Design and Implementation of A Memory Allocator to Achieve Cache Partitioning in the Linux Kernel

Elena Lucherini

supervised by
Giorgio BUTTAZZO
Björn B. BRANDENBURG
Tommaso CUCINOTTA
Alessandro BIONDI

October 3, 2017
Abstract

Predictability is one of the key properties of hard real-time systems. A system is predictable when it is possible to guarantee in advance that the timing constraints of all the tasks in the system will be met. Achieving predictability on modern multi-core systems is, however, very challenging, mainly due to shared architectural resources, such as cache memories. It has been shown that the interference caused by a shared cache can increase the task execution time by up to 40%, with even higher delay spikes that are hard to predict.

The problem of mitigating the effects of cache interference has received much attention in the real-time systems community. A popular and effective approach is a software cache-partitioning technique called page coloring. With page coloring, memory is partitioned into colors. Assigning different colors to different tasks avoids cache interference between the tasks, as they do not contend for the same cache partitions. Therefore, the tasks are spatially isolated in cache.

Various uses of page coloring have been proposed in the literature. However, the existing implementation efforts only take into account single page requests and do not isolate kernel processes, which can still interfere with other tasks.

Conversely to previous works, this thesis targets both user-space and kernel memory. The objective of this work is to provide insights into the design and implementation of a page coloring solution for the latest Linux PREEMPT-RT kernel. The proposed implementation supports both single-page and multiple-page allocation schemes.

Experimental results show significant improvements in task isolation, as our solution reduces the effects of cache interference. Interference affecting worst-case execution times is decreased from 14.6% to a maximum of 4.4%.
Acknowledgments

First and foremost, I would like to thank my advisors, Giorgio Buttazzo (Scuola Superiore Sant’Anna) and Björn B. Brandenburg (Max Planck Institute for Software Systems), for their invaluable support. I am also very grateful to Alessandro Biondi, Tommaso Cucinotta, and Daniel Bristot de Oliveira for their advice in the last months of this thesis.

Many thanks to the guys at MPI-SWS for keeping me company in the seven months I spent in Kaiserslautern. A special ‘thank you’ goes to my office-mates, Arpan and Manohar, my flatmate Rohith, and the other guys of the RTS group. Also, thanks to the guys at the ReTiS Lab for bearing with me in the past few months!

Qualche parola anche per i miei colleghi universitari di sempre: Sara, Francesca, Michele, Antonio e tutti gli altri. Avete reso leggeri cinque anni molto pesanti. Voglio includere, inoltre, gli altri miei amici, quelli vicini così come quelli più lontani. Giulia, le Buste e i ragazzi e le ragazze di Pink Riot: grazie per l’aiuto diretto, gli incoraggiamenti e le distrazioni.

Ultimo, ma non meno importante: un grazie infinito alla mia famiglia per aver sempre creduto in me e per aver sempre supportato le mie decisioni.

수많은 알 수 없는 길 속에
희망한 빛을 난 쫓아가
언제까지라도 함께하는 거야
다시 만난 나의 세계
Contents

List of Figures ii
List of Tables iv
List of Abbreviations v

1 Introduction 1
  1.1 Related Work ............................................. 3
  1.2 Contributions ............................................. 5
  1.3 Organization ............................................. 5

2 Background 7
  2.1 Memory Management in Linux ......................... 7
    2.1.1 Physical Memory ..................................... 7
    2.1.2 Virtual Memory ..................................... 8
    2.1.3 Kernel Dynamic Memory Management ............... 10
  2.2 Cache .................................................. 17
    2.2.1 Cache Interference .................................. 19
    2.2.2 Cache Management Schemes ......................... 20

3 Memory Allocation: A Closer Look 23
  3.1 Tracing the Allocations ................................. 24
  3.2 Determining the Allocation Order ...................... 25
4 Design and Implementation

4.1 Assigning Colors to a Process ........................................... 29
  4.1.1 Overview ............................................................. 29
  4.1.2 Categories: Implementation ........................................ 32
  4.1.3 Number of Colors: Implementation ................................ 33
  4.1.4 Assigning Colors to a Category: Implementation .............. 33

4.2 Single-Page Allocation ................................................... 35
  4.2.1 Buddy Allocator: Implementation Details ......................... 36
  4.2.2 Colored Order-Zero Allocator ..................................... 40

4.3 Multiple-Page Allocation ............................................... 48
  4.3.1 SLUB: Implementation Details ..................................... 50
  4.3.2 Vmalloc: Implementation Details ................................ 54
  4.3.3 Colored SLUB Allocator ............................................ 57

5 Evaluation ............................................................................ 65

5.1 Overview ......................................................................... 65
  5.2 Experiment 1: Last-Level Cache Misses ............................... 67
  5.3 Experiment 2: Running Times ........................................... 71
  5.4 Experiment 3: Allocation Overhead ................................... 72

6 Conclusion and Discussion .................................................. 74

  6.1 Future Directions .......................................................... 74
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Relationship among nodes, zones and pages</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Free page block management</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Example of allocation of an order 2 block, when the smallest available block is of order 4</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>Components of the slab allocator</td>
<td>14</td>
</tr>
<tr>
<td>2.5</td>
<td>Memory address as seen by set-associative caches</td>
<td>18</td>
</tr>
<tr>
<td>2.6</td>
<td>Cache mapping of a virtual address used by a task</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>Page coloring example with four and two partitions</td>
<td>31</td>
</tr>
<tr>
<td>4.2</td>
<td>Color bitmaps</td>
<td>34</td>
</tr>
<tr>
<td>4.3</td>
<td>Overview of the main functions in the buddy allocator</td>
<td>36</td>
</tr>
<tr>
<td>4.4</td>
<td>Illustration of an array of colored freelists</td>
<td>39</td>
</tr>
<tr>
<td>4.5</td>
<td>Zone color bitmap</td>
<td>39</td>
</tr>
<tr>
<td>4.6</td>
<td>Example of colored order-zero allocation</td>
<td>43</td>
</tr>
<tr>
<td>4.7</td>
<td>Color bitmap and round-robin bitmap</td>
<td>45</td>
</tr>
<tr>
<td>4.8</td>
<td>Zone color bitmap: example</td>
<td>45</td>
</tr>
<tr>
<td>4.9</td>
<td>Overview of the main functions in the SLUB allocator</td>
<td>52</td>
</tr>
<tr>
<td>4.10</td>
<td>Overview of the main functions in vmalloc</td>
<td>55</td>
</tr>
<tr>
<td>4.11</td>
<td>Scheme of the colored multiple-page allocator</td>
<td>57</td>
</tr>
<tr>
<td>5.1</td>
<td>Experiment 1: results</td>
<td>69</td>
</tr>
<tr>
<td>5.2</td>
<td>Experiment 1: comparison of baseline and page coloring</td>
<td>70</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Example of mm_page_alloc output .................................. 25
3.2 Example of function_graph output ................................. 26
3.3 A failed order-one allocation falls back to order zero. .............. 27

4.1 PC vmalloc: Modifications to the page descriptor ................. 58
4.2 PC vmalloc: Virtual-to-physical address translation functions .... 59
4.3 Utility functions of the SLUB allocator ............................ 63

5.1 Experiment 1: cases .................................................. 68
5.2 Experiment 2: average and maximum execution times ............ 71
5.3 Experiment 3: average and maximum allocation times ........... 73
## List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>GFP</td>
<td>Get Free Page</td>
</tr>
<tr>
<td>LLC</td>
<td>Last-Level Cache</td>
</tr>
<tr>
<td>LSB</td>
<td>Least Significant Bit</td>
</tr>
<tr>
<td>PC</td>
<td>Page Coloring</td>
</tr>
<tr>
<td>PFN</td>
<td>Page Frame Number</td>
</tr>
<tr>
<td>PGD</td>
<td>Page Global Directory</td>
</tr>
<tr>
<td>PID</td>
<td>Process IDentifier</td>
</tr>
<tr>
<td>PMD</td>
<td>Page Middle Directory</td>
</tr>
<tr>
<td>PTE</td>
<td>Page Table Entry</td>
</tr>
<tr>
<td>SLOB</td>
<td>Simple List Of Blocks</td>
</tr>
<tr>
<td>WCET</td>
<td>Worst-Case Execution Time</td>
</tr>
</tbody>
</table>
1 INTRODUCTION

Predictability is one of the key properties a real-time system must have in order to support critical applications. We say that a system is predictable when it is possible to guarantee in advance that the timing constraints of all the tasks in the system will be met [1]. In the off-line analysis of a task set, we take into account the worst-case execution times (WCETs).

The predictability of a system is, however, affected by a range of factors, including the internal components of the processor itself. For example, while caches statistically improve the average performance of the system, they are a source of unpredictable timing behavior. When a cache miss occurs (that is, when data is not found in the cache), memory must be accessed and the retrieved data must also be transferred to the cache, potentially leading to longer access times. It has been estimated that, in the average case, cache misses occur only 20% of the time: however, statistical observations do not provide any guarantee for the worst-case behavior.

Shared resources in a system are an additional cause of non-determinism. On multi-core platforms, cores share resources with the primary goal of improving system performance and efficiency. Cores have access to small private caches, as well as larger, shared, ones: this leads to reduced average execution times for tasks. However, shared caches introduce cache interference. We can distinguish among intra-task, intra-core and inter-core interference [2]. Intra-task and intra-core interference are common in both single-core and multi-core systems. Intra-task interference occurs when two memory entries accessed by a task are mapped to the same cache set, resulting in the task
evicting its own content from the cache. Intra-core interference occurs when the cache content of a task is evicted as the task itself is preempted. Finally, inter-core cache interference introduces dependencies among cores, and it is only present on multi-core platforms. This kind of interference occurs when tasks running on different cores access the cache at the same time, potentially evicting each other’s cache content.

Cache interference negatively affects the predictability of a system. It has been shown that intra-core interference can increase the task execution time by up to 27%. Similarly, inter-core interference can lead to an increase in execution times of 40% [3], with even higher delay spikes [4].

A simple approach to circumvent the problem in the static analysis of a task set consists in assuming that all accesses in cache will result in a cache miss. However, this leads to highly pessimistic WCET estimations.

The problem of mitigating the effects of cache interference has received much attention in the real-time systems community (see Section 1.1). Among other solutions, a popular and effective approach is a software cache partitioning technique called page coloring, or cache coloring (see Section 2.2.2).

With page coloring, memory is partitioned into colors. Each task is also assigned a color (or a set of colors) that determines from which portion(s) of memory the task can allocate. Additionally, the color corresponds to the cache partition, either exclusive or shared, to which the task memory content is mapped. Assigning different colors to different tasks avoids cache interference between the tasks, as they do not contend for the same cache partitions. The tasks are now spatially isolated in cache - that is, they do not interfere with each other.

Unlike hardware-based partitioning schemes, page coloring does not require additional hardware support other than a set-associative cache, which is already available on most modern multi-core processors.

Various uses of page coloring have been proposed in the literature (see Section 1.1).
However, most of the previous work puts a greater emphasis on the theoretical aspects than on the implementation process. Furthermore, the existing implementation efforts only take into account single page requests and do not isolate kernel memory.

With this thesis, we want to target both user-space and kernel memory. The objective of this work is to provide insights into the design and implementation of a page coloring solution for the latest Linux PREEMPT-RT [5] kernel.

1.1 Related Work

Improved throughput. Page coloring was first presented by Taylor et al. [6] as a technique to reduce cache misses on the MIPS OS: the goal here is to mitigate the variance of program runtimes and improve performance in the average case.

Bray et al. [7] and Kessler and Hill [8] propose page coloring as a replacement of the arbitrary virtual-to-physical page mapping to reduce average access time of physically-indexed caches.

A few other examples of operating systems that employ page coloring to improve the system performance are Solaris [9], FreeBSD [10], and Windows NT [11].

Single-core real-time systems. Page coloring was later implemented in single-core real-time systems to improve predictability. Wolfe [12] is the first to propose an OS-based software cache partitioning mechanism similar to page coloring with the objective of preventing cache interference real-time systems. The author targets static task sets in preemptive single-core systems.

On the other hand, Mueller [13] is the first to present an implementation of page coloring for real-time systems at the compiler level. The proposed compiler generates object files for each task that are separated in code and data partitions. Such partitions are no larger than the cache partitions.

Liedtke et al. [14] introduce page coloring to mitigate the problem of cache interfer-
ence in a single-core system, while Bui et al. [15] provide a genetic algorithm to solve the optimization problem that finds the minimum worst-case system utilization under page coloring.

**Multi-core systems.** The first evaluation of page coloring on a real, non-simulated, multi-core environment, is presented by Lin et al. [16]. The authors take into account different metrics, such as performance, fairness, and Quality of Service, confirming important findings from previous work, such as the average performance improvement.

Other OS-level cache partitioning techniques based on page coloring have been proposed for shared caches in multi-core systems by Cho and Jin [17], Tam et al. [18], and Soares et al. [19].

Guan et al. [20] present a scheduling algorithm for non-preemptive multi-core hard real-time systems using page coloring. The algorithm schedules tasks so that, at any time, any two running tasks do not contend for the same cache partitions.

Kim et al. [21] introduce an OS-level cache management scheme to solve typical issues of page coloring in multi-core real-time systems, namely the memory co-partitioning problem, and the limited number of partitions available. The idea is to assign private partitions to each core.

Ward et al. [22] analyze two cache management techniques to achieve predictability in multi-core resource-allocation framework $MC^2$ (Mixed-Criticality on Multi-Core). Their work shows that, despite the increase of system overhead, the enforcement of cache management policies, such as page coloring, leads to a reduction of WCET and to an improvement of schedulability.

Performance isolation in multi-core platforms has also been treated in Memguard [23, 24, 25], Colored Lockdown [26], and PALLOC [27], as part of the Single Core Equivalence (SCE) package [28]. The aim of the SCE package is to enable users to assume constant WCET for applications running on multi-core systems. This allows a
significant increase of timing determinism, which becomes comparable to that achievable in single-core systems, while preserving the performance benefits of multiple cores.

Previous work that is similar to ours has been proposed by Konstantopoulos [29], who presents an implementation of page coloring on the Linux kernel. In contrast to the objective of our thesis, the implementation only takes into account single-page allocation and does not provide further isolation for kernel processes.

1.2 Contributions

The main contributions of this thesis are the following:

- Insights into the design and implementation of an allocator that supports page coloring in the Linux 4.9 PREEMPT-RT kernel. Our solution isolates kernel processes from user-space tasks, and it supports single-page and multiple-page allocation. The former is implemented in the Linux physical page allocator. Multiple-page allocation is, instead, supported by the general-purpose allocator.

- An evaluation of the developed solution. The results show that the allocator effectively isolates tasks in cache and mitigates the negative effects of cache interference.

1.3 Organization

The rest of the thesis is organized as follows. Chapter 2 introduces the essential concepts necessary to understand the work. Namely, it touches upon: memory management in Linux, with particular emphasis on physical memory, virtual memory, and the most frequently used kernel dynamic memory allocators; caches, cache interference, and methods to eliminate it, with focus on software-based cache partitioning technique page coloring.
Chapter 3 presents a preliminary study on memory allocation in Linux, carried out with the use of an internal function tracer. The objective of the chapter is to investigate whether changing the page allocator to implement page coloring conflicts with implicit requirements in Linux.

Chapter 4 discusses the implementation of our page coloring solution. The chapter describes how to determine the number of colors in the system and how to assign colors to a process, following with details on single and multiple-page allocation.

Chapter 5 illustrates the evaluation of the developed solution. The results show that cache interference is effectively reduced. The experiments focus on three fundamental metrics: cache misses, execution times, and allocation overhead.

The thesis concludes with Chapter 6, which presents a final discussion and possible future directions.
2 BACKGROUND

This chapter presents the main concepts needed to understand this work. Section 2.1 briefly describes how memory is managed in Linux version 4.9, while Section 2.2 introduces caches and cache management mechanisms, with particular focus on page coloring.

2.1 Memory Management in Linux

We first present the organization of physical memory from the operating system’s point of view, following with an overview of virtual memory. The section ends with a description of the kernel dynamic memory allocators.

2.1.1 Physical Memory

In Linux, memory is assumed to be arranged into nodes, as required by the Non-Uniform Memory Access (NUMA) architecture. If the architecture is Uniform Memory Access (UMA), then memory is treated as a single node [30].

Each node is divided into zones, representing ranges within memory. Zones are further divided into pages, which is regarded as the basic unit of memory management. The relationship among nodes, zones and pages is summarized in Figure 2.1.

Zones are identified by their type, which depends on their usage [31]. The layout of the zones varies from architecture to architecture. However, we will provide a general description of the most common memory zones, which are: ZONE_DMA, ZONE_DMA32, ZONE_HIGHMEM and ZONE_NORMAL.
Figure 2.1: Illustration of the main elements in Linux memory management (adapted from [30]). Each node is divided into zones, of different types. Zones consists of pages.

In some architectures, such as x86, Direct Memory Access (DMA) operations can be performed only to the lowest ranges of physical addresses. Thus, pages that are typically used for (but not restricted to) DMA are grouped into ZONE_DMA and ZONE_DMA32. Specifically, ZONE_DMA32 is dedicated to 32-bit devices that perform DMA operations.

Furthermore, some portions of memory may not be permanently mapped into the kernel address space: the corresponding pages, contained in ranges of addresses called high memory, are grouped into ZONE_HIGHMEM. This zone is generally regarded as a memory reserve for the kernel and it is empty in those architectures where all pages can be permanently mapped.

Finally, the pages that are not part of any other zone, are grouped into ZONE_NORMAL. This zone comprises permanently-mapped pages. Most of the kernel allocations take place in ZONE_NORMAL.

2.1.2 Virtual Memory

Modern operating systems use virtual memory to hide the physical address space from processes [32]. Virtual memory has many advantages, some of them being:

- Processes are relocatable, i.e., they can take any place in physical memory.
- Virtualization of memory, in combination with the swapping system, allows processes to allocate more memory than would otherwise be possible. The virtual address space assigned to a process may be larger than the available physical memory. If the running process needs more memory, the content of memory portions that are currently not in use can be written to disk and replaced with content from the running process.

- Each process is oblivious of the memory space belonging to other processes, leading to increased security: if a process requests a memory address that is not mapped, the request will result in a page fault and will not be satisfied.

- Addresses that are contiguous in virtual memory may not be so in physical memory. This solves the problem of fragmentation, that is, when free memory space is scattered inside allocated blocks. If a user-space process makes a request that cannot be satisfied by a set of contiguous physical addresses, it will be assigned addresses that are contiguous in virtual memory, but not contiguous in physical memory. This way, a memory request fails only when there is not enough free available space in memory.

Addresses from the virtual address space are translated into physical ones by a component called Memory Management Unit (MMU). Virtual memory is divided into pages, corresponding to physical page frames (i.e., physical pages): resident virtual pages are mapped to physical memory at process runtime.

The mapping between virtual and physical pages is stored in data structures called page tables. In Linux, three levels of page tables exist, called respectively Page Global Directory (PGD), Page Middle Directory (PMD), and Page Table Entries (PTE). Each process descriptor holds a pointer to the corresponding process’ own PGD, which is an array contained in a physical page frame. Each active entry of the PGD array points to a PMD array, contained in another physical page. In turn, the entries in the PMD array
2.1.3 Kernel Dynamic Memory Management

This section provides a brief overview of the following kernel dynamic memory allocators:

- The physical page allocator, which satisfies requests at page granularity, allocating blocks of physically-contiguous pages.

- The slab allocators, which provide memory allocation for commonly-used data structures through `kmem_cache_alloc()`.

- The general-purpose memory allocator, `kmalloc`, built on top of the slab allocators.

- The virtually-contiguous memory allocator, `vmalloc`, which asks the page allocator for single, generally noncontiguous, physical pages, and maps them to contiguous virtual addresses.
Physical Page Allocation

Physical pages in Linux are allocated through the *buddy allocator algorithm* [33], which combines a power-of-two allocator with free-buffer coalescing.

As shown in Figure 2.2, the free pages are arranged into blocks, each being a power of two number of pages; the exponent is called *order* [30]. The allocator keeps a list, called *freelist*, for each order in each memory zone (Section 2.1.1).

When a free block of order \( n \) (that is, a block containing \( 2^n \) page frames) is requested, the allocator returns the first block from the order-\( n \) list. If there are no free blocks of the requested order, the allocator picks the first block from a higher-order list and splits it into two *buddies*: one is allocated, while the other is inserted into the freelist for the corresponding order.

Figure 2.3 shows what happens when the smallest available block is of order 2, and an order-zero block is requested. The order-2 block is split into two order-1 blocks. One is inserted into the order-1 freelist, while the other is further split into two order-zero blocks: the first order-zero block is inserted into the corresponding freelist, and the second block is allocated.

Upon its release, a block of order \( n \) is returned to the order-\( n \) freelist. Then, the allocator attempts to coalesce pairs of buddy blocks belonging to the same order.

Two blocks are buddies if they belong to the same order, and they are located in contiguous physical addresses.

The merging algorithm is iterative: if it succeeds once, the procedure is attempted again on the next higher order, and so on [32].

The buddy allocator reduces *external fragmentation* - that is, fragmentation at page granularity. However, the algorithm does not eliminate the problem: a request to the buddy allocator will still fail if there are enough free physical pages that are not contiguous.
Figure 2.3: Example of allocation of an order-zero block, when the smallest available block is of order 2 (adapted from [29, 30]). The order-2 block is split into two order-1 blocks. The first block is put back into the order-1 freelist, while the second is further split into two order-zero pages. As in the previous step, one order-zero page is reinserted into the order-zero freelist, while the other is returned to the caller.

In Linux, any process can request memory from the buddy allocator using the `alloc_pages()` function and the `alloc_page()` macro. The former takes as arguments the Get Free Page (GFP) flags to pass down to the buddy allocator, and the order of the request. GFP flags determine the behavior of the buddy allocator. GFP flags are defined in the `include/linux/gfp.h` file and are divided into different categories depending on their usage. For instance, physical address zone modifiers are flags that are used to choose the zone from which to allocate, if possible: an example is the `GFP_DMA` flag, which will request the buddy allocator to allocate from the `ZONE_DMA` memory zone.

The `alloc_page()` macro is defined as `alloc_pages(gfp_mask, 0)`. Thus, the macro only takes one argument: the GFP flags. Unlike the `alloc_pages()` function, and as the macro’s name suggests, `alloc_page()` does not allow the user to decide the order of allocation, as it will automatically request an order-zero block of pages.
Slab Allocator

The buddy allocator treats physical pages as the basic memory unit. However, in many cases memory requests are significantly smaller than the size of a page. In these situations, allocating a full page would lead to a waste of memory, causing a phenomenon called internal fragmentation: although the additional memory has been allocated, it will not be used. Therefore, the system needs an allocator that can serve requests that are smaller than a page.

In Linux, the aforementioned allocator is the slab allocator [34]. The slab allocator serves an additional purpose, motivated by the fact that kernel functions tend to request data structures of the same type [32]: instead of allocating and deallocating the same structures repeatedly, the most frequently requested objects are kept in a set of caches at an initialized state [30]. In this context, a cache is a collection of objects of the same type. For instance, a cache exists for task descriptors to be allocated at the creation of a new task, or for the structures maintained to manage the virtual-to-physical memory mapping. Objects which do not have a dedicated cache can be allocated by the slab allocator through the kmalloc caches, described later in this section.

Figure 2.4 gives an overview of the main concepts of the slab allocator. The area of memory reserved for a cache is divided into slabs. A slab is simply a block of contiguous physical memory containing objects, which can either be free or allocated.

With the slab allocator, it is possible to allocate blocks of memory that are smaller than one page, thus reducing internal fragmentation (i.e., fragmentation within a page) and waste of memory. The slab allocator does not manage page allocation; instead, it asks the buddy allocator (Section 2.1.3) for its pages.

The main function that can be used to allocate memory from the slab allocator is kmem_cache_alloc(). kmem_cache_alloc() takes two arguments: the pointer to the cache from which to allocate, and the flags that are passed down to the buddy allocator.
Caches are collections of slabs, which are, essentially, blocks of physically-contiguous memory. Each slab contains objects of the same kind.

It returns the virtual address of the first location of memory that is allocated to the process. The memory allocated with `kmem_cache_alloc()` is, thus, contiguous both in physical memory and virtual memory. The physical addresses are translated into virtual addresses through a linear mapping: this means that there exists a linear, fixed, relationship between physical and virtual addresses, and the mapping does not require modifying the page tables.

The cache system in the slab allocator serves the purpose of speeding up the allocation of frequently used objects. When a new allocation request comes, the object is taken from the list of free objects in the cache. Each time an object is freed, it is just returned to the cache. The memory that is allocated to the slabs is never freed, unless the underlying page allocator runs out of free memory.

While the original slab allocator (`SLAB`) is still available, the default allocator in Linux from version 2.6.23 on is `SLUB` [35] (the Unqueued SLAB Allocator). A third memory allocator, `SLOB` [36] (acronym for Simple List Of Blocks), was developed to replace SLAB in systems with constrained resources, such as embedded systems. In what follows, however, the focus will be on describing the SLUB allocator.

SLUB keeps the nomenclature and the interface of the original SLAB allocator,
simplifying its internal structure. For example, the numerous queues maintained by the original SLAB to manage its objects have been eliminated, reducing the overhead for obtaining a new object. Furthermore, in SLUB, there is no data structure to describe slabs, which are treated as chunks of memory containing free objects organized in a singly linked list [37]. The head of the list is retrieved from the page descriptors, which contain a pointer to the first object in the freelist.

At creation, all slabs are completely free. When an object is allocated from a slab, the slab is added to the list of partially allocated slabs in the cache descriptor. Each cache holds a list of partially allocated slabs (partial list) for each memory node in the system. When allocating new objects, the system gives priority to partial lists, even across nodes, and only if there are no partial lists does it create a new slab. The allocator does not keep track of full slabs; if an object is freed from a full slab, the slab itself is relocated and put back into the partial list.

In addition to the per-node lists, the allocator maintains a list of active slabs for each CPU using per-CPU variables. Creating a per-CPU variable allows each processor in the system to keep a copy of the variable. Accessing a per-CPU variable causes smaller overhead, as it requires no locking (however, preemption must be disabled to make sure that the current process is not moved to another processor). Since per-CPU lists make object allocation faster, SLUB prefers allocating through per-CPU lists rather than through per-node partial lists.

General-Purpose Allocation

General-purpose allocation is carried out by the kmalloc() function. kmalloc() takes as arguments the size of memory to allocate and the GFP flags, and returns the virtual address of the first location of memory that is allocated to the process.

kmalloc() is built on top of the slab allocators and relies on two sets of slab caches, implemented as arrays: kmalloc_caches and kmalloc_dma_caches. The function
can allocate from both caches. The `kmalloc_dma_caches` set is chosen when a DMA-compatible allocation is explicitly requested through the corresponding GFP flags (`GFP_DMA` or `GFP_DMA32`); in all other cases, the chosen slab cache will belong to the `kmalloc_caches` set.

An architecture-specific parameter, `KMALLOC_MAX_CACHE_SIZE`, identifies the maximum size for which the slab caches are used. If the requested size is larger than `KMALLOC_MAX_CACHE_SIZE`, then the allocation is carried out directly by the buddy allocator, which is called by `kmalloc()` after converting the requested size in bytes into the corresponding allocation order. The virtual address of the allocated block of memory is obtained from the physical address through a linear mapping, just as with the slab allocator.

If the allocation size does not exceed `KMALLOC_MAX_CACHE_SIZE`, then `kmalloc()` calls `kmem_cache_alloc()` on the slab caches mentioned above. The exact cache from which to allocate is chosen based on the requested size. Each entry of `kmalloc_caches` and `kmalloc_dma_caches` is a pointer to a cache of objects of different sizes. For example, `kmalloc-1024` is a cache of objects sized 1024 bytes, while `dma-kmalloc-256` is a cache of objects sized 256 bytes that can be used for DMA operations. The size of the biggest kmalloc and dma-kmalloc caches is equal to `KMALLOC_MAX_CACHE_SIZE`.

**Virtually-Contiguous Allocation**

Recall from Section 2.1.3 that the physical page allocator suffers from external fragmentation: if a request comes that cannot be satisfied with a physically-contiguous memory area, then it will fail even if there is enough non-contiguous free space in memory. This Section briefly describes `vmalloc`, which allocates virtually-contiguous memory that may not be contiguous in physical memory [30].

Vmalloc can only allocate memory within a given range of virtual addresses, which varies with the architecture. For example, in x86_64, all the virtual addresses reserved
to vmalloc start with the prefix ffffec9.

When the vmalloc() function is called, the allocator first searches for a free region within its address space that is large enough to satisfy the memory request. The search is performed through a linked list of structures that describe the available regions. Then, vmalloc makes requests for single pages to the buddy allocator. The received pages, which may be non-contiguous in physical memory, are then mapped to contiguous locations in virtual memory by updating the page tables with newly allocated entries.

Vmalloc does not only depend on the page allocator: it also allocates space for its area structures with kmalloc().

As mentioned in Section 2.1.3, vmalloc solves the problem of external fragmentation. However, this comes at the price of increased overhead introduced by the need of setting up the page tables at each memory request. For this reason, kmalloc() is often preferred to vmalloc(). In the Linux kernel, vmalloc is generally only used to load kernel modules into memory, and to allocate large software buffers.

2.2 Cache

In this section, we present a brief overview of the main concepts of caches, followed by a description of cache interference and of the most common cache management schemes. Please note that here, by cache, we no longer refer to the slab caches described in Section 2.1.3, but rather to the hardware CPU component.

A cache is a fast memory unit used to speed up process execution. Modern CPUs usually feature W-way set-associative caches. Data is stored in a cache in units called cache lines, or cache blocks. A set-associative cache can be seen as a two-dimensional matrix of cache lines, where the rows are called sets and the columns are called ways [38]. A W-way set-associative cache is divided into W ways, with the number of sets, n_sets, determined as follows:
Figure 2.5: Illustration of a memory address as seen by set-associative caches in a 32-bit system. The address is divided in its tag, index, and offset.

\[ n_{\text{sets}} = \frac{\text{size}_{\text{cache}}}{\text{size}_{\text{line}} \cdot W} \]

Each data block is mapped to exactly one set, determined by the value of a subset of bits in the data address, known as its index (Figure 2.5). The data block can then be stored in any way of the given set.

As shown in Figure 2.5, a cache address can be divided in three parts. The first set of bits is called its tag, and it is used for lookups. When the system needs to access data at a given address, it first checks if an entry can be found with the corresponding tag in the cache. If the lookup was successful, the system does not need to access the main memory: this situation is called a cache hit. If the data is not found in the cache (cache miss), then the system must retrieve it from the main memory.

The second set of bits is called its index, as explained above, and the remaining bits are its offset, which represents the offset in a cache line.

In most modern multi-core systems, caches are hierarchically organized in levels. Cores usually present one or more levels of private caches, possibly of different sizes, while the last-level cache (LLC) is generally shared among the cores. For instance, some of the currently available Intel processors present two Level-1 caches (one for storing data, one for instructions) and a Level-2 cache for each core; a Level-3 cache, which is the LLC, is shared among the cores.

2.2.1 Cache Interference
While caches statistically improve the system performance, from a real-time perspective, they are a source of unpredictable timing behavior [1]. Average process execution times benefit from cache hits: \textit{program locality} ensures that the most frequent memory accesses be generally restricted to a small amount of addresses. However, cache misses introduce longer memory access times: when a cache miss occurs, the system must first retrieve the data from main memory and then write it to the cache. Cache misses occur, on average, only 20\% of the time. However, this is only an estimation that cannot be used to derive the worst-case execution times (WCETs) employed in the off-line analysis of real-time task sets.

Shared last-level caches are an additional source of unpredictability in real-time multi-core systems. Each core stores in its private cache both its own data and the data that is shared among cores. When a core writes to a datum stored in other cores’ private caches, their cache content must be invalidated. Invalidation is performed to eliminate outdated copies in cache and has the consequence of increasing the process execution time.

There are different kinds of \textit{cache interference} that can affect real-time tasks running on a multi-core system [2]:

- \textit{Intra-task interference} generally occurs when two memory entries accessed by a task are mapped to the same cache set, resulting in the task evicting its own content from the cache. This kind of interference is common in both single and multi-core systems.

- \textit{Intra-core interference} occurs in single-core systems as well as in multi-core systems. When a task is preempted, its cache content is evicted from the cache. The preempted task will, thus, experience longer data access time as it is rescheduled.

- \textit{Inter-core interference} is typical of concurrent platforms, as it introduces dependencies among cores or hardware threads. It occurs when tasks running on
different cores (or on different hardware threads of the same core) access the cache at the same time, evicting each other’s cache content.

Cache interference negatively affects the predictability of a system. In what follows, we discuss some of the different mechanisms that can be used to mitigate the effects of cache interference.

2.2.2 Cache Management Schemes

Cache management schemes for real-time systems can be classified in cache locking and cache partitioning approaches [2]. With cache locking, a portion of the cache is locked, so that its content cannot be evicted. Cache locking relies on hardware-specific mechanisms and it is effective against intra-task, intra-core and inter-core interference.

On the other hand, with cache partitioning, a portion of the cache is assigned to a given task. Cache partitioning schemes can be classified as follows:

- **Index-based or way-based**: in index-based partitioning mechanisms, the cache is partitioned by sets; in the latter, the cache is partitioned by ways. While the first method generally provides a larger number of partitions, way-based partitioning mechanisms avoid contention on cache ways.

- **Hardware-based or software-based**: hardware-based schemes usually require special hardware support, not available on the majority of commercial off-the-shelf systems. An example is Intel’s Cache Allocation Technology (CAT) [39], which requires that the system be equipped with one of the most recent Intel processors. Software-based mechanisms generally do not require special hardware support.

- **Compiler-based or OS-based**: this is a sub-categorization of software-based cache partitioning techniques. With compiler-based mechanisms, support for cache partitioning is added at the compiler level; OS-based schemes require that the support be introduced at the OS memory allocator level.
In this work, we focus on page coloring, a software index-based cache partitioning mechanism. In page coloring, caches are partitioned at page granularity by manipulating the virtual-to-physical page translation mechanism at the OS level [26]. The following section describes page coloring in more details.

Page Coloring

Figure 2.6 shows how a task’s memory address is mapped to a cache entry. The $g$ least significant bits are used as an offset into a page (of size $2^g$), while the remaining bits, representing the number of a virtual page, are translated into a physical page number.

Let us suppose the cache is physically indexed and physically tagged: that is, the cache location used to store the data block is identified by its physical address. Assuming a line size of $2^l$ bytes and $2^s$ cache sets, the $l$ least significant bits of the physical address represent the offset within a cache line; similarly, the preceding $s$ bits identify the set index. The overlapping intersection bits between the physical page number and the set index determine the color index.

Page coloring divides the cache into $2^{(s+l-g)}$ partitions by making use of the color index: addresses with the same color index contend for allocation in the same cache.
partition. This implies that the entire physical memory is co-divided into $2^{(s+l-g)}$ partitions.

The total number of colors available in the system, which equals to the number of cache partitions (and, thus, memory partitions), can be determined as follows [26]:

$$n_{\text{colors}} = \frac{\text{size}_{\text{LLC}}/\text{assoc}_{\text{LLC}}}{\text{size}_{\text{page}}}$$  \hspace{1cm} (2.1)

The $\text{size}_{\text{LLC}}$ parameter represents the size of the last-level cache, while $\text{assoc}_{\text{LLC}}$ represents its associativity. Parameter $\text{size}_{\text{page}}$ is the size of a page in the system. For instance, the number of colors available in a system with page size of 4 KB and a 2-MB shared 16-way set-associative cache is 32, and the color index is of 5 bits ($2^5 = 32$). With page coloring, we assign color 0 to page 0, color 1 to page 1, and so on, until page 31, which is assigned to color 31. Because only 32 colors are available in the system, page 32 is again assigned to color 0, page 33 to color 1, and so on.

Different colors can be assigned to different tasks or cores. For example, by assigning color 0 to a given task, we can make sure that the task will only request pages of color 0. By controlling the allocations made by a process, it is possible to predict where its data will be mapped in cache. Two tasks requesting pages of different colors will not contend for the same cache sets and they will not evict each other’s content in cache. Thus, the tasks are isolated in cache and do not interfere with each other: the effects of inter-core and intra-core cache interference can then be eliminated.

Page coloring can be implemented at the compiler level or at the OS level, and it only requires the presence of a set-associative cache. Thus, it is supported by most modern multi-core processors.
3 MEMORY ALLOCATION: A CLOSER LOOK

In Section 2.1.3, we discussed the main allocators of dynamic memory in the kernel. We saw that both the physical page allocator and the slab allocator return chunks of physically-contiguous memory, while the use of the kernel virtually-contiguous allocator is generally discouraged.

The Linux kernel prefers managing physically-contiguous blocks of memory rather than chunks that are virtually contiguous but scattered in physical memory. There are two reasons why physically-contiguous memory is preferable: first, devices that access memory through DMA require contiguous physical addresses to allocate large buffers, as they are oblivious to the paging mechanism. Moreover, managing physically-contiguous memory, as an alternative to mapping noncontiguous physical addresses to contiguous virtual ones, avoids the necessity to change the kernel page tables. Frequent requests would cause the translation lookaside buffers (i.e., caches for translations from virtual addresses to physical addresses) to be flushed too often, which would inevitably lead to an increase of average memory access times.

In this chapter, we conduct a preliminary study on the page allocation mechanism. We try to understand if, after system boot and with the exception of DMA requests, processes actually require contiguous allocation of physical pages. What would happen if memory requests of more than one page were contiguous in virtual memory, but not necessarily so in physical memory?

The importance of the issue in our work is motivated by the fact that, by design, pages of a given color cannot be contiguous in physical memory. For example, in a
system with two colors, pages having the least significant bit (LSB) of their page frame number (PFN) set to zero belong to the first color, while pages with PFN having LSB set to one belong to the second color. In light of this, the above question becomes: is it possible to allocate only pages of a single color, which are noncontiguous in physical memory, without conflicting with any implicit requirement in Linux?

To gain a practical understanding of how memory allocation works in Linux, we made use of internal function tracer Ftrace [40], as described next.

3.1 Tracing the Allocations

Recall from Section 2.1.3 that a request’s order describes the size of a block of contiguous physical page frames: an order-\(x\) allocation corresponds to an allocation of \(2^x\) page frames; for example, if the order is zero, only one (\(2^0\)) page is allocated.

The first step in our work is to observe the order of memory requests made by processes, and understand the reason why processes request larger-order allocations, if any.

The function \texttt{alloc\_pages\_nodemask()} is the heart of the buddy allocator. By tracing the \texttt{kmem:mm\_page\_alloc} event, Ftrace provides information on calls to \texttt{alloc\_pages\_nodemask()} such as the order of the allocation request, the name of the process requesting it, and, if the request was successful, the physical address of the first allocated page.

To enable Ftrace, it is first necessary to mount the debug filesystem \texttt{debugfs} (typically on the \texttt{/sys/kernel/debug/} directory). The following command is used to start tracing event \texttt{kmem:mm\_page\_alloc}:

\begin{verbatim}
echo kmem:mm_page_alloc >> /sys/kernel/debug/tracing/set_event.
\end{verbatim}

The tracing starts after writing 1 to the \texttt{tracing\_on} file in the \texttt{tracing} directory. An example of output of the tracer is shown in Table 3.1.
In the first two lines of Table 3.1, we can observe that the cat process (PID: 25028), currently running on CPU 0 ([000]), made an order-zero request, and that it has been allocated the physical page at address ffffea0004178680. Similar information can be deduced for the following allocations made by the bash process.

In our tests, we observe that, while most of the allocations are of order zero, a small number of them are of larger orders. Examples of processes that make larger requests include standard Linux processes, such as the previously mentioned bash, and grep, as well as kernel processes, such as irq/33-eth0. Since both user-level and kernel processes are involved, we can conclude that the order of the allocations is decided automatically in the kernel, and not by higher-level allocators.

It is now important to understand the reason of larger-order requests, and if it is essential that the allocations be contiguous in physical memory.

### 3.2 Determining the Allocation Order

In this section, we describe in more details how the kernel determines the order of a memory request.

By combining the use of the kmem:mm_page_alloc event tracer as explained above, and the function_graph tracer, it is possible to trace all function calls (within a given depth) that result in a memory allocation request. The function_graph tracer simply
probes a function on its entry and on its exit [40], building a call stack. The following command changes the tracer to function_graph:

```
    echo function_graph >> /sys/kernel/debug/tracing/current_tracer.
```

Table 3.2 shows an example of output for an order-one allocation.

<table>
<thead>
<tr>
<th>Function</th>
<th>Time</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>kmem_cache_alloc()</td>
<td>3.739 us</td>
<td>/* mm_page_alloc: page=ffffe0000c40680 pfn=200730 order=1 migratetype=2 */</td>
</tr>
<tr>
<td>_slab_alloc.isra.74()</td>
<td>5.504 us</td>
<td>/* alloc_pages_current */</td>
</tr>
<tr>
<td>_slab_alloc()</td>
<td>0.054 us</td>
<td>mod_zone_page_state();</td>
</tr>
<tr>
<td>new_slab()</td>
<td>9.592 us</td>
<td>/* _slab_alloc */</td>
</tr>
<tr>
<td>alloc_pages_current()</td>
<td>0.045 us</td>
<td>preempt_count_sub();</td>
</tr>
<tr>
<td>get_pages_current()</td>
<td>0.056 us</td>
<td>/* _slab_alloc.isra.74 */</td>
</tr>
<tr>
<td>allocate_slab()</td>
<td>10.499 us</td>
<td>/* kmem_cache_alloc */</td>
</tr>
</tbody>
</table>

Table 3.2: function_graph output.

Let us take a closer look at the call stack. kmem_cache_alloc(), defined in file mm/slub.c of the kernel, allocates a new object from the given cache (Section 2.1.3). If all the slabs in the cache are full, kmem_cache_alloc() allocates a new slab by calling new_slab(). The memory page request is executed by function allocate_slab() (not included in the call stack for brevity), which interfaces the slab allocator with the buddy allocator.

The order of the allocation is calculated by calculate_order() [41] starting from the size of the slab object to allocate. The preferred order is zero, as single-page allocation reduces the risk of fragmentation. However, if the new object is too large, an order-zero allocation would result in a waste of memory in the slab, as the remaining space would
be too small to accommodate other objects (the allocated page frames are never released, unless the kernel is looking for additional free memory [32]). `calculate_order()` ensures that a minimum number of objects is contained in a single slab by assigning the request a higher order if more than \( \frac{1}{16} \) of the slab would be wasted. This also leads to a reduction of overhead, which would otherwise be generated by the more frequent requests to the page allocator.

When allocating a new slab, `allocate_slab()` first attempts to make an allocation of order previously calculated by `calculate_order()`, letting it fail if the system is under memory pressure (i.e., memory shortage [42]). In such case, the function falls back to the minimum order allocation, solely determined by the size of the slab with no further optimization. An example of this behavior, captured by the function graph tracer, is shown in Table 3.3.

```
1) cat-15426 | | /* mm_page_alloc: page= (null) pfn=0 order=1 migratetype=2
gfp_flags=_GFP_IO|_GFP_NOWARN|_GFP_NORETRY|_GFP_COMP|_GFP_NOMEMALLOC|
|_GFP_RECLAIMABLE|_GFP_NOTRACK */
1) cat-15426 | | /* mm_page_alloc: page=ffffea00034379c0 pfn=855527 order=0 migratetype=2
gfp_flags=GFP_NOFS|_GFP_COMP|_GFP_RECLAIMABLE|_GFP_NOTRACK */
```

Table 3.3: A failed order-one allocation falls back to order zero.

The `cat` process first makes an order-one request with the `GFP_NORETRY` flag. With this flag, the `__alloc_pages_nodemask()` function attempts to satisfy the request only once. Since the allocation did not return a page (`page= (null)`), possibly due to memory fragmentation, the system makes a second attempt of minimum order (in this case, zero), successfully completing the allocation. Note that, in the general case where the object size exceeds the page size, the slab allocator will make an allocation of order larger than zero, even after falling back to the minimum order allocation.

The inspection of file `/proc/slabinfo` provides further insights into slab caches. From this file, it is possible to learn information such as the size of each slab object,
how many objects are per slab, and how many pages are per slab. For example, in Linux 4.9.40 PREEMPT-RT, the kernel allocates 5696 bytes for each `task_struct` slab, and each slab contains 5 objects, for a total of 28,475 bytes per slab. This means that, in a system with physical pages sized 4 KB, at least 7 pages are needed for each `task_struct` slab. When the kernel allocates a new `task_struct` slab, we should expect an order-3 request (that is, a request of 8 pages) to the buddy allocator, as the size is rounded up to the nearest power of 2.

**Summary.** We conclude that both kernel and user-space processes make requests of memory chunks that are larger than a page, even after boot. Such requests are served by the buddy allocator with physically-contiguous blocks.

Many allocation requests, such as the ones reported in the example, involve the SLUB allocator. The order of an allocation request is generally decided before calling the buddy allocator. In the case of the SLUB allocator, the order depends on the size of the object and on several optimization parameters.

The approach adopted to circumvent the issue of contiguity in physical memory is presented in the next chapter, along with the implementation details of the proposed solution.
4 DESIGN AND IMPLEMENTATION

This chapter is divided into three sections. Section 4.1 illustrates how colors are assigned to a process. Single-page and multiple-page allocations have been treated as two different problems and are discussed in Sections 4.2 and 4.3, respectively. The motivation behind this choice is described in Section 4.3.

4.1 Assigning Colors to a Process

This section describes how a color (or a set of colors) is allocated to a process, and how to determine the number of colors present in a system, as well as the number of colors to assign to a process. Section 4.1.1 provides an overview of the main concepts and mechanisms, while Sections 4.1.2, 4.1.3, and 4.1.4 describe the implementation details.

4.1.1 Overview

Recall from Section 2.2.2 that the color of a page determines which set in the last-level cache (LLC) maps the page. It is then possible to predict where the content of a process will be mapped in cache if the process only makes requests for pages of a given color (or a given set of colors). In this implementation, we assign different sets of colors to different categories of processes, so as to avoid cache interference among the categories. Each process in the system belongs to exactly one category.

The implemented categories are the following:
• **Kernel**: processes that belong to this category are kernel processes.

• **Reserved**: ideally dedicated to real-time applications that will benefit from performance isolation. Only user-space processes belong to this category.

• **Unreserved**: this category includes all the user-space processes that do not belong to the Reserved category.

A different number of colors can be assigned to each category. Furthermore, tasks in the Reserved category can be optionally isolated from one another.

In order to determine the maximum number of colors allowed in the system (that is, the maximum number of partitions into which the LLC can be subdivided), three parameters are needed: the size of a page in the system, the size of the LLC, and the associativity of the LLC. The maximum number of colors (MAX_COLORS) in the system is, then, calculated with Equation 2.1:

\[
\text{MAX COLORS} = \frac{\text{size}_{\text{LLC}}/\text{assoc}_{\text{LLC}}}{\text{size}_{\text{page}}}.
\]

The LLC can be optionally split into a number of partitions smaller than MAX_COLORS. Let us suppose that MAX_COLORS equals to 4: that is, the LLC and the main memory can support a maximum of four partitions. The color index is 2 bits long. The example is illustrated in Figure 4.1a. We can distinguish the four partitions in main memory by looking at the two LSB of the PFN: physical pages with PFN having the two LSB equal to 00 belong to color 0, while 01 belongs to color 1, 10 belongs to color 2, and 11 belongs to color 3. However, we can consider a smaller number of partitions and group colors with the same LSB together: that is, 00 and 10 can be grouped together, as well as 01 and 11. This corresponds to splitting the LLC and the main memory into two partitions instead of four, as illustrated in Figure 4.1b: physical pages with PFN having
Figure 4.1: In this example, we divide main memory into partitions with page coloring. Figure (a) illustrates how pages are arranged when memory is partitioned in four colors, while Figure (b) shows the case of two colors.

LSB equal to 0 belong to color 0, while those with PFN having LSB equal to 1 belong to color 1.

We introduce a new parameter called {	extsc{num\_colors}}, which represents the number of partitions in which the LLC is actually split. As explained above, {	extsc{num\_colors}} can differ from {	extsc{max\_colors}} and the following relation holds:

\[ 1 \leq \text{num\_colors} \leq \text{max\_colors} \]

In the following, we discuss the implementation details of what has been presented in this section. Specifically, Section 4.1.2 starts with how to decide the category of a process, followed by Section 4.1.3, which describes how to determine the number of colors of each category. Finally, Section 4.1.4 shows how to assign colors to each
4.1.2 Categories: Implementation

In the previous section, we explained that, in this implementation, kernel processes belong to the Kernel category. Kernel processes are identified by the PF_KTHREAD flag in their process descriptor. Processes that do not have the PF_KTHREAD flag set in their descriptor belong to user-space, and they can only be assigned to either the Reserved or the Unreserved category.

The Reserved category is implemented as a cgroup subsystem [43], which allows a hierarchical organization of processes. The cgroup mechanism provides an API that enables programmers to define new kernel subsystems, and a filesystem for control purposes.

After mounting the cgroup filesystem, usually on the /sys/fs/cgroup folder, a process is assigned to a cgroup by writing the process' PID to the cgroup.procs file in the cgroup folder. The information about the cgroup to which the process belongs is embedded in the process descriptor.

In this implementation, the cgroup filesystem contains the page_coloring directory, representing the cgroup of the same name. In the following example, we assume page_coloring contains the rt1 subdirectory. By issuing the following command, all processes invoked from the current shell will belong to the Reserved cgroup:

```
    echo $$ >> /sys/fs/cgroup/page_coloring/rt1/cgroup.procs
```

The page_coloring cgroup can contain multiple subdirectories. This is especially useful if we want to assign a different set of colors to each subdirectory. This way, we can isolate the processes from one another within the Reserved category, by assigning different colors to each subdirectory. The procedure used to assign colors to a subdirectory is described in Section 4.1.4.
Finally, user-space processes that are not assigned to the Reserved category are assumed to belong to the Unreserved category.

4.1.3 Number of Colors: Implementation

The \texttt{NUM\_COLORS} parameter must be decided by the user following the rules described in the first part of the chapter. To initialize \texttt{NUM\_COLORS}, and to set the number of colors that must be allocated to each category, three kernel boot parameters are defined: \texttt{ncolors}, \texttt{kernel\_ncolors}, and \texttt{reserved\_ncolors}. The parameters respectively determine: \texttt{NUM\_COLORS}, the number of colors to assign to the Kernel category, and the number of colors to assign to the Reserved category. The following relations hold true:

\[
1 \leq \text{kernel\_ncolors} \leq \text{NUM\_COLORS}
\]

\[
1 \leq \text{reserved\_ncolors} \leq (\text{NUM\_COLORS} - \text{kernel\_ncolors})
\]

The number of colors dedicated to the Unreserved category is then calculated as:

\[
\text{unreserved\_ncolors} = \text{NUM\_COLORS} - \text{kernel\_ncolors} - \text{reserved\_ncolors}
\]

4.1.4 Assigning Colors to a Category: Implementation

In order to denote which colors are allocated to each category, we define three \textit{bitmaps} that are updated when the kernel boot parameters are parsed. Bit \(i\) of a bitmap is set to 1 if and only if color \(i\) is allocated to the associated category; 0 otherwise. For example, if color 9 is assigned to the Reserved category, then the bit at position 9 in the associated bitmap is set to 1.

Figure 4.2 shows how the bitmaps are organized. All the bits of the bitmap associated to the Kernel category (\textit{Kernel bitmap}) are set to zero, except for the bits from 0 to...
Figure 4.2: Illustration of the set of three bitmaps defined to keep track of the color-to-category mapping: $\text{kernel\textunderscore cmap}$, $\text{reserved\textunderscore cmap}$, and $\text{user\textunderscore cmap}$.

$\text{kernel\textunderscore ncolors} - 1$. The bits in the $\text{Reserved bitmap}$ are set to one starting from position $\text{NUM\textunderscore COLORS} - \text{reserved\textunderscore ncolors}$, until $\text{NUM\textunderscore COLORS} - 1$. Finally, the colors that are not mapped to the Kernel and Reserved categories are assigned to the Unreserved category, and the associated bitmap ($\text{User bitmap}$) is updated accordingly.

As mentioned in Section 4.1.2, processes belonging to the Reserved category can be optionally isolated from one another by creating multiple subdirectories in the $\text{page\textunderscore coloring}$ cgroup filesystem. In the following example, we create subdirectories $\text{rt1}$ and $\text{rt2}$. Then we assign colors 12, 13, and 14 to the processes belonging to the first subdirectory by writing to the $\text{page\textunderscore coloring\textunderscore colors}$ file in $\text{rt1}$. Similarly, we assign color 15 to the processes belonging to $\text{rt2}$.

```bash
mkdir /sys/fs/cgroup/page\textunderscore coloring/rt1
mkdir /sys/fs/cgroup/page\textunderscore coloring/rt2
echo 12-14 >> /sys/fs/cgroup/page\textunderscore coloring/rt1/page\textunderscore coloring.colors
echo 15 >> /sys/fs/cgroup/page\textunderscore coloring/rt2/page\textunderscore coloring.colors
```

Note that writing to the $\text{page\textunderscore coloring\textunderscore colors}$ file is allowed only for colors abiding to the rules explained above. Thus, the previous example is valid only under
the assumption that the bits from 12 to 15 are set to 1 in the Reserved bitmap.

This section described how to assign colors to a given process. In order to implement page coloring, we must make sure that processes belonging to a given category are only allowed to request pages of the color(s) assigned to their category. The next section shows how to implement page coloring for single-page allocations. Specifically, the changes are applied to the lowest level of dynamic memory allocation in the Linux kernel: the buddy allocator (Section 2.1.3).

4.2 Single-Page Allocation

As described in Section 2.1.3, the buddy allocator maintains in each zone a list, called freelist, of blocks of free physically-contiguous pages for each order. This means that each zone stores an array of list heads for each order. A simplified view of the data structure is presented in Figure 2.2. For each order $x$, all elements of the $x$-order list are blocks of $2^x$ physically-contiguous pages. In the Linux kernel, the default value of \textsc{MAX\_ORDER} is 11: thus, the largest blocks are of $2^{11}$ pages.

Section 4.2.1 discusses the implementation details of the buddy allocator, with emphasis on function \texttt{rmqueue\_smallest()}, which selects and returns a block of pages from the freelists. In order to implement page coloring, the main modifications to the buddy allocator will be applied to the \texttt{rmqueue\_smallest()} function: the details are presented in Section 4.2.2.

Please note that, in alignment with the terminology used in the Linux kernel code, starting from this section the term \textit{page} denotes both a single page and a block of physically-contiguous pages of any order.
4.2.1 Buddy Allocator: Implementation Details

This section provides an overview of the main implementation details of the buddy allocator.

Section 2.1.3 explains that any process can request pages from the buddy allocator by calling the `alloc_pages()` function, specifying the order and the GFP flags to pass down to the allocator. Figure 4.3 shows a simplified version of the `alloc_pages()` call graph, down to `__rmqueue_smallest()`, which performs the removal of the page to allocate from the freelists. Note that the call graph only contains the essential functions in the buddy allocator. Omitted calls are marked in the figure with symbol `[…]`.

The `__alloc_pages_nodemask()` function first chooses a memory zone as the preferred zone from which to allocate, and attempts the allocation by calling function `get_page_from_freelist()`; if it does not succeed, it performs some sanity checks before trying again.

Typically, the preferred zone is the first memory zone in the zone list: thus, the preferred zone corresponds to the memory zone containing pages with the lowest
Algorithm 1 \(_\text{rmqueue\_smallest}(\text{zone}, \text{order})\)

1: \(\text{current\_order} \leftarrow \text{order}\)
2: \(\textbf{while} \ \text{current\_order} < \text{MAX\_ORDER} \ \textbf{do}\)
3: \(\ \ \ \text{freelist} \leftarrow \text{zone.freelist}[\text{current\_order}]\)
4: \(\ \ \ \text{page} \leftarrow \text{FirstEntryOrNull}(\text{freelist})\)
5: \(\ \ \ \textbf{if} \ \text{page} \ \text{not found} \ \textbf{then}\)
6: \(\ \ \ \ \ \ \ \text{current\_order} \leftarrow \text{current\_order} + 1\)
7: \(\ \ \ \ \ \ \ \textbf{continue}\)
8: \(\ \ \ \ \textbf{end if}\)
9: \(\ \ \ \text{RemoveEntry}(\text{page}, \text{freelist})\)
10: \(\ \ \ \textbf{if} \ \text{current\_order} == \text{order} \ \textbf{then}\)
11: \(\ \ \ \ \ \ \ \text{return} \ \text{page}\)
12: \(\ \ \ \ \textbf{else}\)
13: \(\ \ \ \ \ \ \ \text{return} \ \text{Expand}(\text{page}, \text{current\_order}, \text{order})\)
14: \(\ \ \ \ \textbf{end if}\)
15: \(\ \ \ \textbf{end while}\)
16: \(\ \ \ \textbf{return} \ NULL\)

addresses in the system. However, if the GFP flags contain zone modifiers (see the example in Section 2.1.3), the preferred zone will be chosen accordingly. For instance, in presence of the GFP\_DMA flag, ZONE\_DMA will be picked as the preferred zone.

The \text{get\_page\_from\_freelist()} function goes through the zone list, starting from the previously chosen preferred zone. For each zone, the function checks if there is enough free memory for the allocation, stopping either when it finds a page of the requested size to return, or if it does not succeed in finding a page of suitable size in any memory zone. In this case, it returns NULL. The search in the freelists for a page of suitable size is performed by function \_\text{rmqueue\_smallest}().

Finally, \_\text{rmqueue\_smallest}() is at the heart of the buddy allocator. The function goes through the freelists of a zone, trying to find the smallest available page. Then, the function removes the page from the freelists and returns it to the caller. Algorithm 1 shows the simplified structure of \_\text{rmqueue\_smallest}(). Particular attention is given to this function, as it will be modified to add page coloring support to the buddy allocator.
Function \texttt{rmqueue\_smallest()} loops through the orders, starting from the order passed to it as an argument. For each order, the function performs the following steps:

- It attempts to find a page in the freelist of the given order. \texttt{FirstEntryOrNull} simply returns the first entry of the freelist, or \texttt{NULL} if the freelist is empty. If no page is found in the freelist and if the current order is smaller than \texttt{MAX\_ORDER}, the function checks the freelist of the next larger order.

- After finding a suitable page, it removes the page from the freelist. The function returns the page to the caller if it is of the requested order. Otherwise, the page is of a larger order. In this case, the function splits the page in smaller blocks with \texttt{Expand}, as shown in Figure 2.3: the \texttt{Expand} function stops when the pages obtained after splitting a larger page are of the requested order. Then, one of the obtained pages is returned to the caller, while the other pages are reinserted into the freelists according to their order.

- If no page is found in any of the freelists checked by the function, then there is not enough memory in the chosen zone. Therefore, the function returns \texttt{NULL}.

As mentioned above, \texttt{get\_page\_from\_freelist()} calls \texttt{rmqueue\_smallest()} on each memory zone until a page of suitable size is found, or if there is not enough memory in any zone (that is, if \texttt{rmqueue\_smallest()} always returns \texttt{NULL}).

We want to modify the \texttt{rmqueue\_smallest()} function to add page coloring support to the buddy allocator. Namely, the function must be able to choose the color of the page to allocate, depending on the category (see Section 4.1) of the requesting process.

In the next section, we will show the modifications to \texttt{rmqueue\_smallest()} that allow the buddy allocator to support page coloring.

Ideally, we want to modify the buddy allocator so that any process requesting memory blocks of any size will receive pages of the assigned colors. In Section 2.2.2,
we showed that, by design, pages of the same color cannot be contiguous in physical memory. However, as explained in Section 2.1.3, and as we substantiated in Chapter 3 with the aid of an internal tracing tool, the main kernel dynamic memory allocators manage physically-contiguous blocks of pages.

In the Linux kernel, there exists a large number of functions interfacing with the buddy allocator. When making requests of order larger than zero, the callers of the buddy allocator expect to receive a physically-contiguous block of pages. If page coloring is implemented for all orders at the buddy allocator level, all callers must be aware of the fact that they will receive pages that are not contiguous in physical memory when they make requests of order larger than zero.

It is impractical (and, occasionally, unfeasible, as is the case of DMA) to modify all the call sites to support physically-noncontiguous page allocation. Therefore, the buddy allocator only supports colored order-zero allocation, and it is used as a building block for the colored multiple-page allocator. The issue will be explored in more details in Section 4.3.
4.2.2 Colored Order-Zero Allocator

This section shows the main modifications to the buddy allocator to achieve colored order-zero allocation. First, we introduce the new data structures used in the implementation; then, we discuss the changes to function \texttt{rmqueue\_smallest()}.

Two new fields are defined in the zone descriptor:

- An array of \textit{colored freelists}. As with the original freelists, which group free pages by their order, colored freelists are defined to group free order-zero pages by their color. There exists one colored freelist per color. Figure 4.4 illustrates the new colored freelists.

- A \textit{zone color bitmap} that is updated according to the colors available in the colored freelists of the zone: if the freelist associated to a given color is empty, then the bit corresponding to that color is set to 0; otherwise, it is set to 1. For example, if bit 4 is set to 1 in the zone color bitmap, then the colored freelist for color 4 is not empty. Figure 4.5 illustrates the new bitmap.

Function \texttt{\_\_rmqueue\_smallest()} is, then, modified to implement the new colored order-zero allocator. In order to avoid confusion, in what follows we refer to the “colored” \texttt{\_\_rmqueue\_smallest()} as \texttt{\_\_colored\_rmqueue\_smallest()}. Similarly, we always refer to the new freelists as \textit{colored freelists}, while the buddy allocator freelists are called \textit{original freelists}. Furthermore, the term \textit{order-i freelist} refers to the original freelist of order \(i\), while the term \textit{color-i freelist} refers to the colored freelist for color \(i\).

Function \texttt{\_\_colored\_rmqueue\_smallest()} first determines the color of the page to allocate, based on the process category and on the availability of the colored freelists. The function returns a page from the colored freelist corresponding to the chosen color. If the colored freelists for the set of colors associated to the process category are all empty, \texttt{\_\_colored\_rmqueue\_smallest()} tries to repopulate them with pages from the
Algorithm 2  
\texttt{colored\_rmqueue\_smallest(zone, order)}

1: if order > 0 then
2: \hspace{1em} return \texttt{\_rmqueue\_smallest(zone, order)}
3: end if
4: (cmap, cmap\_rr) \leftarrow \texttt{GetColorBitmap(current\_process)}
5: clists \leftarrow zone.color\_freelists
6: color \leftarrow \texttt{SelectFreeColor(zone.cmap, cmap, cmap\_rr)}
7: if color found then
8: \hspace{1em} return \texttt{GetPageFromCFreelist(clists[color], zone.cmap)}
9: end if
10: current\_order \leftarrow 0
11: while current\_order < \texttt{MAX\_ORDER} do
12: \hspace{1em} for all pages in zone.freelists[current\_order] do
13: \hspace{2em} \texttt{InsertToCFreelists(clists, page, current\_order)}
14: \hspace{2em} color \leftarrow \texttt{SelectFreeColor(zone.cmap, cmap, cmap\_rr)}
15: \hspace{2em} if color found then
16: \hspace{3em} return \texttt{GetPageFromCFreelist(clists[color], zone.cmap)}
17: \hspace{2em} end if
18: \hspace{1em} end for
19: \hspace{1em} current\_order \leftarrow current\_order + 1
20: end while
21: return \texttt{NULL}

original freelists. Next, the colored freelists of the process colors are checked again and
the function returns a page from one of the newly-updated colored freelists. If there is
no page of the given color in the zone, the function returns \texttt{NULL}.

Algorithm 2 outlines the structure of \texttt{\_colored\_rmqueue\_smallest()}. A similar
scheme was first seen in PALLOC [27]. In the following, we break down the pseudo
code of the function to discuss each step in more details.

- **Lines 1-3.** First, \texttt{\_colored\_rmqueue\_smallest()} checks if the order of the re-
quest is greater than zero. In this case, it redirects the request to the original
buddy allocator. Since we only support colored order-zero allocation at the buddy
allocator level, requests of larger orders are redirected to the original, unmodified,
allocator. If the order is zero, the request is served by the colored allocator.
The function determines which color the process will allocate from. This takes place in two steps:

- **GetColorBitmap** finds the set of colors assigned to the process. It determines the color bitmap associated to the process \((cmap)\), depending on the process category, with the procedure previously described in Section 4.1.2. Note that there is an additional color bitmap that is returned by the function, \(cmap_{rr}\). Its use is explained later in this chapter.

- **SelectFreeColor** determines the availability of the requested colors in the zone by checking the zone color bitmap \((zone.cmap)\). If there is at least one page in the colored freelist associated to one of the requested colors, then **SelectFreeColor** returns the index of that colored freelist (i.e., it returns the color index from which the function will allocate). Otherwise, the colored freelists in the zone do not contain pages of the requested colors. The algorithm behind **SelectFreeColor** is described in more details below.

**Lines 7-8.** If **SelectFreeColor** returned a color index, then **GetPageFromCFreelist** is invoked. The first page from the associated colored freelist \((clists[color])\) is removed and returned to the caller. **GetPageFromCFreelist** also updates the zone color bitmap if the colored freelist is now empty, setting its corresponding bit to 0. If **SelectFreeColor** could not find any color, then we proceed to the next step.

**Lines 10-20.** The function accesses Line 10 when the search for pages of the requested colors failed in the colored freelists. This does not necessarily mean that there are no free pages in the zone of the requested colors. The function tries to populate the colored freelists by taking pages from the original freelists with **InsertToCFreelists**. For each order, the pages of the original freelists are moved to the colored freelists one by one, stopping when it is possible to serve the request from the colored freelists (**SelectFreeColor** and **GetPageFromCFreelist** are invoked
Figure 4.6: Example of colored order-zero allocation: initial situation. Figure (a) illustrates the colored freelists, Figure (b) shows the original order-zero and order-1 freelists, and Figure (c) shows the zone color bitmap associated to Figure (a).

again). Larger-order blocks are split into order-zero pages and distributed among the colored freelists according to their colors.

• **Line 21.** If the search for pages in the original freelists was unfruitful, it can be concluded that there are no available pages of the requested colors (in the current zone). Therefore, the function returns NULL.

**Examples**

Consider the example illustrated in Figure 4.6. Let us suppose process A, belonging to the Reserved category, requests an order-zero page to the colored buddy allocator. Let us also suppose that the Reserved category is assigned colors 5, 6, and 7. Figure 4.6a shows that the color-5 freelist contains two pages, while the colored freelists for colors 6 and 7 are empty. Thus, the bit at position 5 in the zone color bitmap is set to 1, while bits 6 and 7 are set to 0. This is illustrated in Figure 4.6c.

When process A makes a request, *GetColorBitmap* returns the Reserved color bitmap. Then, *SelectFreeColor* selects color 5, since it is the only color assigned to the
Reserved category from which it is possible to allocate. GetPageFromCFreelist removes the first page from the color-5 freelist and returns it to the caller. Because the color-5 freelist is not empty, the zone color bitmap is not updated.

Let us now suppose that the Reserved category is only assigned colors 6 and 7. When process A requests a page, SelectFreeColor cannot return any color: the colored freelists for colors 6 and 7 are empty. The colored freelists must be repopulated with pages from the original freelists. Let us assume Figure 4.6b represents the content of the original freelists of order 0 and order 1: in the order-zero freelist, we find pages of colors 1, 3, and 7; similarly, in the order-1 freelist, we find an order-1 page composed by two pages of colors 6 and 7.

First, the pages from the order-zero freelist are inserted into the colored freelists. Next, SelectFreeColor is invoked again. Because the color-7 freelist is no longer empty, SelectFreeColor returns color 7. Finally, GetPageFromCFreelist allocates the page from the color-7 freelist.

In the previous examples, SelectFreeColor returned the only color from which the process could allocate. Suppose that both color-6 and color-7 freelists are not empty: in this case, the color of the page to allocate is selected in a round-robin fashion. To implement the round-robin algorithm, an additional bitmap is defined: cmap_{rr} (rr stands for round-robin). The solution is discussed next.

Color Selection

This section presents the algorithm behind the color selection in SelectFreeColor. First, we introduce the additional data structure used to enforce round-robin. Then, we describe the main characteristics of the implementation.

The cmap_{rr} bitmap enforces the round-robin algorithm for the Reserved category in a way that, ideally, processes belonging to this category are allocated an equal number of pages for each color. As with the other bitmaps in this work, bit $i$ denotes color $i$. If
its value is set to 1, then color \( i \) can be allocated. After the allocation, bit \( i \) is set to 0. Bit \( i \) will be again set to 1 in two scenarios: (i) after all the other possible colors have been allocated once, or (ii) if the remaining colors cannot be allocated. This is better explained with an example.

Let us assume that the Reserved category is assigned colors 5, 6, and 7. If the first allocation is of color 5, then the second is of colors 6, the third of color 7, the fourth of color 5 again, and so on. The \( cmap_{rr} \) is configured as seen in Figure 4.7: bits 5, 6, and 7 are set to 1, while the others are set to 0. Bitmap \( cmap_{rr} \) is currently equivalent to the Reserved color bitmap (\( cmap \)).

The following example illustrates scenario (i) as described above. Consider process A belonging to the Reserved category (which is assigned colors 5, 6, and 7) that makes three consecutive requests of order zero. Let us suppose that more than one page of all three colors are available in the colored freelists of the current memory zone. After the first allocation of color 5, bit 5 is set to 0 in \( cmap_{rr} \). Now the process can allocate pages of either color 6 or color 7. Similarly, after the next two allocations of colors 6 and 7, in this order, bit 6 and 7 are set to 0. If process A makes a fourth order-zero request, \( cmap_{rr} \) is reset to its initial state: bit 5, 6, and 7 are set to 1, making the bitmap equivalent to the Reserved color bitmap.

Scenario (i) can be summarized by the following statement: \( cmap_{rr} \) is reset to be equal to \( cmap \) when all bits in \( cmap_{rr} \) are equal to zero.
Now we explain scenario (ii). Let us suppose that the colored freelists for colors 5 and 6 contain at least two pages each, while the color-7 freelist is empty. This scenario is illustrated in Figure 4.8, which shows the values of the associated zone color bitmap (zone.cmap). The first two order-zero allocation requests by process A result in two pages of colors 5 and 6, respectively. Now all bits in cmap_rr are set to zero, except for bit 7, and the next order-zero allocation request is expected to result in a page of color 7. However, the color-7 freelist in the current memory zone is empty. We could search the original freelists for a color-7 page, but that would introduce significant overhead. Since the colored freelists of colors 5 and 6 are not empty even after the first two allocations (remember that we assumed that they contained at least two pages each), we can still allocate from one of these two lists. Therefore, cmap_rr can be reset to be equal to cmap, so that colors 5 and 6 can be considered eligible for allocation again. The next order-zero allocation will result in a color-5 page.

Scenario (ii) can be summarized with the following sentence: cmap_rr is reset to be equal to cmap when the set bits in cmap_rr and zone.cmap do not overlap.

Note that, if the set bits of cmap and zone.cmap do not overlap, the colored freelists in the memory zone do not contain pages of the requested colors. In this case, the colored allocator will attempt to repopulate the colored freelists, as explained earlier in this section.

Algorithm 3 summarizes the structure of SelectFreeColor. It receives as arguments the zone, the color bitmap (cmap), and the round-robin bitmap (cmap_rr).

The initial if-statement ensures that the colored freelists contain pages of the requested colors by checking that at least one set bit in cmap and in the zone color bitmap overlaps, as explained above.

The second if-statement covers the two scenarios described earlier in this section. The cmap_rr bitmap is set equal to cmap when: (i) the round-robin has been “completed” and it must be “reset”, and (ii) there are no more pages available of the remaining
Algorithm 3 SelectFreeColor(zone, cmap, cmap_{rr})

1: if (cmap ∩ zone.cmap) == ∅ then
2:   return NULL
3: end if
4: if cmap_{rr} is empty OR (zone.cmap ∩ cmap_{rr}) == ∅ then
5:   cmap_{rr} ← cmap
6: end if
7: cmap_{rr} ← (cmap_{rr} & zone.cmap)
8: color ← FirstSetBit(cmap_{rr})
9: return color

colors in the colored freelists.

The next step consists in setting cmap_{rr} equal to the bitwise AND operation between the zone color bitmap and itself. This way, we make sure that the color is chosen only among the colors available in the colored freelists of the zone.

The first set bit in cmap_{rr} is then selected as the color to allocate from. Finally, the chosen bit is cleared in cmap_{rr}. Then, the colored allocator returns to \_colored\_rmqueue\_smallest() and removes the first element from the designated colored freelist, as described previously.

As mentioned at the beginning of this section, cmap_{rr} is associated to the Reserved category. Therefore, in this implementation, the round-robin algorithm is only applied to the Reserved category. This is justified by the fact that our work aims at improving the isolation and predictability of real-time tasks, which, we expect, will be assigned to the Reserved category. However, the feature can be easily extended to the remaining categories by defining two more cmap_{rr} bitmaps and passing them as an argument to SelectFreeColor.

In the current implementation of Algorithm 2, getColorBitmap returns (cmap, NULL) if the process belongs to a different category than Reserved. Similarly, the cmap_{rr} argument in SelectFreeColor is left NULL. Null-checking on cmap_{rr} was omitted in
Algorithm 3 for brevity.

With round-robin, allocation requests are balanced with respect to colors within the Reserved category. As a result, tasks can distribute their content in all the assigned cache partitions, reducing the risk of tasks evicting their own content.

**Summary.** In this section, we discussed our solution of single-page allocation at the buddy level. A similar scheme was first presented in PALLOC [27].

Note that the freeing operation has not been modified. When a page is freed, it is returned to the original freelists, potentially starting the coalescing mechanism seen in Section 2.1.3. The page is reinserted into the colored freelists on demand, as described in this section.

At the beginning of this discussion, we mentioned that implementing multiple-page allocation at the buddy allocator level is impractical, as it would require changing all the call sites of larger-order requests to enable the callers to manage noncontiguous physical memory blocks. In the next section, we explain the problem in more details and we provide a solution that consists in changing the physical-to-virtual page mapping in the SLUB allocator (Section 2.1.3).

### 4.3 Multiple-Page Allocation

As discussed in the previous sections, implementing “colored” multiple-page allocation in Linux is a nontrivial problem. The main issues are summarized below:

- By design, pages of the same color cannot be contiguous in physical memory (see Section 2.2.2). However,

- the main kernel dynamic memory allocators in Linux manage physically-contiguous blocks of pages.

A solution would be to simply modify the buddy allocator so that, when a process
makes requests of order larger than zero, the allocator returns pages that are not
contiguous in physical memory. However, there exists a large number of functions
directly interfacing with the buddy allocator that need to be aware of the change.

In Linux 4.9, there are more than one hundred calls to function alloc_pages() [44],
many of them making larger-order requests to the buddy allocator. If the buddy
allocator itself is modified to support multiple-page allocation, then the callers must be
aware of the fact that the chunks of memory they get from the allocator are no longer
contiguous in physical memory.

The buddy allocator simply returns the physical address of the first allocated page.
Therefore, in order to support colored multiple-page allocation, the return value should
be modified; for example, we could implement it as a list of addresses of pages of the
same color. However, this would imply modifying all call sites accordingly.

A conceptually simple solution to this problem is to modify the physical-to-virtual
page mapping when requesting multiple pages at once. The resulting pages are contiguous
in virtual memory, but not in physical memory. However, there is no central point
in which the physical-to-virtual address mapping takes place in the buddy allocator.
Furthermore, physical addresses in the Linux kernel are often translated into virtual
addresses through a linear mapping, as seen in Section 2.1.3. After allocating a page
with the buddy allocator, its virtual address can be obtained by simply calling function
page_address().

Our solution is to go a level higher and add page coloring support to the SLUB
allocator, one of the most widely used kernel memory allocators in Linux. As seen in
Chapter 3, SLUB makes frequent multiple-page requests to the buddy allocator. The
idea is to modify the SLUB allocator so that, instead of making a single multiple-page
request to the buddy allocator, it requests as many order-zero pages as needed. The
colored order-zero allocator is, thus, used as a building block for the colored SLUB
allocator.
The idea discussed above is feasible only if we make sure that the allocated pages are contiguous in virtual memory: this cannot be achieved with linear mapping. Therefore, we need to change the physical-to-virtual translation mechanism. The proposed solution, consists in mapping the allocated pages using the page table mechanism, briefly described in Section 2.1.2. The implementation is similar to the vmalloc scheme (see Section 2.1.3).

Further details on the implementation of the original SLUB allocator are provided in Section 4.3.1. Section 4.3.2 gives an overview of the main concepts necessary to understand vmalloc. Finally, Section 4.3.3 illustrates the proposed solution to “color” the SLUB allocator.

4.3.1 SLUB : Implementation Details

We know from Section 2.1.3 that SLUB asks the buddy allocator for pages to allocate commonly-used objects, which can be smaller than a single page. The objects are kept in caches at an initialized state, so that requests to the SLUB allocator can be served faster. Furthermore, general-purpose allocator kmalloc() is built on top of the SLUB allocator.

In this section, we provide insights into the implementation details of the SLUB allocator, with emphasis on the functions that need to be modified to add page coloring support. We introduce the discussion with an overview of the data structures involved.

Data Structures

Structure kmem_cache describes a slab cache. There exists one instance of the structure for each cache in the system. A few of the most relevant structure fields are the following:

- A string representing the name of the cache.

- Two structures with, respectively, the optimal and minimum size of each object in the cache. The size is described by a kmem_cache_order_objects structure,
which also contains information about the order of the allocation requests to the buddy allocator for that particular slab.

- A pointer to a kmem_cache_node array, with as many elements as the number of nodes in the system. These structures contain the per-node lists of partially-used slabs (partial lists) introduced in Section 2.1.3.

- A pointer to the kmem_cache_cpu per-CPU variable associated to the slab cache. This structure contains the per-CPU lists introduced in Section 2.1.3. More details on this structure will be provided in this section.

- Two sets of flags: one set is specific to the SLUB allocator, while the second consists of GFP flags.

Structure kmem_cache_cpu describes the per-CPU lists associated to a given slab cache. For each cache, there exist as many per-CPU lists as the number of CPUs in the system. Per-CPU lists are the preferred way to allocate slabs, as they do not require any locking (although it is necessary to disable interruptions to avoid that the task migrate to another CPU).

The kmem_cache_cpu structure contains:

- A pointer to the next free and allocatable object, freelist. Note that the slab freelist pointer differs from the buddy allocator freelists, since the latter consist of linked lists of free physical pages. The freelist of a slab cache is a pointer to a chunk of memory whose size depends on the cache it belongs to.

- A pointer to the slab from which the structure is currently allocating objects, page. In the SLUB allocator, there is no explicit descriptor for slabs. Recall from Section 2.1.3 that slabs are simply chunks of physically-contiguous memory. Hence, this field is implemented as a pointer to a page.
• A pointer to partially-allocated slabs that are currently not in use, partial. As with the previous field, it is implemented as a pointer to a page.

The next section talks about how the allocator interacts with the newly-defined data structures. To simplify the discussion, we will describe the main functions involved.

Call Graph

Figure 4.9 shows the call graph of function `kmem_cache_alloc()`, used to allocate objects from slab caches. For brevity, the call graph has been simplified and some calls have been omitted. Omitted calls are marked in the figure with [...].

Function `slab_alloc_node()` identifies the CPU on which the process is running and tries to allocate from the related per-CPU list by returning the address contained in pointer `freelist`. Then, the freelist field is updated with `get_freepointer_safe()`,
which moves the pointer to the next free object. If the allocation does not succeed, it means that the current slab, pointed by `page`, is empty. In this situation, SLUB attempts to allocate from the per-CPU partial list by invoking `__slab_alloc()`.

Before `__slab_alloc()` tries to allocate from the per-CPU partial list, it checks the per-CPU list again for new objects that might have been freed. If that is the case, the allocator returns the new object; otherwise, it allocates from the per-CPU partial lists. If the per-CPU partial lists are empty as well, the function invokes `new_slab_objects()` to allocate from the per-node partial list.

Finally, if function `new_slab_objects()` fails at allocating from the per-node partial list, then it attempts to allocate a new slab by asking the buddy allocator for new pages. This is mainly carried out by function `allocate_slab()`, which invokes `alloc_slab_page()` for the requests to the buddy allocator. A list of free objects is loaded from the new slab with the `set_freepointer()` function. The `page` pointer is updated with the newly-allocated slab, and the `freelist` pointer with the first free object. Finally, the address contained in freelist is returned to the caller.

In the worst case, function `alloc_slab_page()` is called twice: first, the allocator attempts to allocate a slab of an optimized order (as discussed in the previous section and in Chapter 3). The allocation fails if the system is under memory pressure. In this case, we fall back to the minimum-order allocation, which depends on the size of the object, with no further optimization.

Note that, as previously mentioned, the freelist always stores a virtual address. The list of free objects built in `allocate_slab()` is, therefore, a list of virtual addresses.

General-purpose allocation with `kmalloc()` works as described in this section, with the additional steps explained in Section 2.1.3.

Freening objects previously obtained from the SLUB allocator is performed with `kmem_cache_free()`. The function detects the slab cache to which the object belongs and returns the object to the slab. The slabs are rarely returned to the buddy allocator:
this generally happens only in the case of memory shortage.

Deallocation of objects allocated with kmalloc() can be performed by kfree().

The next section gives an overview on the implementation of vmalloc. As with the SLUB allocator, we first introduce the data structures used in vmalloc, following with a brief explanation of the main functions involved.

4.3.2 Vmalloc: Implementation Details

Recall from Section 2.1.3 that vmalloc allocates memory that is virtually contiguous, but it may not be physically contiguous. The physical-to-virtual mapping is performed by updating the page tables.

Data Structures

The main data structures in vmalloc are vm_struct and vmap_area. The two structures are used to keep track of the mapping between the physical pages and the virtual address space reserved to vmalloc. Whenever vmalloc is called, both structures are allocated with kmalloc().

Structure vm_struct contains the size of the area to allocate with vmalloc (size), the address of the virtual memory area it describes (addr), the number of pages to request to the buddy allocator (nr_pages), and an array that will be filled with the pages returned by the buddy allocator (pages). A pointer to a vm_struct instance is contained in each structure of type vmap_area, which also contains the start address and the end address of the range of virtual addresses it represents (va_start and va_end, respectively).

The range of virtual addresses that can be mapped with vmalloc varies with the architecture; however, the start address is identified in the kernel by the Vmalloc_START constant, while the end address is denoted by Vmalloc_END.
Figure 4.10: Simplified call graph of vmalloc(), invoked to allocate a block of virtually-contiguous pages of the given size. Omitted functions are denoted with the symbol [...]..

Call Graph

This section gives an overview of the main functions involved in the allocation of virtually-contiguous memory with vmalloc. The allocation process starts when a process invokes the vmalloc() function, passing as arguments the desired size in bytes of the allocation. Figure 4.10 shows the simplified call graph of vmalloc(), until it interfaces with the buddy allocator. As with the previously-discussed allocators, the function calls that have been omitted are marked in the figure with symbol [...].

Function __vmalloc_node_range() determines the size in bytes of the pages needed to cover the allocation request. Then, it allocates memory for an instance of vm_struct and an instance of vmap_area. Finally, it finds a virtual address space big enough for the allocation and stores the related information in the structures, before invoking __vmalloc_area_node().

The __vmalloc_area_node() function interfaces vmalloc with the buddy allocator. A simplified scheme of the function is illustrated in Algorithm 4. The arguments shown are the virtual memory area (an instance of structure vm_struct), and the GFP flags to pass down to the buddy allocator. First, the function determines the number of pages needed, depending on the size of the allocation, and stores the information in the nr_pages of the virtual memory area. Then, it allocates memory for the pages array.
Algorithm 4 _vmalloc_area_node(area, flags)

1: area.nr_pages ← GetNrPage(area.size)
2: area.pages ← kmalloc(array_size, flags)
3: counter ← 0
4: while counter < area.nr_pages do
5:    area.pages[counter] ← alloc_page(flags)
6:    counter ← counter + 1
7: end while
8: MapVmArea(area, area.pages)
9: return area.addr

with kmalloc(). Next, the function makes nr_pages order-zero requests to the buddy allocator, storing the pages obtained in the pages array. Finally, the pages are mapped to the virtual memory area and the start address of the area is returned to the caller.

If not all nr_pages pages can be allocated (that is, in case of memory shortage), the pages that have been obtained are freed. For simplicity, this has been omitted in Algorithm 4.

Note that, as mentioned in Section 4.2.1, the alloc_page() macro always makes order-zero requests. This means that vmalloc supports page coloring in a system where the colored order-zero allocator, described in Section 4.2, is enabled.

When a process wants to free an area allocated with vmalloc, it invokes function vfree(). This function simply unmaps the physical pages from the virtual memory area, calls the buddy allocator to deallocate the pages, and deallocates the associated vmap_area and vm_struct structures by calling kfree().

Section 4.3.1 described the essential implementation details of the SLUB allocator, while this section presented an overview of the virtually-contiguous allocator, vmalloc. In the following, we show how to combine the two allocators to add page coloring support to the SLUB allocator.
4.3.3 Colored SLUB Allocator

Let us consider, as an example, a process that makes a memory allocation request with `kmalloc()`. As seen in the previous sections, `kmalloc()` is built on top of the SLUB allocator, which in turn asks the buddy allocator for pages.

The idea is to change the physical-to-virtual address translation mechanism in SLUB, so that the allocator supports multiple-page blocks of memory that are not physically contiguous. Figure 4.11 summarizes the proposed scheme: the solution is equivalent to adding a layer between the SLUB allocator and the buddy allocator. The new layer is, essentially, a simplified version of `vmalloc`: this is justified by the fact that `vmalloc` supports page coloring if the colored order-zero allocator is enabled. Therefore, we can add page coloring support to the SLUB allocator. In what follows, we will refer to this simplified version of `vmalloc` as `Page Coloring vmalloc` (PC vmalloc).
Page Coloring Vmalloc

Before discussing the implementation details of PC vmalloc, we illustrate the modifications to the page descriptor. Table 4.1 shows the three new fields.

The fields are described as follows:

- **pc_vmalloc_addr**: the PC vmalloc virtual address to which the page is mapped. More information about the address space reserved to PC vmalloc are given later in this section. This field is set to **NULL** if the page is not allocated through PC vmalloc.

- **pc.compound_head**: in the buddy allocator, the *compound head*, or *head page*, of a block of memory is the first page of the block. The concept is especially useful in the SLUB allocator, when it is necessary to translate a (linearly mapped) virtual address of an object, for example to deallocate a block of memory obtained by the buddy allocator: the compound head provides the physical address of the first page of the block, which can then be deallocated using the standard deallocation functions provided by the buddy allocator. In PC vmalloc, the concept of compound head is best described as: the physical page mapped by the first virtual page of the (virtually-contiguous) block of memory. For each page of the block, field **pc.compound_head** contains the physical address of the compound head; the first page of the block stores its own address in **pc.compound_head**.

```c
struct page {
    ...;
    void *pc_vmalloc_addr;
    struct page *pc_compound_head;
    unsigned int pc_compound_order;
    ...;
}
```

Table 4.1: PC vmalloc: New fields in the page descriptor
bool is_pc_vmalloc_addr(void *addr)
struct page *pc_vmalloc_to_page(const void *addr)
struct page *pc_vmalloc_to_head_page(const void *addr)

Table 4.2: PC vmalloc: Virtual-to-physical address translation functions

- pc_compound_order: in the buddy allocator, compound order refers to the order of a block of multiple pages. In PC vmalloc, the concept remains unchanged: the only difference is that the block of multiple pages is contiguous in virtual memory, and (most likely) not in physical memory. In the buddy allocator, the compound order of a page can be obtained with function compound_order(). In PC vmalloc, the value of this field is the same in all the pages of a block.

The main modifications to the original vmalloc provided by PC vmalloc are the following:

- Main function call. The function that interfaces the caller with PC vmalloc is pc_vmalloc(). While the original vmalloc() function only takes as an argument the size of the area to allocate, pc_vmalloc() takes one additional argument: the GFP flags to pass down to the buddy allocator. This is justified by the fact that, in the SLUB allocator, the GFP flags vary with the slab cache. We want to replace with pc_vmalloc() any call to the buddy allocator made by SLUB, with an exception: objects allocated for DMA operations, which require contiguity in physical memory. This situation is explained later in this section.

- New virtual address space. As mentioned in Section 4.3.2, vmalloc can only allocate from a given virtual address space, whose range depends on the architecture. Because PC vmalloc is a separate module from vmalloc, we defined a separate virtual address space for the new allocator. The new range is delimited by constants PC_VMALLOC_START and PC_VMALLOC_END.
• **Independence from kmalloc().** In Section 4.3.2, we specified that all the instances of `vm_struct` and `vmap_area` are dynamically allocated with `kmalloc()`. However, with PC vmalloc, we want to exploit the functionalities of the original vmalloc scheme to interface the SLUB allocator with the buddy allocator. Therefore, `kmalloc()` cannot be used to allocate memory in PC vmalloc. The adopted solution is to define a cache of statically-allocated objects. We implement the cache as two arrays of structures of type `vm_struct` and `vmap_area`, respectively. For each array, two functions are defined: one to allocate an object from the array, the second to deallocate an object. To keep track of the allocated elements of the array, two bitmaps are defined (one for each array). Bit $i$ is associated to the element of position $i$ in the related array: if the value of bit $i$ is 1, then the $i$-th element of the associated array is free; otherwise, the element is allocated. The bitmaps are updated by the allocation and deallocation functions.

• **Virtual-to-physical address translation functions.** We provide a set of new utility functions that are used to retrieve the physical page descriptor mapped to a given virtual address. The three functions are shown in Table 4.2. Function `is_pc_vmalloc_addr()` checks whether the given address belongs to the range of addresses defined for PC vmalloc. This is useful to find which functions can be used in the address translation, as explained later in this section. Function `pc_vmalloc_to_page()` takes a (PC vmalloc) virtual address and searches the page tables for the physical page descriptor mapped by it. Finally, `pc_vmalloc_to_head_page()` finds the compound head associated to a given virtual address. This is done by calling `pc_vmalloc_to_page()` to the virtual address provided; then, the compound head can be found by inspecting the `pc_compound_head` field in the page descriptor. The purpose of the function is similar to that of original Linux kernel function `virt_to_head_page()`, which takes a (linearly mapped) virtual address and finds the associated compound
We summarized the most important changes brought by PC vmalloc. In the following section, we describe how we integrate PC vmalloc with the slab allocator. Recall from the previous section that we want to obtain a scheme similar to the one illustrated in Figure 4.11.

**Integrating SLUB with Page Coloring Vmalloc**

In Section 4.3.1, we introduced function `alloc_slab_page()`, which interfaces the SLUB allocator to the buddy allocator. In its original form, the function simply determines the order of the allocation, invokes `alloc_pages()`, and returns the obtained page to the caller.

We want to modify this function to support allocation through PC vmalloc and integrate page coloring into the SLUB allocator. However, we must take into account the case of DMA operations, which can be performed only on physically-contiguous pages. Our solution is to keep the original version of the SLUB allocator for DMA allocation requests, and to “color” all the other types of requests.

The changes in `alloc_slab_page()` are illustrated in Algorithm 5, which shows a simplified version of the function. Its arguments are the order of allocation, and the GFP flags for the buddy allocator. To avoid confusion and to clarify that Algorithm 5

```
Algorithm 5 colored_alloc_slab_page(order, flags)
1: size ← GetSize(order)
2: if order > 0 AND flags ≠ GFP_DMA then
3:    area ← pc_vmalloc(size, flags)
4:    page ← virt_to_head_page(area)
5: else
6:    page ← alloc_pages(order)
7: end if
8: return page
```

head.

We summarized the most important changes brought by PC vmalloc. In the following section, we describe how we integrate PC vmalloc with the slab allocator. Recall from the previous section that we want to obtain a scheme similar to the one illustrated in Figure 4.11.
represents the modified version of `alloc_slab_page()`, the function has been renamed `colored_alloc_slab_page()`.

The function calculates the size of the allocation in bytes. Then, it checks the order and the GFP flags, and decides whether to request pages through PC vmalloc, or directly to the buddy allocator. If the request is of order zero, it is always redirected to the buddy allocator, since the order-zero allocator already supports page coloring. This way, the additional overhead introduced by page table manipulations is avoided.

If the flags passed as an argument to the function indicate that DMA operations are going to be performed on the requested pages, then the colored allocator is bypassed, and the buddy allocator is invoked directly. Flags associated to DMA operations are `GFP_DMA` and `GFP_DMA32` (the latter was omitted in Algorithm 5 for brevity).

The colored allocator invokes `pc_vmalloc()`. Then, because we want to maintain the original structure of `alloc_slab_page()`, the function finds and returns the physical address of the page with `virt_to_head_page()`. The latter is a virtual-to-physical translation function used in the standard Linux kernel. The function has been modified to support PC vmalloc addresses, as described next.

The rest of the SLUB allocator remains unchanged. Since the original allocator is still being used, SLUB manages two types of virtual addresses: the addresses obtained with linear mapping operations, and the new PC vmalloc addresses. The type of an address can be identified with function `is_pc_vmalloc_addr()`, described above. In order to convert the PC vmalloc addresses correctly, we modified kernel functions `virt_to_head_page()`, `compound_order()`, and `page_address()`. Table 4.3 summarizes the use of these functions, which are related to the conversion from virtual addresses to physical addresses in the SLUB allocator.

The functions are modified with a pattern similar to the following: if the page was allocated through the original SLUB allocator, then the information is obtained with the original code; otherwise we use the equivalent PC vmalloc functions described in
static inline struct page *virt_to_head_page(const void *x)
Returns the compound head of x.

static inline unsigned int compound_order(struct page *page)
Returns the compound order of page.

static inline void *page_address(const struct page *page)
Returns the virtual address of page.

Table 4.3: Utility functions of the SLUB allocator

the previous section. Next, we provide a brief overview of the modifications made to each function.

In virt_to_head_page(), virtual address x is tested with is_pc_vmalloc_addr(). If x belongs to the PC vmalloc address space, then pc_vmalloc_to_head_page() is called.

In compound_order(), pages allocated with PC vmalloc are identified with field pc_vmalloc_addr in the page descriptor: if the field is not NULL, and if it is a PC vmalloc address, then the page was allocated with PC vmalloc. In that case, pc_compound_order is returned instead of compound_order.

Finally, in page_address(), the page descriptor field pc_vmalloc_addr is tested, as with the previous function. If the page was allocated with PC vmalloc, then pc_vmalloc_addr is returned.

Summary. We presented the implementation of a “colored” multiple-page allocator obtained by integrating a simplified version of vmalloc, PC vmalloc, between the SLUB allocator and the buddy allocator. We provided new virtual-to-physical address translation functions to convert virtual addresses from the newly-defined address space into physical addresses. Finally, we modified the existing address translation functions to take into account addresses allocated through PC vmalloc.
In the next chapter, we evaluate the colored allocators. The chapter presents the methodology used in the evaluation, followed by the obtained experimental results.
5 EVALUATION

The preceding chapter presented the design and implementation of the proposed page coloring solution. In Section 4.1, we explained the methodology used to assign colors to a process through the definition of three categories of processes: Kernel, Reserved, and Unreserved. Each category is assigned a different set of colors. Section 4.2 introduced the “colored” single-page allocator, obtained by modifying the buddy allocator to add page coloring support to order-zero allocation. Finally, Section 4.3 described how to use the colored single-page allocator to build a colored multiple-page allocator by modifying the SLUB allocator. In order to change the physical-to-virtual mapping in the SLUB allocator, an additional layer is integrated between SLUB and the buddy allocator. The new layer, called Page Coloring vmalloc, changes the page request mechanism in SLUB, so that it only makes order-zero requests to the buddy allocator and stores the mapping information in the page tables.

This chapter discusses the evaluation of the new allocator. Specifically, in Sections 5.2 and 5.3, we verify that the effects of cache interference have been mitigated. Furthermore, Section 5.4 investigates the effects of the overhead introduced by the colored allocator.

5.1 Overview

With the evaluation of our solution, we want to answer the following questions: Can we avoid cache interference by allocating with the colored order-zero and SLUB allocators? Is the overhead introduced by the colored allocator acceptable?

In section 2.2, we discussed how cache interference negatively affects the predictability
of the system. With page coloring, processes can be isolated in the LLC, so that inter-core and intra-core interference are avoided. Section 5.2 describes Experiment 1, which shows that inter-core cache interference is drastically reduced. Namely, the number of LLC misses of a program running isolated on one core is comparable to the number obtained with the execution of the same program along with a co-runner on another core.

Experiment 2, presented in Section 5.3, shows the effects of cache interference on average and worst-case execution time. In Experiment 2, we show that the execution time of a program running isolated on one core is comparable to the time obtained with the execution of the same program along with a co-runner on another core.

Finally, in Section 5.4, we measure the overhead introduced by the colored allocator. In its worst-case behavior, the colored order-zero allocator must search for a page of suitable colors in the original freelists, traversing them. This introduces overhead with respect to the original buddy allocator. Furthermore, the colored SLUB allocator is an additional source of overhead: unlike the original SLUB, the colored SLUB allocator must update the page tables whenever a process makes a multiple-page request. Experiment 3 shows that the overhead generated by the colored allocator is acceptable.

The hardware platform used in this work is powered by a dual-core Intel Core 2 Duo E8500. For each core, the process has a private 32 KiB 8-way L1 data cache, and a 4-way L1 instruction cache of the same size. The L2-cache is a shared 24-way 6144 KiB cache. The available memory on the machine is of 4 GiB, and the size of a physical page is 4 KB.

Given the parameters of the LLC, the maximum number of colors in the system (\texttt{MAX\_COLORS}) is calculated with Equation 2.1 and it is equal to 64. In the following experiments, we always assume that \texttt{NUM\_COLORS} is equal to \texttt{MAX\_COLORS}. Therefore, both the LLC and the main memory are split into 64 partitions.
In order to provide further isolation, the programs developed for the experiments are compiled with C standard library *musl* [45], using the `-static` option for static linking.

### 5.2 Experiment 1: Last-Level Cache Misses

In order to measure the last-level cache misses, we wrote a C program to perform a multiplication on matrices of varying size. The cache misses are measured for matrices of sizes 500, 750, 1000, 1250, 1500, 1750, and 2000.

The program allocates memory for three matrices of integers of equal dimension. Given that each integer is 4 bytes and that the LLC is 6144 KiB, allocating memory for three matrices of dimension equal to or larger than 750 means allocating more memory than the size of the cache.

The cache misses are detected with the *perf* tools [46], which can measure the last-level cache store misses and load misses, and provide basic statistics on the collected data. Cache load misses are generated when the system attempts to retrieve a datum that is not stored in cache. Similarly, store misses are generated when the system writes to a datum that is not present in cache. In order to have an overall view of the cache misses, the two values are measured and then added together. For each data point, the experiment is repeated for a suitable amount of times, until the standard deviation of the collected data is smaller than 1%.

The *perf* tool is used in conjunction with the *taskset* command, with which it is possible to pin a program to a single core. For instance, the two commands can be combined in the following way:

```
  taskset -c 0 perf stat -r 500 -e LLC-load-misses,LLC-store-misses ./test
```
In the example, `taskset -c 0` pins the execution of what follows to CPU 0. The `perf stat` command gathers statistics about the events specified after option `-e`; in this case, it collects data about the last-level cache load misses and store misses generated by program `test`. With `-r`, it is possible to specify how many times the execution of the program is repeated: in this case, `test` is executed 500 times. Therefore, `perf stat` computes the average values of the last-level cache load misses and store misses generated in the 500 runs, and the related standard deviation.

Table 5.1 summarizes the four cases analyzed in Experiment 1. The baseline is shown in the first column (`no coloring`). The program is executed on Linux 4.9.40 PREEMPT-RT, with no modifications to the memory allocators; the program is executed in isolation on core 0 (`solo`), and, subsequently, with interference caused by a second task running on core 1 (`corun`). The two cases are repeated on the same system after enabling page coloring: this is summarized in the second column of the table (`coloring`).

<table>
<thead>
<tr>
<th></th>
<th>no coloring</th>
<th>coloring</th>
</tr>
</thead>
<tbody>
<tr>
<td>solo</td>
<td>no coloring, solo</td>
<td>coloring, solo</td>
</tr>
<tr>
<td>corun</td>
<td>no coloring, corun</td>
<td>coloring, corun</td>
</tr>
</tbody>
</table>

Table 5.1: Summary of the four cases of Experiment 1.

Figure 5.1a illustrates the baseline of Experiment 1. The blue line shows the cache misses detected in the 'no coloring, solo' case, while the red line represents the cache misses in the 'no coloring, corun' case. The selected co-runner is the same matrix multiplication program, with constant matrix dimension equal to 10,000. By comparing the two graphs, it is possible to notice that the program running with interference generates a significantly higher number of cache misses with respect to the isolated program. The gap is of up to three orders of magnitude.

The two lines overlap around data point 1500, which approximately corresponds to the allocation by the program of an amount of memory twice as large as the cache. From this data point on, there is no significant difference between the two graphs.

With Figure 5.1a, we show the impact of cache interference on system performance.
Figure 5.1: In Experiment 1, we measure the last-level cache misses on Linux 4.9.40 PREEMPT-RT. Figure (a) illustrates the cache misses measured with the system using the original allocator. Figure (b) shows the cache misses measured with the system allocating with the colored allocator. See Section 5.2 for more details.

Namely, the red graph shows that the co-runner evicts the content of the measured program from the cache, causing inter-core cache interference.

We repeated the same experiment using the colored allocator, assigning 16 colors out of 64 to the Reserved category and 16 to the Unreserved category. The remaining 32 colors are assigned to the Kernel category. The measured program is assigned to the Reserved category, running in isolation, while the co-runner (if any) is assigned to the Unreserved category.

The results are illustrated in Figure 5.1b. The gap between the two lines is substantially smaller with respect to the baseline. The largest gap, corresponding to data point 500, is reduced by approximately two orders of magnitude. For all the other matrix dimensions, the number of cache misses detected in the 'solo' case is roughly equal to those of the 'corun' case. The effects of cache interference are, therefore, significantly reduced.

It is important to notice that the smallest number of cache misses is obtained from the 'no coloring, solo' case. This is illustrated in Figure 5.2, which compares Figures
Figure 5.2: Experiment 1: comparison of Figures 5.1a and 5.1b. With page coloring, the number of cache misses is not reduced. Rather, its predictability increases. See Section 5.2 for more details.

5.1a and 5.1b. The blue and red lines belong to Figure 5.1a, while the yellow and purple lines correspond to Figure 5.1b.

With Figure 5.2, it is possible to observe that the reduction of cache interference does not necessarily correspond to the reduction of the total number of cache misses. Rather, it corresponds to the significant reduction of the gap between the 'solo' and 'corun' cases, therefore increasing the predictability. Our results show that, with our solution, the number of cache misses with page coloring is closer to the 'no coloring, corun' case. We believe this effect is mainly related to the fact that, without page coloring, the program could potentially map its data to any cache entry. Instead, with page coloring and with our category settings, the program is limited to \( \frac{1}{4} \) of the total cache size, with higher chances of evicting its own content. This situation causes intra-task interference.
Table 5.2: Results of Experiment 2: average and maximum execution times.

<table>
<thead>
<tr>
<th>matrix dim.</th>
<th>no coloring, solo</th>
<th>no coloring, corun</th>
<th>coloring, solo</th>
<th>coloring, corun</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>avg (s)</td>
<td>max (s)</td>
<td>avg (s)</td>
<td>max (s)</td>
</tr>
<tr>
<td>500</td>
<td>2.086</td>
<td>2.126</td>
<td>2.087</td>
<td>2.197</td>
</tr>
<tr>
<td>750</td>
<td>7.084</td>
<td>8.099</td>
<td>9.051</td>
<td>9.186</td>
</tr>
<tr>
<td>1000</td>
<td>16.849</td>
<td>20.992</td>
<td>21.816</td>
<td>22.039</td>
</tr>
<tr>
<td>1250</td>
<td>36.537</td>
<td>42.513</td>
<td>43.002</td>
<td>43.132</td>
</tr>
<tr>
<td>1500</td>
<td>71.956</td>
<td>73.453</td>
<td>73.955</td>
<td>74.157</td>
</tr>
<tr>
<td>1750</td>
<td>114.189</td>
<td>113.486</td>
<td>115.225</td>
<td>116.52</td>
</tr>
<tr>
<td>2000</td>
<td>172.686</td>
<td>165.099</td>
<td>174.274</td>
<td>173.019</td>
</tr>
</tbody>
</table>

5.3 Experiment 2: Running Times

In this experiment, we measure the execution time of the matrix multiplication program written for Experiment 1. As with Experiment 1, the running times are measured for matrices of sizes: 500, 750, 1000, 1250, 1500, 1750, and 2000.

The execution time is measured with function `getrusage()` from the GNU C library [47]. The function is called before and after the computation, and collects data about the program resource usage. Then, the execution time is obtained by subtracting the start time to the end time.

The statistics measured by `getrusage()` are stored in a structure of type `rusage`. The field of interest in this experiment is `rusage.ru_utime`, which contains the time spent executing the instructions of the program.

As with Experiment 1, the experiment is repeated for the four cases in Table 5.1. Table 5.2 reports a summary of the results obtained in each case by measuring the average and the maximum execution times. The two metrics are analyzed separately in what follows.

Max. Execution Times. The 'no coloring, solo' and 'no coloring, corun' cases represent the baseline of the experiment. The difference of maximum execution times between the 'solo' and 'corun' cases reaches a maximum gap of 14.6%. The experiment
is, then, repeated after enabling page coloring. While there is a slight increase in the maximum times, the gap between the 'solo' and 'corun' cases is reduced, reaching a maximum gap of 4.4%.

**Avg. Execution Times.** Similar results can be observed with the average execution times. The gap between the 'no coloring, solo' and 'no coloring, corun' baseline cases reaches a maximum of 16%. The gap is reduced to a maximum of 4.5% after enabling page coloring.

The experimental results show that our solution provides an improvement of performance isolation in the average and worst-case execution time.

### 5.4 Experiment 3: Allocation Overhead

In this experiment, we evaluate the overhead introduced by the use of the colored allocator. The experiment consists in measuring the allocation times of the matrix multiplication program written for the previous experiments. As with Experiment 1 and 2, the allocation times are measured for matrices of sizes 500, 750, 1000, 1250, 1500, 1750, and 2000.

The experiment is executed twice on the same system. First, the original allocator is used. The second time, the colored allocator is enabled and 16 colors out of 64 are assigned to the benchmark program (through the Reserved cgroup). In both cases, the program run on a single core with no interference.

As with Experiment 2, the allocation times are measured with GNU C library function `getrusage()`. In this experiment, the observed field is called `ru_stime`. The field measures the time spent executing operating system code on behalf of the program. Function `getrusage()` is invoked before and after the allocation of the three matrices, which is performed with C library function `malloc()`. The start time is then subtracted
Table 5.3: Results of Experiment 3: average and maximum allocation times.

<table>
<thead>
<tr>
<th>matrix dim.</th>
<th>original alloc.</th>
<th>colored alloc.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>avg (s)</td>
<td>max (s)</td>
</tr>
<tr>
<td>500</td>
<td>0.001</td>
<td>0.007</td>
</tr>
<tr>
<td>750</td>
<td>0.0036</td>
<td>0.011</td>
</tr>
<tr>
<td>1000</td>
<td>0.0074</td>
<td>0.024</td>
</tr>
<tr>
<td>1250</td>
<td>0.0154</td>
<td>0.029</td>
</tr>
<tr>
<td>1500</td>
<td>0.0288</td>
<td>0.056</td>
</tr>
<tr>
<td>1750</td>
<td>0.0432</td>
<td>0.078</td>
</tr>
<tr>
<td>2000</td>
<td>0.0611</td>
<td>0.087</td>
</tr>
</tbody>
</table>

Table 5.3 compares the average and maximum allocation time of the original Linux memory allocators and the colored allocator. The allocation time increases with the matrix dimension and the colored allocator introduces additional allocation overhead to the original allocators. However, the worst-case allocation time measured in the experiment is lower than 0.85% of the total execution time of the program (see Table 5.2). This value is observed for matrices of dimension 500. The percentage decreases as the matrix dimension increases, reaching 0.14% at dimension 2000. Therefore, the overhead introduced by the colored allocator is considered acceptable.

**Summary.** Experiment 1 and Experiment 2 confirmed that our solution mitigates the effects of cache interference. The metrics taken into consideration were: number of cache misses, and average and worst-case execution time. Experiment 3 showed that the average and worst-case overhead introduced by the colored allocator are acceptable. We conclude that the processes assigned to the Reserved cgroup are effectively isolated in cache at a relatively small overhead cost.
CONCLUSION AND DISCUSSION

In multi-core platforms, predictability is affected by the presence of shared resources, such as caches shared among cores. Shared caches introduce cache interference, which has been shown to significantly increase the task execution times. A popular solution in the literature is a software-based cache-partitioning technique called page coloring.

This thesis addressed the design and implementation of a new allocator for the Linux kernel that supports page coloring. The proposed solution, obtained by modifying the buddy allocator and the SLUB allocator, supports both single and multiple-page allocation. Furthermore, the new allocator provides spatial isolation between kernel and user-space processes by assigning a different set of colors to each process category. Experimental results confirmed the effectiveness of the proposed solution, which is shown to mitigate inter-core cache interference, thus improving system predictability.

It is worth noting that the modified version of the SLUB allocator developed in this thesis has been found to be affected by a technical issue. The issue is attributed to the allocation of network buffers by means of the \textit{kmalloc} kernel function, which might perform memory allocation operations in a non-conventional manner. Nevertheless, the implementation was stable enough to run the experiments discussed in Chapter 5.

6.1 Future Directions

Several directions may be taken to address some of the shortcomings of this work. The most obvious one is to provide fixes to the colored SLUB allocator in order to solve the network buffer issues.
To improve efficiency and utilization of the cache sets, the round-robin mechanism discussed in Section 4.2.2 can be extended from the Reserved category to the remaining categories. Furthermore, an improvement to the colored SLUB allocator would consist in making requests of order larger than zero to the buddy allocator when the request can be served by physically-contiguous pages of colors that can be allocated to the process. For example, if a process is assigned colors 0 and 1, an order-1 request to the colored SLUB allocator could return a physically-contiguous block of two pages of colors 0 and 1, respectively. This would reduce the overhead introduced by making multiple order-zero requests instead of a single one of a larger order. The overhead would be further decreased, as physical-to-virtual translation could be obtained with a linear mapping instead of page table manipulation.

Finally, page coloring support may be added to other components of the kernel memory to provide better performance isolation. For instance, per-CPU memory was mostly excluded from this work, except for the per-CPU slab lists: future work may be aimed at providing page coloring support to the other per-CPU memory reserves, such as the ones in the buddy allocator. Other parts of the kernel memory management that may be supported with future extensions to this work are: the kernel static memory, and the direct multiple-page calls to the buddy allocator. The latter may be excluded if physical contiguity is required for DMA operations.
Bibliography


[37] *The SLUB allocator*, 2007. URL: https://lwn.net/Articles/229984/.


