesis is based upon, in order to give the reader the needed background. Therefore, after a review of the motivations and of the structure of caches section 2.1, section 2.2 explains contention phenomena by discussing their sources and their consequences. Then, section 2.3 shows the details of the Nehalem [17][18, section 2.4] and Sandy Bridge [18, section 2.2] CPU architectures by Intel, which well represent the CMPs available in today's server platforms and are thus the reference platforms for the purposes of this thesis.
Given the importance of physical memory allocation as in section 2.1, section 2.4 presents the goals and the functioning of the physical memory allocator of the Linux kernel, which will be at the heart of the proposed solution. Finally, section 2.5 explains a recent hardware capability that allows CMPs to easily manage large portions of physical memory, which is affecting the design of modern Operating Systems (OSs) and is to be considered in the context of this thesis.

2.1 Motivations and structure of modern caches

This section gives an overview of the structure of a modern CPU cache, starting from the basic reasons that led to the introduction of this component (section 2.1.1), explaining its structure, functioning and types (sections 2.1.2 to 2.1.5) and finally showing how multiple caches are organized into hierarchies within modern CMPs (section 2.1.6).

2.1.1 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 [36]. In particular, also the difference between these two speeds grew up exponentially, in favor of the CPU. This ever-increasing difference created over the years the so-called *processor-memory gap* [13], forcing the CPU to wait for the completion of load and store memory operations and causing long idle intervals that limit the overall Instructions Per Second count (IPC) and decrease performance. Hence, the need of overcoming this bottleneck emerged, and designers chose to introduce an intermediate layer of fast memory that is invisible to both the CPU and the RAM memory, called *cache*[1].

The design of this layer is based on the patterns of memory accesses found in real-world applications, which exhibit, each one to a certain degree, a characteristic called *access locality.* This behavior is the key principle caches are built upon, and is further distinguished in *spatial locality* and *temporal locality.* Without the need of a precise mathematical definition (as given by Bunt and Murphy [9]), we can easily define spatial and temporal locality as follows.

Definition 2.1. (Spatial locality) *if a (virtual) memory address a is referenced, it is likely that the addresses nearby 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 in the near future.*

Combining these two principles, we ca state 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. Caches are indeed designed to recognize these "local" patterns and keep data close to the CPU, and thus have a very different implementation than RAM memories.
In fact, caches are typically implemented with Static Random Access

[1]as of the Oxford Dictionary, a *cache* is defined as "A hidden or inaccessible storage place for valuables, provisions, or ammunition", from the French verb *cacher*, "to hide"

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. For these reasons, the size of cache memories is a careful trade-off that takes into account many parameters.

2.1.2 Structure of a cache

The basic granularity for data management inside the cache is the *line* (also called *block*). 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. 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 stores contiguous words from main memory. Using cache lines bigger than the CPU word allows to leverage spatial locality, because the cache stores also the words that are adjacent to the requested one. To leverage temporal locality, a more complex mechanism has been designed and is explained in the following section.

2.1.3 Cache functioning models

Based on the trade-off between latency and performance, several cache designs exist, each one with a certain ability to leverage temporal locality. The most general model is the *n-way set-associative cache*, where n is usually an even number. This model divides the cache into 2^s *sets*, each set containing n lines. Using both line size and set number in powers of two makes it possible to use subsets of the data address bits to look for a certain byte inside the cache. In particular, looking at fig. 2.1, the less significant l bits determine the offset of the referenced byte inside the line, and are hence called *line offset* bits. Similarly, the higher s bits are the *set number* bits and determine the cache set. In this way, the lookup of the requested datum is easily performed starting from its address.
A third parameter, the *associativity a*, plays a fundamental role in the design and performance of a cache. Each set, in fact, is able to store a different lines having the same set number; to distinguish the lines inside

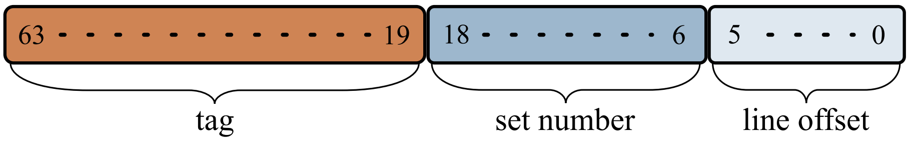


Figure 2.1: Bit fields of a memory address to access a cache

The physical address is divided into three fields for cache access: line offset, set number and tag

a single set, the cache uses the remaining bits of the address, called *tag bits*, as a label associated to each line.

When a datum is requested to the cache, the cache controller reads the three fields of the address, accesses the set and searches the line having the requested tag. The tag value is searched in parallel over (ideally) all the a set lines, thus explaining why this cache is called "set-associative". If the controller finds the line (this event being called *cache hit*), it uses the line offset to fetch the requested data bytes. Otherwise, in case of *cache miss*, the controller fetches the line from main memory and must store it inside the set, in one of the a lines. To determine the line to place incoming data to, the cache controller looks for a free line by reading a specific bit that indicates whether the line is free; if it finds one, it stores the incoming line to the free location. Otherwise, the controller has to replace a line with the incoming one. To do so, the key idea is to choose the line that is less likely to be reused again in the near future, in order to keep in the cache the only data with highest temporal locality. Caches typically determine the reuse probability though the recent history of each line: in obedience to the principle of locality, the more recently a line has been accessed the higher its reuse probability is. Hence, to track the recent history, each line has an associated field called *Least Recently Used (LRU) priority*, which is set to 0 every time the line is accessed and incremented on every access to other lines. Based on this value, the cache controller chooses the least recently accessed line for replacement: it *evicts* this line, eventually writing it to main memory if it was modified while it was inside the cache, and stores the incoming line in place of the evicted one.

Moreover, modern caches also have additional units called *prefetchers*, which attempt to predict future memory accesses and load the corresponding cache lines from main memory in advance, in order to mask the latency of cache misses. These units are widely present in today's processors, which perform aggressive predictions of memory accesses [11] and may even anticipate accesses at various offsets [30].

In general, the parameters of the cache play a fundamental role in determining its performance and it cost in terms of power and area, introducing several conflicting goals. Therefore, the design of a cache is always a trade-off between different objectives. For example, to exploit temporal locality at best the associativity should be maximal, to the point that all the lines a cache can holds lie in the same unique set (hence $s = 0$ and a is very big). In this cache model, called *fully associative* cache, the tag and LRU priority lookups involve a big number of lines, and need either a number of comparator circuits that is quadratically bigger (to perform all the comparisons in parallel) or a very long lookup time, increasing the latency of every operation. Therefore, this model is not feasible in practice.
On the other side, a cache with $a = 1$ and maximal number of sets requires the least amount of area and power, and has minimal latency since no lookup is to be performed, but only the tag comparison between the requested address and the one of the line in the target set; yet, $a = 1$ implies that it is not possible to choose among several lines in case of replacement, thus preventing the cache from leveraging temporal locality. Due to the low performance of this model, called *direct-mapped* cache, it was used only for small, low-latency caches in the past years, where the lithographic technology posed considerable lower bounds to the latency of the transistors.
Nowadays, despite the great progress of silicon lithography, the associativity is still limited to a small set of values, usually ranging from 2 to 20. Conversely, the number of sets is quite high to hold big amounts of data even with limited latency: indeed, modern caches may have more than 2048 sets with a latency around 10 ns.

2.1.4 Cache addressing

As stated in the previous section, the cache uses the data address for the internal lookup. Yet, 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, is unique and is usually managed by the OS only. Instead, applications use virtual addresses to reference memory and cannot handle 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. Furthermore, as the cache needs to know both the set number and the tag, one address space can be used for the set and the other for the tag. Based on which address space a cache uses for which field, four model can be devised.

Instead, a Virtually Indexed, Virtually Tagged (VIVT) cache uses the virtual address for both the set number and the tag. Therefore, the TLB translation is noot needed to perform a complete lookup, minimizing the latency. Yet, caches of this type suffer from two main problems, collectively called *aliasing*. Since each process has a dedicated virtual memory space, it may happen that the same virtual addresse refers two different physical location, depending on the running process: this is called the *homonyms problem*, and forces either the cache to add extra logic and internal state to disambiguate, or the OS to completly flush the cache in case of context switch, with an evident performance penalty. Conversely, different virtual addresses can refer to the same physical locations (as with shared data or IPC mechanisms), thus creating multiple copies of the same data (*synonyms problem*). To solve this issue, a cache must track which lines contain to the same physical address, adding extra logic that depends on the TLB lookup (even if not on the critical path of cache hits). Generally, since the solutions to homonnymity and synonymity problems have a high cost, VIVT caches are rare.

On the opposite side, in the PIPT model, the cache uses only the physical address for both the set number and the tag. With such scheme, all the addresses come from the same address space, ensuring their uniqueness inside the whole cache. Nonetheless, PIPT caches require the TLB translation before accessing, and have thus a higher latency. However, thanks to the progress of the lithography and to the speed of TLBs most of the modern caches are implemented in this way.

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. However, the caches have small latency, and are still used in small, fast caches.

2.1.5 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 a certain type of data, for example those along the critical path of fundamental 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, which pre-decode the variable-length x86 instructions into fixed-length micro-instructions, which enter the CPU pipeline. To speedup the slow decoding phase of variable-length instructions, the pre-decoder stores groups of corresponding micro-instructions inside such cache in order to retrieve them in future, to reduce the bottleneck and the consumption of decoding. This technique, widely employed in the latest architectures by Intel, is particularly suited to speedup the execution of loops, whose body can be entirely kept inside the micro-operation cache and be decoded only at the beginning.

2.1.6 Cache hierarchy

As stated, the design of a cache for a given architecture requires numerous trade-offs. 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. Therefore the cache had to scale-up with the rest of the architecture, without causing bottleneck effects. Since the latency of a single cache is determined by the number of sets and the associativity, even with the last technological enhancements it is impossible 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. These layers increased in number during the years, and today's server processors typically have three cache layers.

The lowest Level 1 (L1) caches are the first ones to be accessed for lookup and are very small, typically 32 or 64 KB, and are usually divided into a *data cache* and an *instruction cache* to realize a so-called *Harvard architecture.* The former cache contains only the applications data, which often exhibit higher locality than instructions, that the latter stores; to leverage this higher locality, data caches have higher associativity. In order to limit latency (around 2 cycles), the L1 caches have small size and employ a VIPT addressing scheme. In CMP architectures, L1 caches are per-core, in order to provide to the core datapath a dedicated, fast 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 has greater size (typically from 256 KB to 8 MB) and has higher associativity (around 8), but also higher latency (typically 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 aliasing phenomenon is present. If this cache is the last layer in the hierarchy, it is shared among cores, otherwise it is per-core. In fact, in the latest years, the increasing amount of data CMPs must elaborate led to the use of such caches as per-core caches, while an upper Level 3 (L3) layer serves as shared cache and coordinates data sharing among cores. These L3 caches have even greater associativity (from 12 to 16), size (from 6 to 40 MB) and latency (from 20 to 40 cycles), presenting a high hit ratio. This level is also called LLC, and is connected to the main memory via a controller, with a latency that is often in the order of 100 cycles. Typically, it is shared among all the cores, even if few CMPs have multiple independent caches shared between couples of cores. Because of the increasing number of cores and of the increasing LLC size, this last layer is often split into more *slices* (or tiles) that are interconnected with the cores. How the intercommunication system is designed depends highly on the model of the CPU, and the same vendor has developed, over the years, multiple solutions with different cost, performance and scalability. Therefore, the contention the cores may experience on the intercommunication varies highly, and we cannot assume a reference model for this system.

A key aspect of cache hierarchies is the *coherency* between multiple caches: when data are shared among cores, the cache lines containing those data are loaded into the lower layers of each core, and a coordination mechanism must control whether these lines are modified and eventually communicate the changes to the other cores (and CPUs in Non-Uniform Memory Access (NUMA) systems) to ensure data coherency. The state of the lines is tracked through the Modified, Exclusive, Shared, Invalid (MESI) protocol [64], which CPU manufacturers adapted over the years to their architectures producing variants like Modified, Exclusive, Shared, Invalid and Forward (MESIF) for Intel [87] and Modified, Owned, Exclusive, Shared and Invalid (MOESI) for ARM and AMD [54, 1].

Finally, CPUs can have specialized caches like a micro-instruction cache or an upper victim cache, but, here too, the solutions vary greatly based on the vendor and on the CPU family.

2.2 Contention on a Shared Cache

To understand the nature of this work and of those presented in the next chapter, we review how contention arises on shared caches. Given the structure of modern CMPs presented in the previous sections, we assume that only the LLC, either an L2 of L3 cache, is shared among cores, to focus on this level. This shared layer is where contention happens, hindering running applications scheduled on the cores.

Cache contention phenomena are divided in two main categories: *thrashing* and *pollution*. Thrashing [25] indicates that lines with high locality are evicted by other lines with high locality too, for the simple reason that both sets of lines are mapped to the same cache set. This contentious pattern causes frequent transfers to and from the main memory, which become the performance bottleneck. Thrashing is typical of co-scheduled, memory-intensive applications with good locality and is hardly predictable as it depends on the physical location of data in main memory, in turn depending on the OS, the workload, etc.
Instead, pollution indicates that lines with low future reuse are evicting lines with higher future reuse. This phenomenon happens because of the LRU policy and of the limited associativity of a set: because of the limited space, the cache controller must evict a line to make room for the incoming one, assuming that this line will be reused in the near future. When this assumption is wrong, pollution happens and the evicted line, having greater locality, will be fetched again. This phenomenon is due to multiple sources, like non-local accesses to buffers [26] or the wrong access predictions of cache prefetchers [80].

The advent of CMP architectures exacerbates contention on the LLC by "mixing" the access patterns of various applications simultaneously running on the cores. In fact, the LRU policy at the basis of caches was conceived for single-core CPUs and is effective in capturing the access pattern of a single application, while it is unable to distinguish the different access patterns imposed by the cores, resulting in sub-optimal performance. It is to be noted that, despite data sharing among applications happened also with single-core CPUs because of time sharing mechanisms, the time granularity of this phenomenon is order of magnitude greater than that of memory references, being the scheduling quantum around 10 ms; this caused the LRU policy to be usually very effective, with the exception of few, rare scenarios. Instead, in modern CMPs the access patterns mix with a much higher frequency, as cores can reference lines at intervals of few clock cycles, or even at the same time.
Although contention is widely studied in the research, commodity CMP architectures lack interfaces to control this phenomenon and prevent access patterns from mixing inside the LLC.

2.3 Intel's Nehalem and Sandy Bridge architectures

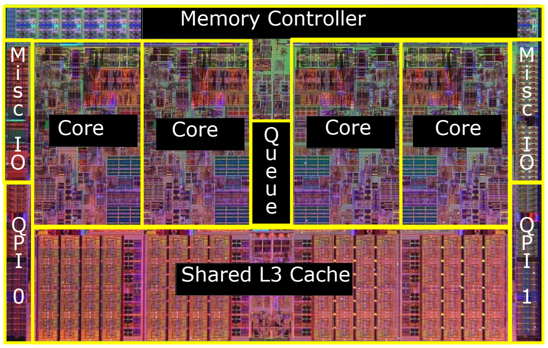
Today's server CMPs have a complex structure, with an ever-increasing number of cores and a large LLC to hold the cores' data. Intel's architectures have a dominant position in the market of server CPUs [39]; in particular, its most recent architectures, named Nehalem [**nehalem_man**, 17] and Sandy Bridge [**sandy_man**], power the servers of recent computing infrastructures. Therefore, facing the contention over the LLC of these architectures is of primary importance to cloud computing infrastructures.
Hence, this section explains the fundamental details of Nehalem's and Sandy Bridge's architectures, on which the proposed solution is conceived.

2.3.1 Architecture of Nehalem

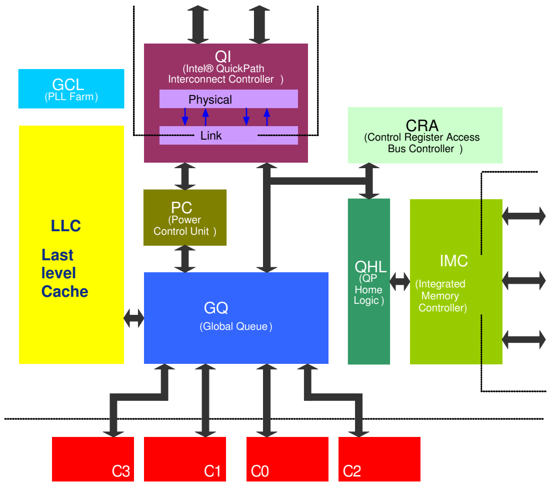
Introduced at the end of 2008, the Nehalem architecture is the evolution of the previous Core architecture. It was designed to be modular, in order to be adapted to the various market segments (mobile, desktop, server) without re-designing large portions of the chip to comprise four or more cores. Inside the single core, the Nehalem platform provides new Single Instruction Multiple Data (SIMD) instructions, called Advanced Vector Extension (AVX) [19, chapter 5.13], and increased Simultaneous Multi-Threading (SMT) support, called Hyper Threading (HT) in Intel's terminology.

Leveraging the progresses of silicon lithography, Nehalem is designed with a L3 cache as LLC, shared among all the cores, with a size from 4 to 24 MB. Lower caches are per-core, with a size of 64 KB for L1 data and instruction caches and 256 KB for the L2 unified cache. The L3 cache is *inclusive*, meaning that it includes all the lines stored inside the caches of all the cores: this property, typical of Intel's LLCs, simplifies data coherency since per-cores caches can send coherency requests (via the MESI protocol of ita variants) directly to the LLC, without long snoop requests that would go through all the other cores.

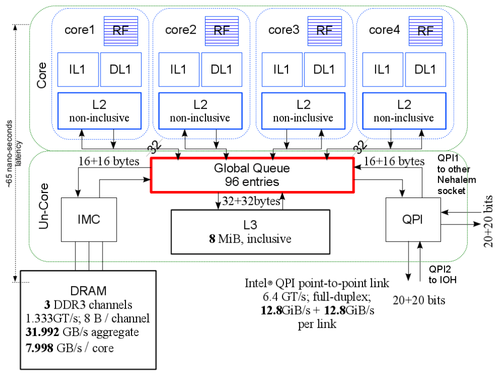
Previous architectures, which had at most four cores, were conceived



(a) Die photo of a Nehalem CPU, from [37]. The GQ is in the middle of the chip to route memory requests and responses



(b) Functional architecture of Intel's Nehalem platform, from [22]. A GQ interfaces the components with the LLC, handling all the memory requests

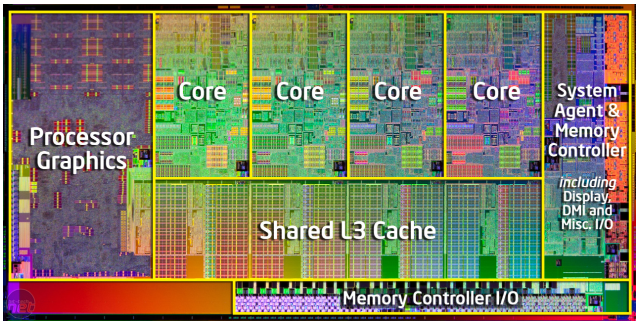


(c) Units connected to the GQ, from [86] with an L2 cache as LLC, and only later, high-end redesigns of the Penryn architecture (Nehalem's predecessor) comprised a shared L3 cache, exploiting the 45 nm lithography. In Nehalem, instead, the L3 cache is a basic element of the architecture, introduced to increase the memory bandwidth needed by the higher number of physical and logical SIMD-capable cores. This L3 cache, despite the physical multi-bank design visible in fig. 2.2(a), is logically managed as a unique element. Figure 2.2(b) and fig. 2.2(c) also show how the on-chip interconnection sub-system, named GQ, connects the LLC to the cores, the integrated memory controller and the other components that handle Input/Output (I/O), inter-CPU cache coherency and power management. This interconnection sub-system is realized through a cross-bar structure to provide an efficient routing medium.

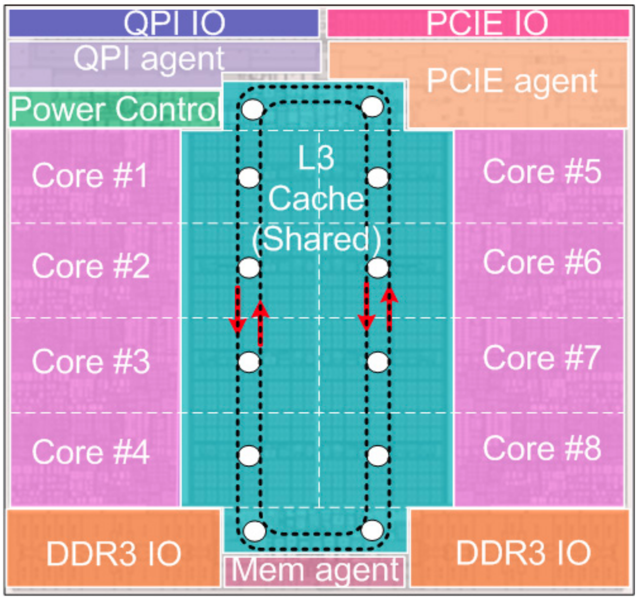
2.3.2 Architecture of Sandy Bridge

Pushing forward the evolution of the Nehalem architecture, Intel released the first Sandy Bridge models in early 2011, based on 32 nm lithography. The cores' internal architecture has undergone several changes like the addition of further AVX instructions and a better energy management; an important change was the addition of an integrated, on-die graphic processor, making this architecture particularly suitable for laptops.
Overall, the architectural model of the Sandy Bridge cache hierarchy also applies to the latest families, namely Ivy Bridge (that was a little more than a die-shrink of Sandy Bridge) and Haswell. Indeed, the new features introduced in Sandy Bridge proved to scale well with the lithography and the customers' needs, and were maintained across the following families. Throughout this thesis, as we focus exclusively on the cache hierarchy, we will mention only the Sandy Bridge family for the sake of brevity, implicitly meaning also the subsequent CPU families.

In the cache hierarchy, the lower L1 and L2 layers are unchanged, while the L3 layer is notably different. Because of the new AVX instruction with operands of 256 bits and of the higher number of cores, the Sandy Bridge architecture needs more bandwidth between the L3 cache and the other elements, such as the graphic, the memory controllers, the System Agent (Intel's name for the CPU power controller), etc. Since



(a) Die photo of a Sandy Bridge CPU with integrated graphic, from http://images.bit-tech.net



(b) Ring interconnection of a Sandy Bridge-EP server CPU with integrated graphic, from www.qdpma.com this architecture is designed to ship more cores than Nehalem, it also require better scalability of the cache bandwidth. To meet these goals, Intel split Nehalem's unique cache into several parts called *slices*, one per-core with equal size and characteristics. To guarantee to each element access to a broad cache space, each element can access all the slices through an interconnection sub-system. Figure 2.2(a) shows the on-die elements on a Sandy Bridge CPU with integrated graphic: all these elements communicate with each other, and in particular with the L3 slices, through a ring interconnection. Figure 2.2(b), instead, highlights the ring interconnection, a major novelty of the Sandy Bridge design [52]. The white boxes in fig. 2.2(b) are the interfaces that sense requests from the elements and assert replies on the rings. These interfaces are called *cache boxes* or *ring stops*. To let the bus bandwidth scale with the number of elements, the ring is fully pipelined: during a clock cycle, data flow from one cache box to the following one, according to the bus direction. Furthermore, two rings are present, with data flowing in opposite directions based on the bus. Each slice has one cache box per ring, and a cache box serves both the slice and the core it is coupled with. Since, in total, two cache boxes are present per-slice, a slice can read requests from other elements and send replies simultaneously, with a routing protocol that chooses the best ring based on the bus occupation. To guarantee high bandwidth, ring buses consist of four rings for data, requests, acknowledge replies and snoops, respectively. In particular, the data ring is 32 B wide, so that a cache line (64 B) is transferred in two clock cycles. To avoid the performance bottlenecks of a centralized control, the arbitration protocol is distributed on each element and the coherency protocol running on the snoop ring is based on MESIF. The protocol governing the rings is, overall, undocumented, but Intel claims that, using separate buses for data, coherency and coordination, cache boxes can assert data and control messages on every clock cycle.

With such organization, multiple requests can be served simultaneously, provided that they are directed to different slices. To ensure an even distribution of requests among slices and avoid bottlenecks, Intel computes the L3 slice a line must reside in by means of an undocumented *hash* function: physical addresses are hashed at the source and the data request asserted on the ring. This feature further ensures the scalability of the entire architecture and shows to the software a “fictitious” associativity that is higher than the real one. Yet, a first drawback of this addressing scheme is that the L3 access latency cores experience is variable, depending on the hop distance of the cache slice from the requesting core. Hence, this latency varies from around 21 cycles in case of hit in the local slice to almost 40 in case of hit in a distant slice. Finally, another drawback of this organization regards the power saving capabilities. If the frequencies of the cache and of the cores differ, data requests suffer from a penalty that is proportional to the ratio between the two frequencies. Therefore, Intel’s designers chose to place all the cores and the slices in the same voltage/frequency domain, avoiding intolerable latencies and performance bottlenecks due to different frequencies.

2.4 Buddy memory allocator

Since modern shared caches are PIPT, the cache set where data are placed depends on their physical address, either directly or through a hash function. Hence, for the purpose of this word, it is fundamental to understand how data in LLC can be controlled by mean of their physical address, which in turn depend on a component of the OS called *physical memory allocator*. This component manages the physical memory of the machine: when a subsystem of the kernel (such as the page fault handler, the Direct Memory Access (DMA) drivers, etc.) needs a portion of physical memory, it issues a request to the physical memory allocator, which allocates a contiguous physical area of at least the requested size. CPU architectures with virtual memory, such as those we consider throughout this work, pose several constraints to these memory areas; one of them is the minimum granularity of memory management, which is called *page*. Therefore, in these architectures the physical memory is allocated in multiples of the page. This size is always a power of 2: for example, it is 4KB on x86 architectures. The other main constraint is that memory pages have memory addresses that are always *page-aligned*. This alignment simplifies the virtual-to-physical mapping, that is performed by the TLB.

Because of the page alignment, a memory address can be split into two bit fields: the less significant bits are the *page offset* and indicate the requested byte inside the page, while the most significant bits are the *page address*. This is the value the TLB has to translate from the virtual address space of a process to the physical address space of the machine. After the translation, the TLB outputs the physical page address and pads it with the offset bits to obtain the final physical memory address. Modern CPUs support also bigger page sizes, as section 2.5 discusses, but the basic granularity of memory management is usually small and architectures maintain rigorous legacy retro-compatibility.

Because of this small size, modern machines may have millions of pages: for example, an x86 system with 4GB of RAM memory has more than one million pages, and modern 64 bit servers may ship 64GB or more memory. The physical allocator is in charge of managing all these pages, and, in a modern operating system like Linux, must fulfill requests from many subsystems, with varying granularity and at a considerable request pace. Moreover, its performance should not degrade with the time and it should minimize *memory fragmentation*, which is a degradation phenomenon of memory areas. Memory fragmentation is due to internal and external fragmentation, and its limitation is a key design goal for allocators. *Internal fragmentation* is the waste of memory due to over-allocation, which happens because the allocator returns a memory area bigger than the requested one. Internal fragmentation depends on the page size: if the request is not an exact multiple of the page size, the allocator returns a higher multiple of pages and space wastage happens. Instead, *external fragmentation* indicates the interleaving between free and used areas: if external fragmentation is high, the allocator cannot allocate contiguous areas of big size because there are used pages "in the middle".

Due to all the requirements we highlighted, a physical memory allocator is designed with several goals:

- *efficiency*: when a page is requested, it should be returned as fast as possible
- *scalability*: since machines have very different amounts of memory (from few MB to hundreds of GB), the allocator should handle memory areas efficiently

- keeping *internal fragmentation* low
- keeping *external fragmentation* low

Efficiency and scalability highly depend on the implementation of the allocator, and thus vary. Instead, the main strategy to keep external fragmentation low is to reserve an amount of memory 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 application 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) and may cause severe performance degradation of the allocator, which could store many small memory fragments in its own data structures and perform long lookups. Therefore, complex heuristics exist to keep external fragmentation low and vary among OSs.

In Linux, the heart of the physical memory allocator is the Buddy algorithm [48]. It is known in the literature from almost 40 years and is widely used because of its capabilities. 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 [50].
The following sections show the functioning of this algorithm, with a focus on the implementation found in the Linux kernel. In particular, section 2.4.2 explains the data structure the algorithm is based on, section 2.4.2 explains how the algorithm leverages this data structure to perform the operations typical of memory allocations and section 2.4.3 explains an additional heuristic that is key in Linux' implementation.

2.4.1 Buddy data structure

Within the buddy algorithm, physical memory is divided in parts called *buddies*. A buddy is a unique memory area with a size controlled through a parameter called *order*, where a buddy of order i is composed of 2^i physically contiguous pages. The order, is hence, the key parameter of a

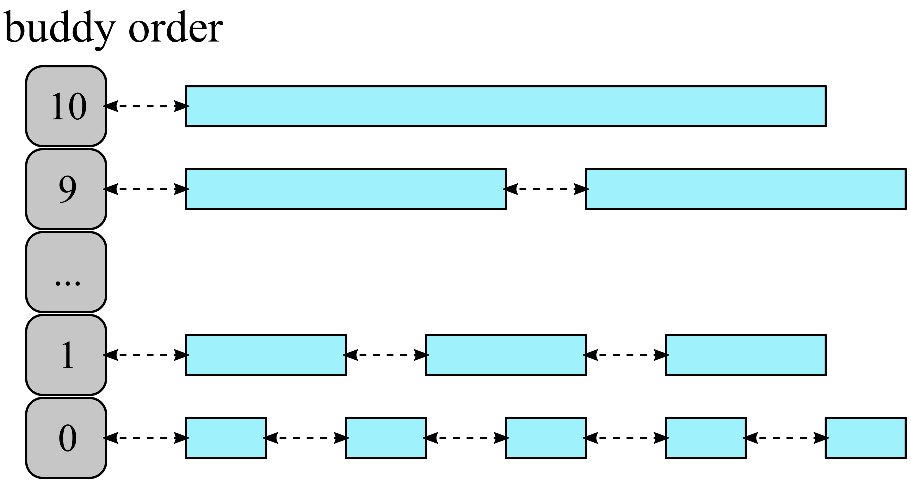


Figure 2.2: Lists of free areas of the Buddy allocator

buddy, and has an upper bound M that depends on the implementation: hence, an implementation manages areas of memory whose size ranges from 1 pages to 2^M pages. Buddies are *order-aligned*, meaning that the first page of the buddy has a page address aligned to a 2^i memory boundary. Hence, the page address of the first page of the buddy has the least significant bits set to 0, and is called *buddy address*. Thanks to these constraints, a buddy with a given order can be identified uniquely through its buddy address. The buddy algorithm is so called because it manages memory through buddies, trying to group them together to handle areas of size as big as possible. This grouping of areas reduces the number of elements the allocator handles, allowing scalability and efficiency.

In Linux, the main data structure of the Buddy algorithm is an array of doubly-linked lists, depicted in fig. 2.2; each list links all the buddies of the a certain order, so that there are as many lists as buddy orders. Thanks to this structure, the allocator can satisfy memory requests of a certain order in constant time, as any buddy inside the list fits the request, the allocator simply extracts the head of the list. Conversely, when a buddy is released to be freed, it is added to the head of the list, thus still in constant time. For insertion and removal, the buddy allocator performs additional operations described in the next section.

2.4.2 Buddy algorithm

As from the previous section, the heart of the buddy algorithm is its efficient data structure. However, the mere insertion of buddies does not allow the grouping of them into bigger, less numerous memory areas, to maintain efficiency and scalability. On the opposite side, a list can be empty, so that a request of a given order cannot be satisfied. Solving these issues is the key goal of the buddy algorithm.

The buddy allocator responds to memory requests by allocating buddies of a certain order, which must be greater than or equal to the requested memory amount. In Linux the physical allocator accepts requests expressed only in buddy orders, and it is up to the higher kernel (typycally the *slab* allocator [57]) levels to round up requested sizes to the closest higher order. If this order is n, the allocator subsystem looks in the list of order n for free buddies. If no buddy is present, it splits a buddy at order $n + 1$ in two buddies of order n; this operation is called *buddy splitting* and allows the allocator to vary the granularity of buddies in order to fit requested sizes. After splitting, the allocator stores one of the two halves in the buddies list of order n, and returns the other. Similarly, if also the list of order $n+1$ is empty, the allocator checks the list of order $n+2$ to contain buddies; if the list does, the allocator splits an $(n+2)$-buddy in two $n+1$ halves, stores one $n+1$-half int the proper free list and further splits the other half as previously shown. Instead, if also the $(n + 2)$-list is empty, the allocator checks higher order lists and recursively splits buddies in the same fashion. Finally, if also the M-order list is empty, the allocation cannot be satisfied.
With these splitting scheme, the time complexity of the allocation is $O(M)$: since M is typically small (the default value in Linux is 10) every allocation has a strict time bound. Moreover, thanks to the logarithmic relation between the buddy order and the buddy size, even big requests can be fulfilled quickly. Starting the lookup from the smallest buddies is key to keep external fragmentation low. For the same purpose, 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

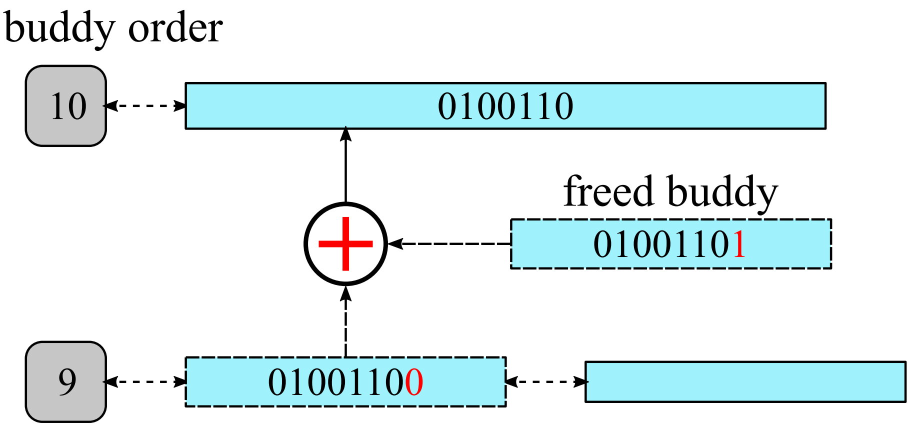


Figure 2.3: Coalescing of two buddies

When a buddy of order 9 is freed, it is coalesced with the "twin" buddy and the resulting 10-order buddy is placed in the proper list; the twin buddies differ only in the least significant bit of their address

efficient and to decrease fragmentation, particularly with kernel drivers of fast devices that typically need large contiguous buffers [29].

As opposed to the splitting process, for the insertion of a free buddy of order n the allocator tries to re-group the incoming buddy with another of the same order, to create an $n+1$-order buddy. This operation is called *buddy coalescing*. Given an incoming buddy of address a, the coalescing is possible only if a physically contiguous buddy of the same order is also free. Furthermore, since all buddy addresses are order-aligned, the new buddy must be aligned to a $n+1$-order address. Hence, each buddy can be coalesced with only another one, whose address is $a \bigoplus 2^n$: performing this XOR operation n-th bit of the buddy address gives another buddy that is physically contiguous to the incoming one. Hence, a buddy a of order n is strictly associated to the buddy $a \bigoplus 2^n$ of order n [2]. The XOR operation also guarantees that one of the two buddies has the n-th lower bit set to 0, while the lower bits are already set to 0: this address will become the address of the new $(n+1)$-order buddy. In such a way, Linux checks the buddy with address $a \bigoplus 2^n$ to be free,

[2]This strict coupling is the reason for the term "buddy".

and in this case it coalesces the two buddies in a bigger one, as fig. 2.3 shows. Linux can perform this check in constant time, as it describes the physical memory as an array of pages: each page stores the information of the buddy it represents (if the page is order-aligned), and kernel can access each page in the array by using the page (or the buddy) address as array offset. Like the splitting operation, also the coalescing operation is recursive, so that buddies of increasing order can be grouped together if both buddies are free.
With this mechanism, the buddy allocator maintains scalability and efficiency during time: if applications and drivers properly free memory after use, large physical areas are quickly made available again.

2.4.3 Linux mobility heuristic

Despite the splitting and coalescing mechanism, long running applications may retain memory pages for a long time and cause external fragmentation the allocator cannot face. Moreover, kernel allocations are likely to be needed for a long time, further increasing this issue. At the end, after hours or days with volatile and non-volatile allocations interleaving in memory, external fragmentation can be considerable because of small "in the middle" fragments that cannot be coalesced because they are in use.
To limit this phenomenon, Linux further subdivides buddies according to their *mobility*, that is their probability of being freed in a short period. The more a page is *movable*, the higher the probability of being freed shortly after allocation. For example, buddies used for kernel data structure have almost no mobility, while userspace allocations have maximum mobility. It is to be noted that mobility is not enforced by any hardware constraint or mechanism, but is simply a consequence of the expected behavior of software layer requesting the memory. Therefore, the kernel is free to choose certain memory areas as a memory pool to serve allocations of a given mobility. For example, Linux chooses a pool for high-mobility allocations: this makes it possible to group in a contiguous area all the high-mobility allocations, so that it is very likely that, shortly after the requests, large areas are freed, allowing coalescing. Likely, low-mobility allocations are clustered too, so that they cannot fall

"in the middle" of higher mobility areas, preventing coalescing.

To implement this heuristic, Linux groups buddies in pools and assigns each page a so-called *migratetype* to indicate the pool it belongs to [57, Section 3.5.2]. In addition, to enable fast lookup of buddies with specific mobility, the doubly-linked lists at the heart of the buddy allocator (described in previous sections) are further subdivided per migratetype. At boot time, these migratetype pools are populated according to pre-defined quotas. During runtime, if the number of buddies in a pool falls below a pre-defined watermark, the kernel moves buddies from another pool, according to fallback lists: pools, in fact, should "borrow" buddies from a pool with similar mobility, and "steal" buddies with very different mobility only as a last chance. To change the pool a buddy belongs to, it is sufficient to change the migratetype if its first page and to move the buddy from the original list to the head of the new one. All these operations are done in constant time.
This heuristic proves to be very effective in keeping external fragmentation low, and has become a key aspect of Linux' implementation of the buddy algorithm.

2.5 Hugepages

Despite the buddy allocator is can manage physical areas of different sizes, the mapping between physical and virtual areas is still limited to the granularity the hardware enforces via the TLB. In today's, memory rich systems, a large memory area, even if physically and virtually contiguous, results in many mappings the TLB must handle. 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. For example, the x86 architecture allows page sizes of 2 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. As we will explain in the following, this capability of modern architectures is not compatible with the solution that is the objective of this work. Hence, this section aims at giving a very basic overview of this capability, to later discuss the reasons of incompatibility.

Pages of “non-legacy” size are called *hugepages* in Linux terminology and *large pages* in Windows terminology. Throughout this work, we will refer to them as hugepages. To exploit hugepages, the kernel must allocate a large contiguous physical area of the same size, which is mapped into a corresponding virtual area. Like small, legacy pages, also hugepages must be aligned to a memory boundary that is a multiple of the hugepage size.
Hugepages support is actually limited both in Linux and Windows due to several reasons: the difficulty of modifying the physical allocator and the virtual memory system, the reduced interest on behalf of potential users and the fragmented hardware support. In Linux, for example, the system administrator should explicitly enable hugepages at boot time, and applications can request them only via custom interfaces. Nonetheless, the research community is investigating the benefits and drawbacks of this possibility, and some results have already been presented [55, 100].

State of the Art 3

Urbem quam dicunt Romam, Meliboee, putavi
stultus ego huic nostrae similem, quo saepe solemus
pastores ovium teneros depellere fetus.
Sic canibus catulos similes, sic matribus haedos
noram, sic parvis componere magna solebam.
Verum haec tantum alias inter caput extulit urbes,
quantum lenta solent inter viburna cupressi.

Virgil, Bucolics I

In this chapter, the state-of-the-art work that is of interest for this thesis is presented, with an emphasis on cloud platforms. The main objective behind these works is to enhance the usage of CMPs, in particular of the LLC.
The techniques to improve the usage of the LLC follow different guidelines and have a very diverse impact on the hardware/software layers of modern platforms. Section 3.1, indeed, reviews how the task scheduling policies can take in account the usage of the LLC, based on information collected at runtime or given a priori. Then, the following sections focus more specifically on the techniques that reduce the contention over the LLC. These works span from new hardware models to changes to the OS layer, exploring a wide variety of solutions.

For the sake of clarity, this chapter divides the presented techniques into *hardware* and *software* techniques, partially following a taxonomy the author of this work already presented [73]. Hence, section 3.2 gives an overview of the main hardware techniques, emphasizing how the research is alive around contention issues in current CMP architectures. Then, section 3.3 reviews several software techniques starting from the explanation of page coloring, the mechanism at the basis of the present work.

3.1 Performance-aware scheduling techniques

To tackle contention on the LLC without deep hardware or software changes, the research proposed numerous scheduling techniques that take in account how applications impact on shared caches. Overall, these techniques attempt to minimize contention by "re-arranging" running applications, moving those that cause intolerable contention to other CMPs. To guide the re-scheduling policy, these techniques need a metric of contention or of performance, which allow 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 [56, 84, 28]. 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 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, and typically suffer because of co-location with other applications. Batch applications, instead, tend to have a continuous usage of resources, leaving limited room for the execution of co-located latency-sensitive applications. The bursts of these applications, in turn, need sudden availability of CMP resources and in particular of LLC space, leading to slow warm-up activities called *cache inertia*, which increase the latency percentile and degrade QoS. Data centers typically solve this problem by limiting co-location, in particular with and latency-sensitive applications, thus potentially under-exploiting resources. To address this situation, the research proposes different policies and techniques. In more detail, cloud platforms present specific scenarios, like Virtual Machines (VMs) or multi-thread workloads, whose peculiarities may improve the scheduling policies.

Addressing multi-threaded workloads, Chen et al. [14] propose to leverage data sharing to improve the usage of the CMP. They evaluate two scheduling policies, showing that, in particular, Parallel Depth First (PDF) [7] is best from this point of view. In more detail, PDF characterizes an application with a task graph, reaching a fine description granularity of the application's parallelism, and assuming fine-grained tasks to share most of the data in LLC. Based on this description and on profiling information, Chen et al. [14] finds the optimal scheduling granularity to be used for the execution of the PDF scheduler.

Tang et al. [85] perform an in-depth study over the performance of some cloud-typical applications, showing the importance of co-scheduling based on resources usage. They characterize applications via specific low-level parameters like LLC miss rate and propose heuristics to maximize the benefit from sharing and to minimize contention. In particular, they stress the importance of thread-to-core mapping to improve the final performance of the workload. Similarly, Kang et al. [45] exploit both hardware counters and information from instrumented libraries, in particular about lock operations and synchronization patterns, to optimize the co-location of tasks on the Tilera64 Network on Chip (NoC) [5] at runtime.

Instead, Mars et al. [56] attempt to predict performance degradation offline by adding a controllable pressure to the memory subsystem. In this way, they measure the sensitivity of each application to the LLC and to the sharing of memory resources. The collected measurements are then used to devise a co-location scheme for latency-sensitive applications, whose QoS fulfillment increases. Pushing this approach further, [97] continuously monitors running applications with a tunable memory-stressing application, in order to capture applications' phases and adapt the workload. With this characterization approach, the utilization of the infrastructure further increases as well as the percentage of QoS fulfillment.

Since VMs are at the base of many cloud services, a broad literature is available that specifically addresses the co-location of virtualized workloads. For example, Gong and Gu [32] perform a high level modeling of resource usage through PAC, a runtime VMs monitor that tracks resource usage by finding certain patterns called *signatures*. Based on signatures, the hypervisor periodically chooses a schedule of VMs that distributes the workloads over the machines in order not to exceed a threshold of resources usage. Instead, Govindan et al. [33] adopt an approach similar to [56], with a tunable LLC-stressing application used as a proxy for incoming workloads. Via this proxy, [33] profiles running VMs and predicts the degradation due to contention. Finally, [61] devises a runtime model of interference by measuring how co-location affects the meeting of QoS goals the applications declare, and, based upon the information collected, employs a tracking approach to schedule resources and fulfill the goals.

3.2 Hardware techniques

Several works address contention in the LLC proposing hardware changes to the cache. In the context of this thesis, two main aspects are to be stressed. On one side, the *techniques* are the mechanisms that allow alleviating the contention in the LLC. The techniques presented in this section are implemented in hardware, and are driven by a *policy*, implemented either in hardware or in software. A policy measures the applications behavior and chooses how to manage the LLC with respect to pre-defined goals. This distinction is fundamental in our review, and will be consistently adopted in our nomenclature in the following.

The benefit of implementing a technique in hardware lies primarily in the reduction of the overhead and in the availability of fine-grained information about running applications. Instead, policies can be implemented either in hardware or in software, this choice being more disputable: a hardware-only implementation has lower latency and is transparent to the software layer, while a software implementation (usually inside the OS) can consider multiple high level goals like QoS requirements and resource usage.

Overall, hardware techniques derive from very different concepts: some change the actual implementation of the LRU algorithm, which is an inherent cause of contention in CMP architectures (as from section 2.2), while others have a more disruptive approach, re-designing the cache structure.
A first, non-disruptive technique for LLC partitioning is called *way partitioning*: according to a user-given bitmask, each core can access a subset of the lines inside each set. In case of eviction, the LRU policy works only on the cache lines assigned to the core, so that each set is effectively partitioned among the cores. Some special-purpose archi-

tectures like Octeon [62] or some prototype CMPs [16] adopt this LLC partitioning mechanism, but commodity CMPs do not. However, way partitioning has the drawback of decreasing the associativity available to a core: in fact, since each core accesses only a subset of the lines, it has a smaller set of candidates for eviction. As an extreme example, if a core controls 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 decision, paying the benefits of isolation with a much lower performance.
Based on way partitioning, Utility-based Cache Partitioning (UCP) [69] computes how many lines each core needs in order to maximize the global workload performance. Therefore, UCP computes the *utility* of each configuration, that is to say how much an application benefits in terms of LLC misses when the partition size changes. To collect this measure, it introduces a novel component, called Utility MONitoring (UMON), which monitors the utility of each application. Then, based on the recent measurements, UCP minimizes the total number of misses and assigns to each core a certain number of cache ways.

Unlikely, other techniques change the LRU implementation by recording which core loaded each line . This allows the LLC controller to work with specific subsets of lines, controlling the eviction candidates on the basis of a certain policy. For example, Sharifi et al. [76] considers the scenario of a multi-thread application, whose threads are assumed to have similar characteristics: to balance the performance of the threads, [76] penalizes the core with highest IPC (the *victim* core) in favor of the others. Exploiting the modified LLC controller, the cache evicts some of the lines of the victim core and assigns them to the slowest one in case of line insertion.

Another fundamental parameter to control the functioning of a cache is the LRU value of the lines. For example, as the evictions from the a set are performed according to the LRU value, loading a line with a higher value than usual potentially increases its persistence inside the set, making the line more unlikely to be evicted in the future. Such solution, combined with a policy that decides the proper LRU values on the basis of given goals, allows to adapt the cache functioning to the workload. Leveraging this technique, Seshadri et al. [74] explicitly address thrashing and pollution. They employ a First-In, First-Out (FIFO) structure called Evicted Address Filter (EAF), which is added to the cache to capture the frequency of recently evicted lines. The goal of this structure is to predict whether evicted cache lines are likely to be referenced again in the near future, thus overcoming the limitation of the LRU policy with mixed access patterns. Indeed, the EAF 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 are likely to be "captured" by the EAF, and thus to be kept in cache. Moreover, [74] implements the EAF by means of a Bloom filter [59], a hash-based memory structure that limits the area overhead while. This solution, at the same time, limits thrashing: since a Bloom filter must be periodically flushed, thrashing lines, which the EAF often captures, are often "lost" thanks to the flush operation, being re-inserted with low-probability.

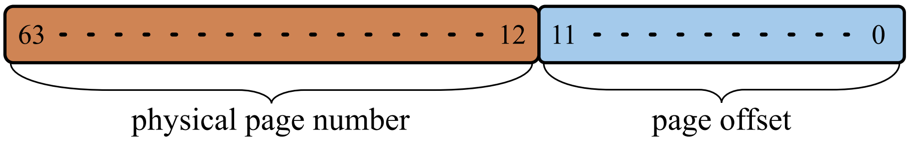
With a completely different vision that brings to deep changes, *Vantage* [72] is based on the *z-cache* model [71]. A z-cache maps the incoming data to a line by means of a tunable hash function, whose parameters can be changed to control the mapping. Leveraging this control, Vantage is meant tp explicitly partition the LLC. Exploiting the utility curves of UCP [69], Vantage selects at runtime a new partition size for each core and resizes the existing partitions accordingly. A novel idea behind Vantage is to use an unmanaged area of the cache (not assigned to any core) to resize partitions, a very delicate operation. When a partition is modified, the lines belonging to the old core are moved to the unmanaged area by simply tagging the line as "unmanaged", and the core receiving the additional cache space is free to load its new data into its managed area. Moreover, Vantage lets partitions outgrow into the unmanaged area to avoid that, with many partitions, conflicts or bottleneck effects arise in the unmanaged area. Thanks to its strong statistical background and to the properties of the hash functions used in z-caches, [72] provides strong guarantees in terms of performance and adaptation. In fact, an advantage of this approach is that it is able to react to applications changing phases faster than actual caches, yet at the cost of a profound re-design of the cache.

3.3 Page coloring and software techniques

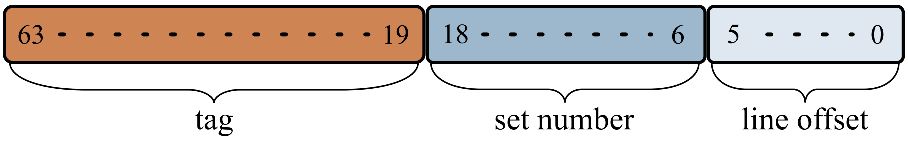
Software techniques are of high interest for this work as they involve only the OS layer and can thus be deployed on real machines without changing the application layer or the hardware infrastructure.

The mechanism at the basis of software techniques is *page coloring*, which depends on the way modern caches map data to cache lines. Indeed, section 2.1.4 explained how modern architectures use the physical memory address to map data into the LLC. Figure 3.1 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 3.1(a) 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, fig. 3.1(b) 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.
To control where data are stored inside the LLC, the key idea is to control the physical address: in fact, in fig. 3.1(c), some bits are in common between the set number and the physical page number, which is under the control of the OS. This bits are called *color bits*, and a configuration of them is called *page color*. Controlling the page color allows the control of the LLC sets data are mapped to, thus enabling a fine-grained placement of data into the LLC.

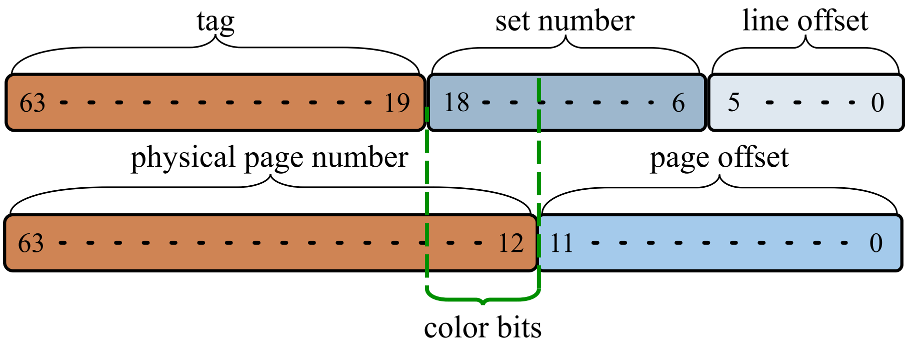
The number of color bits is, in general, $l + s - \max_{(l,p)}$; since, yet, cache lines are typically much smaller than memory pages due to the finer granularity of data management in caches, we can assume without loss of generality that $l+s-\max_{(l,p)} = l+s-p = c$. In the architecture represented in fig. 3.1, $c = 7$. Hence, Xeon W3540 has 7 color bits and 128 page colors. It is important to note that a color may correspond to multiple cache sets because of the minimum granularity of the page size. For example, in the configuration of fig. 3.1(c), a color spans $2^{s-c} = 2^{p-l}$ sets. This is the minimum amount of cache sets that can be allocated: in the example of fig. 3.1(c), these sets correspond to $2^{13-7} \times 16 \times 64B = 64KB$, being 16 the associativity and 64 B the cache line size.



(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 3.1: Bit fields of a physical memory address

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 consequently control their placement inside the LLC, possibly partitioning the cache.

This technique and its benefits are well-known in the literature [8, 46]. Yet, the advent of CMPs architectures exacerbated the cache contention as section 2.2 explains, thus enforcing the need of applications isolation. Given the lack of definitive hardware solutions, this need makes page coloring a hot research topic, further justified by the recent spreading of cloud services with QoS requirements.

The main drawback of page coloring is its static behavior, 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, which is particularly detrimental, for example, in case of a write operation inside the page.
However, some solutions exist that allow re-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 need careful design to limit the overhead, and in particular to minimize re-coloring.

The next sections review the relevant the work based on page coloring. In particular, section 3.3.1 shows the usage of page coloring to limit the occurrence of cache pollution, while section 3.3.2 shows how page coloring-based techniques can realize LLC partitioning via 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% [26]) 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.

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

With a more flexible approach, Xiaoning at al. [26] 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) [77] 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. [83] 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 Zhang, Dwarkadas, and Shen [101] 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. [53]. 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.

Recently, Ye et al. [98] developed two novel re-coloring policies that take in account also time sharing of cores and QoS requirements. The first policy recolors a number of pages proportional to the memory footprint, but proves to be sub-optimal since it often recolors cold pages. Instead, the second policy tracks page hotness through the per-page used bit (present in the x86 architecture), prevents the application from accessing the hot pages and re-colors these pages when the application accesses them; this mechanism is able to capture the application's access profile, and recolors only the most used pages. Moreover, [98] validates the proposed solutions also on a low-end Sandy Bridge architecture, showing a preliminary evaluation of page coloring on this recent architecture. This evaluation suggests that page coloring can be a beneficial technique also on sliced caches and opens new research directions.

Finally, Jin et al. [43] apply page coloring to the Xen hypervisor to show that also VMs benefit from LLC partitioning, thus reproducing the basis of a cloud infrastructure. Proceeding in this direction, in [93] they add a dynamic re-coloring mechanism, 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.

Design 4

This chapter shows the design of our proposal, which is a modification of the Linux kernel to implement page coloring. Our solution aims to guarantee to applications isolation in the LLC and is designed considering the architecture of modern CMPs. In this work we propose to study the constraints these architectures pose to our page coloring and to explore the possibilities cache isolation opens, along with the necessary trade-offs. Throughout this chapter, section 4.1 explain the overalls vision behind our work, while section 4.2 explains how the proposed solution integrates with 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, section 4.5 explains the modifications to the buddy data structure and to the allocation algorithm.

4.1 Vision

The final objective of this work is to allow distributed computing platforms to provide better QoS via isolation of running applications. The principal 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 detrimental effects of contention, in particular those exposed in section 2.2. Modern CMP architectures, as from chapter 3, exacerbate contention, but are ubiquitous in today's computing environments. On these architectures resource partitioning, if available, is actually limited to a coarse granularity; instead, unpartitioned resources degrade performance and cause unpredictability. Focusing on the LLC to mitigate contention on this resource is the key point to enhance isolation, and hence performance predictability and QoS guarantees. Since we also want the proposed solution to be feasible - and possibly beneficial - in real systems, we cannot propose hardware changes, but we concentrate instead on the possible ways by which software can enforce performance isolation through LLC. Looking at the actual state of the art in chapter 3, several techniques face LLC contention phenomena proposing different solutions, but only *page coloring* provides strong guarantees of isolation even in co-location on a per-application basis. This aspect is fundamental in our choice: fulfilling QoS requisites according to SLAs requires a precise, tunable per-application control that is effective in “crowded” and “noisy” environments. Therefore, to bring page coloring to a realistic distributed computing environment, we propose a re-implementation of page coloring that is tailored to modern architectures, in particular to those shown in section 2.3.

Considering the high variability of distributed computing environment, the broad customers audience and the legacy, we assume to have no control over the single application, nor we can enforce specific programming practices or frameworks. Ideally, we would 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. Given these assumptions, the implementation of page coloring within an OS makes a good fit, since it requires updating only the OS and understand applications’ resource requirements.

Another key decision of our solution lies in how the LLC partition size is chosen. Ideally, a production-ready mechanism should be capable of deciding the size of the LLC partitions automatically, taking into account hardware constraints and QoS requirements, in an adaptive way. However, in the context of this work we focus on providing recent commodity CMP architectures with an LLC partitioning mechanism, implemented through page coloring. To demonstrate the importance of this technique even in newer architectures, we implemented a solution relying on *static* partition sizes provided by the user. Thus, the solution we develop in this thesis may serve as a starting point for experimenting dynamic policies (as those in chapter 3).

4.2 Approach

Implementing 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. Our focus on cloud infrastructures leads us to consider an OS that supports the diverse application and hardware scenarios these environments offer. Among others, we wish an OS that supports virtualization natively, as many cloud environments are based on this capability. Linux, for example, is released under GNU General Public License (version 2) (GNU GPLv2) license and supports virtualization natively though [51] and hardware emulation through the widespread QEMU [68]. Other open source OSs, such as FreeBSD, do not have this native support, nor are as widespread as Linux is. Concerning Xen, the open source hypervisor, we have to note that it is capable of managing VMs only, and not normal applications; while Linux manages VMs as normal processes. Since we desire to provide isolation capabilities to any kind of process, either a native application or a VM, the most natural choice is Linux.

The choice of the Linux kernel, thus, drives our design. The solution we designed, called *Rainbow*, introduces page coloring into Linux' buddy allocator, accounting for the aforementioned objectives. As from section 2.4, the physical allocator is a critical component: it is accessed frequently and is 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). Therefore, any modification must be carefully designed, in order to continue meeting the allocator's goals exposed in section 2.4 while affecting only the minimum possible amount of kernel code.

Although the core of *Rainbow* affects the internal memory allocator, we need to provide an easy-to-use and flexible interface to applications. Considering the possibilities Linux offers, we chose to develop a *cgroup* [15], providing userspace applications with a filesystem-like interface to request an LLC partition.

In the following we explore the main design decisions behind *Rain-*

bow, starting from how modern CMP architectures of section 2.3 affect page coloring.

4.3 Cache hierarchy model and page coloring

This section explains the major assumptions about the cache hierarchy at the base of *Rainbow*. In particular, section 4.3.1 goes through these assumptions in the context of a generic CMP architecture with multiple layers of cache. Given these assumptions, section 4.3.2 and section 4.3.3 explain the consequences of page coloring adoption on the caches hierarchies of Nehalem and Sandy Bridge architectures.

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 should be general enough to be applied to a wide class of commodity CMP architectures.

The first assumption is that the architecture features a unique LLC shared among all the cores, with a set-associative indexing scheme. This assumption is verified in most of today's commodity CMPs, with the exception of few models that are however rare. Moreover, the most recent architectures, like Sandy Bridge, are designed according to this model. Instead, lower levels can also have a different architecture, for example a Harvard architecture with separate caches for code and data, without impacting our design. Instead, the line size is assumed to be constant across the layers of the hierarchy, as verified in all CPU architectures.
Another fundamental assumption is that the LLC be a PIPT-addressed cache. This allows the control of the data placement in the LLC on behalf of the physical allocator, a key requirement in our vision.

Together, these assumptions allow the physical memory allocator to control *all* the applications co-running on the CMP cores in a way that is totally transparent to the user.

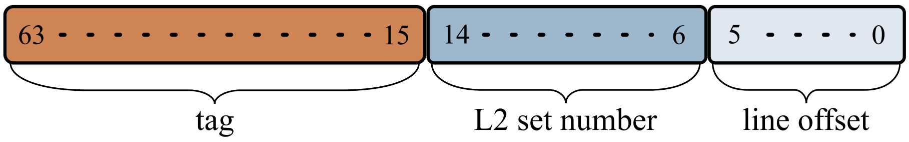


Figure 4.1: Physical address bit fields for L2 cache access

These assumptions fit a wide variety of commodity architectures, by Intel, ARM and IBM, in addition to those we target in this work. 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 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.

In addition to these assumed common features, modern CMPs use PIPT-indexed caches also in lower levels, thanks to the small latency of their TLBs. These caches are typically per-core L2 caches, while L1 caches are often VIPT-indexed, having strict latency requirements to feed the core pipeline.
Overall, these features, being commonly met in recent commodity CMPs, are taken as a reference model throughout our design, and possible variations do not have a strong impact, as discussed in the following sections.

4.3.2 Partitioning Nehalem's hierarchy

Recalling section 3.3, page coloring is the only software technique able to partition the LLC; it exploits the address bits in common between the LLC set and the page address for partitioning, as fig. 3.1(c) depicts. Nehalem CMPs, having a classical LLC addressing scheme, follow this model and can be partitioned in the same way as previous CPUs. Therefore, the page coloring implementation used throughout the state-of-the-art work of section 3.3 could theoretically be employed.
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. fig. 4.1 shows the use of the physical address to access the L2 cache of our target platform, which differs from fig. 2.1 as the set number consists of 9 bits instead of 13.

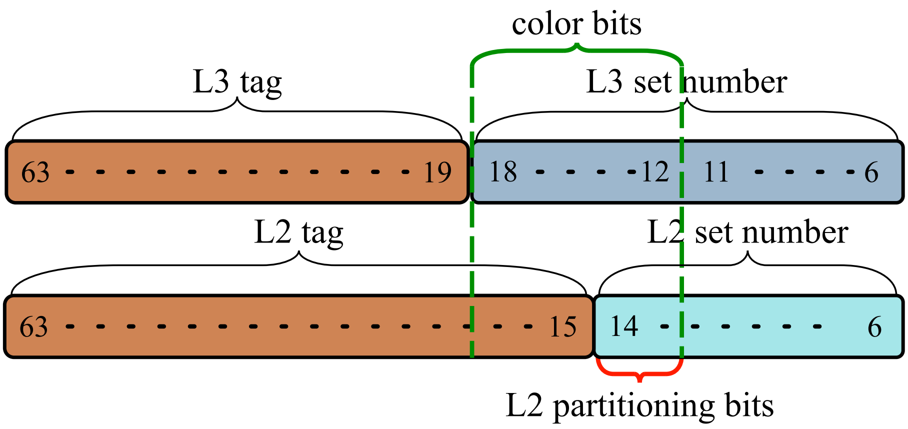


Figure 4.2: Overlap of color bits and L2 set bits in a Intel Xeon W3540

Among these bits, some overlap with the LLC color bits, as in fig. 4.2, where the 3 most significant bits of the L2 set number are in common with the LLC color bits.

This overlap causes also the L2 cache to be partitionable through the subset of common bits we identified. Partitioning this level of cache can penalize the running thread because it would restrict the the data it can keep inside this layer; this, in turn, increases the accesses to the LLC and hence the overall memory access latency. Even with SMT-enabled cores, partitioning the L2 cache can be a detrimental operation, since it is very hard to balance the partition sizes, and an unbalanced partitioning may result in dangerous bottleneck effects. Therefore, in this work we choose not to partition this layer, thus preventing the use of the common bits. This decision is another key characteristic of *Rainbow*, that ensures the full exploitation of the lower cache levels capacity in CMP architectures. Thus, the remaining, highest color bits are those effectively usable for LLC partitioning, and are 4 in the example of fig. 4.2. This limits the number of partitions to 16, a granularity that, though not fine, is enough for the purpose of this work.

Unlike the L2 cache, the lowest L1 cache (assumed to be doubled for data and instructions) is usually virtually indexed, so that partitioning the upper layers does not affect the data placement inside this layer. However, even if this layer was physically indexed, its size is very limited,

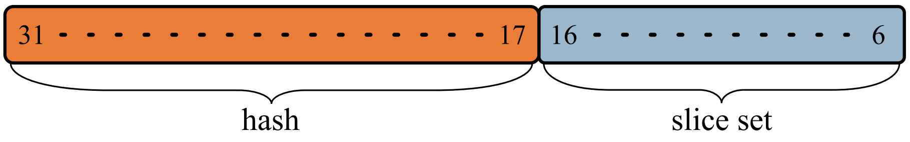


Figure 4.3: Utilization of address bits in Intel Core i7-2600

so that the set bits of both L1 caches are fewer than those of the LLC, with no overlap.

Exploiting mathematical formalism, we can finally define the notion of page color for the Intel Xeon W3540 of fig. 4.2 in order to identify the single LLC partitions of minimal size. If we represent the i-th bit of the address b in fig. 4.2 with the notation b_i, thus $b = b_{47}b_{46}...b_0$ with positional notation [1], for the Nehalem architecture we can define the color as a function $c : \{0 \ldots 2^{48} - 1\} \rightarrow \{0 \ldots 15\}$ of the address in the following way:

$$color_N(b) = b_{18}b_{17}b_{16}b_{15}$$

4.3.3 Partitioning Sandy Bridge's hierarchy

With Sandy Bridge, Intel adopted a hash-based LLC addressing scheme, as discussed in section 2.3.2. Based on the physical address, the hash function computes the LLC slice the line resides in. The hash function is undocumented, but past work has unveiled interesting details. In particular, Hund, Willems, and Holz [38] reconstruct the hash function of a specific Sandy Bridge model for security purposes, showing that the hash employs all of the higher bits as input. In fig. 4.3, representing the Intel Core i7-2600 CMP used in [38], bits 17 to 31 are used for the hash computation, while bits 6 to 16 address the index inside a specific slice. Wishing to use the same coloring mechanism devised for Nehalem, we must take in account the parameters of the L2 cache, which is identical to that of Nehalem CMPs. Hence, fig. 4.4 shows that bits 12 to 14 still overlap with the L2 set index, so that only bits 15 and 16 are available for partitioning, with 4 possible partitions inside each slice. Since the hash function maps an address to a slice in an unpredictable but statistically fair way, even controlling bits 15 and 16 any process requiring more than

[1]physical addresses are composed of at most 48 bits in today's machines

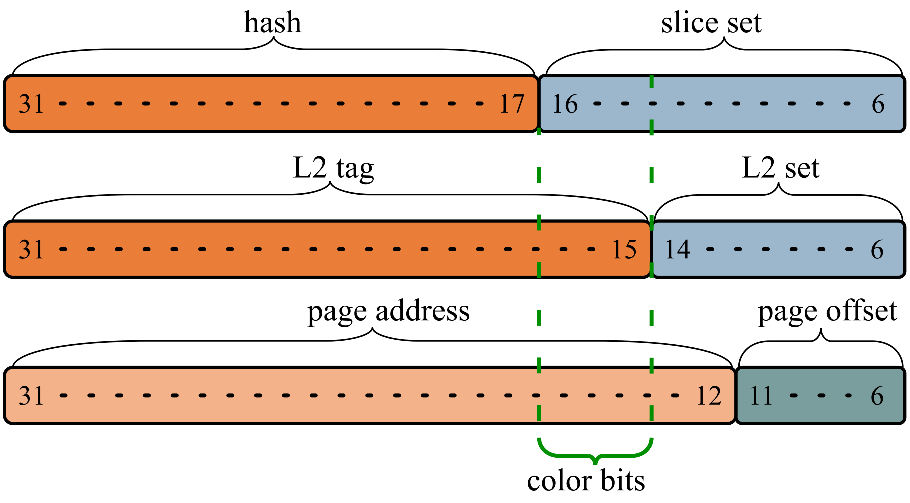


Figure 4.4: Color bits in Intel Core i7-2600

$2^{16} = 64KB$ of memory can receive any slice, because the bits higher than 16 vary. Thus, any process with a realistic amount of memory, if colored by means of bits 15 and 16, would receive the same partition on all the slices (4 in the case of the Intel Core i7-2600). Therefore, the final number of controllable partitions would be 4. This is, yet, a too coarse granularity for an environment with multiple, different processes such as the one we target.

To have a better control over the partitioning, we must assume the knowledge of the hash function, which can be found in a similar way to [38]. With this information, we can control both the cache sets inside the slice and the slice itself, for a total of 16 partitions, an acceptable granularity. To devise a more formal model, we can define the hash function h as a function from the address space to the set of slices: $h : \{0 \ldots 2^{48} - 1\} \to \{0 \ldots 3\}$. Using a positional notation, which is more useful in our context, the hash function can also be represented as $h(b) = h_2(b)h_1(b)$, where h_1 and h_2 are the specific hash functions that compute each bit, as in [38, page 198]. Thus, we can define the color also for a Sandy Bridge platform by concatenating the hash function and the per-slice color bits, in the following way:

$$color_{SB}(b) = h_2(b)h_2(b)\ b_{16}b_{15}$$

In this way, with this notion of color in a Sandy Bridge CMP we uniquely identify each partition inside a slice by means of bits 15 and 16 and each slice by means of bits h_1 and h_2.

For the sake of generality, this notion could be extended in a similar way to different CMPs than Intel Core i7-2600; for example, for an eight-core CMP the hash function is defined as $h(b) = h_3(b)h_2(b)h_1(b)$, and the color definition immediately follows. Studying the characteristics of recent Intel's CMPs, we find that many features such as the number of per-slice sets (2048 in Intel Core i7-2600) and the number of L2 sets do not vary, even if the number of cores does; thereof, bits 15 and 16 always identify the per-slice partition. This makes the definitions in this section general enough to model many CMPs without further effort.

Finally, the assumption of knowing the hash function is hardly satisfied in practice. Nonetheless, it is possible to reconstruct this information as Hund, Willems, and Holz [38] did and chapter 5 explains a repeatable methodology to find this function. However, generally, it is not possible to rely on single attempts of reconstruction in order to build a CMP-abstraction framework embedded within the OS, as a production-ready implementation would require. To this final aim, the cooperation of CMP manufacturers is fundamental.

4.4 Consequences of page coloring on applications

The strict dependence of the LLC mapping on the physical page address due to the PIPT addressing scheme has several consequences. If a task receives an LLC partition, becoming a *colored* task, it undergoes some limitations, which this section discusses. In particular, section 4.4.1 shows how page coloring limits the physical memory available to the colored task, while section 4.4.2 shows why hugepages become unavailable with page coloring.

4.4.1 Cache-Memory constraint

Page coloring essentially consists in choosing certain memory pages to control data mapping to the LLC. Therefore, if a task receives a set of

colors the physical allocator can choose only the pages of those colors to fulfill the task's memory requests. Thus, the task is inherently associated those pages, which are a subset of the entire memory of the machine. This means that, if a task has already consumed all these pages and requires further memory, its requests cannot be satisfied, unless using pages from other colors. But this last possibility causes the tasks to use LLC areas that are used by other tasks, which gives lieu to contention over the shared LLC sets.
As *Rainbow* aims to enforce strict isolation within the LLC, a key design decision we made is to use only the user-reserved colors. Thus, since *Rainbow* assumes the number of colors (and, implicitly, the corresponding memory pages) as a user's input, it is fundamental that this input be properly chosen with respect to the memory footprint of the task, in addition to the desired final performance.

If the user underestimates the number of colors, the allocator cannot satisfy the requests and upper kernel layers may decide to swap memory pages to a disk in order to perform the allocation. But this operation has an overhead that is typically intolerable. Furthermore, if the pressure on the kernel memory subsystem increases over a certain limit, the allocation might fail and the process terminate erroneously.

Quantifying the memory percentage an LLC partition reserves to a process is simple in the case of the Nehalem platform. If N is the number of memory pages and C is the number of colors (always a power of 2 for both Nehalem and Sandy Bridge, as from section 4.3.2 and section 4.3.3) and N is a multiple of C, then each color is associated to exactly $n = \frac{N}{C}$ pages. Hence, a process that receives c colors can allocate over $\frac{N}{C} \times c$ pages. Thereof, the percentage of memory a process can use is equal to the percentage of LLC it receives from the user.
The assumption that N is a multiple of C is a good approximation of reality: indeed, the RAM memory in modern machines is provided in sticks of relatively big capacity (around 1, 2, 4 GB of more), this amount being very close to a power of 2. Therefore, the single stick holds a number of pages that is, approximately, a power-of-2 multiple of C, and more sticks, even if of different sizes, sum up to a page amount that is a multiple of C. A more precise quantification would require considering the exact size of each memory stick, which depends on the vendor and

the model. However, since colors repeat over the memory pages due to the color bits being a subset of the page address, the differences among colors are negligible in today's machines with millions of pages.

In the case of a Sandy Bridge platform, it is not possible to compute the number of pages per color in advance, because the hash function controls the bits h_1 and h_2 in an unpredictable way. Nonetheless, we can assume the output of the hash function to be evenly distributed, as from its design goals (section 2.3.2). Since the configurations of bits 15 and 16 repeat over the memory pages as the colors in Nehalem, we assume negligible differences in the number of pages with respect to each bits configuration. Within these pages, the hash bits are evenly distributed, overall causing the page colors c_{SB} to be evenly distributed as well. Therefore, also with Sandy Bridge we can well approximate the percentage of memory to the percentage of LLC.

In general, this memory constraint can be a limitation for applications with a big memory working set and a small cache working set. Yet, applications with such characteristics are rare in practice, as the LLC footprint is roughly proportional to the memory occupation for compute-intensive applications. For other applications, a huge working set is often due to large I/O buffers, which usually have low cache affinity and can be swapped out after use due to low reuse probability. To avoid situations where kernel I/O buffers take most of the task's memory, we choose to allocate these buffers all over the RAM memory, without the restrictions of coloring. This also allows ensures to the kernel high availability of memory for its functioning, which is fundamental to preserve the machine responsiveness and stability. However, we believe that, in general, an approximate knowledge of the task's input on behalf of the user is sufficient to limit the cache-to-memory constraint: for example, some cloud infrastructures require end users to provide an a-priori estimation of the memory they use.

4.4.2 Limits on physical page size

The second limitation is the impossibility of using hugepages (see section 2.5), available in some modern architectures.

Figure 4.5 shows the very different granularities of hugepages and

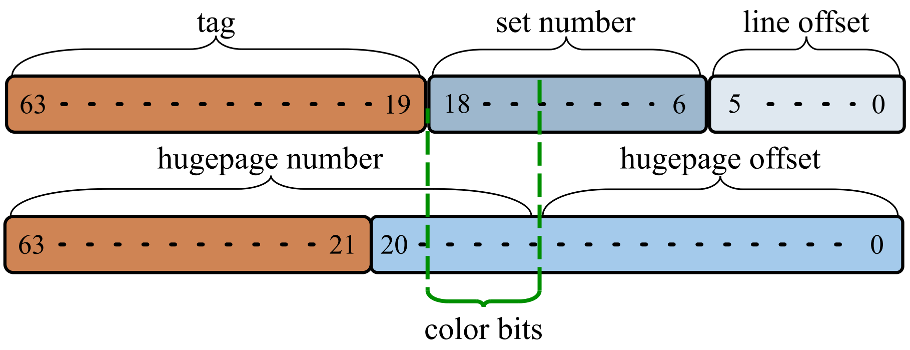


Figure 4.5: Bit fields for hugepages and LLC

No bits are in common between hugepage address and LLC set index

of the LLC management that cause the color bits to overlap entirely with the page offset. Therefore, in Nehalem a single hugepage covers all the possible colors of the platform and is mapped to all the cache sets, denying isolation. Similarly, with Sandy Bridge the hugepage offset overlaps with bits 15 and 16 and with part of the page offset, used for the hash: the lack of control over these bits causes the data to be spread evenly among all the LLC slices, and in any set of the slice.

Since a hugepage causes data to be mapped to any set, neither isolated applications nor the others can leverage this feature. Unfortunately, the PIPT addressing makes this limitation impossible to relax. Page coloring and hugepages could coexist only with hardware changes like a more diverse granularity, but any such modofications are not planned in future CMPs.

4.5 Rainbow Buddy

Implementing page coloring requires noticeable modifications to the buddy algorithm and data structure. The aim of these modifications is to permit the allocation of a page of a specified color. These modifications are based on the LLC parameters identified in section 4.3. In particular, section 4.5.1 explains the modifications to the data structures and section 4.5.2 shows those to the basic algorithms for insertion and removal of buddies.

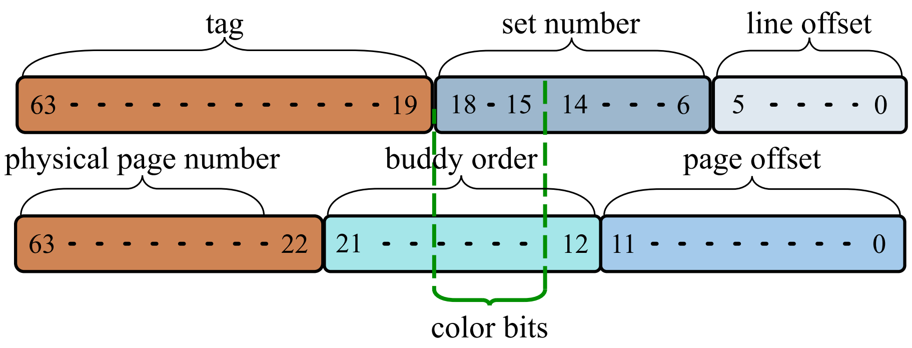


Figure 4.6: Overlap of color bits and buddy bits in Nehalem

4.5.1 Mcolors and data structure modifications

To allow "colored" allocations, 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 by choosing one of the colors allocated to the application. We have to note that the notion of "color" applies natively to pages, that is to say, in the context of the buddy allocators, to buddies of order 0; and the color varies over consecutive pages. When the buddy order increases, more and more lower bits of the buddy address become 0 (see section 2.4.1). Above a certain order the address bits being forced to 0 start overlapping with the color bits, as fig. 4.6 shows, and the configurations available for the color bits are less and less as the order increases. Thereof, after a certain order all the color bits are forced to 0. For Nehalem, all the colors are available if the zeroed bits are below bit 15, corresponding to orders 0 to 3 (included). Hence, for the buddies of these order the notion of color applies as well. Always considering Nehalem, buddies with order from 4 to 6 (included) have more and more color bits forced to 0, progressively reducing the possible color configurations from 8 to 2, and buddies with order higher than 6 have all the color bits being 0, thus one configuration. Considering, for example, a 4-order buddy, it spans two 3-order buddies which have two different colors, with the first 3 bits in common and the lowest bit variable. Hence, it is natural to identify the color of a 4-order buddy with the highest 3 bits only, as the fourth bit may vary. And, similarly, a 5-order buddy contains two 4-order buddies, and has only two fixed color bits; and so on. This leads us to define the notion of *multi-color* or *mcolor* in order to generalize the color of a buddy with respect to its order. Therefore, if b is the address of the buddy and d its order, we can mathematically define the mcolor for Nehalem as

$$mcolor_N(b,d) = \begin{cases} b_{18}b_{17}b_{16}b_{15} & \text{if } 0 \le d \le 3 \\ b_{18}b_{17}b_{16} & \text{if } d = 4 \\ b_{18}b_{17} & \text{if } d = 5 \\ b_{18} & \text{if } d = 6 \\ 0 & \text{otherwise} \end{cases}$$

For Sandy Bridge, the presence of a hash function leads to troubles. In case of orders from 0 to 5 the definition of mcolor we have given still applies, simply replacing $b_{18}b_{17}$ with h_1h_2. Similarly, in case of order 7 or greater, our definition applies equally. The case of order 6 is particular because of the unpredictability of the hash function: in fact, a 6-order buddy with hash $h_1h_2 = 10$ can, for example, be split in two 5-order buddies with hash $h_1h_2 = 10$ and $h_1h_2 = 01$ respectively, because bit 17 might be XORed with both h_1 and h_2, flipping both hash bits when it is 1 (as in the second buddy). As will be clear in the following, a definition of mcolor also for Sandy Bridge is still possible, but depends on the specific hash function. Therefore, we leave the details on this topic to chapter 6.

Leveraging the notion of mcolor, it is natural to split each order list into several sub-lists, one per mcolor, to keep the lookup operation fast and maintain efficiency. Figure 4.7 depicts the new structure *Rainbow* buddy allocator uses: an array of lists per mcolor, which can be accessed in constant time by adding the mcolor to the base pointer.
Therefore, the final subdivision of buddy lists takes in account three aspects: order, migratetype (recalling section 2.4.3) and mcolor. In the following section, for the sake of simplicity we will ignore the presence of the migratetype; in fact, this parameter is constant during an allocation and does not vary often across allocations, since userspace requests (the only colored allocations) are typically fulfilled with buddies of the highest mobility.

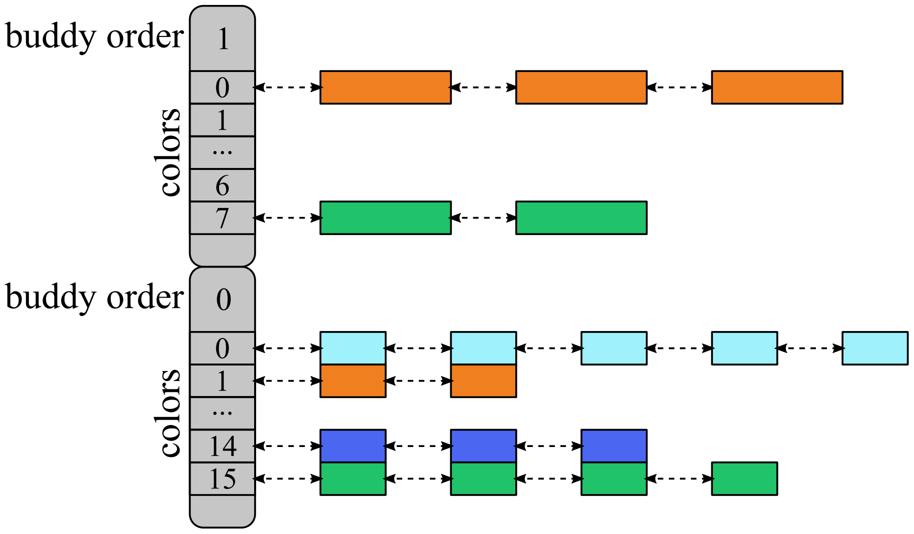


Figure 4.7: Rainbow Buddy data structure

The designed data structure with colored lists per-order; in the higher levels multiple colors are aggregated

4.5.2 Algorithm modification

Leveraging the data structure we designed in the previous section, we consequently have to modify the buddy algorithm, with the goal of maintaining its efficiency.

Within the *Rainbow* buddy allocator, the coalescing procedure is, overall, unchanged, since it depends on the physical contiguity of the two buddies. The only change is the final insertion, in which the insertion list now depends not only on the order and migratetype but also on the mcolor of the buddy. The insertion algorithm is explained in the following pseudo-code snippet. Here, `MCOLOR` is the function that returns the buddy mcolor from section 4.5.1, either for Nehalem or for Sandy Bridge, and the buddy migratetype, being constant during the coalescing procedure, is not shown. The code snippet shows a recursive procedure that summarizes the insertion functioning in *Rainbow*.

```
globaldata: list_head buddies[MAX_ORDER][MAX_COLORS
    ]

```

```
procedure INSERT_BUDDY
input:  buddy_address addr, buddy_order ord
output: none
behavior:
  mcolor = MCOLOR(addr, ord)
  // address of the physically contiguous buddy
  twin_addr = addr + (1 XOR ord)

  if (ord < MAX_ORDER - 1) AND (BUDDY_IS_FREE(
     twin_addr))
    new_buddy = COALESCE_BUDDIES(addr, twin_addr,
       ord)
    INSERT_BUDDY(new_buddy, ord + 1)
    return
  else
    INSERT_INTO_LIST_HEAD(buddies[ord][mcolor],addr
       )
end procedure
```

As from section 2.4, the “twin” physically contiguous buddy is computed by means of a XOR operation on the order-th bit of the buddy address. The procedures `BUDDY_IS_FREE` and `COALESCE_BUDDIES` act on the array of physical pages (described in section 2.4.1) to store or retrieve information about the buddy (order, migratetype, allocation status, etc.), run in constant time and their details are of little interest here; however, the names explain their roles.

The splitting procedure, instead, is more complex. The key issue is that, if a buddy of a desired color is not present, the allocator must split a higher order buddy in which the requested color is present, to guarantee the allocation time be constant and avoid searching. To do this, we need a way to select the proper mcolor from the requested color. For the Nehalem architecture, we note that, increasing buddy order, the lower color bits are discarded in order to compute the mcolor. In a similar way, we can discard the lower bits of the requested color to have the mcolor of a certain order. For example, if a page of color 13 is requested and no buddy of order from 0 to 3 is present of color 13, the allocator has to split a buddy of higher order in which a page of color 13 is present. Color 13, or 1101_2 in binary format, corresponds to mcolor 110_2 of order 4: if a buddy of such order is present, it is split into two halves, differing in the lowest bit, thus being 1100_2 and 1101_2 respectively. After the split, the allocator continue the split with the second buddy to reach order 0 and store the former in the 3-order free list. Similarly, if no 4-order buddy of color 110_2 is present, the allocator has to split a 5-order buddy of color 11_2, choose the first half and split it further; and so on. This procedure, overall, is called *color chasing* and is the key innovation of the splitting procedure. Leveraging mathematical formalism, if the requested color is $c = c_3c_2c_1c_0$ and the requested buddy order being checked is d, the mcolor to look for is:

$$mcolor\_lookup_N(c,d) = \begin{cases} c_3c_2c_1c_0 & \text{if } 0 \leq d \leq 3 \\ c_3c_2c_1 & \text{if } d = 4 \\ c_3c_2 & \text{if } d = 5 \\ c_3 & \text{if } d = 6 \\ 0 & \text{otherwise} \end{cases}$$

Again, in Sandy Bridge this function depends on the hash, and we postpone the discussion to chapter 6.

The following pseudo-code snippet summarizes the splitting procedure, which starts from `EXTRACT_PAGE`: this procedure looks for a page of the requested color. If it cannot find one, it invokes the procedure `SPLIT_BUDDY`, which looks for a suitable higher-order buddy performing color chasing by means of the functions *mcolor* and *mcolor$_l$ookup* defined previously (in capital letters inside the code snippet).

```
globaldata: list_head buddies[MAX_ORDER][MAX_COLORS
    ]

procedure SPLIT_BUDDY
input:  buddy_order ord, mcolor col
output: buddy_addr
behavior:
  buddy_order local_ord
  buddy_addr local_addr, twin_addr
  mcolor local_mcol, addr_mcol

  if ord == MAX_ORDER
    // no buddy exists
    return nil

  local_mcol = MCOLOR_LOOKUP(col, ord)
  if LIST_IS_EMPTY(buddies[ord][local_mcol])
    // no buddy found in this order
    local_addr = SPLIT_BUDDY(ord + 1, col)
    if local_addr == nil
```

```
      return nil
  else
    // get buddy for splitting
    local_addr = REMOVE_LIST_HEAD(buddies[ord][
        local_mcol])

  twin_addr = local_addr + (1 XOR ord)

  // color chasing
  addr_mcol = MCOLOR(local_addr, ord - 1)

  if first_mcol != local_mcol
    // twin_addr is the buddy to return
    SWAP_VARS(local_addr, twin_addr)
  INSERT_INTO_LIST_HEAD(buddies[ord - 1][addr_mcol
      ],twin_addr)

  return local_addr
end procedure

procedure EXTRACT_PAGE
input:  mcolor mcol
output: buddy_addr
behavior:
  if LIST_IS_EMPTY(buddies[0][mcol])
    return SPLIT_BUDDY(0,mcol)
  else
    return REMOVE_LIST_HEAD(buddies[0][mcol])
end procedure
```

The procedure SPLIT_BUDDY starts by checking if buddies of the requested order may exist. Then, it looks for a suitable buddy to split, recursively requesting one if none is present in the current order. After finding the buddy to split, the procedure and computes the address of the “twin” buddy of smaller order. Then, it performs color chasing by checking which of the two buddies has the desired mcolor, either the local (variable local_addr) or the twin (variable twin_addr). In case the twin is the desired buddy, the procedure swaps the two variables so that, finally, local_addr contains the needed buddy and twin_addr the other one, to be stored in the free list.

4.6 *Rainbow* interface

To exploit the LLC partitioning facilities introduced with *Rainbow*, an application needs a suitable interface. In particular, an application should be able to declare the size of its LLC partition to the kernel via a userspace interface, so that the kernel, then, allocates its memory properly. An additional system call is a technically viable solution to realize such interface, but is not in line with Linux development guidelines [2]. Moreover, since several parameters are of interest for the new *Rainbow* capabilities, the system call interface can become complex and unclear.

In the present years, Linux developers prefer filesystem-like interfaces to expose non-core functionalities, like the generic *sysfs* interface [82] and, in particular, the *cgroups* [15] facility. A cgroup is, essentially, a hierarchical interface on a system resource (memory, I/O, CPUs, etc.) exposed as a folder hierarchy. A user can partition a resource by creating a subdirectory inside the main one. Each subdirectory contains several files, which control the resource parameters (for example, the memory amount, the I/O bandwidth, the set of allowed cores, etc.). To change one of those parameters, it is sufficient to write the proper file, thus changing the resource allocation. Similarly, to constrain a process to a resource partition by associating the process to a cgroup, it is sufficient to write the Process IDentifier (PID) to a specific file.

Therefore, we chose to introduce a new cgroup called *cacheset*, which serves as an interface to *Rainbow*'s capabilities. A user creates an LLC partition by making a subdirectory inside the main cacheset directory and chooses the number of colors of the partition by writing a file, and the tasks using that LLC partition by writing another. This flow well fits the way distributed computing platforms are managed: typically, each server has a centralized manager that monitors running applications and creates new ones on demand. This manager can exploit the cacheset cgroup in the way described above, associating a task to a certain cgroup based on the task's memory footprint, the QoS requirements and any high-level policy.

[2]Linux developers are unwilling to introduce new system calls, unless a wide audience of users shows an evident and frequent need of it.

On the kernel side, Linux automatically redirects the filesystem operations to the LLC management functions, in a transparent way to the user. Since recent Linux versions have a dedicated, independent interface to add a cgroup, no change is needed to the filesystem management. Thereof, adding cacheset consists essentially in "hooking up" into the Linux cgroup interface, as explained chapter 6.

Reconstructing Sandy Bridge hash function 5

Tell me, o Muse, of the man of many devices, who wandered full many ways after he had sacked the sacred citadel of Troy. Many were the men whose cities he saw and whose mind he learned, aye, and many the woes he suffered in his heart upon the sea, seeking to win his own life and the return of his comrades.

Homer, Odyssey I
translated by A.T. Murray

Section 2.3.2 explains the novel architecture of Intel's Sandy Bridge family, which goes almost unchanged also through the following Ivy Bridge and Haswell families. For our purposes, the key innovation is the aforementioned introduction of a hash-based addressing scheme, that computes the LLC slice a cache line maps to from the line address. Since this hash is undocumented, the control over the LLC *Rainbow* needs, as from section 4.3.3, is reduced to the extent that it is impossible to control the slice data are mapped to, resulting in only 4 partitions available. Therefore, in the same section 4.3.3 we assumed the knowledge of such hash function, as it is fundamental for *Rainbow*'s goals.
This chapter, thus, presents the methodology we used to reconstruct this information, also unveiling some inner details of the CMP model we worked with. In particular, section 5.1 explains the assumptions we leveraged and, consequently, the methodology we followed to reconstruct the hash function. Then, section 5.2 explains how the experiments were conducted, in particular how the hardware and software were configured. Section 5.3 shows the first tests performed on our CMP, which are interpreted in 5.4 to find 6 possible hash functions that fit with the measurements. Section 5.5 shows another set of tests performed to finally

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6
h2	⊕		⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕	⊕		⊕											
h1	⊕	⊕	⊕		⊕		⊕		⊕		⊕		⊕	⊕												

Figure 5.1: Reconstructed hash function of Intel Core i7-2600

find the hash of our CMP. Based on the findings of this last section, section 5.6 infers additional details about the architecture.

5.1 Assumptions and methodology

Reconstructing a hash function can be very complex task, in particular with cryptographic hashes. Yet, these hashes, if implemented in hardware, have an area cost and a latency that are not compatible with the requirements of a component inside a CMP core: indeed, Intel's hash function must work at more than 3 GHz frequency and must provide the result within less than 30 cycles, the average latency of an LLC hit in Sandy Bridge. Therefore, this hash must be simple and have small area and power overhead.
Hund, Willems, and Holz [38] unveiled the hash of a Intel Core i7-2600, showing that it is basically a XOR operation on bits of the tag, as fig. 5.1 depicts. Since the hash function h must return a number from 0 to 3, its output uses two bits, each one computed according to a different XOR-based hash: in positional notation, $h(b) = h_2(b)h_1(b)$. A posteriori, we could try to infer the rationale of this design choice. A first reason is that the XOR operator has evenly distributed output values, thus without biasing the hash and, thereof, the slice choice: all the 4 configurations have equal probability. Secondly, XOR ports have very small latency, area and power overhead, in particular with Intel's full custom Application Specific Integrated Circuit (ASIC) lithography.

Thanks to these characteristics, we can expect this design choice to hold across all the models of the Sandy Bridge family, and, with good confidence, also across other families and models. The bits used within each hash can change, but we assume the XOR operator to be the only one Intel uses. We have to stress this is a critical assumption: if it was wrong, then much more complex techniques than those we use here would be necessary. But our findings finally proved our assumption to be correct.

Once the bit-combining operator is known, the remaining task is to discover *which* bits h_1 and h_2 combine. Since our CMP model, a Xeon E5-1410, is different from that in [38], we cannot assume the hash function to be exactly the same. However, exploiting the few hints in [38], we can learn from their reconstruction methodology to reach the same result with our CMP.
Therefore, the main step consists in finding collisions among memory addresses. Using Intel's performance counters, we can detect whether an address causes an LLC miss [21, chapter 18]. By reading proper sequences of addresses (as detailed in the following), we can find whether an address has evicted another one: in such case, the two addresses collide, hence are mapped to the same LLC set and must have equal hash. Following the procedure in [38], we will collect groups of memory addresses that collide in the LLC. In particular, given a fixed memory address called *prober*, we find several other addresses, called *colliders*, that collide with the prober. Then the comparison of these addresses let us infer the influence of each bit over h_1 and h_2.

5.2 Environment setup and preliminary considerations

The machine Hund, Willems, and Holz [38] use ships 4 GB of RAM: this limits the physical memory bits to 32, so that the hash function works with bits from 0 to 31. Higher bits are 0 and, even if they are used in the hash, they do not affect the XOR operator. Instead, the machine we use for the measurements has 6 GB of RAM and uses thus also bit 32: to learn the influence of this bit too, we have to extend the collection of colliding addresses also to the upper 2 GB of memory, where this bit is set.

The necessity of knowing physical memory addresses forced us to make measurements with ring-0 privilege level, thus inside an OS. We chose Linux, which allows users to insert executable *modules* at runtime and provides good documentation and interfaces for a deep control over the OS and the hardware, even from external modules[57, chapter 7]. Another key functionality of Linux was the possibility of accessing the whole memory space with ease, since the x86_64 version of Linux maps the entire physical memory of the machine to the kernel address space, thanks to the huge availability of virtual memory with 64 bit machines [95].

Since our aim is to collect the *exact* memory addresses that collide with each other, it is fundamental to have very precise measures. Yet, in today's architectures, many features can potentially disturb the measurements, leading to noisy results that are useless for our goals. Moreover, Intel's performance counters are not conceived for measurements of high precision, but to characterize long execution phases of applications (in the order of milliseconds or more), when the hardware monitors typically count millions of events: for this purpose, differences between the counters and the "true" values of few hundreds of events are negligible, while in our testbed they would totally invalidate our results. Therefore, it is absolutely necessary to disable all sources of noise in the testbed.

A first example of a disturbing feature is speculative prefetching, which loads data according to forecasts based on the memory access pattern. In particular, modern architectures have prefetching units in the pipeline front-end, to prefetch instructions from the L1, and inside each layer of cache, to prefetch lines from the upper memory layer. Prefetchers can disturb measurements by pre-fetching unneeded lines that evict other lines, thus increasing the miss counter. Since it is not possible to know which address caused the miss, misses due to prefetchers could be confused with those artificially caused by out tests, thus invalidating the measurements. In our machines, it is possible to disable cache prefetchers from the configuration panel of the Basic Input-Output System (BIOS), while instruction prefetchers cannot be disabled. However, since they prefetch from the L1, we expect them not to affect the measures.

Sandy Bridge CMPs aggressively adjust the frequency based on the cores utilization through Dynamic Voltage-Frequency Scaling (DVFS), with a hardware/software control. To stabilize the measurement envi-

ronment and limit unpredictable behavior like event loss or delays, we set in software the minimum frequency and disable any power-saving mechanism. Although the hardware may still decide to make small DVFS regulations independently, we assume these actions to be limited and not to pollute measurements sensibly.
Since our test kernel module runs on a single core, other cores are a major source of noise: in fact, they simultaneously run other applications, system services and interrupt handlers (like the clock interrupt handler, that drives preemptive kernels). According to our experience, it is fundamental to disable the other cores to achieve much higher predictability and precision; for the same reason, all interrupts on the working core are disabled when the module is loaded.

Modern CMPs are designed to aggressively execute instructions out of order, leveraging speculative mechanisms that proved to be very effective. These features often change the order of instruction execution in order to hide pipeline and memory latencies. Since it is impossible to know which memory address caused the miss event we eventually measure, we can rely only on the memory accesses we trigger by software to understand which memory address caused the miss event. Hence, we have to enforce the execution of memory accesses in exactly the same order they appear in software. Since it is impossible to disable speculative, out-of-order execution, we can only attempt to enforce an in-order execution. Therefore, we disable all the compiler optimizations, which reschedule instructions, we force each memory access by the C `volatile` qualifier and we explicitly protect accesses with a hard memory barrier (the x86 `mfence` instruction[20, section 3.2]), which forces both the compiler and the pipeline to maintain the order of memory accesses.
Finally, we learned from our experience that Intel's performance counters might have delays in the updating the register values, in particular with consecutive accesses. Still from our experience we learned that, most of the times, the register is updated as expected after multiple reads. Despite all these precautions, we experienced that some noise is inevitable.

We made the tests taking into account all these details, writing the modules' code with care, barriers and pointer access modifiers, and also checking the output assembly code. However, in describing the tests we made we will omit to show all these precautions, not to overwhelm the reader with an amount of details that can hide the basic concepts.

5.3 Collecting colliding addresses

Finding two colliding addresses in the LLC requires considering all its parameters. Looking at fig. 5.1, we notice that bits 6 to 16 are used for the set index and bits 0 to 5 are used for the line. Therefore, two addresses that have bits from 0 to 16 in common map to the same set number and line offset. To compute these addresses, we use a parameter called *stride*, which is equal to 2^{17}: two addresses are mapped to the same set number if their distance is a multiple of the stride. Given a probe address, we can find multiple colliders by repeatedly summing the stride to the probe and checking whether the probe is evicted after every access. Yet, adding the stride increments the tag, which is the input of the hash function: therefore, two addresses differing of a stride are mapped to the same set number, but might unpredictably be mapped to different slices, without colliding. Hence, the number of memory locations we have to traverse to find a collision is not a priori known. Since the associativity of the Xeon E5-1410 is 20, in the best case 20 memory locations with offset equal to the stride can map to the same set inside the same slice, filling the set; in such case, if also the 21st memory location maps to the same slice, it evicts the probe address, and we can detect this event by reading the probe and checking whether a miss happens. However, this scenario is impossible in practice, because a hash function like that of fig. 5.1 changes frequently with the lower bits. Thereof, we expect the data we access to be evenly distributed among the 4 slices: in this more realistic case, 78 accesses are needed on average to fill a set, and about 80 to evict the probe and trigger a miss when the probe is subsequently read. And, in our measurements, we found very close numbers.

We can summarize this test with the following code snippet, which looks for a collider by checking whether the global probe has been evicted.

```

globaldata: integer stride
        address probe
```

```
      integer collider_offset

procedure FIND_COLLIDER
input:  none
output: address
behavior:
  integer i, miss_count, jumps
  address first_collider

  // compute address of first collider
  first_collider = probe + collider_offset * stride

  for jumps from 20 on
    // bring probe into LLC
    READ_ADDRESS(probe)

    // bring colliders into LLC
    for i in [0,jumps)
      READ_ADDRESS(first_collider + i * stride)

    miss_count = READM_MISS_COUNT()
    READ_ADDRESS(probe)

    if (READM_MISS_COUNT() > miss_count)
      // last collider hash evicted the probe
      return first_collider + i * stride
end procedure
```

For the sake of simplicity, we wrapped the low-level details with the functions READ_ADDRESS, which reads the memory address with the hard barries, and READM_MISS_COUNT, which reads the miss counter register using Intel's dedicated performance counter [21, chapter 19.5].

By changing the variables probe and collider_offset, we can find multiple couples probe -collider. In particular, by varying only collider_offset we can find multiple colliders associated to the same probe, which are useful for the following steps. Furthermore, by adding to each probe and each collider an additional, constant offset lower than the stride, we can map addresses to a given set number, not necessarily 0, collecting conflicting addresses with the lower 17 bits being not all 0. This will let us verify that bits lower than 17 are not used in the hash, as fig. 5.1 found. Overall, with these experiments we collected several probes p_a, p_b, etc. and multiple colliders for each probe, represented as sets $\{c_{a,1} c_{a,2}, ...\}$, $\{c_{b,1}, c_{b,2}, ...\}$, etc., accounting for more than 2'7000'000 millions addresses.

5.4 Results interpretation

With these data, we can have a first insight on the hash bits and on the measurement noise. To evaluate how noisy the measures are, we perform a first test on our data. Considering a single probe p and its colliders $\{c_1,\ c_2, ...\}$, we know that the hash of all these addresses are equal as they conflict in the LLC. If we consider any two colliders c_i and c_j of the same probe, their Hamming distance $H(c_i, c_j)$ cannot be 1, that is to say they cannot differ for a single bit: if this happened, the hash would be different, because h_1, h_2 or both would change because of this only bit. Thus, finding couples of colliders of the same probe that have Hamming distance of 1 gives an idea of the noise affecting the measurements. In our experiments, less than 0.03% of the measures fulfilled this condition; once two such colliders are found, they are discarded.

Assuming that all the bits 17 to 32 are used within at least one of the two hash functions, as in [38], the problem of reconstructing the hash $h = h_2h_1$ may be reduced to a clustering problem. Specifically, we expect only three clusters of bits:

1. the cluster of bits only in h_1
2. the cluster of bits only in h_2
3. the cluster of bits both in h_1 and in h_2

To find these clusters, we look for colliders of the same probe having Hamming distance of 2. Since these colliders have the same hash, we infer that the two changing bits do not change the overall hash. This is due to the property of XOR: if two bits change at the same time, their XOR does not change. This, in turn, implies that the two bits are in exactly one of the following *configurations*:

1. both bits are used in h_1
2. both bits are used in h_2
3. both bits are used both in h_1 and in h_2

For each couple of colliders at Hamming distance 2 we found the two different bits i and j, and counted how many times i and j appeared across the whole dataset. This count represents the *likelihood* $l_{i,j}$ of bits i and j to be in the same configuration.

In a similar way, we need an estimation of the *unlikelihood* $u_{i,j}$ of bits i and j to be in the same configuration. The higher the unlikelihood, the higher is the probability that i and j are not in the same configuration. To find such couples, we can consider two probes p_a and p_b with Hamming distance 1, thus guaranteed to have different hashes, and two colliders $c_{a,i}$ of p_a and $c_{b,j}$ of p_b with Hamming distance 2. Since the hashes of $c_{a,i}$ and $c_{b,j}$ must be different, the two different bits must have a different configuration, otherwise they would "flip" the XOR twice and cause the hash values to be equal. Counting the occurrences of such couples, we devised an estimation of the unlikelihood $u_{i,j}$ for each couple of bits.

Finally, likelihood and unlikelihood allow us to cluster bits based on their configuration. To this aim, we developed a simple clustering based on an Integer Linear Programming (ILP) model. Introducing the binary variable $s_{i,j}$ that represents whether bits i and j are in the same cluster, we can write the objective function as

$$\text{minimize} \sum_{i=17}^{32} \sum_{j=i}^{32} [(1 - s_{i,j}) + l_{i,j} - s_{i,j} \times u_{i,j}]$$

To indicate whether a bit i is inside cluster c, we also introduce the binary variables $t_{i,c}$, and add the constraint

$$s_{i,j} \geq t_{i,c} + t_{j,c} - 1 \qquad \forall c \in [1, n]$$

where n is the number of clusters. Finally, since a bit must stay in a cluster only, we also add the constraint

$$s_{i,j} \leq 1 - t_{i,c} + 1 - \sum_{kin[0,n]\setminus\{c\}} t_{j,k} \qquad \forall c \in [1, n]$$

Trying with different values of n, three clusters of bits emerged:

$$f_1 = 18, 25, 27, 30, 32$$
$$f_2 = 17, 20, 22, 24, 26, 28$$
$$f_3 = 19, 21, 23, 29, 31$$

Bits 6 to 16 have very similar likelihood and unlikelihood in every couple they appear, but the values are very low compared to those of other couples, as representing occurrences of noise. This confirmed the independence of the hash from bits lower than 17.

5.5 Finding the correct hash function

The three clusters found are in accordance with the configurations in fig. 5.1; however, we still do not know which configuration each cluster corresponds to. Indeed, any cluster may correspond to any configuration, and the two hash functions h_1 and h_2 are the XOR-combination of a common and a specific configuration. Therefore, there are 6 possibilities, corresponding to 6 different hashes h. If we denote with x_i the XOR of all bits in f_i and separate with the comma the two functions h_2, h_1 (to keep the usual positional notation), the 6 hashes are:

$$\begin{aligned}
h_a &= (x_1\ XOR\ x_2), (x_3\ XOR\ x_2) \\
h_b &= (x_1\ XOR\ x_3), (x_2\ XOR\ x_3) \\
h_c &= (x_2\ XOR\ x_1), (x_3\ XOR\ x_1) \\
\\
h_d &= (x_3\ XOR\ x_2), (x_1\ XOR\ x_2) \\
h_e &= (x_2\ XOR\ x_3), (x_1\ XOR\ x_3) \\
h_f &= (x_3\ XOR\ x_1), (x_2\ XOR\ x_1)
\end{aligned}$$

where the last three hashes are obtained by "swapping" the first three ones. If we closely look at these functions, we notice that it is possible to map each function to another one in a bijective way. This because, once we know the result of a hash, we can uniquely derive the values of x_1, x_2 and x_3, and from these values compute any other hash. This implies that a bijection between any two hashes exists, further implying that these functions have the same output distribution, thus being equal in terms of "spreading" capability. Hence, Intel could use any of these functions in its CMPs: for example, Hund, Willems, and Holz [38] showed that Intel Core i7-2600 uses h_e.

To find which hash function our CMP uses, we decided to exploit the variable latency of the ring interconnection: trying a specific hash function, we can compute in which slice an address is supposed to be mapped. Then, we verify if this hypothesis is correct: for example, if the test is done on core 0 [1], accessing slice 0 must have minimum latency with respect to the other slices. In this way, we can check whether a specific hash is correct.

Therefore, we wrote another Linux module to measure the latency when accessing a given memory address. This module has to access the LLC and measure the access latency, without accessing the L2. To do this, we have first to fill the LLC with the data we want to read and then fill the L2 with other data from the LLC, so that the measured access will hit in L3 and not in L2. The following code snippet summarizes this procedure.

```

globaldata: address buffer
      integer llc_size
      integer l_size
      integer offset

procedure MEASURE_LATENCY
inputs: none
outputs: integer
behavior:
    integer i

    for i in [0, llc_size)
      READ_ADDRESS(buffer + i)

    for i in [llc_size - l2_size,  llc_size)
      READ_ADDRESS(buffer + i)

    READ_ADDRESS(buffer + offset)
    return READ_LAST_ACCESS_LATENCY()
end procedure
```

Here, `buffer` is a generic memory area used to read data, and `offset` controls which data are read. By varying this parameter of the stride value, it is possible to read data from a desired slice.

[1]Linux assigns several identifying numbers to cores, some directly coming from the hardware and others based on SMT capabilities and boot order, in turn depending on the BIOS. Here we use the hardware ID of a core, which follows the same numbering of the slices and is called *ApicID* in Linux.

	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6
h2		⊕		⊕	⊕		⊕		⊕	⊕	⊕	⊕	⊕	⊕		⊕											
h1	⊕		⊕		⊕	⊕	⊕	⊕	⊕		⊕		⊕		⊕	⊕											

Figure 5.2: Reconstructed hash function of Xeon E5-1410

Table 5.1: Minimum LLC latencies measured in Xeon E5-1410 when accessing a slice from a core

	slice 0	slice 1	slice 2	slice 3
core 0	18	20	26	27
core 1	20	18	31	26
core 2	26	27	18	20
core 3	31	26	20	18

In our experience this measure was affected by a lot of noise: even reading the same address multiple times, the measured latency varies in a certain range that depends on the slice. For example, if accessing slice 0 from core 0, the latency typically varies from 18 to 23 cycles, while the farthest slice from core 0 (slice 3) has a latency in the interval from 27 to 31 cycles. These variations are likely due to the complexity of the architecture, in particular to the conflicts the core may experience on the ring bus and when accessing a slice, which could already be busy to service a prior request and enqueue the core's access. Therefore, we concentrate on the minimum values, which describe the access latency in ideal conditions. Looking at these values, we found that couples of cores have symmetrical latencies when accessing slices, and also have the same minimum latency, 18, that we assume to be the latency towards their local slice. However, the only hash that maps that maps each core's numbers to the same slice number is hash h_d, shown in fig. 5.2. This hash is different from that of [38], probably because of the different model.

Table 5.1 summarizes the minimum latencies found when accessing the slices from each core. We note, again, symmetry between couples of cores, guessing that the routing protocol of Sandy Bridge routes cache lines to the best direction, that is to say the one with the least hops.

5.6 An educated guess: reconstructing the topology

Analyzing the latencies, we can try to partially reconstruct the topology of the Xeon E5-1410. In particular, table 5.1 shows which are the closest slices to each core. For example, core 0 is close to slice 1, while slices 2 and 3 are more distant. Similarly, core 3 is close to slice 20 and distant from slices 1 and 0. Still looking at core 0, the difference between the latency of slice 0 and slice 1 is 2 cycles. This is likely due to the 1-hop distance the request and the response have to travel. Therefore, we can infer that there is a single hop distance between cores 0 and 1, and similarly between cores 2 and 3. Instead, the latency between, for example, core 0 and slice 2 is higher, suggesting that there are intermediate steps like memory ports, I/O interfaces, PCIExpress interfaces, QuickPath Interconnect (QPI) bridges for inter-processor coherency, etc. This also appplies to the distance between core 1 and core 2, where the number of hops is, probably, even higher. Moreover, we can note that, when two cores differ for only the least significant bit, they are close, otherwise they are more distant. This is a typical characteristic of routing protocols for high-bandwidth, hypercube interconnections, whose key features probably inspired Intel's engineers.

Looking at the maximum values in table 5.1 (31), we can note that it appears only in two cases, when the communication occurs from core 1 to core 2 and from core 3 to core 0, while in the opposite cases the latency is 27. This is likely due to an asymmetry in the routing protocol, for example because it sends the response along a path different from the path of the request. This, in turn, might serve to balance the load of each ring segment, to avoid bandwidth saturation and, consequently, long latencies to access shared resources such as memory or I/O interfaces.
Considering all these insights, we attempt to reconstruct the topology of Xeon E5-1410 in fig. 5.3, were the intermediate hops (with dashed lines) are supposed to be between cores 1 and 3 and between cores 0 and 2. The red spots depict the cache boxes and two buses with opposite directions are assumed, as in fig. 2.2(b).

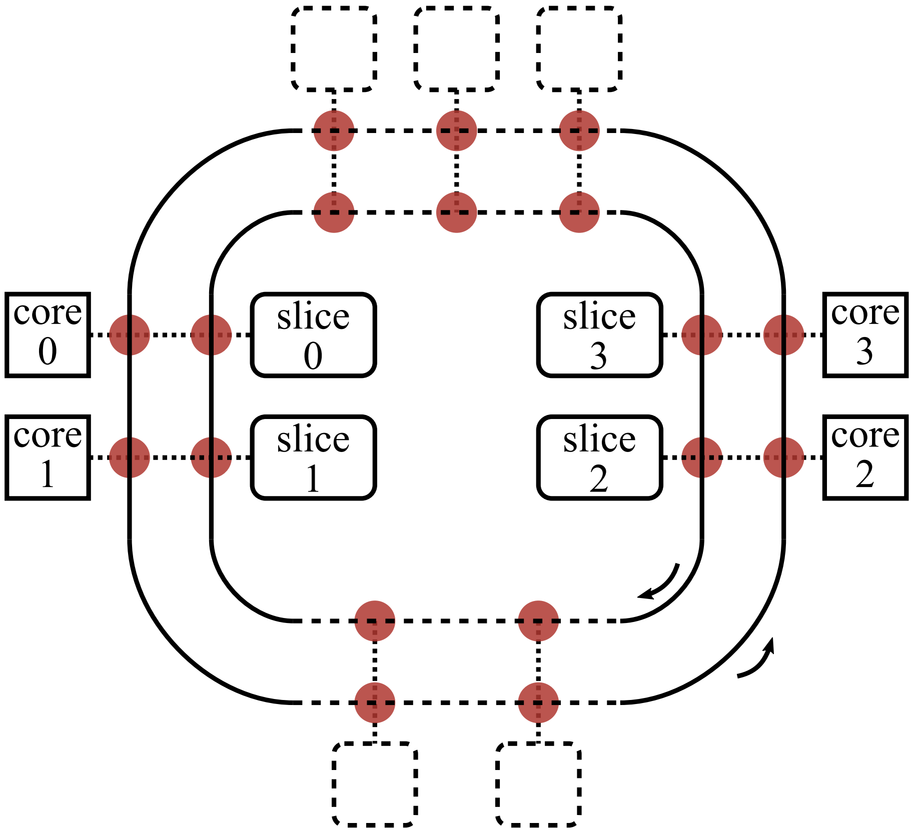


Figure 5.3: Reconstructed topology of Xeon E5-1410

The intermediate hops are dashed as their function is not known

Implementation 6

This chapter discusses the details of the implementation of *Rainbow* on the basis of the design choices of the previous chapter. section 6.1 shows the way *Rainbow* is implemented inside the Linux kernel, while section 6.2 gives an overview of the various components added with *Rainbow*.
Section 6.3 explains how *Rainbow* represents colors and manages them through mcolors; in particular, we present how the knowledge of the hash function for Sandy Bridge allows *Rainbow* to associate a color to a buddy uniquely, and how the color chasing mechanism is implemented thanks to the information in chapter 5. Section 6.5 shows how colors are associated to applications via the cgroup interface and section 6.4 discusses how *Rainbow*'s buddy allocator uses this information.

6.1 Implementation approach

The design proposed for *Rainbow* in chapter 4 is implemented on the Linux kernel version 3.17.1.

This choice is due to two main reasons. The first is the need of making modifications starting from a recent version of the memory allocator: in fact, this subsystem has undergone important changes over the years, and is now very different from earlier versions. To derive a modern implementation for *Rainbow*, a modern allocator is the only choice. The second reason lies in some of the changes the latest versions bring, related to cgroups. While older versions didn't offer a clear interface to add a cgroup, requiring developers to deal with low-level filesystem management details, from version 3.16 Linux developers introduce a unique, high-level API that prevents disruptive interventions and eases development. In version 3.17.1 this interface has definitely been stabilized and polished.

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 was to add the new source code to the kernel working tree and to switch between the default allocator and *Rainbow* at compile time.

Particular care is devoted to keeping higher levels of the Linux memory management system unchanged, in order to have the smallest possible impact on the existing codebase and functionalities. Therefore, normal applications should not be isolated and should allocate across all the available pages. Most importantly, it is fundamental to preserve the functionality and performance of the kernel without restricting the memory it can allocate from, also in presence of colored tasks.

6.2 Coloring the memory of a task

Before showing the details of the implementation, it is useful to provide an overall view of how *Rainbow* works, and of how it integrates within Linux in a non-disruptive way. Therefore, this section shows the main steps required to run isolated applications and how the novel mechanisms of *Rainbow* act within a running Linux kernel.

When a task starts first, no coloring information is attached to it; if no colored applications are running, the task can allocate everywhere in memory. Otherwise, a global data structure stores the colors not allocated to any set of tasks (recalling that, via the cacheset cgroup, the administrator can create groups of tasks sharing the same LLC partition), and the task allocates from this pool of pages.

If, at some point, the administrator colors the task by writing its PID into a cacheset, it restricts the pages available for future allocations to the cacheset pool. Therefore, the administrator can choose to isolate a task from the very beginning or only from a certain point on, for example when its LLC usage grows over a threshold or when it starts causing sensible pollution to other applications.

Once colored, any memory request from the task will be fulfilled using the cacheset pool. The only mechanism for userspace tasks to obtain physical pages is by causing a page fault. After this event, the kernel recognizes the faulting application and satisfies the memory request by calling the physical memory allocator. This component searches a suitable page via the buddy algorithm and returns it to the fault handler, which in turn updates the virtual-to-physical mapping of the task's memory. In *Rainbow*, the memory allocator checks whether the requesting task has an associated color set and eventually uses it to restrict the allocation to the allotted colors. If *Rainbow*'s allocator cannot find any coloring information, it uses the default color pool.

6.3 Color management

6.3.1 Representation of color sets

During a colored allocation, the allocator should be able to select a color from a set in an efficient way. Therefore, *Rainbow* needs a suitable representation of a set of colors. Section 4.5.1 introduce the concept of mcolors to generalize the notion of color to the whole buddy allocator, where the size of managed memory areas varies with the order. Thus, a LLC reservation enforced through page coloring could be represented at any granularity by associating mcolors to a task rather than colors. However, to allow the most possible fine-grained partitioning capability, we choose to associate base colors to a task.

High level structures exist to represent sets of data, with or without an order, like lists or trees, but they have complex management algorithms and typically require several memory accesses, resulting cache-unfriendly. To devise the right implementation for *Rainbow*, it is important to note that the information about coloring, once set, is unlikely to change: hence, the pooling facility the cacheset cgroup creates makes it easier to create different hierarchical pools rather and moving applications between pools rather than chaning a single color pool. Furthermore, the number of colors an architecture provides is typically small.

Therefore, it is natural to describe a set of colors with a bitmask, for which Linux provides pre-built creation and management facilities.

Together with this information, it is important to store also the number of colors a mask contains, to evaluate the memory it can use, and the last color the task allocated from, in order to distribute allocations on colors in a round-robin manner. *Rainbow* organizes these data in a structure, as listing 6.1 shows.

Listing 6.1: Colors set representation

```c
struct color_info {
  colormask_t cmask;
  unsigned int last_color;
  unsigned int count;
};
```

This information must be associated to the task once this is colored. The most direct and efficient way is storing this structure inside the one that contains all the information of a process in memory, that is the `task_struct` structure at the base of Linux application management. Therefore, *Rainbow* adds a field `struct color_info *cinfo` for this purpose. During the task creation, this fiels is initialized to the value of the task's father, in order to maintain isolation in particular when a task creates threads (Linux spawns threads via the `fork` system call and represents them with dedicated `task_structs`, in a similar way to independent processes). The default value for this field is `NULL`, meaning that the task is not colored.

In a similar way to a task's color information, *Rainbow* defines two global variables to store global coloring information of type `struct color_info`. The first one is `cinfo_kernel`, which stores all the colors the machine has, is initialized at runtime and is not modified; this is the pool the kernel allocates from. The second variable is `cinfo_allowed`, which stores the colors no cacheset still uses: non-colored applications allocate from this pool, which does not intersect with any cacheset pool, thus guaranteeing isolation of colored applications from non-colored ones. This variable is updated whenever a cacheset is created or destroyed.

6.3.2 Implementation of mcolors

The definition of mcolors given in section 4.5.1 simplifies the management of the buddy data structure when looking for a specific color, avoiding lookups along lists of buddies. Similarly, in section 4.5.2 we introduced the function $mcolor\_lookup_N(c, d)$, which returns the d-order mcolor from a color c. These notions are at the heart of *Rainbow*, and their implementation must be efficient.
Linux identifies a buddy through its first page, where the information about the buddy are stored. In turn, Linux identifies a physical page with a so-called Page Frame Number (PFN), which is simply the page physical address without the offset bits, after a right-shift of 12 bits on x86 architectures. This is the starting point from which mcolors are read for any buddy order.

In Nehalem, the implementation of the function $mcolor_N(b, d)$ of section 4.5.1 follows closely from the mathematical definition: since the number of bits considered for the mcolor increases with the order, a right-sift can discard lower bits based on the order, as in listing 6.2.

Listing 6.2: Macro to retrieve the buddy mcolor in Nehalem

```
#define page_mcolor(page, order)                        \
  ( (((unsigned long)page_to_pfn((page)) &
      linear_color_bitmask) \
     >> (cshifts[order]) )
```

`linear_color_bitmask` is a bitmask that keeps only the colors bits from the PFN. The variable `cshift` is an array of shift indexed per order and initialized at boot time: for orders 0 to 3, the shift is 3, for order 4 it is 4 and so on, as from the definition of $mcolor_N(b, d)$. The function `page_to_pfn`, insted, is provided by the kernel and translates a `struct page` variable (that represents a page in Linux ' pages array) to its PFN.
In a similar way, the function $mcolor\_lookup_N(c, d)$ in Nehalem is implemented in listing 6.3

Listing 6.3: Macros to compute the mcolor from a color in Nehalem

```
#define mcolor_from_color(color, order) (vcolor >>
   (cshifts[order] - cshifts[0]))
```

As anticipated in the design, the case for Sandy Bridge is, in general, more complex. Following the definition of page color for Sandy Bridge in section 4.3.3, the implementation can be derived, as visible in listing 6.4.

Listing 6.4: Function to retrieve the page color in Sandy Bridge

```
unsigned int page_color(struct page *p)
{
  unsigned long _pfn = page_to_pfn(p);
  unsigned int hash = sb_hash(_pfn);
  return ((hash << (2)) | (( _pfn &
      linear_color_bitmask) >> 3));
}
```

Here, `sb_hash` is the function that computes the hash starting from the page address, as reconstructed in section 5.5, and `linear_color_bitmask` keeps only the "linear part" of the color bits, that is to say those color bits that are not used for the hash (in Intel Core i7-2600 and Xeon E5-1410, bits 15 and 16). These bits are right-shifted of 3 positions since bits 12 to 14 are not used as color bits (we recall that `_pfn` is already right-shifted by 12, being a PFN) and finally concatenated on the left with the hash.

The hardest, hash-dependent point with Sandy Bridge is implementing the $mcolor_{SB}$ function; here, the presence of the hash creates issues: at order 6, where Nehalem uses only bit 18 to indicate the mcolor, it is not possible to choose bit h_2 (the highest bit in the hash) for the same purpose, since, even knowing h_2, it is impossible to predict the future configuration h_2h_1 on a general basis. Therefore, we have to rely on the knowledge of the hash function for our implementation. The basic insight is that bit 17 is used in both the hashes h_2 and h_1. If we consider the hash computed after zeroing bit 17, we can find that if bit 17 is 1, the final hash is the bit-flip of the previous one, otherwise is is the same; for example, if the hash with bit 17 equal to 0 is 01_2 and bit 17 is 1, then the final hash is 10_2. Then the hint consists in grouping together buddies depending on their hash having equal bits or not. If, for example, the allocator looks for a buddy with hash 10_2, hence with different bits, it should pick a buddy whose hash without bit 17 has different bits. If, still as an example, the hash without bit 17 is 10_2, the allocator should split this buddy and select the first half, having bit 17 equal to 0, which does not change the hash. Conversely, if the hash without bit 17 is 01_2, taking the second half of the buddy, with bit 17 equal to 1, “flips” the hash and gives 10_2. With this rationale, at order 6 the buddies should have mcolor 0 if they have hash, without bit 17, of equal bits (00_2 or 1_2), and mcolor 1 if the bits are different. Higher orders, instead, have all color 0, while lower orders compute the color by concatenating the hash with one or both color bits 16 and 15.

All this could be implemented in a function that, considering the order passed in input, can select the proper case. However, we believe this to be an inefficient implementation. Therefore, we decided to implement this function as a lookup table indexed by page color and order, as in the following code

Listing 6.5: Lookup table to retrieve the mcolor in Sandy Bridge

```
uint8_t color_jumps[NR_COLORS][MAX_ORDER];
```

so that the $mcolor_{SB}$ function is implemented as

Listing 6.6: Function to retrieve the mcolor in Sandy Bridge

```
unsigned int page_mcolor(struct page *p, unsigned
    int order)
{
  unsigned int color = page_color(p);
  return color_jumps[color][order];
}
```

and the $mcolor\_lookup_{SB}(c, d)$ as

Listing 6.7: Function to retrieve the mcolor in Sandy Bridge

```
#define mcolor_from_color(color, order) (
    color_jumps[color][order])
```

Either using a function or a lookup table keeps the implementation hash-dependent, since their rationale is based on the particular role of bit 17 within the hash. An idea for a general enough implementation is using a lookup table with values initialized at boot time: once the kernel recognizes the CMP it runs on, knowing its hash function, it can provide the logic to fill the lookup table properly. Again, this qould require the cooperation of CMP manufacturers to have exact information.

6.4 Rainbow Buddy

In this section we show the major modifications to the Buddy algorithm, according to the design proposed in section 4.5. Section 6.4.1 shows how the Buddy data structures have been modified, while section 6.4.2 shows the changes of the algorithm.

6.4.1 The modified Buddy structure

In the Linux kernel, the physical memory of a machine is firstly divided into nodes, to manage the memory of different NUMA nodes. Then, each node is divided into zones. 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 [1]. Applications are preferably given memory from a zone called "Normal", but the allocator falls back to the other areas if no memory is present.

Within Linux, a zone is described by many fields; among them, the field `struct free_area free_area[MAX_ORDER]` is the main data structure described in Figure 2.2, and the constant `MAX_ORDER` is the number of buddy orders allowed for the current architecture (10 by default). In particular, the `struct free_area` implements the structure of the buddy allocator for a single order, comprising the various per-migratetype lists. In fact, this data structure is natively defined as

Listing 6.8: The original buddy data structure

```
struct free_area {
  struct list_head  free_list[MIGRATE_TYPES];
  unsigned long   nr_free;
};
```

[1]For example, in a x86-64 system three zone usually exist: the *DMA* zone consists in the first 16 MB of memory and is meant for old DMA controllers, which use only 24 bits for addressing; 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.

To split each list into per-color sublists according to section 4.5.1, the definition changes into

Listing 6.9: The new buddy data structure

```
struct free_area {
  struct list_head  free_list[MIGRATE_TYPES][
      NR_COLORS];
  unsigned int last_color[MIGRATE_TYPES];
  unsigned int count_migtype[MIGRATE_TYPES];
  unsigned long   nr_free;
};
```

In the original buddy data structure of listing 6.8, the allocator can check if there are available buddies for a certain migratetype by simply checking the list head. Instead, with the modifications of listing 6.9, it would be necessary to check each list. To avoid such operation, we added a counter storing the number of buddies for each migratetype. The allocator can check this counter before going through the lists, but it has to update the count in case of buddy removal or insertion.

In general, the allocations of a physical page happens frequently in a running system, especially if with high load conditions. In Linux, the structure of the buddy allocator has a centralized management, and is protected with a spinlock. Since this mechanism does not scale with CMPs, Linux employs per-core pools to fulfill page requests, while for higher-order buddies the allocator goes through the buddy data structure. These pools are refilled at runtime in batch and are meant to fulfill most of the memory requests, as applications are given only single pages after a page fault. The per-zone data structure, called `per_cpu_pageset`, that implements this pool is originally defined as

Listing 6.10: The per-cpu-pages original structure

```
struct per_cpu_pages {
  int count;     /* number of pages in the list */
  int high;    /* high watermark, emptying needed */
  int batch;     /* chunk size for buddy add/remove
      */
  /* Lists of pages, one per migrate type stored on
       the pcp-lists */
  struct list_head lists[MIGRATE_PCPTYPES];
};
```

Like for the zone `free_area`, *Rainbow* splits the buddy list and adds counters to store the actual availability of buddies per-migratetype.

Listing 6.11: The modified per-cpu-pages structure

```
struct per_cpu_pages {
  int count;     /* number of pages in the list */
  int high;    /* high watermark, emptying needed */
  int batch;     /* chunk size for buddy add/remove
      */
  struct list_head lists[MIGRATE_PCPTYPES][
      NR_COLORS];
  unsigned int last_color[MIGRATE_PCPTYPES];
  unsigned int count_migtype[MIGRATE_PCPTYPES];
};
```

The `free_area` is initialized at boot time by forcibly freeing all the memory pages; in this way, the allocator triggers the coalescing procedure that groups buddies into buddies of order higher and higher. Finally, the allocator initializes the `per_cpu_pages` with a batch of pages, according to pre-defined quotas. Even with our modifications the initialization follows the same procedure, but for the fact that buddies are stored in the proper list according to their mcolor.

6.4.2 The modified Buddy algorithm

In Linux, the process of allocating physical memory is very complex, as it considers the multiple subdivisions of memory areas (nodes, zones, etc.) and the per-cpu and migratetype pools, with heuristics to move pages among pools and fallback lists to move to other zones or nodes in case of memory exhaustion on the local node. Therefore, an allocation can be performed in several, more and more complex, attempts. Overall, the highest entry point for the physical memory allocator if the function `__alloc_pages_nodemask`, which determines the migratetype based on the caller kernel subsystem and iterates over the allowed machines nodes. Then, `get_page_from_freelist` iterates over the zones, in turn calling `buffered_rmqueue`, which function first looks for a page in the per-cpu pagesets, and in case it cannot find one, it goes through the buddy allocator. Hence, `buffered_rmqueue` is the initial point for the necessary modifications.

The first change to this function consist in retrieving the information of the reserved colors of the process that faulted. To this aim, we added the following code snippet at the beginning

Listing 6.12: Code to choose the color pool to allocate from

```
if((gfp_flags & (GFP_DMA | GFP_DMA32 | GFP_ATOMIC))
    ||
  (order > 0) || !ccount(get_allowed_cinfo())) {
  ci = get_kernel_cinfo();
} else if(!colored_task(current)) {
  ci = get_allowed_cinfo();
} else {
  ci = get_task_cinfo(current);
}
```

The first `if` statement checks whether the target for the allocation is the kernel (for a DMA allocation, for example), and eventually selects as color set the `cinfo_kernel` pool; `ci`, in fact, is a pointer to a `struct color_info` and is passed to lower layers of the allocator to provide information about the `cinfo_allowed` is used, or the task's pool if available.

Once the color pool is chosen, if a page is requested the allocator attempts to allocate from the per-cpu pageset. To choose an allowed color, *Rainbow* adds the following code to iterate on the bitmask over the task's allowed colors.

Listing 6.13: Color-aware pageset search

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

Here, `find_color_round` iterates over the task's colors by jumping to the initial color if it reaches the last; furthermore, it returns if it has iterated over all the colors. In particular, within listing 6.13, the starting color given in input to `find_color_round` is color successor to the one used in the last allocation. This information, stored inside each `struct` color_info as a hint to the allocator, permits to iterate in a round-robin manner over colors, thus distributing allocations over all colors. In fact, starting from a fixed color would use all the pages of the first color before using others, causing a lot of task's data to be mapped to a single LLC portion and under-exploiting the cache. Moreover, distributing allocations decreases the probability of emptying one list, in turn decreasing the number of iterations required to find a color with available pages.

If instead the allocation order is greater than 0, the allocator goes through the per-order free_areas, passing the variable ci as the color pool.

Lower levels make several attempts to allocate a page from migratetype pools, in case previous attempts fail. In particular, buffered_rmqueue calls __rmqueue, which makes a first attempt to allocate a page from the requested migratetype via __rmqueue_smallest, whose failure forces __rmqueue to call __rmqueue_fallback, which iterates through all the migratetypes to find a suitable page. Both __rmqueue_smallest and __rmqueue_fallback implement the basic concepts of the buddy algorithm, looking for a buddy of the desired order and eventually splitting a larger one. These functions receive the color pool ci from buffered_rmqueue , and similarly search for a color with available buddies starting from the last used color. If no buddy is available, the search moves to a higher order, where it is fundamental to look only among buddies of the suitable mcolor, so that the split procedure is guaranteed to return the desired color. To this aim, these functions employ the macro mcolor_from_color previously defined to compute the mcolor from the desired color and the current order they are looking in. Once a buddy to split is found, both functions call expand to perform the splitting. This function splits a buddy in two buddies of lower order, stores one inside the list of free buddies and further splits the other, until it obtains a buddy of the requested order. To finally obtain the desired color, *Rainbow* changes the splitting procedure according to section 4.5.2 in order to implement color chasing. While in the standard implementation the half to split is determined a priori (the first one), *Rainbow* determines it at runtime, with the following check:

Listing 6.14: Color chasing implementation

```
if(mcolor_from_color(color,high) !=
    mcolor_from_color(__page_color,high) ) {
  struct page *tmp = twin;
  twin = page;
  page = tmp;
  __page_color = page_color(page);
}
list_add(&twin->lru, &area->free_list[migratetype][
    page_mcolor(twin,high)]);
```

where `page` is the first half of the split buddy, `twin` the second half, `color` the desired color, `__page_color` the color of the first half and `high` the current order. As visible in the `if` statement, if the mcolor of the first half is different from the desired mcolor, the variables holding the two halves are swapped, and the undesired half stored in the list of free buddies according to its color.

6.5 Implementation of the cacheset cgroup

Using the new facilities of Linux version 3.17.1, the cacheset cgroup integrates into the kernel to provide userspace with a file-like interface. On the kernel side, cacheset hooks into the Linux cgroup subsystem by defining the variable

Listing 6.15: Main data structure for cacheset cgroup

```
struct cgroup_subsys cacheset_cgrp_subsys = {
  .css_alloc  = cacheset_css_alloc,
  .css_free = cacheset_css_free,
  .attach   = cacheset_attach,
  .can_attach = cacheset_can_attach,
  .exit = cacheset_exit,
  .bind   = cacheset_bind,
  .legacy_cftypes = files,
  .early_init = 1,
};
```

which follows a strict name convention to be automatically recognized as a cgroup at compilation time. This structure exports the kernel and userspace interfaces to cacheset, in particular the functions that are called during the lifetime of cacheset. For the userspace interface, the field `legacy_cftypes` contains a list of file names and functions: when an

application writes a file, the related functions are called for input parsing, for providing information, etc.; in this way, a user can, for example, set the number of desired colors by writing a color sequence to a file, and *Rainbow* applies this setting to the cgroup.
The other fields inside `cacheset_cgrp_subsys` constitute the kernel interface. When, for example, a process tries to hook into a cacheset my writing its PID into a file, the kernel calls the function `can_attach` to perform initial checks, and the function `attach` to finally perform the attachment.

Internally, all these functions represent a cacheset with the data structure in listing 6.16. For example, `attach` links the color set information to the `task_struct` of the inserted process and perform bookkeeping, like incrementing `task_count` and `thread_count`.

Listing 6.16: The cacheset data structure, to store information of a single cacheset

```
struct cacheset {
  struct cgroup_subsys_state css;
  unsigned int nesting_level;
  atomic_t task_count;
  atomic_t thread_count;
  struct color_info cinfo;
  spinlock_t cinfo_lock;
};
```

Among other fields in listing 6.16, `css` connects other cacheset in a tree-like manner, and `cinfo` the set of colors.

Experimental Results 7

In this chapter we present the results obtained through *Rainbow*. The final goal is to achieve isolation of co-running applications inside the LLC using the capabilities *Rainbow* provides. This isolation must be guaranteed in both the Nehalem and the Sandy Bridge platforms. Section 7.1 explains the experimental environment setup by describing the two platforms used for testing *Rainbow*, the tests adopted to show *Rainbow*'s effectiveness and the measurement methodology. Following these guidelines, section 7.2 shows how the test applications behave with different LLC reservations, in order to devise the tests' sensitiveness to this resource and provide a reference behavior for co-location. Section 7.3 analyzes how *Rainbow* is able to isolate diverse co-running applications on an on-demand basis, hence with an LLC partition of varying size. Similarly, section 7.3.1 measures the effectiveness of *Rainbow* with a more regular workload, where polluting I/O patterns are more limited. Finally, section 7.5 summarizes the results of this chapter.

7.1 Experimental environment

Our evaluation comprises several aspects. The first aspect to be evaluated is *Rainbow*'s effectiveness to partition the LLC, guaranteeing isolation to requesting applications. The second aspect to be studied is how LLC partitioning impacts on the target architectures for which *Rainbow* is designed. To this aim, it is important to choose the experimental environment and the test cases properly. Therefore, section 7.1.1 provides the details about the environment setup, in which *Rainbow* will run the applications chosen in section 7.1.2. Finally, section 7.1.3 explains the measurement details and tools.

7.1.1 Testbed and coloring parameters

For our test, we employed two different machines, one equipped with a Nehalem CMP and the other with a Sandy Bridge CMP. These machines are representative of low- and middle- end servers typical of computing environments. The Nehalem machine has a 64 bit quad-core Intel Xeon W3540 CMP, with a clock frequency of 2.93 GHz and 12 GB of RAM. In particular, Xeon W3540 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

The parameters of the caches 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)

Since the physical page size in x86 architectures is 4 KB, he fundamental parameters for page coloring are:

- $13 + 6 - 12 = 7$ bits in common between LLC set index and page address
- $9 + 6 - 12 = 3$ bits of L2 set index that overlap with the previous bits
- $7 - 3 = 4$ bits of color, thus with 16 possible partitions of 512 KB each; these color bits are bits 15 to 18

Similarly, the other platform is a Sandy Bridge CMP Intel Xeon E5-1410 running at 2.8 GHz frequency with 6 GB of RAM. In this architecture, only the upper LLC is different from Nehalem's, as it is 20-way associative, still with 8192 sets, and is split in four slices of 2048 sets each, connected through the familiar ring-interconnection. Overall, its capacity is 10 MB. Therefore, the parameters *Rainbow* leverages for this CMP are the following:

1. $11 + 6 - 12 = 5$ bits in common between per-slice LLC set index and page address
2. still 3 bits of L2 set index overlapping with LLC set index,
3. $5 - 3 = 2$ bits of per-slice color, thus with 16 possible partitions of 512 KB each; these color bits are bits 15 and 16
4. 2 bits of hash, for a total of 4 bits for the color

Because of the complexity of modern architectures, several capabilities can affect the measurements in an unpredictable way. Cache prefetchers aggressively load data according to speculative decisions based on the access pattern, and can create pollution or become a lieu of contention [28]. Hyper-Threading support [40] causes two threads to share many resources inside a single core (lower caches, physical registers, execution units, ...), unpredictably disturbing each other. Finally, the Turbo Boost capability [41] aggressively boosts the core frequency for limited time periods, but is entirely under the control of the hardware according to internal, undisclosed policies. Therefore, we disabled all these sources of noise.

The software environment comprises the Ubuntu Linux distribution, version 12.04 Long Term Support (LTS), running on a *Rainbow*-modified

Table 7.1: Selected SPEC CPU2006 tests

Test	**Input**
libquantum	control
gcc	g23
omnetpp	omnetpp
leslie3d	leslie3d
xalancbmk	t5
sphinx	ctlfile
astar	rivers
bzip2	text

kernel. As the default setting, the maximum order for buddies is 10, so that the largest allocatable area spans 4 MB.

7.1.2 Test applications

To stress *Rainbow*'s capabilities, CPU-intensive applications are needed. A common reference in the literature, the SPEC CPU2006 benchmark suite [78], offers a wide variety of CPU-intensive applications with diverse characteristics and cache access patterns [16]. SPEC tests are single-threaded applications, a characteristic that allows us to evaluate the features of *Rainbow* while co-locating different applications without any disturbance from the kernel scheduler or from external, thread-management libraries. In particular, multi-threaded applications have more complex access patterns towards the shared LLC, which could hinder the evaluation of *Rainbow*.

Since many SPEC applications run with more input sets sequentially, they can have different phases that can present different access patterns. Therefore, we chose to separate all the possible inputs of each application and to evaluate each one separately, collecting only the execution time. At the end of the evaluation, we selected 8 benchmarks with different access patterns in order to have a representative set of patterns. For each application we chose the input set causing the longest run. The selected applications and inputs are reported in table 7.1.

7.1.3 Experimental methodology and tools

To characterize the LLC pattern of an application and its implications on the overall behavior. Two different metrics are needed. For the pattern, the most suitable metric is the LLC miss rate, which shows the ability of an application to effectively exploit the LLC. To characterize the overall behavior of an application, also the runtime is a suitable metric that suggests how the performance change.
To collect these metrics, several tools are available; for these work, we chose `perf`, [67] which is integrated in Linux and exploits the hardware performance counters available in modern x86 architectures. In particular, we used the *perf stat* command to choose the events to measure and run the test.

Throughout our experiments, all the applications are forced to run on a fixed core (core 0, unless specified otherwise) through the `taskset` command. This avoids overheads due to the kernel moving the application, which causes the application to re-load data into the L2 and L1 caches. Moreover, power-controlling mechanisms have been disabled and the fixed maximum frequency is set pn all the cores.

To launch an application and give it a LLC partition, a small launcher application has been developed, which takes in input the cacheset and the target application to run. This launcher retrieves its own PID, writes it into the cacheset proper file and launches the target application with an *exec* call, so that the target application "inherits" the LLC partition from the launcher.

7.2 Application characterization

To characterize the access pattern, we build an accurate profile of each application's behavior with respect to varying size of caches. The obtained profiles will indicate the sensitivity of each application to the cache space, guiding the choice of suitable workloads to show the effectiveness of *Rainbow* in a co-location scenario. Each application is run 10 times with a different LLC partition, on both the Nehalem and the Sandy Bridge machines. Since LLC-sensitive applications typically reach an execution plateau when the cache space increases, it is interesting to profile them in particular when the LLC reservation is small, where the behavior is typically more diverse. In the following measurements, the test application is the only one running inside the machine, and thus its behavior is optimal with respect to a co-location scenario.

7.2.1 Profiles in Nehalem

Figure 7.1 shows the profiles of the test applications running in our Nehalem machine: the red curve plots the execution slowdown with respect to the case with full cache (8 MB) and its scale is on the left side, while the blue curve plots the miss rate and its reference axis is on the right. Each point in the plots is the average of all the 10 measures. Moreover, for each measure the plots also show the Standard Error of the Mean (SEM) with bars, which are hardly visible in most cases.
As we expected, the LLC profiles are quite diverse: for example, libquantum is almost insensitive to the LLC size, while bzip2 and omnetpp benefit from more space. However, the miss rate of two cache-friendly applications as bzip2 and omnetpp can still differ slightly, as the miss rate curves of these two applications show. In general, this is due both to the access pattern and the size of the working set. In the specific case of bzip2 and omnetpp, bzip2 has a larger working set than omnetpp: hence, we can infer that the difference is due only to the access pattern. In fact, while bzip2, a compression application, has very strong locality, omnetpp scans a large part of the dataset to perform an event-drive simulation of a complex ethernet network.

In general, looking at fig. 7.1 we can note that the SEM is small across the measures. This suggests that the selected applications have regular access patterns, hence with a predictable behavior. Secondly, the predictability of the runs suggest that the design and the implementation allow *Rainbow* to effectively control the LLC.
An exception is sphinx, whose SEM is clearly visible. Sphinx is a speech recognition system based on Viterbi search [90] and on beam search heuristic [75], which is a graph-search algorithm. Furthermore, sphinx loads multiple files from disk to recognize multiple words. Therefore, sphinx has irregular access patterns that depend on input data and on the position of memory areas in each single run. In fact, with more than

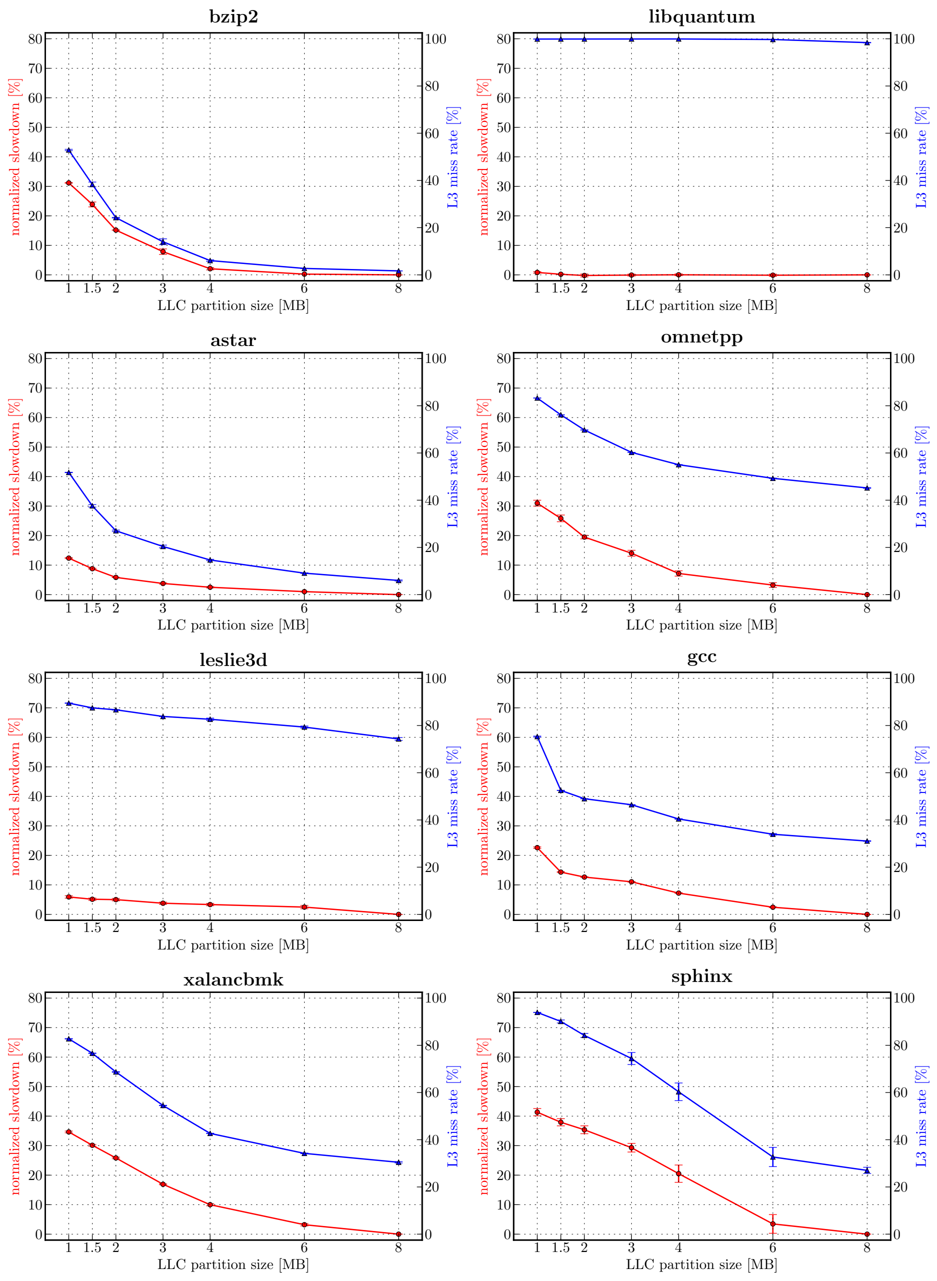


Figure 7.1: Applications profiles in Nehalem with different cache partitions

6 MB of LLC its performance and miss rate is more moderate, as most of its working set fits in the LLC.

7.2.2 Profiles in Sandy Bridge

Performing the same measurements in Sandy Bridge, we obtained the plots in fig. 7.2. Here, applications run on core 0 and are given colors starting from slice 0 and then reaching the other slices in order. Each slice has a size of 2.5 MB and has increasing latency when accessed from core 0 (as from table 5.1). Here, the limited RAM memory (6 GB) poses some constraints on the tests: bzip2 and gcc, in fact, are not run with only 1.25 MB of LLC because of the cache-memory constraint identified in section 4.4.1. Thus, 1.25 MB over 10 MB corresponds to 12.5% of the LLC size, constraining the memory to only 750 MB, while these two applications have a barely superior memory footprint (around 850 MB). Therefore, the profile of bzip2 and gcc start from 1.88 MB of LLC.

With respect to fig. 7.1, we can see, overall, that the runs are steadily more sensitive to the increasing LLC partition size. Indeed, if we consider the slowdown with 1.25 MB of LLC, it is in every profile higher than that in 1 MB of section 4.4.1. This is even more interesting if we consider the higher associativity of Sandy Bridge (20) with respect to Nehalem's (16). A higher associativity should decrease the miss rate with the same amount of LLC, as it clearly happens with regular applications, whose miss rate curves values in Sandy Bridge are slightly lower than the corresponding values in Nehalem. Thanks to the higher associativity, the difference between the worst and best miss rate should be smaller. Instead, this difference is comparable, and it is even higher in the case of libquantum. Unlike with Nehalem, in Sandy Bridge libquantum shows some degree of LLC-sensitiveness. This sensitiveness can be explained with the hash-based addressing scheme. If a cache line can be mapped to a slice only, it can be mapped to a single set, whose associativity is 20. If, instead, it can be mapped to two slices, it can end up in two different sets, perceiving an associativity that is, roughly, the double. Therefore, as the LLC an application can use increases, also the associativity it perceives increases, with a consequent gain in terms of reduced miss rate. Previous work on Intel's caches [94] also found similar phenomena, which are, overall, new and not systematically studied.

Finally, we have to note the strange behavior of leslie3d and xalancbmk with 5 MB of LLC, in which the miss rate increases with respect to the 3.75 MB point. Further investigation found that it is due to their working set having internal memory areas that have low access rate: if these areas map to a considerable extent with areas of higher access rate, they cause pollution. Moreover, the round-robin policy for color allocation within *Rainbow* could, for certain configurations, cause polluting overlaps between memory areas having different locality. In particular, xalancbmk outputs a very large amount of data (around 100 MB), with a cache-unfriendly streaming pattern.

7.3 Isolation with mixed workloads

With the cache profiles collected previously, we can classify applications based on their LLC "sensitivity", that is to say on how much applications benefit from receiving more cache space. Therefore, we focus on the slowdown for this classification.

We classify applications as *sensitive* when their slowdown with the least amount of cache is equal or greater than 30%, while the others are *insensitive*. The value of 30% derives from the direct observation of the plots in figs. 7.1 and 7.2, and allows us to well separate applications in the two distinct groups. This holds for both the architectures, where the same value gives the same classification results. In Sandy Bridge, bzip2 and gcc have not been profiled with 1.25 MB of LLC for the reasons explained above, and thus their classification is not directly available. To estimate their slowdown, we computed the slowdown slope in Nehalem in the range 1.25 - 1.88 MB and used this value to compute the slowdown with Sandy Bridge and 1.25 MB of LLC, resulting with the same classification of Nehalem. Table 7.2 shows the classification of the SPEC benchmarks used for the evaluation.

Starting from the classification, we randomly chose four workloads with a sensitive application, called *target*, and three other applications, called *polluters*, that run simultaneously. To have a diverse mix, the first polluter is chosen from the sensitive applications while the other two pol-

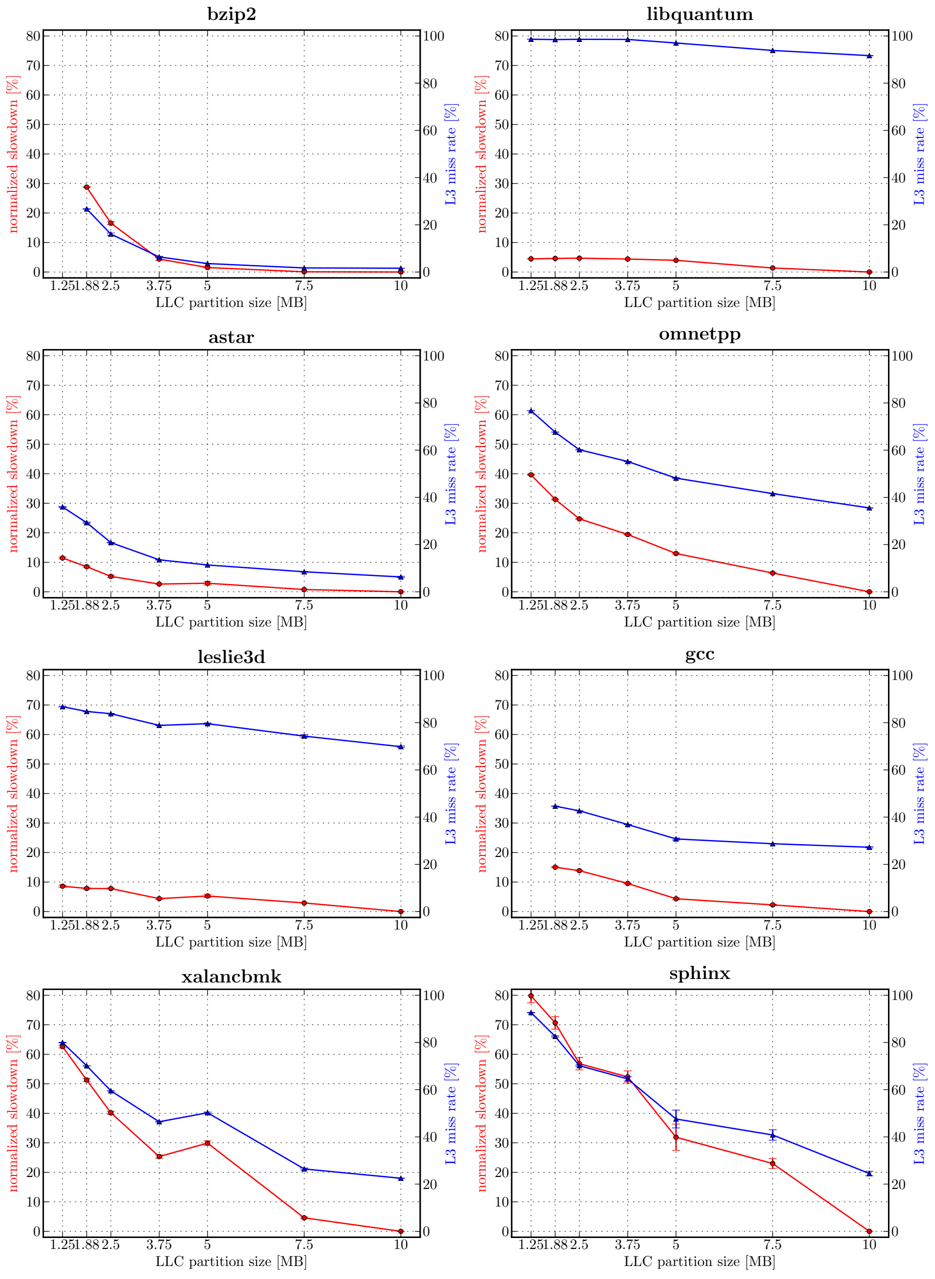


Figure 7.2: Applications profiles in Sandy Bridge with different cache partitions

Table 7.2: Classification of applications

Classification	**Applications**
sensitive	bzip2, omnetpp, xalancbmk, sphinx
insensitive	libquantum, leslie, astar, gcc

Table 7.3: Test workloads

Workload	**Target**	**Polluters**
W1	bzip2	xalancbmk, leslie3d, gcc
W2	omnetpp	sphinx, libquantum, astar
W3	xalancbmk	omnetpp, libquantum, gcc
W4	sphinx	bzip2, leslie3d, astar

luters are insensitive applications. Table 7.3 shows the four workloads, with the target and the polluters.

The experiment consists in isolating the target, running on core 0, through *Rainbow*'s capabilities, while the polluters run on the other three cores. In particular, the target is assigned an LLC partition, while the polluters contend for the rest of the cache. Like previous measurements, the size of the target's LLC partition is varied throughout the tests, in order to reconstruct a profile of how each target behaves in co-location with the polluters. Furthermore, if a polluter terminates before the target, it is immediately restarted.
It is fundamental to note that, in such experiments, the target can receive only a part of the LLC, while a considerable part is left to the polluters. This is particularly important because of the cache-memory constraint of section 4.4.1: if the target receives, for example, 80% of the LLC, then it also receives 80% of the system memory, and only 20% of RAM is left to the three polluters, which could severely conflict for memory pages and cause swapping.

7.3.1 Co-location in Nehalem

Figure 7.3 shows the targets' profiles obtained in co-location on the Xeon W3540. Each plot is named after the workload name and the target application. The red continue line shows the slowdown, while the blue continue line shows the miss rate. Instead, the dash-dotted lines show

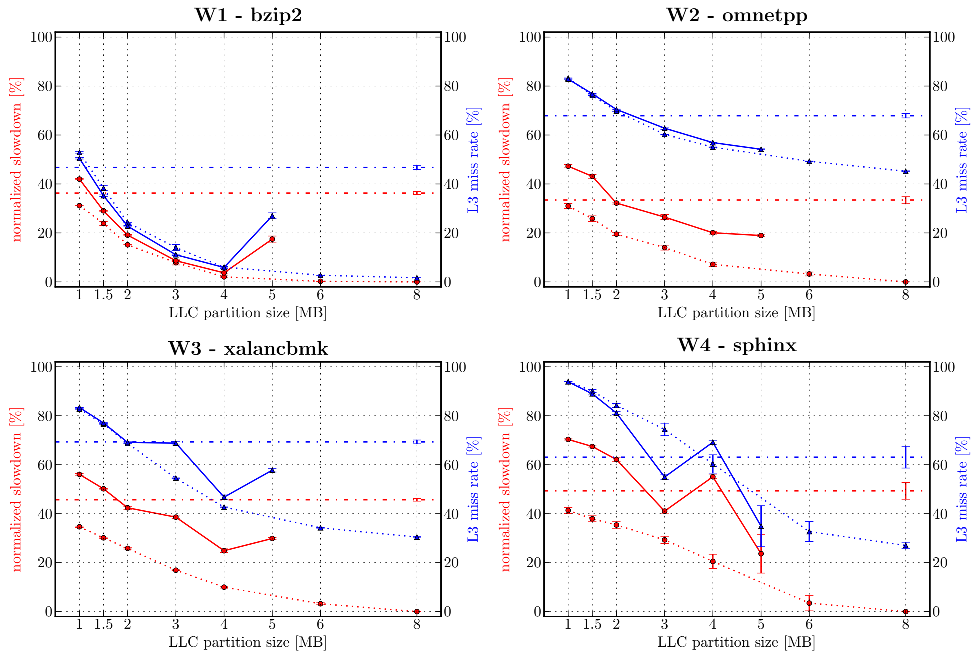


Figure 7.3: Profiles of the workloads in table 7.3 on Nehalem, with different cache partitions

how the target behaves when not partitioned, being free to contend the LLC with the polluters. For the sake of comparison, the plots also show the stand-alone target profiles of fig. 7.1 with dashed lines and the same colors. For the unpartitioned execution, the SEMs for the slowdown and for the miss rate are reported on the right side.

In co-location, it is evident that regular applications benefit more from LLC partitioning: in fact, in W1 and W2 the miss rate soon drops below the dash-dotted line, in a controllable way; the same happens for the slowdown, which is tightly related to the miss rate. In particular, bzip2 experiences strong contention when running un-partitioned, and immediately benefits from *Rainbow*. In fact, despite it has a large memory footprint, it has a small cache working set thanks to the zip compression algorithm, which seeks symbols subsequences inside small chunks of the entire dataset. Omnetpp has, instead, a greater cache working set, thus needing a larger partition. Unlikely, less regular patterns like W3 and W4 benefit less from isolation, since their miss rate and slowdown do not have a monotonically descending behavior. We will explain these behaviors, in a more general view, in the following paragraphs.

Focusing on the single profiles, we can see very different behaviors with respect to the stand-alone execution. This is visible with W1, W3 and W4. In particular, we can notice that with 5 MB of LLC the miss rate increases with respect to 4 MB. Furthermore, W3 and W4 have miss rate curves that do not follow the stand-alone miss rate curve closely. The only very regular pattern is that of W2, where the miss rate in co-location is very close to the stand-alone profile.

Investigating these irregularities, we can anticipate the results of the following sections to explain the divergent patterns. Through further measurements, we found that the deviations from the stand-alone profiles in fig. 7.1 are due to I/O activity. In fact, the workloads in table 7.3 contain applications with very diverse execution times and I/O phases overlapping with the target's execution. For example, in W1, xalancbmk and leslie3d have execution time longer than the target bzip2, while gcc has shorter execution time and is loaded twice during the execution of bzip2. Furthermore, gcc has a considerable working set (around 350 MB) and sudden spikes at the beginning, to load inputs from disk. The same holds for W3, with xalancbmk lasting much longer than gcc. Similarly, in W4 all the applications last much less than sphinx, and are reloaded multiple times, causing numerous I/O bursts; this holds in particular for bzip2, which loads its large working set multiple times. Furthermore, since sphinx has non-optimal access patterns, it receives limited benefits from partitioning if, at the same time, pollution happens; indeed, its performance is better than the unpartitioned case only when a large portion of LLC is reserved, because most of its large working set fits inside the partition, contrasting pollution.

In mode depth, I/O activity is detrimental to LLC isolation because of the allocation patterns of DMA drivers. These drivers, indeed, reserve large amounts of contiguous memory during the system boot. Since it is impossible to predict at boot time which application (and which LLC partition) a driver will serve, DMA memory pools are not restricted to specific colors, spanning potentially all the colors. Moreover, the I/O subsystem leverages specific interfaces for allocation, which are outside the buddy subsystem. Therefore, if repeated I/O activities happen when the target is running, it experiences pollution because of accesses to buffers. Furthermore, when the memory reserved to the target increases due to a larger LLC partition, the probability of overlapping with buffers data in LLC and of incurring in pollution increases. This explains the increase in the miss rate with more than 4 MB of LLC. Pollution due to buffers is visible by comparing the plots in fig. 7.1 with those of the following sections, whose workloads have less intensive I/O activity.

Concerning the slowdown curve, it seldom follows the stand-alone behavior closely: even in the case of a regular pattern like W2, the red lines are well distinct. This is due to non-partitioned resources, like the on-chip interconnection and the memory bandwidth, which are still shared among co-running applications and cause uncontrollable contention. This is especially evident for W2, where the target application, omnetpp, maintains a high miss rate (above 50%) that causes frequent accesses to main memory and to the on-chip interconnection, experiencing contention. Instead, bzip2 has a much lower miss rate and a slowdown that is close to the optimal profile, since most of its data remain inside the LLC and the contention on the memory bandwidth is limited.

7.3.2 Co-location in Sandy Bridge

Similarly to the previous section, fig. 7.4 shows the profiles when running on Xeon E5-1410. Here we notice slightly less improvements due to *Rainbow*, since the LLC has greater performance than Nehalem's, in particular higher capacity and associativity. However, isolation is still fundamental to meet a strict performance requirement.

Comparing the co-location profiles with the stand-alone references, we note similar behavior to fig. 7.3. Very regular applications like bzip2 and omnetpp still have comparable curves with respect to the stand-alone execution, while other applications suffer from pollution in an unpredictable way. Overall, these results on Sandy Bridge confirm our findings with Nehalem, still highlighting the role of I/O with respect to strict LLC isolation.

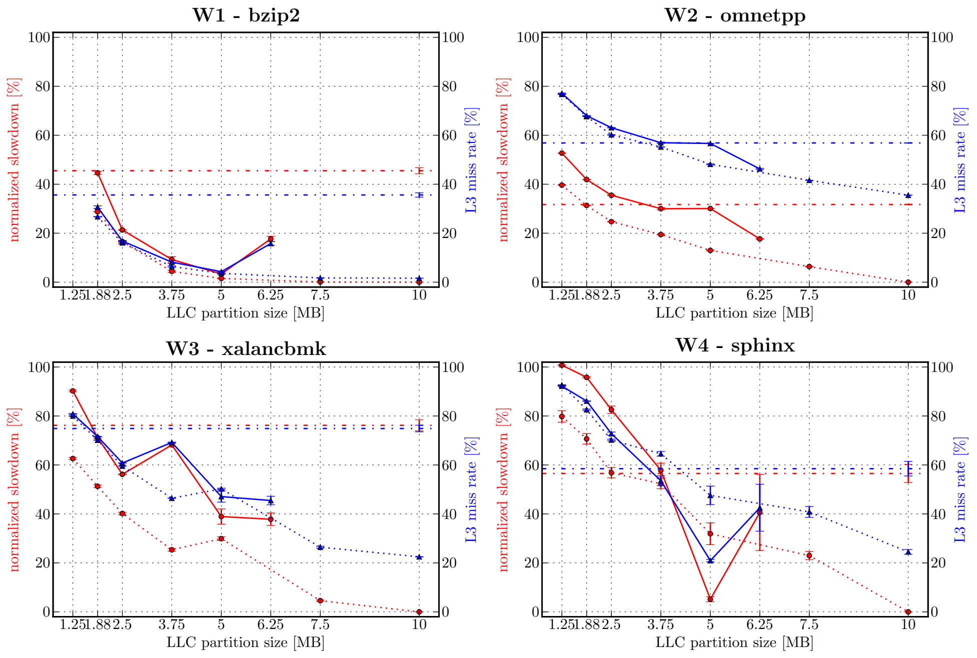


Figure 7.4: Profiles of the workloads in table 7.3 on Sandy Bridge, with different cache partitions

Table 7.4: More regular test workloads

Workload	**Target**	**Polluters**
X1	bzip2	xalancbmk, leslie3d, *astar*
X2	omnetpp	sphinx, libquantum, *xalancbmk*
X3	xalancbmk	omnetpp, libquantum, *leslie3d*
X4	sphinx	bzip2, leslie3d, *sphinx*

7.4 Isolation with more regular workloads

After the findings of previous sections, in order to test *Rainbow*'s effectiveness we choose four new workloads with lower I/O activity. These workloads derive from the those in table 7.3 by replacing the applications identified as more polluting with others, performing less I/O. Table 7.4 shows the four new workloads, emphasizing the new polluters. The choice of the applications is determined by their execution time: the

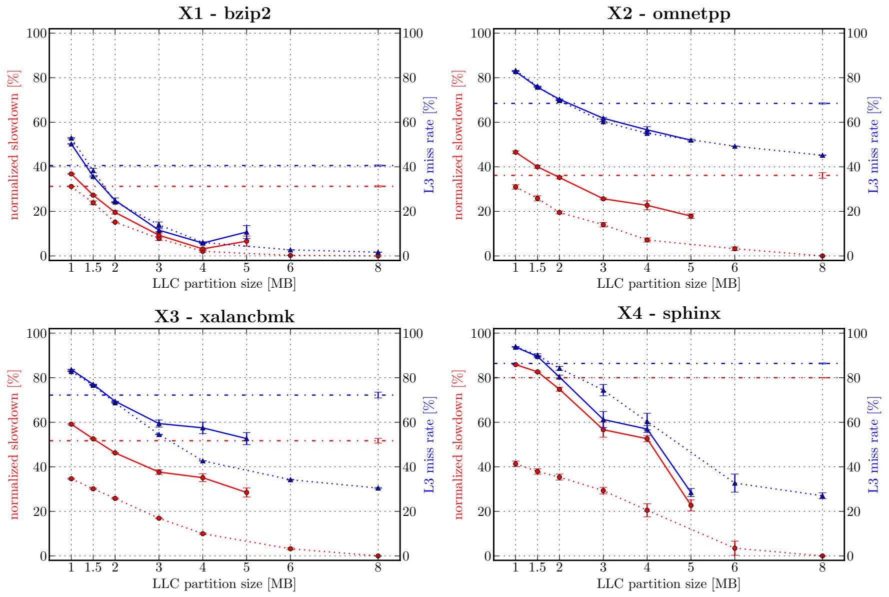


Figure 7.5: Profiles of the workloads in table 7.4 on Nehalem

new polluters have an execution time that is roughly equal to or longer than the target's. In particular, the target of W8, sphinx, has maximal duration with respect to any other application: hence, we chose another instance of sphinx as polluter as the only way to limit I/O activity during the target's execution.

Repeating the measurements with the same methodology of the previous section, we also profiled the workloads in table 7.4. Figure 7.5 plots the workloads' profiles on Nehalem, while fig. 7.6 plots the profiles for Sandy Bridge.

In general, the plots show a more regular behavior of the targets, whose curves are now monotonically decreasing and closer to the standalone execution, with the only exception of X3 in Sandy Bridge. A posteriori, this justifies the workload choice after the findings of section 7.3.1, and highlights the effect of I/O on running applications, even if isolated within an LLC partition. In particular, X4 has steeply descending curves, which clearly indicate the presence of contention with the polluters; this contention is due to the different polluter, which is

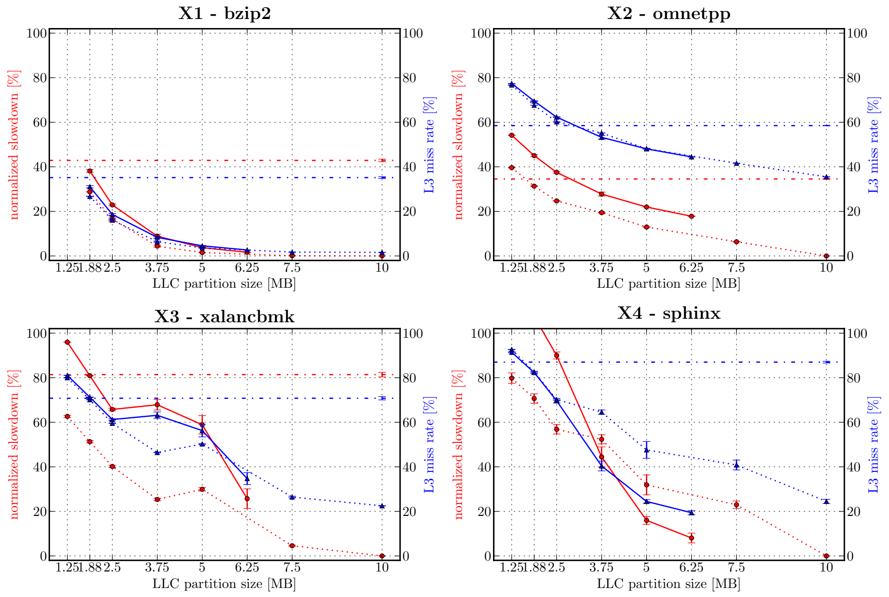


Figure 7.6: Profiles of the workloads in table 7.4 on Sandy Bridge

another instance of sphinx. On Sandy Bridge, such effects are even more visible, with the slowdown falling out of the plot. Looking at X3 and X4 in both fig. 7.5 and fig. 7.6, is evident how contention on the LLC hinders the targets' execution and, consequently, how *Rainbow* can provide guarantees on the execution on a per-demand basis that are unachievable without LLC partitioning. Similar considerations apply to X1 and X2, even if the benefits are smaller, due to the better locality of their targets that the cache can leverage. Focusing on Sandy Bridge, the curves of X3 and X4 are steeper with respect to Nehalem's for the reasons explained in section 7.2, indicating that *Rainbow* can provide even higher benefits.

7.5 Overall results

Throughout the previous sections, we measured how *Rainbow* is effective in isolating applications in the LLC, allowing to control their performance on an on-demand basis. Despite the effects of I/O, *Rainbow* can positively affect running applications, in particular in the cases where the isolated application has an access pattern that cache-unfriendly. In this case, indeed, the LLC is not able, per se, to optimally manage the application's data, and contention can severely worsen the final performance. This is particularly evident with the Sandy Bridge architecture, where a controlled LLC partition brings benefits in terms of cache space and of increasing associativity (as from section 7.2.2).

Conclusions 8

As you set out for Ithaka
hope the voyage is a long one,
full of adventure, full of discovery.
Laistrygonians and Cyclops,
angry Poseidon - don't be afraid of them:
you'll never find things like that on your way
as long as you keep your thoughts raised high,
as long as a rare excitement
stirs your spirit and your body.

Konstantin Kavafis, Ithaka
translated by Edmund Keeley

Reviewing the work done within this thesis, this chapter derives the conclusions from what has been presented so far. Considering the goals behind *Rainbow*, section 8.1, discusses the contributions of this work and with its limits with respect to current state of the art approaches in the field. Instead, section 8.3 shows possible work and research directions starting from *Rainbow*.

8.1 Contributions

Considering the state of the art in chapter 3, this work provides several contributions. A first contribution of this work is the systematic identification of the constraints posed by page coloring when used with modern commodity CMPs, as discussed in particular in section 4.4. A second contribution is the implementation of page coloring with recent CMP architectures like Sandy Bridge, with a thorough evaluation of all the related aspects. To the best of our knowledge, no prior work addressed the novel challenges of the hash-based addressing scheme these

architectures adopt[1]. Instead, this work shows how page coloring can be implemented even on these architectures, overcoming the difficulties due to the unpredictable mapping.
This contribution is particularly important in the context of cloud platforms, a spreading phenomenon nowadays. In fact, also these infrastructure can benefit from page coloring, despite having modern hash-addressed caches.

The third contribution is the adoption of the cgroup interface, which allows a centralized control of the LLC partitions on behalf of a workload manager, a component typical of cloud-like environments. Moreover, this interface allows ease of use and deep control at the same time, and is becoming a widespread interface paradigm in today's Linux installations. From this point of view, *Rainbow* is “on track” with recent facilities for applications control, and provides a suitable interface to experiment policies and allow application characterization.

8.2 Limits of the present work

Despite the novel contributions this work brings, several aspects still deserve more investigation and effort.

The first issue to tackle regards I/O activity, in particular the pollution caused by DMA drivers. After the results in chapter 7, the need to coordinate I/O buffers activity with *Rainbow*'s mechanisms is clear in order to achieve the strictest isolation. Looking at the state-of-the-art in chapter 3, research work already exists that investigates these aspects [26], but it is to be integrated into *Rainbow*'s design, so that the allocation of both the applications' memory and the buffers are automatically managed by the kernel.

To test *Rainbow*'s capabilities in production scenarios, a deeper evaluation is needed. The choice of the SPEC suite, in fact, imposes that all the test applications are single-threaded and CPU-intensive, representing only a part of today's workloads running within distributed environments. Therefore, it is important to validate *Rainbow* with a more

[1]The only prior work that, at the date of puslishing, implements page coloring on a Sandy Bridge CMP is [98], which, yet, does not claim to control the LLC slice data are mapped to

diverse set of tests. In particular, cloud-like applications with network I/O patterns and multi-threaded implementation are a key scenario. Related to this aspect, it is also important to have a better understanding of how the cache-memory constraint discussed in section 4.4.1 plays with real workloads, whose memory requirement can be unknown or unpredictable when the decision about LLC partitioning is made. In fact, a bad choice can later harm isolation, especially if an application with growing memory requirement is assigned colors that are already in use for other tasks.

More investigation is also required to study the impact of renouncing to hugepages. Even if this feature is rarely used in today's applications (and no SPEC test uses it), the ever increasing amount of data to elaborate will potentially push the usage of hugepages. With this vision, the incompatibility of hugepages and page coloring requires high-level decisions to choose whether to exploit the former or the latter feature, and a deeper study on the pros and cons of them.

8.3 Future work

Following the above-mentioned limits of the current work, we stress the importance of testing *Rainbow* in real-world, cloud-like scenarios, that should comprise also batch workloads (e.g., Hadoop) and latency-sensitive applications (e.g., web search).

To envision the employment of page coloring in real-world environments, the main lack is a dynamic mechanism. This mechanism should monitor at runtime the status of each application, devise a proper LLC partitioning scheme and apply it, possibly resizing partitions and removing colors to an application. To implement this feature while guaranteeing strict isolation, re-coloring is needed. Yet, it has a high cost that is still to be evaluated on latest hardware and is potentially open to new solutions. Furthermore, a dynamic mechanism opens many research possibilities about the policies that should drive re-coloring, and which metrics are to be considered.

A promising scenario for *Rainbow*, VMs are widespread in cloud environments, as they enforce isolation among co-running workloads of different users. To further enhance this isolation also at the level of the

LLC, *Rainbow* would be particularly interesting to test, in particular with latest Sandy Bridge CMPs. Moreover, the ever-increasing availability of LLC space inspires complex scenarios in which, for example, a 2-levels LLC partitioning mechanism is in place: the first level runs in the hypervisor and partitions the LLC among the VMs, while the second level runs inside each VM and sub-divides the LLC among applications. However, such an organization requires a complex orchestration between the hypervisor and guest OSs, which cannot control the physical mapping and perform page coloring directly.

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, as more CPU manufacturers are showing interest in the market of distributed platforms and servers, and some already have market niches, *Rainbow* could be ported and tested on other architectures like ARM, Sparc and IBM Power.

8.4 Final considerations

Through page coloring, this work achieved control over the LLC, a fundamental component in today's computing platforms. Through *Rainbow*, the control over this resource is given to the software, which can manage the LLC according to high-level constraints and objectives.
Yet, other resources are still hardly partitionable, or not partitionable at all: CPU-to-memory bandwidth, I/O bandwidth, etc. Anyway, a centralized coordination of partitioned resources is missing, so that unbalanced situations can happen without the software to be aware of them. Moreover, the ever-increasing demand of computational power and the spreading of heterogeneous computational resources with higher energy efficiency open new dimensions in the spaces of monitoring and of possible control actions. Because of these challenges, the research community advocates both software and hardware changes. One the software side, the research is still looking for general enough solution to schedule a

wide set of resources for computation and communication. On the hardware side, manufacturers are unwilling to provide control interfaces to the software layer, mainly because this increases the complexity of the hardware and implies deep design shifts.

Facing this lack of control interfaces, this work has attempted to give an enabling technology to achieve this control over today's LLCs, fulfilling a request that, in the broad view of this section, opens more and more research opportunities.

List of abbreviations

API Application Programming Interface. 2, 82
ARM Advanced RISC Machines. 33
ASIC Application Specific Integrated Circuit. 68
AVX Advanced Vector Extension. 21, 23

BIOS Basic Input-Output System. 70, 77

CMP Chip-MultiProcessor. i–iv, 1–3, 5–8, 11, 18–21, 35, 36, 38, 39, 42, 45, 47, 48, 50–53, 55, 58, 67–71, 76, 87–89, 96, 97, 113, 114, 116
CPU Central Processing Unit. 2, 11–13, 15–27, 43, 44, 50, 51, 65, 98, 114, 116

DMA Direct Memory Access. 26, 28, 88, 91, 107, 114
DRAM Dynamic Random Access Memory. 13
DVFS Dynamic Voltage-Frequency Scaling. 70, 71

EAF Evicted Address Filter. 40
eDRAM embedded Dynamic Random Access Memory. 17

FIFO First-In, First-Out. 40

GNU GPLv2 GNU General Public License (version 2). 49
GQ General Queue. 22, 23

HT Hyper Threading. 21

I/O Input/Output. 23, 57, 65, 79, 95, 107–111, 114–116
ILP Integer Linear Programming. 75
IPC Instructions Per Second count. 12, 16, 39, 44
IT Information Technology. 1

L1 Level 1. 18, 21, 23, 44, 51–53, 70, 96, 99, 116
L2 Level 2. x, 18, 19, 21, 23, 51–53, 55, 77, 96, 97, 99
L3 Level 3. 18, 19, 21, 23, 25, 26, 77, 96
LLC Last Level Cache. i–iv, xi, 1, 3, 5–8, 19–23, 26, 35–45, 47–53, 55–58, 65–69, 72, 74, 77, 82, 83, 92, 95–100, 102, 103, 105–108, 110–112, 114–117
LRU Least Recently Used. 14, 15, 20, 38–40
LTS Long Term Support. 97

MESI Modified, Exclusive, Shared, Invalid. 19, 21
MESIF Modified, Exclusive, Shared, Invalid and Forward. 19, 25
MOESI Modified, Owned, Exclusive, Shared and Invalid. 19
MRC Miss Rate Curve. 44
MTBF Mean Time Between Failures. 2, 3

NoC Network on Chip. 37
NUMA Non-Uniform Memory Access. 19, 88

OS Operating System. 11, 16, 26, 28, 35, 38, 41–43, 48, 49, 55, 69, 70, 116

PDF Parallel Depth First. 36, 37
PFN Page Frame Number. 85, 86
PID Process IDentifier. 65, 82, 94, 99
PIPT Phisically Indexed, Physically Tagged. 6, 7, 16, 18, 26, 50, 51, 55, 58
PIVT Phisically Indexed, Virtually Tagged. 17

QoS Quality of Service. i, iii, 2–4, 6, 7, 36–38, 42, 45, 47, 48, 65
QPI QuickPath Interconnect. 79

RAM Random Access Memory. 12, 16, 27, 56, 57, 69, 96, 97, 102, 105
ROCS Run-time Operating system Cache-filtering Service. 44

SEM Standard Error of the Mean. 100, 106
SIMD Single Instruction Multiple Data. 21, 23
SLA Service Level Agreement. 2, 3, 48
SMT Simultaneous Multi-Threading. 21, 52, 77, 116

SPEC Standard Performance Evaluation Corporation. 7, 98, 103, 114, 115
SRAM Static Random Access Memory. 12
SRC Stall Rate Curve. 44
SRM Selected Region Mapping. 43, 44

TAT Turn-Around Time. 2
TLB Translation Lookaside Buffer. 16–18, 26, 27, 33, 51
TLP Thread Level Parallelism. 2

UCP Utility-based Cache Partitioning. 39, 40
UMON Utility MONitoring. 39

VIPT Virtually Indexed, Physically Tagged. 17, 18, 51
VIVT Virtually Indexed, Virtually Tagged. 16, 17
VM Virtual Machine. 36–38, 46, 49, 115, 116

Bibliography

[1] Inc. Advanced Micro Devices. "Memory Coherency and Protocol". In: *AMD64 Architecture Programmer's Manual Volume 2: System Programming.* Intel Corporation, May 2013. Chap. 7.3. URL: `http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/24593_APM_v21.pdf`.

[2] M. Alhamad, T. Dillon, and E. Chang. "Conceptual SLA framework for cloud computing". In: *Digital Ecosystems and Technologies (DEST), 2010 4th IEEE International Conference on.* 2010, pp. 606–610. DOI: `10.1109/DEST.2010.5610586`.

[3] Shane Amante et al. "Inter-provider quality of service". In: *White paper draft* 1 (2006).

[4] Luiz André Barroso and Urs Hölzle. "The Case for Energy-Proportional Computing". In: *Computer* 40.12 (Dec. 2007), pp. 33–37. ISSN: 0018-9162. DOI: `10.1109/MC.2007.443`. URL: `http://dx.doi.org/10.1109/MC.2007.443`.

[5] Shane Bell 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.* IEEE. 2008, pp. 88–598.

[6] Andreas Berl et al. "Energy-efficient cloud computing". In: *The Computer Journal* 53.7 (2010), pp. 1045–1051.

[7] Guy E. Blelloch and Phillip B. Gibbons. "Effectively Sharing a Cache Among Threads". In: *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures.* SPAA '04. Barcelona, Spain: ACM, 2004, pp. 235–244. ISBN: 1-58113-840-7. DOI: `10.1145/1007912.1007948`. URL: `http://doi.acm.org/10.1145/1007912.1007948`.

[8] Brian K. Bray, William L. Lunch, and Michael J. Flynn. *Page Allocation to Reduce Access Time of Physical Caches.* Tech. rep. 1990.

[9] Richard B Bunt and Jennifer M. Murphy. "The measurement of locality and the behaviour of programs". In: *The computer journal* 27.3 (1984), pp. 238–245.

[10] Rajkumar Buyya et al. "Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing As the 5th Utility". In: *Future Gener. Comput. Syst.* 25.6 (June 2009), pp. 599–616. ISSN: 0167-739X. DOI: 10.1016/j.future.2008.12.001. URL: http://dx.doi.org/10.1016/j.future.2008.12.001.

[11] S. Byna, Yong Chen, and Xian-He Sun. "A Taxonomy of Data Prefetching Mechanisms". In: *Parallel Architectures, Algorithms, and Networks, 2008. I-SPAN 2008. International Symposium on.* May 2008, pp. 19–24. DOI: 10.1109/I-SPAN.2008.24.

[12] Rodrigo N Calheiros et al. "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms". In: *Software: Practice and Experience* 41.1 (2011), pp. 23–50.

[13] Carlos Carvalho. "The gap between processor and memory speeds". In: *Proc. of IEEE International Conference on Control and Automation.* 2002.

[14] Shimin Chen et al. "Scheduling Threads for Constructive Cache Sharing on CMPs". In: *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures.* SPAA '07. San Diego, California, USA: ACM, 2007, pp. 105–115. ISBN: 978-1-59593-667-7. DOI: 10.1145/1248377.1248396. URL: http://doi.acm.org/10.1145/1248377.1248396.

[15] *Control Group Linux documentation.* URL: %5Chttps://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.

[16] Henry Cook et al. "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. Tel-Aviv, Israel: ACM, 2013, pp. 308–319. ISBN: 978-1-4503-2079-5. DOI:

10.1145/2485922.2485949. URL: http://doi.acm.org/10.1145/2485922.2485949.

[17] Intel Corporation. *First the Tick, Now the Tock: Next Generation Intel Microarchitecture (Nehalem)*. White paper. 2008. URL: http://www.intel.com/pressroom/archive/reference/whitepaper_Nehalem.pdf.

[18] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation, Sept. 2014. URL: http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html.

[19] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developers Manual Volume 1: Basic architecture*. Intel Corporation, Jan. 2015. URL: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf.

[20] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developers Manual Volume 2: Instruction Set Reference, A-Z*. Intel Corporation, Jan. 2015. URL: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf.

[21] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developers Manual, Volume 3: System Programming Guide*. Intel Corporation, Jan. 2015. URL: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf.

[22] Intel Corporation. “Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors”. In: Intel Corporation, 2009. URL: https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.

[23] Michael Cusumano. “Cloud Computing and SaaS As New Computing Platforms”. In: *Commun. ACM* 53.4 (Apr. 2010), pp. 27–29. ISSN: 0001-0782. DOI: 10.1145/1721654.1721667. URL: http://doi.acm.org/10.1145/1721654.1721667.

[24] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: http://doi.acm.org/10.1145/1327452.1327492.

[25] Peter J. Denning. “Thrashing: Its Causes and Prevention”. In: *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I.* AFIPS ’68 (Fall, part I). San Francisco, California: ACM, 1968, pp. 915–922. DOI: 10.1145/1476589.1476705. URL: http://doi.acm.org/10.1145/1476589.1476705.

[26] Xiaoning Ding, Kaibo Wang, and Xiaodong Zhang. “SRM-Buffer: An OS Buffer Management Technique to Prevent Last Level Cache from Thrashing in Multicores”. In: *Proc. of EuroSys.* 2011.

[27] Constantinos Evangelinos and Chris N. Hill. “Cloud Computing for parallel Scientific HPC Applications: Feasibility of Running Coupled Atmosphere-Ocean Climate Models on Amazon’s EC2”. In: *In The 1st Workshop on Cloud Computing and its Applications (CCA.* 2008.

[28] Alexandra Fedorova, Sergey Blagodurov, and Sergey Zhuravlev. “Managing Contention for Shared Resources on Multicore Processors”. In: *Commun. ACM* 53.2 (Feb. 2010), pp. 49–57. ISSN: 0001-0782. DOI: 10.1145/1646353.1646371. URL: http://doi.acm.org/10.1145/1646353.1646371.

[29] *First buddy half optimisation.* URL: https://www.kernel.org/pub/linux/kernel/people/akpm/patches/2.6/2.6.7/2.6.7-mm1/broken-out/buddy-reordering.patch.

[30] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. “Stride Directed Prefetching in Scalar Processors”. In: *Proceedings of the 25th Annual International Symposium on Microarchitecture.* MICRO 25. Portland, Oregon, USA: IEEE Computer Society Press, 1992, pp. 102–110. ISBN: 0-8186-3175-9. URL: http://dl.acm.org/citation.cfm?id=144953.145006.

[31] Íñigo Goiri et al. “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. Ischia, Italy: Springer-Verlag, 2010, pp. 34–47. ISBN: 3-642-15680-0, 978-3-642-15680-9. URL: http://dl.acm.org/citation.cfm?id=1884547.1884551.

[32] Zhenhuan Gong and Xiaohui Gu. “Pac: Pattern-driven application consolidation for efficient cloud computing”. In: *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on.* IEEE. 2010, pp. 24–33.

[33] Sriram Govindan et al. “Cuanta: Quantifying Effects of Shared On-chip Resource Interference for Consolidated Virtual Machines”. In: *Proceedings of the 2Nd ACM Symposium on Cloud Computing.* SOCC ’11. Cascais, Portugal: ACM, 2011, 22:1–22:14. ISBN: 978-1-4503-0976-9. DOI: `10.1145/2038916.2038938`. URL: `http://doi.acm.org/10.1145/2038916.2038938`.

[34] Marjan Gusev and Sasko Ristov. “The optimal resource allocation among virtual machines in cloud computing”. In: *CLOUD COMPUTING 2012, The Third International Conference on Cloud Computing, GRIDs, and Virtualization.* 2012, pp. 36–42.

[35] Ahmed Hemani et al. “Network on chip: An architecture for billion transistor era”. In: *Proceeding of the IEEE NorChip Conference.* Vol. 31. 2000.

[36] John L Hennessy and David A Patterson. “Computer architecture: a quantitative approach”. In: 5th ed. Elsevier, 2012. Chap. 2.1.

[37] Glenn Hinton. *Key Nehalem Choices.* Presentation at Stanford University. 2010. URL: `http://web.stanford.edu/class/ee380/Abstracts/100217-slides.pdf`.

[38] R. Hund, C. Willems, and T. Holz. “Practical Timing Side Channel Attacks against Kernel Space ASLR”. In: *Security and Privacy (SP), 2013 IEEE Symposium on.* May 2013, pp. 191–205. DOI: `10.1109/SP.2013.23`.

[39] *Intel Forecasts Sales That May Top Estimates on Server Chips.* Oct. 2013. URL: `http://www.bloomberg.com/news/articles/2013-10-15/intel-forecasts-sales-that-may-top-analyst-estimates-on-servers`.

[40] *Intel Hyper-Threading Technology.* `http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html`.

[41] *Intel® Turbo Boost Technology.* `http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html`.

[42] Bart Jacob et al. *On Demand Operating Environment: Managing the Infrastructure.* IBM, International Support Organization, 2004.

[43] Xinxin Jin et al. "A Simple Cache Partitioning Approach in a Virtualized Environment". In: *Proc. of ISPA.* 2009.

[44] Gueyoung Jung et al. "Mistral: Dynamically Managing Power, Performance, and Adaptation Cost in Cloud Infrastructures". In: *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on.* 2010, pp. 62–73. DOI: 10.1109/ICDCS.2010.88.

[45] Mikyung Kang et al. "Design and Development of a Run-Time Monitor for Multi-Core Architectures in Cloud Computing". In: *Sensors* 11.4 (2011), pp. 3595–3610. ISSN: 1424-8220. DOI: 10.3390/s110403595. URL: http://www.mdpi.com/1424-8220/11/4/3595.

[46] R. E. Kessler and Mark D. Hill. "Page Placement Algorithms for Large Real-indexed Caches". In: *ACM Trans. Comput. Syst.* 10.4 (Nov. 1992), pp. 338–359. ISSN: 0734-2071. DOI: 10.1145/138873.138876. URL: http://doi.acm.org/10.1145/138873.138876.

[47] JongWon Kim et al. "Explicit Non-reusable Page Cache Management to Minimize Last Level Cache Pollution". In: *Proc. of ICCIT.* 2011.

[48] Kenneth C. Knowlton. "A Fast Storage Allocator". In: *CACM* (1965).

[49] Younggyun Koh et al. "An analysis of performance interference effects in virtual environments". In: *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on.* IEEE. 2007, pp. 200–209.

[50] David G. Korn and Kiem-Phong Vo. "In Search of a Better Malloc". In: *USENIX Summer.* Portland, Oregon, USA, June 1985, pp. 490–506.

[51] *KVM home page.* URL: http://www.linux-kvm.org/page/Main_Page.

[52] Oded Lempel. *2nd Generation Intel Core Processor Family: Intel Core i7, i5 and i3*. Presentation at HotChips 2011. URL: `http://www.hotchips.org/wp-content/uploads/hc_archives/hc23/HC23.19.9-Desktop-CPUs/HC23.19.911-Sandy-Bridge-Lempel-Intel-Rev%207.pdf`.

[53] Jiang Lin et al. “Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems”. In: *Proc. of HPCA*. 2008.

[54] ARM Ltd. *Migrating a software application from ARMv5 to ARMv7-A/R Application Note 425*. URL: `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0425/ch03s12s01.html`.

[55] HJ Lu et al. “Using hugetlbfs for mapping application text regions”. In: *Proceedings of the Linux Symposium*. Vol. 2. 2006, pp. 75–82.

[56] Jason Mars et al. “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. Porto Alegre, Brazil: ACM, 2011, pp. 248–259. ISBN: 978-1-4503-1053-6. DOI: `10.1145/2155620.2155650`. URL: `http://doi.acm.org/10.1145/2155620.2155650`.

[57] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. Birmingham, UK, UK: Wrox Press Ltd., 2008. ISBN: 0470343435, 9780470343432.

[58] Peter Mell and Tim Grance. “The NIST definition of cloud computing”. In: (2011).

[59] Michael Mitzenmacher. “Compressed Bloom Filters”. In: *IEEE/ACM Trans. Netw.* 10.5 (Oct. 2002), pp. 604–612. ISSN: 1063-6692. DOI: `10.1109/TNET.2002.803864`. URL: `http://dx.doi.org/10.1109/TNET.2002.803864`.

[60] Gordon E. Moore. “Cramming More Components Onto Integrated Circuits”. In: *Electronics* (1965).

[61] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. “Q-clouds: Managing Performance Interference Effects for QoS-aware Clouds”. In: *Proceedings of the 5th European Conference on Computer Systems*. EuroSys ’10. Paris, France: ACM, 2010, pp. 237–250. ISBN:

978-1-60558-577-2. doi: 10.1145/1755913.1755938. url: http://doi.acm.org/10.1145/1755913.1755938.

[62] *Octeon processors family by Cavium Networks.* http://www.cavium.com/newsevents_octeon_cavium.html. Sept. 2004.

[63] Simon Ostermann et al. "A performance analysis of EC2 cloud computing services for scientific computing". In: *Cloud Computing.* Springer, 2010, pp. 115–131.

[64] Mark S Papamarcos and Janak H Patel. "A low-overhead coherence solution for multiprocessors with private cache memories". In: *ACM SIGARCH Computer Architecture News* 12.3 (1984), pp. 348–354.

[65] Milad Pastaki Rad et al. "A Survey of Cloud Platforms and Their Future". In: *Computational Science and Its Applications -ICCSA 2009.* Ed. by Osvaldo Gervasi et al. Vol. 5592. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 788–796. isbn: 978-3-642-02453-5. doi: 10.1007/978-3-642-02454-2_61. url: http://dx.doi.org/10.1007/978-3-642-02454-2_61.

[66] Pankesh Patel, Ajith Ranabahu, and Amit Sheth. "Service level agreement in cloud computing". In: (2009).

[67] *Performance analysis tools for Linux.* http://lxr.free-electrons.com/source/tools/perf/Documentation/perf.txt?v=3.9.

[68] *QEMU home page.* url: http://www.linux-kvm.org/page/Main_Page.

[69] Moinuddin K. Qureshi and Yale N. Patt. "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches". In: *Proc. of MICRO.* 2006.

[70] B.P. Rimal, Eunmi Choi, and I. Lumb. "A Taxonomy and Survey of Cloud Computing Systems". In: *INC, IMS and IDC, 2009. NCM '09. Fifth International Joint Conference on.* Aug. 2009, pp. 44–51. doi: 10.1109/NCM.2009.218.

[71] Daniel Sanchez and Christos Kozyrakis. "The ZCache: Decoupling Ways and Associativity". In: *Proc. of MICRO.* 2010.

[72] Daniel Sanchez and Christos Kozyrakis. "Vantage: Scalable and Efficient Fine-Grain Cache Partitioning". In: *Proc. of ISCA.* 2011.

[73] A. Scolari et al. “A Survey on Recent Hardware and Software-Level Cache Management Techniques”. In: *Parallel and Distributed Processing with Applications (ISPA), 2014 IEEE International Symposium on.* Aug. 2014, pp. 242–247. DOI: `10.1109/ISPA.2014.41`.

[74] Vivek Seshadri et al. “The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing”. In: *Proc. of PACT.* 2012.

[75] Stuart C. Shapiro. *Encyclopedia of Artificial Intelligence.* 2nd. New York, NY, USA: John Wiley & Sons, Inc., 1992. ISBN: 0471503053.

[76] Akbar Sharifi et al. “Courteous Cache Sharing: Being Nice to Others in Capacity Management”. In: *Proceedings of the 49th Annual Design Automation Conference.* 2012.

[77] Livio Soares, David Tam, and Michael Stumm. “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.* IEEE Computer Society. 2008, pp. 258–269.

[78] *SPEC CPU2006.* URL: `http://www.spec.org/cpu2006`.

[79] Shekhar Srikantaiah, Aman Kansal, and Feng Zhao. “Energy Aware Consolidation for Cloud Computing”. In: *Proceedings of the 2008 Conference on Power Aware Computing and Systems.* HotPower’08. San Diego, California: USENIX Association, 2008, pp. 10–10. URL: `http://dl.acm.org/citation.cfm?id=1855610.1855620`.

[80] Viji Srinivasan, Edward S Davidson, and Gary S Tyson. “A prefetch taxonomy”. In: *Computers, IEEE Transactions on* 53.2 (2004), pp. 126–140.

[81] V. Stantchev. “Performance Evaluation of Cloud Computing Offerings”. In: *Advanced Engineering Computing and Applications in Sciences, 2009. ADVCOMP ’09. Third International Conference on.* Oct. 2009, pp. 187–192. DOI: `10.1109/ADVCOMP.2009.36`.

[82] *Sysfs Linux documentation.* URL: `https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt`.

[83] David Tam et al. “Managing Shared L2 Caches on Multicore Systems in Software”. In: *Proc. of WIOSCA.* 2007.

[84] Lingjia Tang, Jason Mars, and Mary Lou Soffa. “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. San Jose, California: ACM, 2011, pp. 12–21. ISBN: 978-1-4503-0708-6. DOI: 10.1145/2000417.2000419. URL: http://doi.acm.org/10.1145/2000417.2000419.

[85] Lingjia Tang et al. “The Impact of Memory Subsystem Resource Sharing on Datacenter Applications”. In: *SIGARCH Comput. Archit. News* 39.3 (June 2011), pp. 283–294. ISSN: 0163-5964. DOI: 10.1145/2024723.2000099. URL: http://doi.acm.org/10.1145/2024723.2000099.

[86] Michael E. Thomadakis. *The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms*. Tech. rep. Texas A&M University, 2011.

[87] Michael E Thomadakis. “The architecture of the Nehalem processor and Nehalem-EP SMP platforms”. In: *Resource* 3 (2011), p. 2.

[88] Luis M. Vaquero et al. “A Break in the Clouds: Towards a Cloud Definition”. In: *SIGCOMM Comput. Commun. Rev.* 39.1 (Dec. 2008), pp. 50–55. ISSN: 0146-4833. DOI: 10.1145/1496091.1496100. URL: http://doi.acm.org/10.1145/1496091.1496100.

[89] Christian Vecchiola, Suraj Pandey, and Rajkumar Buyya. “High-performance cloud computing: A view of scientific applications”. In: *Pervasive Systems, Algorithms, and Networks (ISPAN), 2009 10th International Symposium on*. IEEE. 2009, pp. 4–16.

[90] A.J. Viterbi. “Error bounds for convolutional codes and an asymptotically optimum decoding algorithm”. In: *Information Theory, IEEE Transactions on* 13.2 (Apr. 1967), pp. 260–269. ISSN: 0018-9448. DOI: 10.1109/TIT.1967.1054010.

[91] Zhitao Wan. “Sub-millisecond level latency sensitive Cloud Computing infrastructure”. In: *Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT), 2010 International Congress on*. 2010, pp. 1194–1197. DOI: 10.1109/ICUMT.2010.5676670.

[92] Lizhe Wang et al. “Cloud Computing: a Perspective Study”. English. In: *New Generation Computing* 28.2 (2010), pp. 137–146. ISSN: 0288-3635. DOI: 10.1007/s00354-008-0081-5. URL: http://dx.doi.org/10.1007/s00354-008-0081-5.

[93] Xiaolin Wang et al. “A dynamic cache partitioning mechanism under virtualization environment”. In: *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*. IEEE. 2012, pp. 1907–1911.

[94] Henry Wong. *Intel Ivy Bridge Cache Replacement Policy*. URL: http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/.

[95] *x86_64 Linux memory mappings*. URL: http://lxr.free-electrons.com/source/Documentation/x86/x86_64/mm.txt.

[96] Kaiqi Xiong and H. Perros. “Service Performance and Analysis in Cloud Computing”. In: *Services - I, 2009 World Conference on*. July 2009, pp. 693–700. DOI: 10.1109/SERVICES-I.2009.121.

[97] Hailong Yang et al. “Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers”. In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ISCA ’13. Tel-Aviv, Israel: ACM, 2013, pp. 607–618. ISBN: 978-1-4503-2079-5. DOI: 10.1145/2485922.2485974. URL: http://doi.acm.org/10.1145/2485922.2485974.

[98] Ying Ye et al. “COLORIS: A Dynamic Cache Partitioning System Using Page Coloring”. In: *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. PACT ’14. Edmonton, AB, Canada: ACM, 2014, pp. 381–392. ISBN: 978-1-4503-2809-8. DOI: 10.1145/2628071.2628104. URL: http://doi.acm.org/10.1145/2628071.2628104.

[99] Matei Zaharia et al. “Spark: cluster computing with working sets”. In: *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. 2010, pp. 10–10.

[100] Panyong Zhang et al. “Evaluating the Effect of Huge Page on Large Scale Applications”. In: *Networking, Architecture, and Storage, 2009. NAS 2009. IEEE International Conference on*. 2009, pp. 74–81. DOI: 10.1109/NAS.2009.18.

[101] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. “Towards Practical Page Coloring-based Multi-core Cache Management”. In: *Proc. of EuroSys*. 2009.