Energy-Aware Interface for Memory Allocation in Linux

Lars Rechter,
Martin Jensen

June 2021

AALBORG UNIVERSITY
STUDENT REPORT

10th Semester
The Technical Faculty of IT and Design
Department of Computer Science
Selma Lagerlöfsvej 300
9220 Aalborg Øst
https://www.cs.aau.dk

Title: Energy-Aware Interface for Memory Allocation in Linux

Subject: Programming Technology

Project period:
Spring 2021
01/02/2021 - 14/06/2021

Group No:
pt103f21

Group Members:
Lars Rechter
Martin Jensen

Supervisor:
Bent Thomsen
Lone Leth Thomsen

Pages: 78

Abstract:

In this master thesis, we extend the Linux kernel to support grouping frequently accessed (hot) and infrequently accessed (cold) data on different memory hardware. By doing this, the memory hardware with cold data can reduce energy consumption by going into low power states. We manage this separation in the kernel by adding an additional zone for cold data, which is adjustable at compile time. Processes can allocate memory in the cold zone with an extension to the mmap system call. We test the memory layout of our machine with the benchmark STREAM, showing that the modified kernel behaves as desired in terms of memory separation. Additionally, we implement a proof of concept in-memory database to benchmark the power consumption and run time performance of our modified kernel. The results show a smaller overhead than expected, but no reduction in power usage. We attribute the unchanged power usage to the memory power management strategy of the memory controller in our test machine.

Summary

Computers are faster and more common now than ever, rendering the need to optimise programs, specifically for speed, less prevalent. As data centres consumed about 200 TWh in 2019, reducing the energy usage, by a small amount on single machines, can yield significant results on a global scale. Therefore, we aim to optimise the energy consumption, while accepting a small trade-off with the computation speed.

On computers, processes run on top of an operating system (OS), which provides a layer of abstraction to the running processes and the programmer. As the Linux kernel is both open source and popular to run on servers, we make our modifications to this OS kernel.

The memory controller can put DRAM into low power states. Literature shows that a reduction in power consumption of memory can be achieved by separating the memory into frequently (hot) and infrequently (cold) accessed parts, as this allows the memory controller to put the cold memory into low power states. To reduce memory power consumption, we want to modify the Linux kernel to group memory in this fashion on the underlying hardware.

There are multiple different ways of implementing a separation between hot and cold memory in the Linux kernel. We outline three approaches; creating an additional memory zone in the kernel for cold data, dividing each zone into a hot and cold part, or determining whether page frames are cold based on their address index. We choose to implement the additional zone, as this approach makes use of the original infrastructure in the kernel, rather than creating new infrastructure.

Additionally, there are multiple ways of interfacing with the memory separation in the modified kernel. The simplest approach is modifying the mmap system call, to be able to allocate cold or hot memory based on an input. However, more advanced approaches can make it easier for the programmer to benefit from the memory layout; such as a malloc interface, modifying a run time environment to support hot and cold memory, or letting the kernel move the memory based on how often it is accessed.

We implement a new memory zone in the Linux kernel, that we call the cold zone. The size of the cold zone is adjustable when compiling the Linux kernel.

The cold zone integrates with the memory allocation infrastructure of the kernel with few changes.

To interface with the new zone, we modify the mmap system call to take an extra flag. This flag is used to indicate that the memory allocation should be performed in the cold zone. The modifications to mmap are backwards compatible with the Linux kernel from version 5.11. Thus, programmers can write code to take advantage of our modified kernel, while the same code will also run on Linux kernels without our changes.

To test our modifications, we perform two series of benchmarks. First, we modify STREAM to use our mmap interface. STREAM is used to test memory bandwidth and with our modifications, in combination with our hardware setup, we can detect whether attempts at allocating memory in hot or cold memory lead to the desired allocation in physical memory. Results show that the modified kernel successfully separates the memory allocations based on the memory request.

Our second experiment is a benchmark inspired by a similar work. The benchmark is an in-memory database. A memory intensive program, such as this, allows us to test how the cold zone can be utilised, with cold data eviction policies in the database. We implement the database using our modified mmap interface. Running the benchmark and collecting timing and power measurements are done with code from an earlier semester project that we have worked on. We run different benchmarks with varying workloads and database sizes. Our results show a run time overhead that is lower than expected, but we observe no changes to the power consumption in memory.

While we are not successful in reducing the power consumption on our test machine in the database benchmark, we can see that our modifications in the kernel work as intended in the STREAM benchmark. Therefore, we attribute the unchanged power usage to the hardware not utilising the extended periods of time, where the cold memory can go into low power state.

As future work, the modified kernel can be extended with more programmer friendly memory interfaces, dynamically changeable settings for the cold zone, support for NUMA, and portability to other architectures.

Contents

Chapter 1

Introduction

Historically, the main priority in computing has been to increase execution speed [1]. This stems from the fact that computers were rare and expensive, making it important to utilise them to the fullest. Today computers are not as expensive nor as rare. Thus, optimising software execution speed is no longer as important as it once was, leaving room for other considerations, such as energy usage.

There are three domains where energy efficiency already has a significant role, being IoT, mobile, and servers [2]. With IoT and mobile settings, there is often a very tangible limit on the energy that the system can use - the battery. Therefore, power becomes a crucial resource one must consider when developing applications in these domains. Servers are different, as the reason for the interest in energy-awareness in this domain revolves around reducing power consumption, both to reduce CO_2 emissions and the price of powering the facility. Data centres consumed 200 TWH in 2019 [2], meaning that reducing the energy consumption of these, even by a margin, has great effect.

In servers, large amounts of memory may draw as much power as the CPU [3]. Memory devices have energy-aware options, to save energy in periods where they are idling [4]. However, this feature is used sparingly in many computer systems, due to memory interleaving [4]. Thus, if we can utilise these energy efficient options in the memory hardware, we are able to make a trade off between energy consumption and run time performance.

In this thesis, we work with energy efficiency in the Linux kernel. We have chosen to work with Linux, in part, due to its popularity - a big part of data centres run Linux [5]. Additionally, the Linux kernel is open source [6], meaning that we can access the code and modify the Linux kernel to increase energy efficiency in memory. Moreover, due to the popularity of Linux, a lot of resources are available regarding the inner workings of the kernel.

We begin our work based on the following problem statement:

How can the Linux kernel be modified to support energy efficient memory management?

In Chapter 2 we go through theory and background knowledge on the topic of computer systems in general, the role of memory hardware, how memory is managed in Linux, and related works. Chapter 3 contains thoughts on the most important requirements for the project. With this in mind, we specify the design of the modified kernel and how to interface with it. Based on this design, Chapter 4 describes how we have modified the Linux kernel to support energy efficient memory allocation. The implementation is also tested, which is documented in Chapter 5. Here we run two benchmarks to test the memory layout of our machine and to test the performance of the system both in terms of run time and energy efficiency. Chapter 6 describes what we have achieved and how the choices we have made influence the final result. Lastly, in Chapter 7 we conclude on the thesis and present ideas for future work.

Chapter 2

Background

In this chapter, we go through some underlying theory of how memory works. Memory management is a complex topic that requires cooperation between the operating system (OS) and the underlying hardware. First, we give an overview of how computers and OS's work in section 2.1. This lays the ground for Section 2.2, in which we describe the functionality of memory on the hardware level, and Section 2.3, where we go through how Linux manages memory. Lastly, in Section 2.4 we list works that are related to our topic, of which we draw inspiration and knowledge for this thesis.

2.1 Introduction to Computers and Operating Systems

In the early 1980s, computers used approximately equally long time on a CPU cycle and fetching data from an address in memory [7, p. 20],[8, p. 13]. Since then, the size and speed of both components have grown - with the CPU outgrowing the memory regarding speed. To mitigate the difference in speed, additional layers of hardware caches have been introduced to lower the cost of accessing the main memory. The caches allow for small amounts of data to be stored for lower latency access to recently (or frequently) used memory addresses. Figure 2.1.1 shows a comparison between different operations in the machine. Here a clock cycle is normalised to 1 second to add context to the

length of other operations. With this normalisation, it is clear how expensive it is to access main memory.

Another important change in the hardware, is the possibility for multiple cores or even CPUs. Therefore, it is necessary that there is a system for distributing tasks to different cores and CPUs [9, pp. 5–6]. In most computers, this is done by the scheduler of the OS. Similarly, the main memory is controlled by the OS, and allocated to different processes as needed [9, pp. 7–8]. Additionally, with the non-uniform memory access (NUMA) architecture on machines with multiple CPUs, some CPUs are closer to some parts of the memory than other parts. Thus, the OS must consider the specific computer layout when performing scheduling and memory allocation [10]. We limit NUMA from this project, as NUMA further complicates the already complex topic of memory management.

Event	Latency	Scaled
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access (DRAM, from CPU)	120 ns	6 min
Solid-state disk I/O (flash memory)	50–150 µs	2–6 days
Rotational disk I/O	1–10 ms	1–12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: San Francisco to United Kingdom	81 ms	8 years
Internet: San Francisco to Australia	183 ms	19 years
TCP packet retransmit	1–3 s	105–317 years
OS virtualization system reboot	4 s	423 years
SCSI command time-out	30 s	3 millennia
Hardware (HW) virtualization system reboot	40 s	4 millennia
Physical system reboot	5 m	32 millennia

Figure 2.1.1: Overview of latency of fetching data from devices other than CPU registers [11]

Controlling the different resources of the computer is an extensive task. Therefore, the OS is an important abstraction layer between programs and hardware, to avoid resource management in program development. With the OS virtualising the hardware resources, the programs merely have to use the interfaces supplied by the OS. Another advantage of this is that it allows the OS to handle malicious or buggy programs that, without privilege, interact with other programs running on the system.

To manage the allocation of CPU and memory to the processes on the system, the OS uses virtualisation. The individual processes are oblivious to the fact that the OS schedules them to run for a share of time, before they are preempted and switched out with other processes waiting to be run. A process may even be scheduled to be run on other CPUs in the system, which is transparent to the process [9, p. 25]. For memory, the OS manages a virtually contiguous memory space for the process, which is mapped to physical memory. Thus, the process does not need to be aware of other processes in the memory space. Rather, the process simply uses the OS interfaces to obtain and release memory.

2.2 Memory Hardware

To establish a vocabulary and basic understanding of memory on a hardware level, we introduce the most important hardware features. First, we provide a general, high level overview of the memory layout in Section 2.2.1. Then, Section 2.2.2 introduces the notion of memory power states. Lastly, Section 2.2.3 describes hardware interleaving of memory. Furthermore, it is discussed how interleaving influences different workloads on the machine.

2.2.1 Memory Layout

This section serves to provide an outlined overview of the hardware components of the memory system. In computers dynamic random access memory (DRAM) is the main memory of the system. This is provided by a dual in-line memory module (DIMM), typically inserted into the motherboard of a computer. The connection between the CPU and the memory is provided by memory channels and managed by the memory controller of the CPU. Computers can have multiple channels to connect the memory controller to more DIMMs at the same time. Additionally, it is also common for a channel to be shared by more DIMMs. By having multiple DIMMs distributed over different memory channels, it is possible for the memory controller to load data from multiple DIMMs in parallel. As the CPU is faster than DRAM by a factor of 100, the improved bandwidth of using multiple channels can reduce the impact of the memory bottleneck (see Figure 2.1.1 on the preceding page).

To avoid accessing main memory, some data from memory can be stored in hardware caches, which are faster to access than main memory by multiple

factors. With caches, main memory is only accessed on a cache miss or during context switches. [12, pp. 54–56][9, pp. 94–96]

The DIMMs can be further divided into smaller units. The largest of these units is called a rank. Each rank is connected to the memory channel and is managed by the memory controller. The memory controller keeps track of which DIMMs and ranks are currently using the channels to transfer data [4]. The ranks can be further divided into banks, rows, and columns. However, we aim to use the power states at the rank level as these preserve the data in memory while offering energy savings of up to 80% as can be seen in Table 2.2.1. Power states on lower level memory hardware, such as PASR, do not maintain the data of the memory and has a smaller potential memory energy saving of approximately 5-15% [13]. Thus, we will not further elaborate on memory units below rank level.

2.2.2 Power States

DRAM, such as DDR3 and DDR4, has multiple power states to save energy whenever the memory is not being written to or read from. A lower power state yields a higher delay on accessing data on the rank, as it must awake from its current power state [4]. Therefore, low power states are only entered after a period of idling for the DIMM. The transfer between the power states on a DIMM is handled by the memory controller [14, p. 514]. Therefore, power states are only influenced by the OS indirectly through memory access patterns. To indicate the magnitude of power savings and latency, Table 2.2.1 shows the approximate power usage and exit latency of DDR4 DRAM. The table shows that up to 80% memory energy usage can be saved, when allowing DRAM to enter low power states such as the self refresh state. It can also be seen that a greater delay is introduced with greater energy savings. Thus, it is crucial to carefully manage the power states, as overusing self refresh may impair the execution speed, and only using stand by might consume an unnecessary amount of energy.

State	Power, W	Exit latency, ns
Stand by	~ 1.5	0
Power down	0.9	~ 5
Self refresh	0.3	~ 500

Table 2.2.1: DDR4 Memory States [4]

2.2.3 Memory Interleaving

As mentioned in Section 2.2.1, the CPU is restricted whenever it needs to access main memory, as main memory is slow compared to the CPU. Therefore, the memory access time should be minimised if possible. In memory hardware, interleaving is one way to reduce the memory access time, by increasing the memory bandwidth. Interleaving occurs at multiple levels of memory units, being banks, ranks, and channels [15][14, pp. 345–346]. We will not further elaborate on nor work with bank interleaving, since it does not affect rank power states. Rather, the term interleaving, in the remainder of this report, refers to interleaving across ranks and channels.

Interleaving is used to increase the bandwidth of memory by distributing the allocation of sequential memory across multiple memory modules. It is performed by using a subset of bits in the memory address to determine the device that the data should be stored in. For example, for N-way interleaving K bits are needed, where $K = log_2(N)$, to interleave the data across N devices (for example memory channels). A graphical example of interleaving across four memory devices is shown on Figure 2.2.1. In the figure, it can be seen how contiguous addresses from 0 to 15 are interleaved across the 4 devices rather than allocated to a single device. With interleaving, whenever the CPU requests a contiguous block of memory, all available memory devices can fetch their part of the request, utilising the combined bandwidth of the devices.

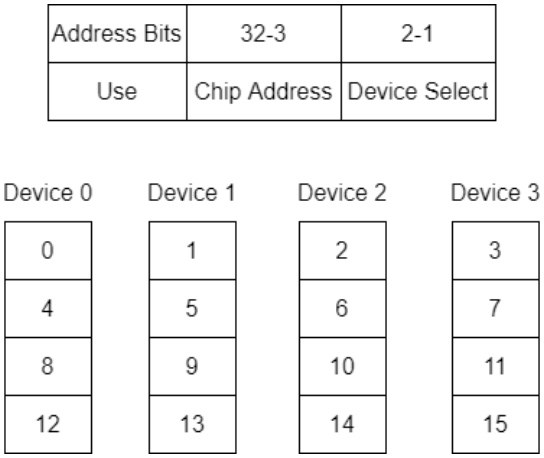


Figure 2.2.1: 4-way interleaving across four memory devices.

The effect of interleaving on program performance varies based on the workload of the program. In the benchmark STREAM [16], the sustainable memory bandwidth is tested. Here, the program performance scales linearly with the memory bandwidth. However, most other workloads do not use memory this way. In [15], the author experiences that most benchmarks perform at the same level (at the most extreme a 4.8% run time increase) after disabling interleaving and thereby, lowering the maximum memory bandwidth.

2.3 Memory Management in Linux

In this section, we give an introduction to how Linux manages the memory of the computer as of kernel version 5.11.0. First, Section 2.3.1 provides an overview of the virtual memory management system in Linux. Section 2.3.2 contains details on how processes access memory from the OS. Following, Section 2.3.3 describes how the physical memory is allocated to a process requesting memory. Then we dive into the details of the Linux memory subsystems in Section 2.3.4, Section 2.3.5, and Section 2.3.6. Based on the memory subsystems, we summarise the memory allocation flow when allocating memory via mmap in Section 2.3.7. Lastly, we describe some of the differences between hardware architectures in terms of code for memory allocation in Section 2.3.8.

2.3.1 Memory Virtualisation

To describe how Linux, in combination with hardware, tackles the topics described in Section 2.1, this section provides a simplified overview of Linux' memory management system.

The most important concept in memory management for any general purpose OS is memory virtualisation. With memory virtualisation, the OS provides the illusion that each program has start address 0, contiguous memory, and as much memory available as the architecture allows. To translate addresses from virtual to physical memory, the OS divides memory into equally sized blocks, called pages [12, p. 36]. When referring to a page we distinguish between a page of data (e.g., 4 KiB data from a process) and a page frame from physical memory (e.g., physical address 4096 to 8191) [12, p. 46]. A page of data can be moved around by the OS from one page frame to another. This only requires that the mapping from virtual to physical memory is updated. These mappings are

managed by the OS in page tables describing which processes use which page frames. Thus, accessing an address in memory from a user process requires multiple steps. First, the OS must retrieve the physical address of the page frame that is used by the process. This is done by using the most significant bits of the virtual address to determine the start address of a page frame by going through the page table. The remaining bits are used as an offset into the page frame to find the physical address. Finally, the value in this address can be read and returned to the process.

To reduce the size of the page table, Linux uses multiple levels of page tables, which are shown graphically in Figure 2.3.1. Dividing the page table into multiple levels, allows the OS to store only the relavant tables at different levels for each process rather than the entire page table [12, pp. 57–59] To translate a virtual address to a physical one, Linux goes through each of the page table levels. Linux uses the first X number of bits in the virtual address to index into the upper-most page table. In Linux this is denounced as the "Page Global Directory". The value obtained from this lookup is the address of the next level page table. This address is used with the index given by the next Y bits to obtain the address of the next page table. The OS continues through the typically four levels of page tables in Linux. In the last level page table is the page frame number, corresponding to the virtual address.

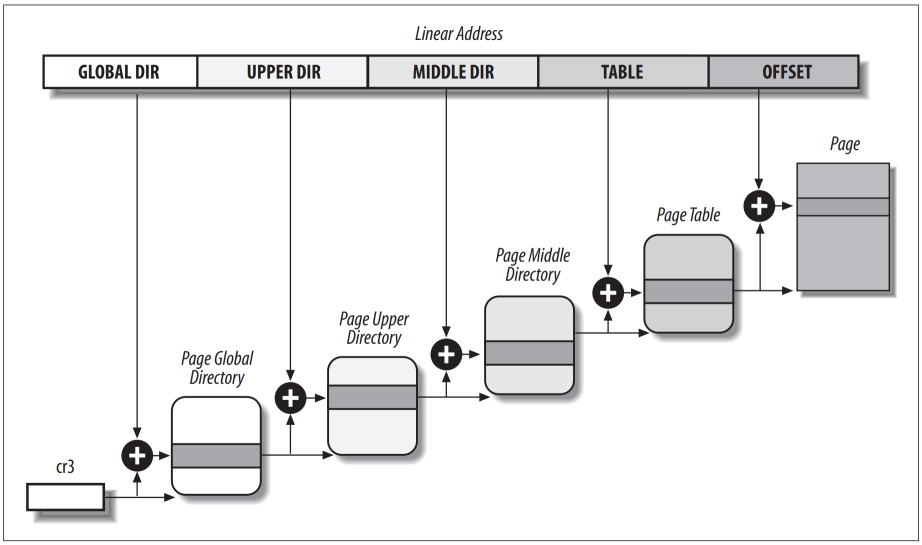


Figure 2.3.1: Multi-level page tables in Linux [12]

Virtual memory provides the illusion that processes have all memory of the system to themselves. This eases memory management from the perspective of the programmer [9, pp. 13–14]. Another advantage with virtualisation is security. By using the page table indirection, the OS can determine whether each process is accessing the memory it is allowed to access. This prevents processes from reading or writing data to each others memory spaces, without explicit permission [9, p. 146]. Lastly, the OS has an effective tool to combat fragmentation. Fragmentation can be divided into 2 categories; internal and external. Internal fragmentation revolves around allocating too much memory for a process, resulting in memory that is allocated yet unused. External fragmentation is the holes that occur between the memory allocations of processes. Internal fragmentation is limited by the 4 KiB size of the page frames of the virtual memory system. Additionally, the OS can combat external fragmentation, as the pages of a process can be placed anywhere in physical memory.

While the virtual memory system helps in many aspects, the act of accessing a memory location is further complicated - meaning longer memory access time. As described, whenever a process needs to access memory, it must access it an additional time for each page table level. This problem is mitigated by a memory address cache called the translation-lookaside buffer (TLB) [9, pp. 183–194]. This hardware cache saves the physical address of the page frame, thus removing the need to look up the address on subsequent accesses to the given page frame. Thus, there is a speed overhead for the processes on the first access to a page, although the subsequent accesses are made with the entry from the TLB. Note that this hinges on the number and demand of TLB entries.

Lastly, as processes assume they have as much memory they need, Linux must handle when processes use more memory than available in the system. To support this, Linux only allocates the virtual pages that are actually used by the process in the physical memory [9, pp. 169–179][12, pp. 35–36]. Additionally, during high memory pressure, Linux may swap some of the least recently used pages to secondary storage. Thereby, the processes are supplied with the memory they need, without losing the data that is evicted from memory. To the process, the act of swapping is transparent and swapped pages may be moved back to the main memory if accessed again.

2.3.2 Programmer Interface

The Linux kernel has two system calls called `brk` and `mmap` that handle memory requests from user space. One way of requesting more memory, is to ask the OS to resize the heap. This is done through the brk system call, by giving the desired new end address of the heap as a parameter to the OS [12, pp. 395–397]. Thus, brk can both expand and shrink the heap. A helper function to expand the heap is sbrk, which takes a value describing how much the heap should be incremented, rather than a new address. When the OS receives a request to enlarge the heap, it grants the memory if the request is legal and there is free memory available to supply the request.

The mmap system call [17] is more powerful than brk, as it is capable of allocating a linear memory address space at any point in the virtual address space, rather than only on top of the heap. The underlying function `do_brk` can be implemented with `do_mmap`, though this is slightly less efficient [12, p. 397]. With mmap the user can supply a preferred address to allocate to the desired size of memory, and information describing the protection, sharing, etc. of the memory. The specified address is used to allocate the memory, if there is space for the allocation at that virtual address. If there is not sufficient space at the desired address or no address is supplied, then mmap allocates the memory wherever it is possible. An error is returned if a sufficiently large amount of contiguous memory is not available in the virtual memory space of the process.

To ease the process of allocating memory for the programmer, the `malloc` function is available in the standard C library glibc[1]. Malloc abstracts away most details of brk and mmap such that the user only has to worry about the size of the memory to request and other parameters such as write protection. Malloc may even request more memory of the OS than the user requests in order to limit the number of system calls necessary in the future. The counterpart to malloc is `free` which is used to deallocate memory that is no longer needed.

2.3.3 Physical Memory Management

To represent the physical memory of the system, Linux divides the physical memory space into zones and page frames. Figure 2.3.2 shows the relation between physical memory in the form of DIMMs, zones, the buddy allocator,

[1] https://www.gnu.org/software/libc/

and page frames. In the figure, the physical memory of the DIMMs is split into zones. Each zone has a buddy allocator that maintains a list of free memory blocks. The memory blocks consist of a number of contiguous page frames, which can be used to store virtual memory. The following sections further elaborate on page frames, zones, and the buddy allocator.

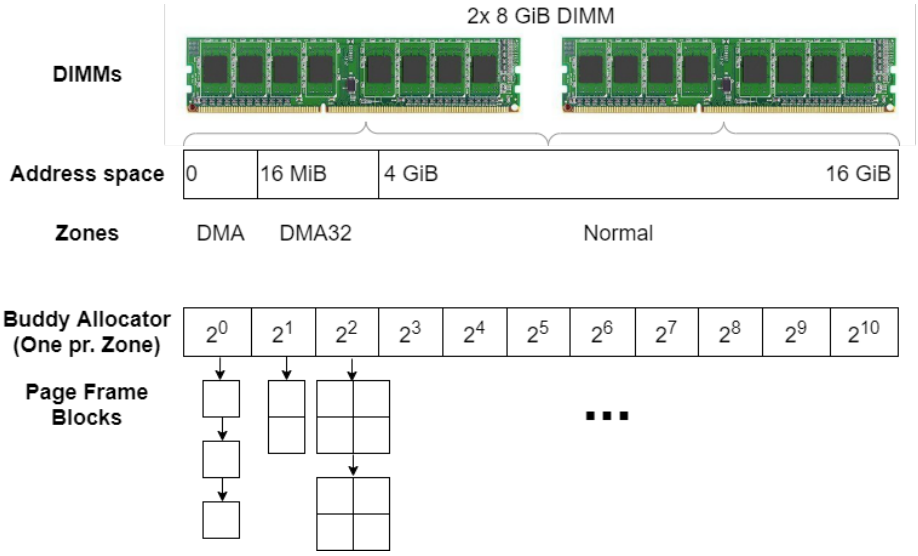


Figure 2.3.2: Graphical representation of the physical memory management in the kernel.

2.3.4 Page Frames

In Linux, page frames are 4 KiB in size. A page frame is represented by the page type in the Linux source code [6]. A page contains information on how many references there are to it by running processes, as multiple processes may share memory. It also contains flags that describe the state of the page frame. Lastly, multiple page frames are contained in a zone, which are elaborated in Section 2.3.5 on the next page.

The Linux kernel will consider certain page frames "reserved". This represents page frames that are used for kernel code or those that are unavailable to the OS. Unavailable page frames include those in address ranges with memory used by the BIOS or other machine specific functionality [12, pp. 65–66].

2.3.5 Zones

A zone represents a contiguous amount of physical memory. Zones exist for the purpose of dividing memory into parts that can be supported by certain devices and functionality [12, pp. 299–301]. For example, older IO devices may only support direct memory access (DMA) from the lowest 16 MiB or lowest 4 GiB of memory. Due to their address restrictions, the DMA and DMA32 zones may be relatively small compared to the normal zone. As the normal zone supports fewer types of memory requests, the OS will prioritise allocating to this zone if the request does not specifically require access to a lower zone. If the normal zone cannot offer the requested memory, the OS will iterate through the lower index zones. Zones are in Linux represented by a `zone` type [6, 15]. The zones that are available to the system can be configured when compiling the kernel.

The types of zones that can be represented in Linux can be seen in Listing 1 on the following page. Zones are represented as enum constants representing the hierarchy between the zones. This hierarchy is used when allocating memory to select the zone with the largest value that the request allows. I.e., a request to DMA32 tries to allocate memory in this zone first, and if this fails, then it tries the DMA zone. The presence of most zones is dependent on the configuration file that is used during compilation of the kernel. The enum constant `__MAX_NR_ZONES` on line 16 represents the number of zones that are available in the system.

Each instance of a zone is represented by a type called `zone`. A zone has members to describe its starting page frame number (PFN) and its size including or excluding the reserved page frames in the zone. Each zone contains a list of watermarks to describe limits on what is considered high or low memory loads. The watermarks are used for determining when to free memory from zones, how much memory to free, and whether allocating more memory in the zone is restricted.

2.3.6 Buddy Allocator

The buddy allocator is the mechanism used to allocate a block of contiguous page frames when processes request memory in a given zone. Additionally, it is responsible for collecting adjacent freed page frames to avoid external fragmentation [9, pp. 166–167][12, pp. 311–317].

```
enum zone_type {
#ifdef CONFIG_ZONE_DMA
  ZONE_DMA,
#endif
#ifdef CONFIG_ZONE_DMA32
  ZONE_DMA32,
#endif
  ZONE_NORMAL,
#ifdef CONFIG_HIGHMEM
  ZONE_HIGHMEM,
#endif
  ZONE_MOVABLE,
#ifdef CONFIG_ZONE_DEVICE
  ZONE_DEVICE,
#endif
  __MAX_NR_ZONES
};
```

Listing 1: The zones that Linux 5.11.0 can represent. The actual number of zones depends on the compile time configuration [6].

When trying to allocate a block of memory in a specific zone, the buddy allocator goes through its free list. The free list is a list, where each element is itself a list of memory blocks. Each of these lists contains blocks consisting of a specific number of page frames. The sizes are orders of two, from 2^0 to 2^{10}. Thus, when allocating a block of memory the buddy allocator first finds the list with the smallest blocks that can contain the request. If the free list for that size is empty, the buddy allocator looks at the blocks one order higher. However, this block is too large for the request and would be wasteful to allocate to the request. Therefore, the block is split into two blocks (buddies) of equal size. One of the blocks is used for the allocation and the other is added to the previously empty free list of that size blocks. This way of splitting a page one order higher can be done as many times as necessary to achieve the requested block size with minimal excess memory. The process of finding a fitting memory block for an allocation is shown in Listing 2. Here the for loop iterates through each order that is equal to or higher than the request, and tries to find a page of that order to allocate to.

When memory is freed, the buddy allocator checks whether the buddy of the given block is also free. If this is the case, the two buddies of order 2^N are merged into one block of order 2^{N+1}. This merged block has a new buddy of

```
1  static __always_inline
2  struct page *__rmqueue_smallest(struct zone *zone, unsigned int
   ↪  order, int migratetype)
3  {
4    unsigned int current_order;
5    struct free_area *area;
6    struct page *page;
7
8    for (current_order = order; current_order < MAX_ORDER;
   ↪  ++current_order) {
9      area = &(zone->free_area[current_order]);
10     page = get_page_from_free_area(area, migratetype);
11     if (!page)
12       continue;
13     del_page_from_free_list(page, zone, current_order);
14     expand(zone, page, order, current_order, migratetype);
15     set_pcppage_migratetype(page, migratetype);
16     return page;
17   }
18
19   return NULL;
20 }
```

Listing 2: Buddy allocator removing the smallest possible block of memory from the free list [6].

equal size, which may also be free to be merged. The buddy allocator keeps merging blocks of contiguous memory this way until either the buddy of a block is not free or the highest order of the buddy allocator has been reached.

2.3.7 The Memory Allocation Flow

Once a process has requested memory using the system calls described in Section 2.3.2, the OS has to support the process in using the memory. However, the memory is not physically allocated before it is used by the process that requested it [15]. Rather, the system calls create a virtual memory area, which is described by the type `vm_area` in the kernel. The first time the newly allocated memory is read from or written to, the system extracts the allocation options saved in vm_area. These options are passed to the function `alloc_pages_nodemask`, which is the main physical memory allocation algorithm.

The algorithm consists of two approaches; the fast allocation path and the slow allocation path [6]. First, the kernel tries the fast allocation path. This path makes the simple allocations, for instance, if there is lots of free space in the requested zone or the zones below. To allocate the memory, the algorithm first extracts information from the vm_area. The most important pieces of information are how much memory is required and which zone it should be allocated in. Using this information, the algorithm iterates through the buddy allocator of the preferred zone, as described in Section 2.3.6. If it is not possible to find a block in this zone, then the algorithm tries the next zone in the hierarchy, as described in Section 2.3.5.

If it is not possible to find a memory block for the allocation in the fast path, then the OS must make space for this allocation. This happens in the slow path, where the OS has multiple options to free memory. The first option is reclaiming memory by cleaning up the page cache. The page cache is a memory cache that contains files that have been loaded into memory from disk. The OS assumes that the user needs these files again soon, and as it is expensive to access the disk, the OS stores it in memory if there is space. The second option is simply swapping the memory to the disk, such that it can be retrieved later if necessary. As accessing the disk is expensive, the OS tries to swap out pages that it most likely will not need in the near future. The third option regards compacting the current memory allocations. This is done by combating external fragmentation [18]. The last resort for the OS is to start the out of memory killer. The out of memory killer is a process that locates and terminates memory intensive applications on the system [12, pp. 710–711].

2.3.8 Architectures

As different architectures may handle memory differently from each other or have differing feature sets, Linux must support this. As can be seen in Listing 3 some flags for virtual memory requests are simply left unavailable to 32 bit architectures in the Linux kernel. This is just one place, where the architecture of the system impacts the programming of the OS. The Linux code base has a folder called "arch", containing specific code for the different architectures. This folder contains 24 subfolders, corresponding to 24 different architectures that Linux supports. Furthermore, some of the architectures contain files specific to either 32-bit or 64-bit systems, such as the x86 architecture.

Since some low-level features require architecture-specific code, a change to such a feature will possibly require a similar change to each architecture that is supported by the kernel. Additionally, other architectures must support the change, which is not guaranteed - such as the extra protection flag bits in page table entries on 64-bit machines.

```
#ifdef CONFIG_ARCH_USES_HIGH_VMA_FLAGS
#define VM_HIGH_ARCH_BIT_0  32  /* bit only usable on 64-bit
↪  architectures */
#define VM_HIGH_ARCH_BIT_1  33  /* bit only usable on 64-bit
↪  architectures */
#define VM_HIGH_ARCH_BIT_2  34  /* bit only usable on 64-bit
↪  architectures */
#define VM_HIGH_ARCH_BIT_3  35  /* bit only usable on 64-bit
↪  architectures */
#define VM_HIGH_ARCH_BIT_4  36  /* bit only usable on 64-bit
↪  architectures */
#define VM_HIGH_ARCH_0  BIT(VM_HIGH_ARCH_BIT_0)
#define VM_HIGH_ARCH_1  BIT(VM_HIGH_ARCH_BIT_1)
#define VM_HIGH_ARCH_2  BIT(VM_HIGH_ARCH_BIT_2)
#define VM_HIGH_ARCH_3  BIT(VM_HIGH_ARCH_BIT_3)
#define VM_HIGH_ARCH_4  BIT(VM_HIGH_ARCH_BIT_4)
#endif /* CONFIG_ARCH_USES_HIGH_VMA_FLAGS */
```

Listing 3: Bits, only available on 64 bit machines, are used as virtual memory allocation flags [6].

2.4 Related Work

Little research exists on the topic of memory power efficiency, especially on DIMM or rank level. However, the articles [4] and [15] contain research on how DIMMs and ranks can be used in a more energy efficient manner. In this section, we give a short summary of both.

In [4] the authors show that the energy consumption of memory can be lowered for an in-memory database system, called DimmStore, by utilising rank-aware memory. Rank-aware memory is a possibility only after disabling memory interleaving on the system. Also, memory addresses are mapped to physical DIMMs in a modified Linux kernel. With this mapping the system is able to allocate multiple DIMMs worth of memory for the database system to evict infrequently accessed data. The result is that some DIMMs can stay in low power states for

longer as their data is accessed less frequently. Some memory is left to the OS to handle as regular memory. This memory will contain the database code and some of the frequently accessed data from the database along with the regular OS and processes.
The authors bring a series of suggestions for future work on their implementation. For example, they call for a standardised interface to request rank-aware memory, rather than having to manage this separately from Linux. Their implementation physically accesses memory that is left unavailable to the Linux kernel. This disallows memory virtualisation, which processes usually have and could potentially cause problems if multiple processes attempt to use the same physical memory. Also, accessing physical memory from a user process, as is done in [4], requires the process to have super user privileges. This could pose a security risk and therefore may not be fit for general purpose applications.

Another attempt at reducing energy usage of memory is made in [15]. The author measures the energy used by different memory hardware in the computer. The memory hardware differs in the usage of energy when reading from and writing to the hardware. With this in mind, the Linux kernel is modified, such that memory requests through the malloc interface can describe the main workload of the memory. Then if the main workload of the memory is writing, it is allocated on memory hardware that is efficient at writing. To obtain this control of which hardware contains the different memory blocks, rank and channel interleaving is disabled in the system. This leads to a small slow down in some of the benchmarks, though most benchmarks, especially on a slower processor, has the same runtime. This implementation reduces the memory energy usage of the benchmarks by up to 27.8%.
Since the memory management changes are implemented directly into the Linux kernel, the article provides inspiration regarding how to manage the physical memory in the kernel. Specifically, the author utilises the zone structure, as zones already are a representation of the physical memory.

Chapter 3

Design

As reported in [4], it is possible to achieve a memory energy reduction of up to 50% by grouping rarely used data to the same ranks. The article uses the principle of hot and cold data, which means the data is often used and rarely used respectively. This allows the ranks with cold data to stay in the low power state for extended periods of time. Therefore, we want to apply this way of handling memory to the Linux kernel, to enable energy savings on desktop computers and servers with Linux. Another advantage of the Linux kernel is that it is open source, allowing us to access and alter the code to support energy efficient memory management.

To establish a terminology, we call data that is frequently accessed "hot", and infrequently accessed data "cold". Similarly, the physical memory dedicated to storing hot data is referred to as hot memory, and the physical memory dedicated to storing cold data as cold memory. We want to group the two kinds of data on the memory hardware to allow the cold memory to go into low power states.

In this chapter, we go through the process of adding hot and cold memory in the Linux kernel. In Section 3.1 we describe the different requirements for the modified kernel. Next, Section 3.2 contains different approaches to separating hot and cold memory in the kernel. Lastly, Section 3.3 considers the options of how to use the chosen approach to separate memory.

3.1 Requirements

To describe the priority of different requirements and trade-offs, we make a MoSCoW analysis [19]. In the remainder of the section, we introduce the requirements in a list for each category of the MoSCoW. After each category we elaborate on the requirements in the category.

Must Have

- Allow allocation of hot and cold data separately in physical memory.
- Work with x86_64 architecture

To enable power savings in programs by using the principle of hot and cold data, we must modify the kernel to support a new type of memory allocation for cold memory. Furthermore, the kernel must allocate cold memory in separate memory hardware to allow this memory hardware more time in low power state.

Since most personal machines and servers use the x86_64 architecture [20], we must support the x86_64 architecture.

Should Have

- Energy usage reduction
- Possibility to enable and disable our implementation in the kernel

As [4] shows that energy can be saved in memory by allocating cold data in separate DIMMs, this should be achievable with our implementation as well.

The next requirement regards enabling and disabling the updated memory management when compiling the kernel. This is important in the Linux kernel as different users have different needs which Linux has to fulfil. Thus, some Linux users might want to disable our memory management, if it results in unnecessary overhead for them. Disabling our additions to the kernel should not break or alter any existing functionality in the kernel.

Could Have

- Change the size of cold memory on run time
- Automatically allocate hot and cold memory
- Support for different architectures

Another desirable feature is adjusting the size of the cold and hot zone at run time. This can be helpful for adjusting the separation of memory depending on the current memory requirements. Additionally, users can hardly be expected to know the optimal memory layout for their machine from the start. Having the possibility of altering the sizes of the hot and cold memory on run time eases trying out different layouts. This helps the Linux user to find the best division of hot and cold memory for them.

To make the memory separation as transparent as possible to any user and programmer, the allocation of data to hot and cold memory can be automatically and entirely handled internally in the OS. This has a large possible energy reduction, as the memory separation affects any process on a computer running with the modified kernel. It is, however, a complex task to monitor and correctly predict how to allocate memory for every process on the system.

The fundamental idea of separating hot and cold data in the memory of a computer is applicable across different architectures. While portability is important in the kernel, it is also difficult to implement and test. The kernel contains many different files that are specific for the architecture the system is running on. Thus, any changes to memory management that reside in architecture specific files, must be implemented for each of the architectures that we choose to support. Additionally, the kernel testing must be carried out on machines corresponding to all supported architectures. Thus, we are not prioritising this requirement.

Won't Have

- Detection of hardware layout

It is deemed beyond the scope of the project to automatically detect the hardware layout of the main memory in the computer. The boundary between hot and cold data should match the boundary between two hardware elements in the memory, e.g., two ranks. This allows the DRAM running the cold part to go into self refresh. While it is possible to obtain the memory layout for the OS, it is not possible to control the memory controller options. Thus, it is necessary for the user to configure the memory to support the modified kernel.

3.2 Memory Separation Approach

To support splitting the physical memory of the system in hot and cold, to allow cold DIMMs and ranks to save power, we need to alter the code in the kernel that manages the physical memory. In Figure 2.3.2 on page 12, we provide a simplified overview of how the kernel manages the physical memory currently. In the following sections, we modify this figure to showcase three different designs, to make the kernel aware of the two different kinds of data.

3.2.1 Cold Memory Zone

One approach to separating memory is to introduce a memory zone for cold data. This integrates with the existing zone infrastructure of the Linux kernel, by adding another zone that can be chosen when allocating memory. This means that little code has to be altered for allocating memory. Additionally, no abstraction is added to the memory management, as can be seen in Figure 3.2.1. One drawback to this approach is that it clashes with the idea that zones in the Linux kernel exist for compatibility reasons [15]. Currently, the sizes of most zones are determined at compile time and by the amount of memory available at the system boot. Allowing for dynamically resizing the normal and cold zones at run time may require more significant changes to the kernel.

3.2.2 Regions of Page Frames

Another approach is to have an abstraction between the zones and the buddy allocator. We call this abstraction regions. Regions are used to split the contiguous memory covered by a zone. By doing this, one region can be used for cold data and another for hot data. This approach is independent from the rest of memory management. However, by adding a new data structure to manage cold and hot data, we need to implement more code for memory allocation. This, in turn, allows us to have more control over allocation policies in contrast to modifying existing implementations such as zones. Moreover, this approach may allow for a dynamically sized separation between hot and cold data, if the regions are located within the same overlying memory zone. For a visual representation of the placement of regions in the representation of physical memory in Linux, see Figure 3.2.2. Of course, adding an additional level of memory management may introduce overhead to the kernel.

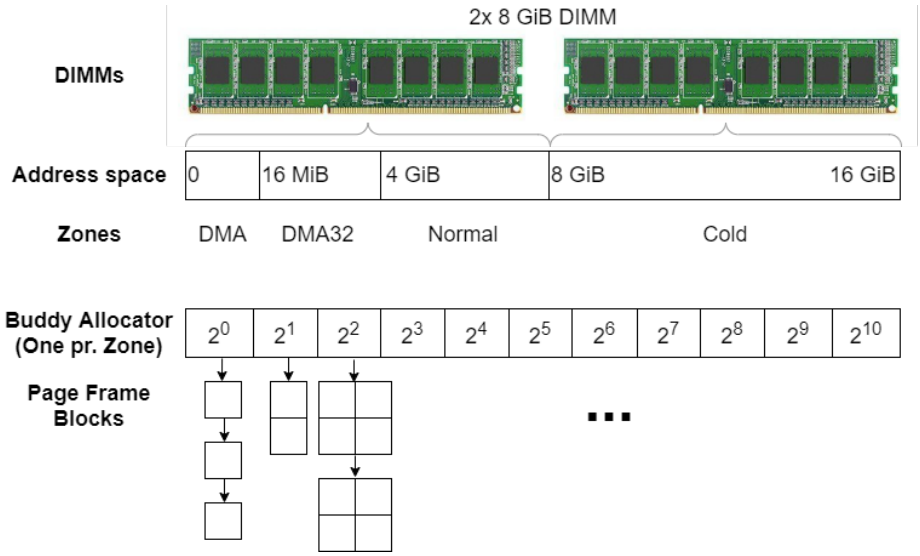


Figure 3.2.1: Graphical representation of the physical memory management in the kernel with the memory zone for cold data.

3.2.3 Hot and Cold Page Frames

Each physical page frame is represented in the Linux kernel by the page type that contains, among other, the start address of the page. By having a global variable that describes the boundary between hot and cold memory, each page frame can infer whether the page frame resides in cold or hot parts of memory by comparing its start address to the boundary. This may be adjustable at run time, by altering the address that separates the hot and cold page frames. See Figure 3.2.3 for a representation of how this affects the representation of memory in Linux. As page frames are common in many architectures, this approach is generic across those. This approach creates some overhead, as the buddy allocator must consider both hot and cold memory blocks when searching for either kind of memory.

3.2.4 Choice of Approach

Each of the three approaches described are valid, and have their own trade-offs. However, in this project we take the approach of adding an additional zone specifically for cold data. This approach requires only minor modifications to the Linux kernel as zones already represent contiguous physical memory. Thus, the

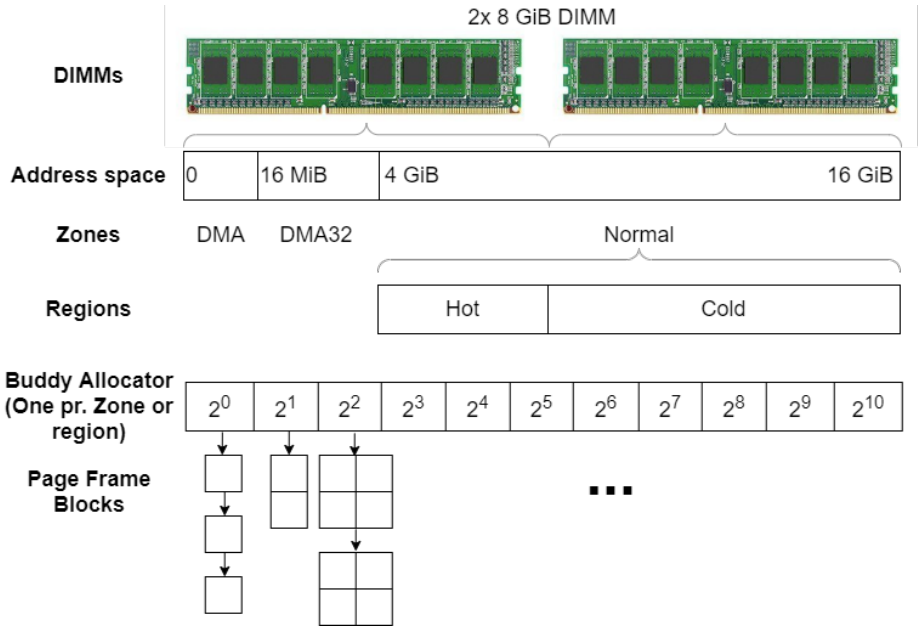


Figure 3.2.2: Graphical representation of the physical memory management in the kernel with the memory regions.

new zone will automatically be taken into account by the memory management of Linux. By introducing the cold zone we avoid having to add a new layer of abstraction in the form of regions. Additionally, the page frame approach can introduce a significant overhead in the buddy allocator, as we must iterate through both hot and cold pages when looking for either. Alternatively, we must have two buddy allocators in each zone to manage the different memory types, though this is a less modular way of having regions. The consequence of choosing the zone approach is that we must delimit the possibility of dynamically scaling the amount of hot and cold memory, as the zone approach is statically defined at compile time.

3.3 Utilisation of Memory Separation

If a separation between hot and cold memory exists, the OS must provide some method to take advantage of it. In this section, we discuss the pros and cons of either having the OS automatically manage hot and cold memory in contrast to exposing an interface for explicit hot and cold memory allocation.

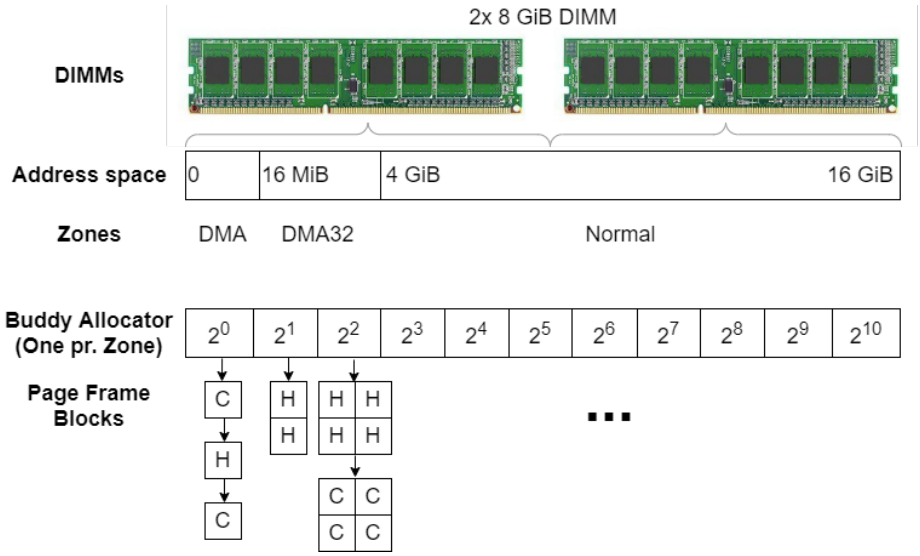


Figure 3.2.3: Graphical representation of the physical memory management in the kernel with the page frames storing whether they can have hot or cold data. Page frames that contain a C and H are cold and hot respectively.

3.3.1 Automatically Managed by the OS

One approach is to have the OS automatically place data in hot or cold memory depending on the memory usage pattern. For example, a process that allocates a lot of memory, but accesses the data infrequently, should have that data allocated to cold memory. Grouping infrequently used data in cold memory areas allows the underlying hardware to use its power saving capabilities for longer periods of time.

There is a big potential for saving energy with this approach as it affects all processes running on the machine. Moreover, this approach is completely transparent to the processes, as no changes to their code are needed.

To make this approach possible, the memory behaviour of processes must be monitored. This can be done by maintaining the flags that describe the state of a page frame. For example, the `PG_lru` and `PG_active` flags describe, whether a page has recently been accessed by its owner process [12, pp. 691–693]. This information is used to determine whether a page of data can be swapped to disk in the Linux kernel. Likewise, the same flag or a new flag can be used to determine if the page should be evicted from cold memory.

A swapped page of data is moved back to main memory as soon as it is needed by its process, as the CPU cannot access data from the secondary storage directly. It is not as critical to move data back to hot memory for this approach as the CPU can read data from cold memory. There is, however, still an interest in moving pages of data back to hot memory if they are accessed frequently, to prevent cold memory from being kept in high power states. One way of handling this is to move pages from cold to hot memory with a probability for each memory access. This approach is used in DimmStore [4].

3.3.2 New System Call

Another approach is to make a new system interface to allow explicit memory requests to the regions with hot and cold data. Such an interface can be made as an mmap-like call or maybe modify the allocation flags of the current mmap to support the two types of allocation. An interface like this is useful when the programmer knows which data in their program is accessed infrequently.

An interface like this is vulnerable to improper use by programmers. For example, if cold memory is used frequently by a process, the cold memory hardware is prevented from entering the self refresh state. Processes violating the expectation of memory usage in cold memory reduce the power savings for the ranks of memory that are affected.

Directly calling the mmap-like system call should only be necessary in specialised cases as is the case with regular mmap. In the following, we present some possible uses, for a new system call that are easier to use than mmap.

Malloc

It is simpler, faster, and uses less memory to use malloc instead of mmap in most cases [21]. Currently, as described in Section 2.3.2 on page 11, the user can call the `malloc` function to request memory. Malloc then, if necessary, calls the system calls mmap or sbrk to request more memory for the process. Therefore, in conjunction with a new mmap-like system calls a malloc-like function can use the new mmap functionality. The new malloc function has the same responsibility and functionality as the existing malloc, with the exception that it calls the mmap to obtain cold memory. A new malloc-like function has the same pros and cons as the existing malloc function. This includes the necessity for the programmer to manually free memory after use.

Run Time Environments

Run time environments of higher level languages could use a new mmap-like system call to handle allocation of hot and cold memory for the programmer. .NET has a run time environment[1] (CLR) that performs just-in-time (JIT) compilation and manages memory with a generational garbage collector [22]. The new system call could, for instance, be used to separate the hot data from the cold in the highest generation of objects in the generational garbage collector. The upside to using a new system call for cold memory allocation in a managed run time environment for a programming language, is that the change affects all programs run with that environment. Thus, the programmer does not need to consider the memory usage of their code. Moreover, as run time environments may differ, each run time environment may choose to use the new system call in different ways, depending on what is advantageous for the given program.

3.3.3 Choice of Interface

As it is the simplest and most direct way of using the changes we have made to the kernel, the choice of interface for this project is the new system call. The system call allows further extensions to build on this functionality, to aid developers in using our modified kernel to reduce memory power usage. For instance, a run time environment could be changed to use the modified mmap system call to allocate specific data in cold memory.

[1] https://docs.microsoft.com/en-us/dotnet/standard/clr

Chapter 4

Implementation

Based on the design in Chapter 3, this chapter describes the steps we take to implement a Linux kernel that supports separating hot and cold data in physical memory. The chosen design revolves around adding an additional zone in the kernel, which is used specifically for the cold data. The process of adding the zone and implementing the zone in the allocation strategy of the kernel is detailed in Section 4.1. Additionally, the section describes how the modified kernel handles inefficient use of the cold zone. Lastly, the kernel configurations to enable, alter, or disable the added functionality is described. Following, Section 4.2 documents the additions to the mmap system call. This revolves around adding a flag to allow allocation of memory in the cold zone. Our modification are available on our Github repository: `https://github.com/Nitram1995/linux/`.

4.1 The Cold Zone

To implement a split between hot and cold data in the physical memory of the computer, we must change the way the Linux kernel views and manages memory. Specifically, the kernel needs a way of differentiating the two types of data and a strategy for placing the data on the DIMMs. In this section, we describe how we implement this in the kernel. First, the cold zone is described along with the changes to incorporate it in the kernel. Then we describe the

compile options we require the Linux user to set, in order to use the modified kernel.

In Section 3.2.4 on page 23 it is decided that the best approach for our implementation is to store hot and cold data in different zones. There are two major benefits to this approach. The first is that zones are an abstraction of the physical memory layout - meaning we can use zones to group hot and cold data. The second benefit is that zones are implemented in a modular way in the kernel, as different machines and architectures require different zones to run [15]. Thus, adding another optional zone in the kernel is well supported.

4.1.1 Adding the Cold Zone to the Kernel

To add another zone in the Linux kernel, we need to alter the enum called `zone_type`. The updated enum can be seen in Listing 4. We add the `COLD_ZONE` zone as the last type in the enum in lines 363-365. Note that `__MAX_NR_ZONES` is not an actual zone type, rather it is used to keep track of the total number of zones in the kernel, which is possible due to how enums can be used as integers in the C programming language. The placement of the cold zone in this enum determines the order in which the memory allocation algorithm allocates to the available zones. This is due to the kernels memory allocation strategy, where it tries to allocate memory in the specified zone, and if this is not possible it tries to allocate memory in the zones below. Thus, by placing the cold zone as the last zone, it will only be allocated to if it is the specified zone for the allocation. This has the advantage that we have complete control over what is allocated in the cold zone, allowing us to keep hot data out of the physical memory storing cold data. Another advantage is that requests for memory in the cold zone can fall through to the lower zones, if the cold zone is full. Thus, cold data may end up being stored in hot memory instead of the system having to swap cold data to disk, if the cold memory is full. A disadvantage is that the system can be inefficiently used by only allocating to the hot zone. This causes the hot zone to start swapping to disk as it runs out of memory, while the cold zone might be completely empty. To combat this, we add a compile option, as described in Section 4.1.3, allowing the system to overflow from the hot zone to the cold zone.

Adding the cold zone to the zone_type enum takes care of most things, such as creating the zone type and instantiating a zone for cold data on boot. Although,

```
348 enum zone_type {
349 #ifdef CONFIG_ZONE_DMA
350   ZONE_DMA,
351 #endif
352 #ifdef CONFIG_ZONE_DMA32
353   ZONE_DMA32,
354 #endif
355   ZONE_NORMAL,
356 #ifdef CONFIG_HIGHMEM
357   ZONE_HIGHMEM,
358 #endif
359   ZONE_MOVABLE,
360 #ifdef CONFIG_ZONE_DEVICE
361   ZONE_DEVICE,
362 #endif
363 #ifdef CONFIG_ENERGY_EFFICIENT_MEMORY
364   ZONE_COLD,
365 #endif
366   __MAX_NR_ZONES
367 };
```

Listing 4: The updated zone_type enum from include/linux/mmzone.h with the comments removed for readability.

we do need additional changes to fully support the cold zone. An important addition is choosing the desired size of the different zones. Given that zones and zone sizes differ in the different architectures, this change must be made in an architecture specific file. As the x86_64 architecture is "must have" in Section 3.1, we edit the `init.c` file in the folder `arch/x86/mm`. To support additional architectures, this change has to be ported to their architecture specific files.

Lines 1005-1010 in Listing 5 show the updated code to describe the zone sizes, if the kernel is compiled with the cold zone. Else, we default to the original code in the else case in lines 1013-1018. The sizes of the different zones are determined by the `max_zone_pfns` array. This array has an entry for each zone, which contains the address of the highest page frame in the zone. Thus, when we add the cold zone as the last zone, we give it the highest possible address as the end address. This is done in line 1010, where we set it to `max_low_pfn`, which is the highest address of the physical memory. In 32-bit systems the highest address is at 4 GiB, so if the system has more memory, this has to

be handled in a special case. This is managed by ZONE_HIGHMEM. However, for 64-bit systems the highest address is limited by the amount of memory in the computer, rendering the highmem zone unnecessary.

```
1005 #ifdef CONFIG_ENERGY_EFFICIENT_MEMORY
1006         max_zone_pfns[ZONE_NORMAL]          = min(MAX_NORMAL_PFN,
             ↪  max_low_pfn);
1007 #ifdef CONFIG_HIGHMEM
1008         max_zone_pfns[ZONE_HIGHMEM]         =
             ↪  max_zone_pfns[ZONE_NORMAL]
1009 #endif /* CONFIG_HIGHMEM */
1010         max_zone_pfns[ZONE_COLD]          = max_low_pfn;
1011
1012         printk("EEL: CONFIG_MAX_NORMAL_SIZE: %u\n",
             ↪  CONFIG_MAX_NORMAL_SIZE);
1013 #else
1014         max_zone_pfns[ZONE_NORMAL]         = max_low_pfn;
1015 #ifdef CONFIG_HIGHMEM
1016         max_zone_pfns[ZONE_HIGHMEM]         = max_pfn;
1017 #endif /* CONFIG_HIGHMEM */
1018 #endif /* CONFIG_ENERGY_EFFICIENT_MEMORY */
```

Listing 5: The zone sizes in the modified kernel from init.c in arch/x86/mm.

In addition to adding the cold zone size, it is also necessary to limit the size of the normal zone, as this originally spanned the remainder of the addressable memory. To do this, we add a user configurable constant called MAX_NORMAL_PFN. This is used in Listing 5 on line 1006, to set the maximum page frame number for the normal zone. We use the min function to ensure that the normal zone does not try to access more memory than is available in the system.

4.1.2 Allocating to the Cold Zone

Allocation to the cold zone works in the same way, as allocating to any other zone. Thus, adding the functionality to support the cold zone requires little code. The additional code recognises the cold flag in the function gfp_zone, which is used to determine the preferred zone to allocate memory in. Listing 6 shows the additional code in the function, which returns the cold zone, if the cold zone allocation flag is set.

When allocating memory with the cold zone as the preferred zone, the function get_page_from_freelist first tries to allocate in the cold zone. If this fails,

```
460   #ifdef CONFIG_ENERGY_EFFICIENT_MEMORY
461   if(flags & __GFP_COLD)
462   {
463     return ZONE_COLD;
464   }
465   #endif
```

Listing 6: The only necessary addition to the allocation algorithm in the kernel to support the cold zone. This is added to the file include/linux/gfp.h in the kernel.

it will iterate through the lower zones and attempt allocation in those zones. If no zone can provide the requested memory, the slow path is used to make space for the requested memory. The slow path is, as the name suggests, a slower algorithm for allocating memory that may invoke various procedures to reclaim memory for the allocation. The slow path also includes iterations through the cold zone and any zones below the cold zone.

4.1.3 Overflowing Hot Memory to Cold Zone

The user can choose a non-strict separation between the normal zone and the cold zone. This means that if the system is unable to allocate a memory request in the normal zone (e.g., due to high memory pressure), it may attempt allocation in the cold zone. This is preferable over swapping from the normal zone to secondary storage, as swapping is expensive in terms of time and power consumption on both disk and CPU. To allow overflowing, three steps are added to the "slow path" of the memory allocation.

As can be seen in line 4778 in Listing 7, we only allow memory from the normal zone or higher to be allocated in the cold zone. Zones lower than ZONE_NORMAL are not considered since data from such zones (for example, ZONE_DMA) cannot be moved to higher physical addresses transparently to their owner process. This is due to some hardware only supporting direct memory access to lower physical memory addresses.

The first addition to the memory allocation algorithm, is the attempt to reclaim the page cache in the preferred zone and then reattempt the memory allocation in that zone. The page cache is memory that contains data from files that are no longer referenced by any process. The data is kept in memory to limit latency if the same file is loaded into memory later. To limit the risk of hot data being

```
4777 #if defined(CONFIG_ENERGY_EFFICIENT_MEMORY) &&
     ↪ defined(CONFIG_NON_STRICT_EEM)
4778     if(ac->highest_zoneidx >= ZONE_NORMAL && ac->highest_zoneidx
         ↪ < ZONE_COLD){
4779         /* Attempt reclaiming page cache, then retry */
4780         int ret;
4781         ret =
             ↪ node_reclaim(ac->preferred_zoneref->zone->zone_pgdat,
             ↪ gfp_mask, order);
4782         switch(ret) {
4783             case NODE_RECLAIM_NOSCAN:
4784             case NODE_RECLAIM_FULL:
4785                 break;
4786             default:
4787                 page = get_page_from_freelist(gfp_mask, order,
                     ↪ alloc_flags, ac);
4788                 if (page){
4789                     goto got_pg;
4790                 }
4791         }
4792
4793         /* Attempt with cold zone used as regular zone */
4794         enum zone_type oldzone = ac->highest_zoneidx;
4795         ac->highest_zoneidx = ZONE_COLD;
4796         ac->preferred_zoneref =
             ↪ first_zones_zonelist(ac->zonelist,
             ↪ ac->highest_zoneidx, ac->nodemask);
4797         page = get_page_from_freelist(gfp_mask, order,
             ↪ alloc_flags, ac);
4798         if (page){
4799             goto got_pg;
4800         }
4801         /* If cold zone failed, reinstate preferred zone */
4802         ac->highest_zoneidx = oldzone;
4803         ac->preferred_zoneref =
             ↪ first_zones_zonelist(ac->zonelist,
             ↪ ac->highest_zoneidx, ac->nodemask);
4804     }
4805 #endif
```

Listing 7: The modified slow path allocation, to allow data from the normal zone to overflow to the cold zone under high memory pressure. The code is added to the file mm/page_alloc.c in the kernel.

allocated in the cold zone, we reclaim the page cache before overflowing into the cold zone. Clearing the page cache is done in lines 4778 to 4791 in Listing 7. The function `node_reclaim` is called in the effort to reclaim the page cache of the preferred zone. If some memory was freed `get_page_from_freelist` is called in a new attempt to allocate memory in the preferred zone. If this is successful, an overflow to the cold zone has been averted.

If the first attempt at freeing memory in the preferred zone fails, the memory allocation is allowed to overflow to the cold zone. In lines 4794 to 4800, the preferred zone is updated to the cold zone and an attempt to allocate in this zone is made with `get_page_from_freelist`. If successful, memory is allocated in the cold zone. This is transparent to the process.

If it was not possible to allocate in the cold zone, the old preferred zone is reinstated in lines 4802 to 4803 and the algorithm will continue with the other approaches to make the memory allocation possible, including swapping to secondary storage. The algorithm loops back to line 4777 if further attempts at freeing memory are unsuccessful. This may yield a different result from the last iteration based on the system state.

4.1.4 Compilation Options

To allow the kernel to be modular, there are many compilation options the Linux user can set, to obtain a different kernel. Following this scheme, we add compilation flags to the kernel to enable or disable our changes. In lines 4777-4778 on Listing 7, two flags are used in combination with an `#if defined`. This is C preprocessor syntax to note that this code should only be included in the compilation, if the given flags are defined. By using this around the code we have added in different places in the kernel, the kernel will remain unaffected by our changes, if the flags are disabled in the compilation.

To give the user an overview of the different available compilation flags, the kernel includes a graphical program called Kconfig. This program gathers the different flags by their domains. For instance, our flags reside in the memory category, as our changes affect the memory management of the kernel. Listing 8 shows the lines added to the mm/Kconfig file in the linux kernel. The code for a compilation flag must have; the name and type of the flag, a description, whether it should be enabled by default, and a help option with further information.

```
878 config ENERGY_EFFICIENT_MEMORY
879   bool "Add cold zone for allocation of infrequently used data."
880   default y
881   help
882     Enabling this allows allocation to 'Cold Zone' in memory.
      ↪  Grouping infrequently accessed data in this zone leads to
      ↪  lower energy consumption in memory, as this data is
      ↪  allocated physically close together. Entire memory ranks
      ↪  or DIMMs that only store this data can spend more time in
      ↪  the energy efficient low power states. To take advantage
      ↪  of this, memory interleaving must be disabled on the
      ↪  DIMMs or ranks that are used in the cold zone.
883
884 config NON_STRICT_EEM
885   bool "Allow overflowing hot data to the cold zone."
886   default y
887   help
888     System may allocate hot memory in cold zone from energy
      ↪  efficient memory under high memory pressure in the
      ↪  normal/hot zone. This prevents poor performance when the
      ↪  cold zone is not utilised.
889
890 config MAX_NORMAL_SIZE
891         int "Normal zone max size as GiB"
892         default 8
893         help
894           Determines the address that separates the hot and the
↪  cold zone
```

Listing 8: The user defined flags in mm/Kconfig that enable and disable cold memory and allow overflowing the normal memory to cold memory.

As can be seen in Listing 8, we have added two flags and a configurable integer. The first flag is called `ENERGY_EFFICIENT_MEMORY`, which makes the kernel create the cold zone. Without this flag selected none of our changes are compiled. The second flag is called `NON_STRICT_EEM`, and allows the normal zone to overflow to the cold zone under high memory pressure, as described in Section 4.1.3. Lastly, the configurable integer `MAX_NORMAL_SIZE` allows the user to specify the address, where the separation between the hot and cold zone is. The input is converted from GiB to a page frame number, to represent the address in an understandable manner to the Linux user.

4.2 Supporting Cold Memory Allocation with mmap

This section explains how we expand the mmap system call to support cold data. First, we show how to use the interface to allocate both hot and cold data. Then Section 4.2.2 details, how we add support for allocating cold memory in the mmap system call. Additionally, it is described how the system call is connected to the allocation of physical memory, as this is necessary to implement the allocation of data in a specific part of memory.

4.2.1 Usage

An example of allocating memory for both hot and cold data is shown in Listing 9. Deallocating the cold memory is done with munmap, like any other memory allocation obtained through mmap. As can be seen in lines 6 and 7, the only difference between allocating regular memory and cold memory is the presence of the `MAP_COLD` flag.

```
1  #include <sys/mman.h>
2  #define MAP_COLD 0x200000
3  #define PAGE_SIZE 4096
4
5  int main(void){
6    void *regular_mem = mmap(NULL, PAGE_SIZE, PROT_WRITE|PROT_READ,
     ↪  MAP_ANONYMOUS|MAP_PRIVATE, -1, 0);
7    void *cold_mem    = mmap(NULL, PAGE_SIZE, PROT_WRITE|PROT_READ,
     ↪  MAP_ANONYMOUS|MAP_PRIVATE|MAP_COLD, -1, 0);
8
9    munmap(regular_mem, PAGE_SIZE);
10   munmap(cold_mem, PAGE_SIZE);
11   return 0;
12 }
```

Listing 9: Example of allocating a page of regular memory and a page of cold memory.

4.2.2 Memory Allocation

When a process invokes the mmap system call, the OS takes control to allocate a piece of memory from the process address space. The arguments given to mmap are first translated to Linux specific VM_flags, such that a virtual memory area

can be allocated for the process. The virtual memory is only allocated physically once the owner process accesses it, as is described in Section 2.3.7 on page 15.

When the OS handles the system call, the protection flags and other flags passed to `mmap` are translated to `VM_flags`. This is done with the functions `calc_vm_prot_bits` and `calc_vm_flag_bits`. We add a single line of code to `calc_vm_flag_bits`, which is line 158 in Listing 10. The VM_flags are stored in the `vma` struct that describes the virtual memory area that has been allocated for the process.

```
150 static inline unsigned long
151 calc_vm_flag_bits(unsigned long flags)
152 {
153   return _calc_vm_trans(flags, MAP_GROWSDOWN,  VM_GROWSDOWN ) |
154          _calc_vm_trans(flags, MAP_DENYWRITE,  VM_DENYWRITE ) |
155          _calc_vm_trans(flags, MAP_LOCKED,     VM_LOCKED    ) |
156          _calc_vm_trans(flags, MAP_SYNC,       VM_SYNC      ) |
157 #ifdef CONFIG_ENERGY_EFFICIENT_MEMORY
158          _calc_vm_trans(flags, MAP_COLD,       VM_COLD      ) |
159 #endif
160          arch_calc_vm_flag_bits(flags);
161 }
```

Listing 10: The function from include/linux/mman.h used to convert mmap flags to vm_flags.

When the process tries to use the virtual memory that has been allocated to it, a page fault occurs as the physical memory is allocated to the process. The `VM_COLD` flag is used to determine the flags for the physical memory allocation (GFP_flags). The addition to this process is the translation from the `VM_COLD` to the `GFP_COLD` in line 2179 in Listing 11.

Now the request for cold memory from the calling process is translated to necessary parameters for physical memory allocation in the cold zone. The process of allocating in the cold zone is described in Section 4.1.2 on page 31.

4.3 Summary

To summarise, we have modified the Linux kernel to have an additional memory zone that is supposed to contain cold data. The additional cold zone is configurable when compiling the kernel, in terms of the address that separates hot

```
2168 struct page *
2169 alloc_pages_vma(gfp_t gfp, int order, struct vm_area_struct *vma,
2170     unsigned long addr, int node, bool hugepage)
2171 {
2172   struct mempolicy *pol;
2173   struct page *page;
2174   int preferred_nid;
2175   nodemask_t *nmask;
2176
2177 #ifdef CONFIG_ENERGY_EFFICIENT_MEMORY
2178   //Sets __GFP_COLD flag if VM_COLD flag is set in the vm_flags
      ↪  of the vma.
2179   gfp |= _calc_vm_trans(vma->vm_flags, VM_COLD, __GFP_COLD);
2180 #endif
```

Listing 11: The first lines of the function from mm/mempolicy.c are used to allocate physical memory based on a virtual memory area.

and cold physical memory. To utilise the cold zone, we have added an additional flag in the mmap system call. Using this flag, the process can request memory that is located in the cold zone, if it is possible (i.e. there is room in the cold zone).

Chapter 5

Experiments

In this chapter we describe the experiments we conduct to test our implementation from the previous chapter. First, we present the setup and specifications of the computer, that is used for the experiments, in Section 5.1. Then we explain the experiments and the motivation for having them in Section 5.2. Lastly, in Section 5.3, we present our expectations to the benchmarks and the results from the experiments.

5.1 Test Setup

5.1.1 Computer Specifications

The test machine used to run the benchmarks has the following specifications:

General	
Processor	Intel Xeon W-1250P, 8 core (16 threads) at 4.10 GHz
Storage	512 GB NVMe SSD
Memory	32 GiB DDR4: 1x16 GiB SK Hynix HMA82GU7DJR8N-XN 2x8 GiB Kingston KHX2666C16/8G
OS	Ubuntu 20.04.2 with Linux kernel 5.11[6]

Table 5.1.1: Specifications of the test machine.

Ubuntu Linux 20.04.2 ships with kernel version 5.4 [23]. We upgrade the kernel of the OS to version 5.11, to work with the newest kernel available at the start of the project.

5.1.2 Memory Layout

To allow a part of the physical memory of the test machine to go into low power states, it is important that interleaving is disabled. In similar articles [4, 15], disabling interleaving is an option in the BIOS. However, the BIOS in our test machine does not have this option, meaning we have to find a different way to handle interleaving.

The motherboard in the test machine has two memory channels and two DIMM slots for each channel. Since it is possible to read and write data to both channels at the same time, the memory controller tries to interleave as much memory as possible across the channels. In our case, we have a 16 GiB DIMM and two 8 GiB DIMMs. If we put the two 8 GiB DIMMs in one channel and the 16 GiB in the other, we would have a memory space of 32 GiB, all of which is interleaved. This is undesirable to us, as it prevents the memory hardware from going into low power states. Rather, we place the 16 GiB DIMM and one 8 GiB DIMM in one channel, and the remaining 8 GiB DIMM in the other channel. With this skewed memory setup, interleaving is either disabled or the memory controller interleaves it in flex mode [24]. If flex mode is used, the first 16 GiB memory is interleaved between the two 8 GiB DIMMs, while the latter 16 GiB is only run on the 16 GiB DIMM, allowing it to go into low power states when this part of memory is unused. To test the memory layout of the machine, we perform experiments, which are described in Section 5.2.1.

5.2 Benchmarks

5.2.1 Memory Bandwidth with STREAM

To determine the memory layout, we test the different parts of our memory space. As mentioned in Section 2.2.3 on page 7, the biggest difference in performance between interleaved memory and regular memory is in memory bandwidth. Thus, by using the memory benchmark STREAM, we should be able to determine, whether interleaving is disabled or flex mode is used in the machine [16].

We modify the STREAM benchmark to optionally use dynamically allocated memory, obtained via the `mmap` system call. We also add the ability to allocate data in either hot or cold memory based on an argument given to the program.

To perform the memory allocation in different memory hardware, we use the modified kernel. We perform one test, where the memory is allocated normally, i.e. in the hot memory. Then we perform another test, where the memory is allocated in the cold memory. We expect the machine to use flex mode for interleaving, meaning that the normally allocated memory is interleaved and the cold memory is not.

When using the STREAM benchmark, it is necessary to select the size of how much memory the benchmark must use to conduct the test. As we are testing two 16 GiB parts of memory, we have set the size to 13.4 GiB or an array size of 600 million elements in the benchmark. The reason that we do not use all 16 GiB is that the normal zone must also preserve memory for running the OS and it always keeps additional memory ready. Thus, with a higher memory pressure the system will start to swap memory to disk, which invalidates the test.

5.2.2 Memory Power Consumption with the Energy-Aware Map

As the results from the DimmStore database [4] show large possible energy savings in memory, we want to recreate their results with the modified kernel. Unfortunately, we have not been able to obtain the source code of DimmStore. Therefore, we create a proof of concept benchmark, called the Energy-Aware Map (EAM), that uses some of the key features described in the article. We implement the EAM in C++.

The EAM can save data associated with a given key. To manage the hot and cold memory, we allocate a memory space in both hot and cold memory corresponding to input parameters. This allows us to move data to a memory space that matches how often it is used. The two areas are represented by a type called `memory_area`, that handles memory allocation, underlying CRUD (Create, Read, Update, and Delete) functionality in the memory, and maintains a LRU (least recently used) list. The LRU list is used to determine, which data is cold and can be evicted from the hot memory area. Whenever the hot memory area is full, a number of LRU data values are evicted to the cold memory area. Similarly it is necessary to have a way of returning cold data to the hot memory area, if it is frequently accessed. To implement this, each cold element has a probability of getting moved to the hot memory area on each access to the element. Thus, if an element is accessed many times, it has a high probability

of getting moved to the hot area, while an element that is infrequently accessed will seldom be moved to the hot area. With these two mechanisms, the database ensures that cold data is in the cold memory area and hot data is in the hot memory area.

Configuration

There are multiple parameters that can influence the EAM in different ways. This section explains which configurations we have set and how this influences the EAM. Lastly, Table 5.2.1 summarises the different parameters that we use for the EAM.

The first parameters are the sizes of the memory areas in the system. We use 10 GiB in the hot area and the full size of the cold zone as the cold area. Using 10 GiB as the hot area leaves room for the OS and other processes in the hot memory. Additionally, we maintain data structures, such as the LRU lists described above, in hot memory that take up space. To fill the database, we use structs of size 1024 bytes (1 KiB). This is also the data size used in DimmStore.

To run our EAM in a similar fashion to DimmStore, we implement the three phases of running the benchmark; initialising the database, warm-up, and the actual run. We use the same values for the different phases as well. After the database is initialised, we run the benchmark with the desired load as a warm-up. Then we run the benchmark, where we measure the performance. The duration of the warm-up period and the benchmark run is five minutes each for the maximum workload. For benchmarks where we scale the workload, the run time is scaled such that each benchmark performs the same number of transactions. This is the same approach as in [4]. Running the benchmark for this long stabilises the result in regards to minor influences, such as system interrupts or background processes.

During the benchmark run, we need to decide which operations should be performed in the database. The EAM supports CRUD, meaning any of these four operations can be used. To match the benchmark from the DimmStore article [4], we perform 80% read and 20% write transactions. A read transaction consists of reading a value in the database. A write transaction consists of reading a value from the database and then writing a new value in its place. Whether to perform a write transaction is determined by the `rand` function. To ensure that the benchmark workload is the same for all implementations, we seed the

rand function with `srand`. The two functions are included in the C standard library[1].

Now that we know what transactions to perform, the next question is which data values to select. We want to select different values, although with a skew such that some values are accessed more often than others. To do this, we use the Zipfian distribution with a 0.95 skew parameter. The distribution ensures that few elements are accessed often, while most other elements are accessed infrequently [25]. This is also the distribution used in DimmStore, and in their benchmark the skew parameter has little influence on the power usage [4]. Rather than implementing the Zipfian distribution ourselves, we have used the code from [26] and modified it with the code from [27] to reduce the cost of generating the numbers.

The two last configurations regard how elements are moved back and forth between the hot and cold memory areas. The first is how many elements to evict to the cold area whenever the hot area runs out of memory. A small number means that the move action has to be performed many times, while a large number introduces the risk of moving hot elements to the cold area. In the benchmark, we move 64 elements at a time, being 64 KiB of data. The other configuration is the probability for elements to be unevicted on access. This has the same considerations; too high means many cold data elements in the hot area, while too low increases the amount of time a hot element can force the cold area into a high power state. We use a probability of 1/64. Both the size of data to evict and the probability of unevicting data is the same as in [4].

Parameters	**Values**
DB size	10 GiB hot and 18 GiB cold = 28 GiB
Workload	30-300 ktps, 30 ktps intervals
Uneviction probability	1/64
Eviction size	64 KiB
Zipfian skew parameter	0.95

Table 5.2.1: The parameters with which the benchmark is run.

Running the Benchmarks

In Figure 5.2.1 on the next page, the components of our benchmark setup can be seen.

[1] https://www.cplusplus.com/reference/cstdlib/

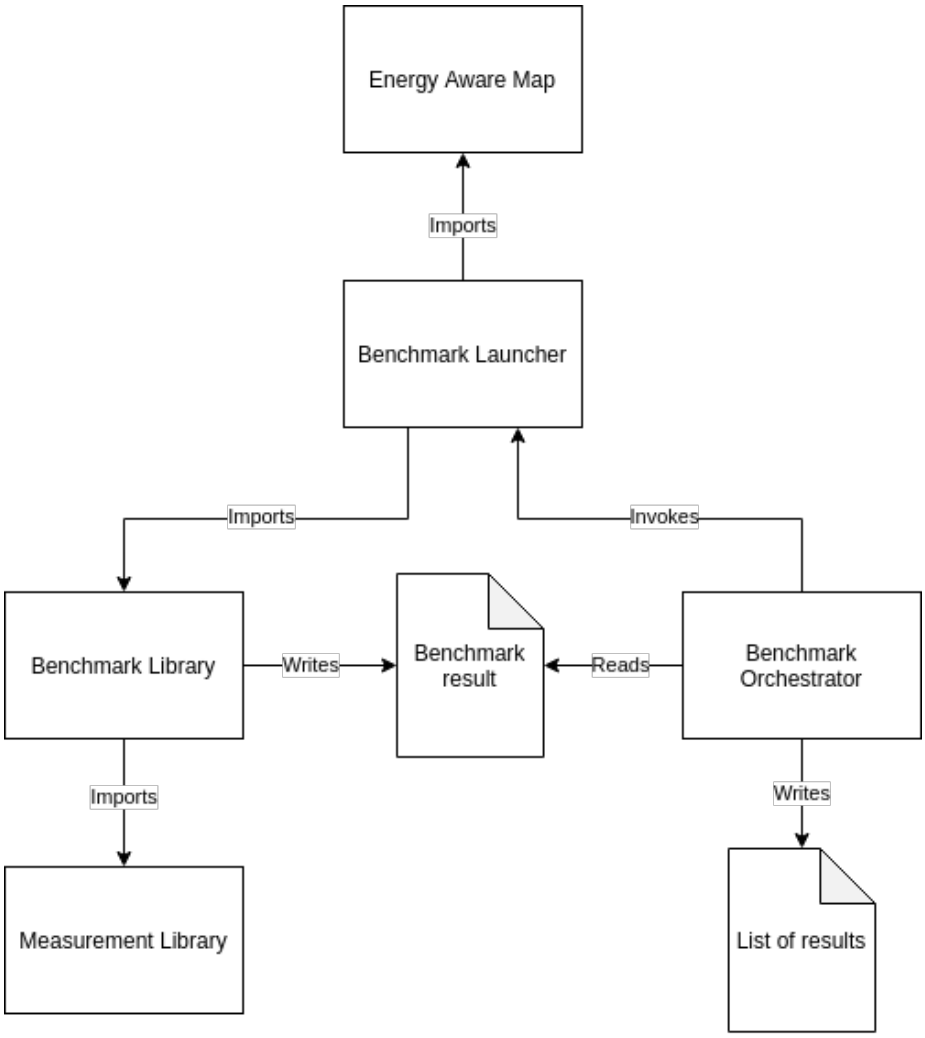


Figure 5.2.1: The structure of the components that are used to obtain results for our benchmark.

Starting from the bottom right corner of the figure, the benchmark orchestrator parses a list of benchmark configurations to run. For each of these configurations it invokes the benchmark launcher, which executes the benchmark. The EAM at the top of the figure is the benchmark implementation, which is described in Section 5.2.2 on page 41. As it is written in C++, it is compiled to a shared

library, which is then imported and called by the benchmark launcher that is implemented in C#. The benchmark launcher is able to invoke the initialisation and the warm-up phase before starting the benchmark run. To measure the EAM, the benchmark launcher uses a benchmark library that measures multiple values for a single function call. The benchmark library imports the measurement library, which is used to collect results from the system clock and Intel RAPL registers. The benchmark orchestrator collects the result from the benchmark library and adds it to the list of results. In this section, we describe each of these components in more detail.

```
1  const unsigned int arrSize = 256;
2  // 1 KB
3  struct data {
4          int arr[arrSize];
5  };
6
7  extern "C" unsigned long initDB(unsigned long hotSizeBytes,
8                                  unsigned long coldSizeBytes,
9                                  unsigned long elementsCount);
10
11 extern "C" void ycsbLoad(const unsigned int ktps,
12                 const unsigned int runTimeSec,
13                 const unsigned long elements,
14                 unsigned long map);
15
16 extern "C" void printStats();
17
18 extern "C" unsigned int sizeOfData(){
19         return sizeof(data);
20 };
```

Listing 12: Header file for C++ benchmark that is compiled to a shared library.

The EAM

The EAM is implemented in C++, as this language has low level memory management tools and easy access to the `mmap` system call for memory allocation. Additionally, C++ supports more abstract features such as classes and generic programming. We also implement functions for interacting with the EAM. These functions, such as *initDB*, take arguments to describe the size of the database and the workload to deliver. The functions must be callable from

another program. Therefore, we compile the implementation to a shared library and expose necessary functions in a header file, which can be seen in Listing 12.

Benchmark Launcher

To setup and run a benchmark, we use a benchmark launcher program that can call the shared library. We implement the program in C# of .NET core[2] for this purpose, as it allows us to utilise the benchmark library, also written in C#, to perform our measurements of time and energy consumption. To call the EAM, the functions from the library must be declared in the C# program. This can be seen in lines 9 to 21 in Listing 13.

```
9         [DllImport(@"./eel_benchmark.so",
↪   CallingConvention=CallingConvention.Cdecl)]
10        public static extern void ycsbLoad(uint ktps,
11                                            uint runTimeSec,
12                                            ulong elements,
13                                            ulong map);
14        [DllImport(@"./eel_benchmark.so",
↪   CallingConvention=CallingConvention.Cdecl)]
15        public static extern ulong initDB(ulong hotSizeBytes,
          ↪  ulong coldSizeBytes, ulong elementsCount);
16
17        [DllImport(@"./eel_benchmark.so",
↪   CallingConvention=CallingConvention.Cdecl)]
18        public static extern void printStats();
19
20        [DllImport(@"./eel_benchmark.so",
↪   CallingConvention=CallingConvention.Cdecl)]
21        public static extern uint sizeOfData();
```

Listing 13: The declaration of functions in C# from the shared library written in C++.

Our benchmarks consist of an initialisation phase, a warm-up phase, and a benchmark phase. We are only interested in obtaining results from the actual benchmark phase, meaning we must be able to make our measurements only for some of the code. Therefore, we must separate the initialisation and warm-up phases from the actual benchmark. We do this by generating functions for performing the initialisation, warm-up, and the benchmark it self in lines 51 to 59 in Listing 14. The initialisation function and the warm-up function are called

[2] https://dotnet.microsoft.com/

immediately afterwards in lines 61 to 65. The benchmark function, however, is given as an argument to the benchmark library, which performs the benchmark and collects measurements in the Run method.

```
44              var bm = new Benchmark(1, false);
45
46              // Converts the dbSize to the corresponding number of
                ↪  elements
47              ulong elementsCount = dbSize * gb / sizeOfData();
48              uint runtime = benchmark == 0 ? totalTransactions /
                ↪  ktps : totalTransactions;
49
50              // Make database
51              Func<ulong> initFunc;
52              if(isHotCold) // With a hot and a cold zone
53                  initFunc = () =>
                    ↪  initDB(dbHotSizeBytes,dbColdSizeBytes,elementsCount);
54              else // Only one zone for all data
55                  initFunc = () =>
                    ↪  initDB(dbHotSizeBytes+dbColdSizeBytes,0,elementsCount);
56
57              // Determine which benchmark to run
58              Action<ulong> warmUpFunc = (db) =>
                ↪  ycsbLoad(ktps,runtime,elementsCount,db);
59              Func<ulong, ulong> mainFunc = (db) => {
                ↪  ycsbLoad(ktps,runtime,elementsCount,db); return
                ↪  0; };
60
61              System.Console.WriteLine("Performing init");
62              var db = initFunc();
63
64              System.Console.WriteLine("Performing warmup");
65              warmUpFunc(db);
66
67              System.Console.WriteLine($"Performing benchmark.
                ↪  Expected to take {runtime} seconds");
68              bm.Run<ulong,ulong>(mainFunc, db, (input) => {
                ↪  printStats(); });
```

Listing 14: Creation of functions for initialisation, warm-up, and running the benchmark. Benchmark library is called with the function for running the benchmark as argument.

Benchmark Library and Measurement Library

The benchmark library is part of our previous project described in [28]. The most essential part of this library can be seen in Listing 15. This is the generic `Run` method that takes a function to benchmark and an action to print output from the benchmark. In lines 86 to 88 measurements are started, the benchmark function is run, and measurements are ended and stored. The measurements are performed by a measurement library, that is imported. The measurement library is also part of our earlier work [28]. It provides the actual capability to measure and store time and RAPL measurements. When a benchmark is completed, the benchmark library calls a method in line 103 to output its measurements to a CSV file.

Benchmark Orchestrator

Lastly, to run a series of benchmarks sequentially, we have implemented a benchmark orchestrator program. This is a helper program that automates our process of running several configurations of the same benchmark. This program parses a CSV file containing benchmark configurations with associated names. For each configuration in the configuration file, the runner starts the benchmark launcher as a process. The benchmark orchestrator awaits the end of the process and then it collects the result from the run and appends it to the result file. This can be seen in Listing 16.

```
71          //Performns benchmarking
72          //Writes progress to stdout if there is more than one
            ↪ iteration
73          public void Run<R>(Func<R> benchmark, Action<R>
            ↪ benchmarkOutput)
74          {
75              //Sets console to write to null
76              System.Console.SetOut(benchmarkOutputStream);
77
78              elapsedTime = 0;
79              _resultBuffer = new List<Measure>();
80              for (int i = 0; i < iterations; i++)
81              {
82                  if(iterations != 1)
83                      print(System.Console.Write, $"\r{i + 1} of
                        ↪ {iterations}");
84
85                  //Actually performing benchmark and resulting IO
86                  start();
87                  R res = benchmark();
88                  end();
89
90                  if (benchmarkOutputStream.Equals(stdout))
91                      print(System.Console.WriteLine, "");
92                  benchmarkOutput(res);
93
94                  if (elapsedTime >= maxExecutionTime){
95                      print(System.Console.WriteLine, "\nEnding
                        ↪ benchmark due to time constraints");
96                      break;
97                  }
98              }
99
100             if (iterations != 1)
101                 print(System.Console.WriteLine);
102
103             saveResults();
104
105             //Resets console output
106             System.Console.SetOut(stdout);
107         }
```

Listing 15: The `Run` method of our benchmark library. This controls the measurements when running the given function. [28]

```
30              // Initialise results file
31              var filePath = "results.csv";
32              var header =
                ↪  "name;duration(ms);pkg(µj);dram(µj);dram(W);temp(C)"
                ↪  + "\n";
33              File.WriteAllText(filePath, header);
34
35              foreach(var (name, path, process) in runs)
36              {
37                  System.Console.WriteLine($"Running benchmark
                    ↪  {name}");
38
39                  var p = Process.Start(process);
40                  p.WaitForExit();
41
42                  var res = string.Join('\n',
                    ↪  File.ReadAllLines(path + "/tempResults.csv")
43                      .Skip(1)
44                      .Select(str => name + ";" + str));
45
46                  appendResults(name, res, filePath);
47
48                  Thread.Sleep(1500); //Sleeping shortly
49              }
50          }
```

Listing 16: The benchmark orchestrator. This program parses a list of benchmarks to run and invokes programs to do so. It collects the results and outputs them to a file.

5.3 Experiment Results

5.3.1 STREAM

By running the STREAM benchmark described in Section 5.2.1, we have obtained the results shown in Table 5.3.1 and Table 5.3.2. Table 5.3.1 shows the result of the STREAM benchmark when allocating the memory in the hot zone. We can see that the sustainable memory bandwidth ranges from 17900 to 20800 MiB/s in the different functions tested in the benchmark. Here a higher value is better, as a higher value means that the memory can transfer more data at a time.

Function	Best Rate MiB/s	Avg time (s)	Min time (s)	Max time (s)
Copy	18269.7	0.526	0.525	0.528
Scale	17944.7	0.537	0.535	0.539
Add	20788.6	0.694	0.693	0.695
Triad	19838.3	0.727	0.726	0.729

Table 5.3.1: The results of running the STREAM benchmark with 13.4 GiB memory allocated in the normal zone.

Function	Best Rate MiB/s	Avg time (s)	Min time (s)	Max time (s)
Copy	11487.0	0.836	0.836	0.837
Scale	11417.2	0.841	0.841	0.842
Add	13020.5	1.107	1.106	1.107
Triad	12795.0	1.126	1.125	1.127

Table 5.3.2: The results of running the STREAM benchmark with 13.4 GiB memory allocated in the cold zone.

The results from running the benchmark with memory allocated in the cold zone are shown in Table 5.3.2. Here we get a sustainable memory bandwidth of 11400 to 13000 MiB/s. Thus, the bandwidth in the hot memory is more than 1.5 times larger than the bandwidth in the cold zone. This performance matches the memory layout described in Section 5.1.2 if the system uses flex mode. The hot memory must be allocated on the two interleaved DIMMs and the cold memory must be allocated on the single DIMM. This should allow the single DIMM to go into low power states, whenever the cold memory is unused for an extended period of time.

To verify our hypothesis that the cold memory is slower due to the memory not interleaving the highest 16 GiB of addresses, we perform the same STREAM benchmark with another memory layout. This memory layout has the 16 GiB memory stick in one channel and the two 8 GiB in the other channel. This layout allows the memory controller to interleave all physical memory across the two available channels. The results of running STREAM on our modified kernel can be seen in Table 5.3.3 and Table 5.3.4. As can be seen from the best transfer rate in the two tables, there are no significant differences between the normal zone and the cold zone with this memory layout, unlike the results with the other memory layout.

Function	Best Rate MiB/s	Avg time (s)	Min time (s)	Max time (s)
Copy	20115.3	0.478	0.477	0.478
Scale	19624.0	0.489	0.489	0.490
Add	22404.8	0.644	0.643	0.645
Triad	21383.4	0.675	0.673	0.675

Table 5.3.3: The results of running the STREAM benchmark with 13.4 GiB memory allocated in the normal zone with fully interleaved memory.

Function	Best Rate MiB/s	Avg time (s)	Min time (s)	Max time (s)
Copy	20120.7	0.477	0.477	0.478
Scale	19641.7	0.489	0.489	0.489
Add	22456.1	0.642	0.641	0.642
Triad	21455.3	0.673	0.671	0.673

Table 5.3.4: The results of running the STREAM benchmark with 13.4 GiB memory allocated in the cold zone with fully interleaved memory.

Based on the results, we determine that the flex mode interleaving is enabled on the machine. The memory layout of our test machine is visualised in Figure 5.3.1.

5.3.2 The EAM

We have multiple tests of the EAM that utilises the hot and cold memory zones implemented in the kernel. The EAM is tested against the original kernel, where all memory is allocated in the normal zone with mmap. Additionally, we also test the overflowing functionality from the hot to the cold zone in the modified kernel. Here we compile the kernel to have a cold zone, but we allocate all memory to the hot zone to test if this causes performance issues.

To test the different aspects of the EAM in these three kernel setups, we perform three different benchmarks. The first benchmark measures the power usage of the memory of the system at various workloads. Based on the results in [4], we expect that the kernel that separates hot and cold memory should have a lower power usage. The second benchmark measures the power usage of memory scaling with the size of the elements in the database. Here we expect that the power savings of the kernel that separates hot and cold data are larger with smaller databases. Lastly, we make a stress test, where we see how fast and energy efficient each of the kernel setups can execute a large number of transactions. Our expectations for this test is that the original kernel is faster

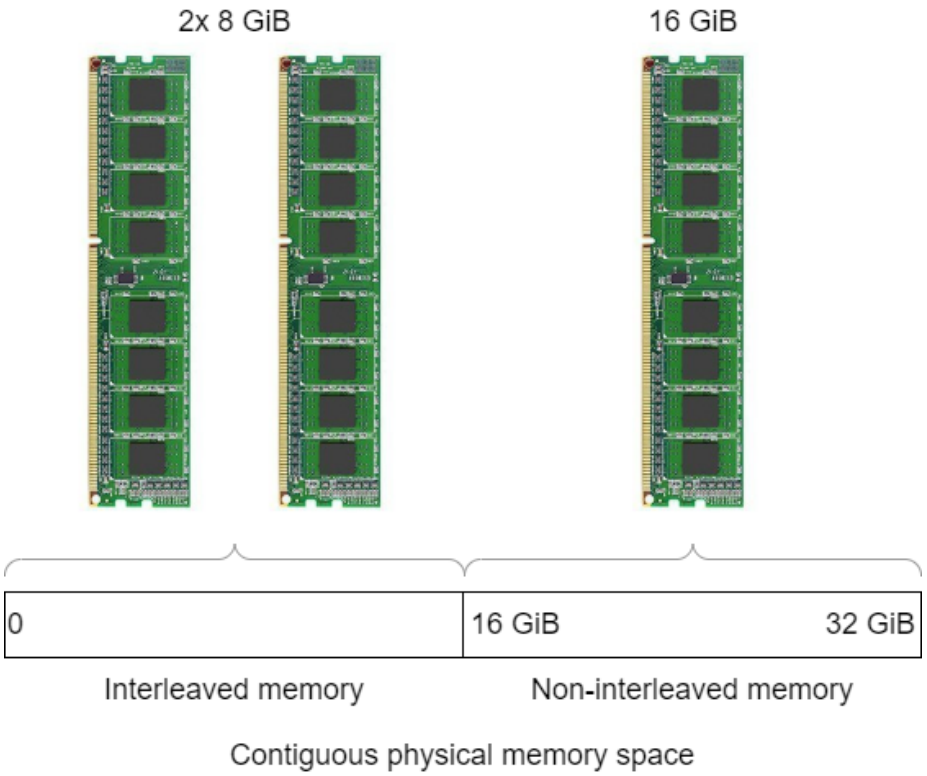


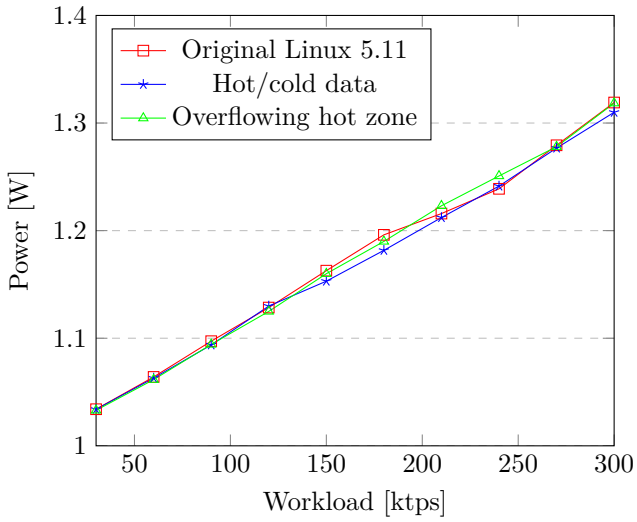
Figure 5.3.1: Memory layout in experiments. The first 16 GiB of memory is interleaved across two 8 GiB DIMMs with the remaining 16 GiB DIMM being non-interleaved.

and uses less CPU power, while the kernel that separates hot and cold data uses less power in memory. The modified kernel that allows overflowing is expected to perform similar to the original kernel, though it might have a small overhead.

The first test of the kernels revolves around scaling the workload. We scale the workload from 30 to 300 kilo transactions per second (ktps), with a benchmark size of 20 GiB. This is a database size that is large enough to ensure that the benchmark uses both memory areas. To ensure that each benchmark performs the same number of transactions, the tests with lower workload must execute for a longer time. The result of this benchmark is seen in Figure 5.3.2.

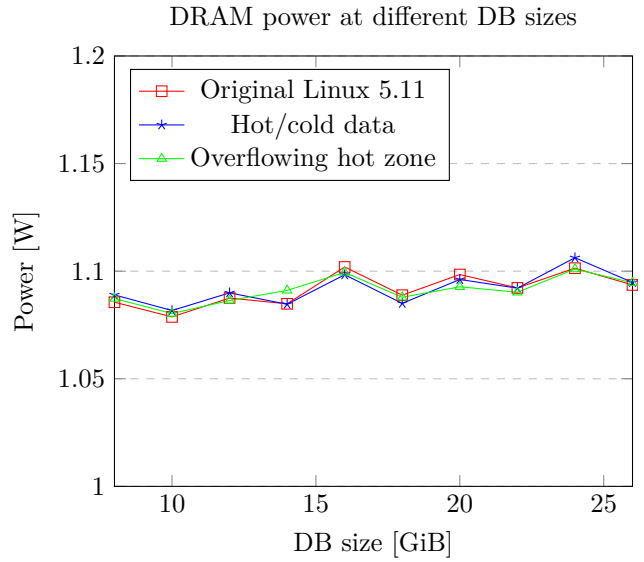
The next benchmark revolves around testing the power consumption of the memory with varying number of elements in the database. The number of elements is described as the total size of all the elements in the database. We test

Figure 5.3.2: DRAM power at different workloads with a 20 GiB database.



from a 8 GiB database to 26 GiB with 2 GiB increments. The benchmark is performed with a workload of 90 ktps, matching the value used in [4]. Figure 5.3.3 shows the results of the benchmark.

Figure 5.3.3: DRAM power at different DB sizes with 90 ktps workload.



To test both run time and energy efficiency when the database is under high pressure, we have made a stress test, where the database must handle 200 million transactions as fast as possible with a 20 GiB database. Table 5.3.5 shows how long each of the kernels take to handle the transactions, how much energy was consumed by the CPU and the RAM, and the power consumption of RAM.

name	time(s)	pkg(J)	dram(J)	dram(W)	temp(C)
Hot/cold data	245.4	5406	463.2	1.887	48.5
Original Kernel	244.6	5583	454.5	1.858	48.5
Overflowing hot zone	245,5	5539	448.7	1.827	46.5

Table 5.3.5: The results of running 200 million transactions on different kernels, with a 20 GiB database.

The results are different from our expectations for the benchmarks. While we do see the power consumption scale proportionally with the workload in Figure 5.3.2, no substantial difference arises between the regular Linux 5.11 kernel and our modified kernel. This is the case both with and without the ability to overflow hot data into the cold zone. Our expectation of a greater difference between the regular kernel and our modified one is not met, as the memory power consumption should remain low in the cold zone due to the memory of the cold zone entering a low power state. Continuing the same trend as the first benchmark, the results of the second benchmark, in Figure 5.3.3, do not yield the results we expect. Once again we see that there is little difference in the power consumption of the different kernels when we scale the database size, where we expect the kernel with hot and cold data to use less memory power than the others. The results from the two graphs can be seen in full detail in Appendix A on page 79.

Lastly, in the high pressure test seen in Table 5.3.5, the three kernels perform similarly as well. In this test we expect the original kernel to be slightly faster than the other two due to overhead introduced in the modifications. The original kernel is 0.33% faster than the modified kernel with strict separation between the hot and cold data. The difference in run time between the original kernel and the modified kernel that allows overflow is 0.37%.

Chapter 6

Discussion

To start the discussion, we summarise the direction of the thesis, the changes we have made to the Linux kernel, and the benchmarks that we have performed. The thesis revolves around implementing a way to make the memory management in the Linux kernel energy efficient. To further define this, we have made a MoSCoW analysis in Section 3.1 on page 20, where we prioritise different requirements. As we have chosen to improve the kernel by separating hot and cold data in memory, the "must have" requirement is implementing this functionality in the x86_64 architecture of the Linux kernel. We discuss our implementation of the memory management in the kernel in Section 6.1, and the user interface we have made to support this allocation in Section 6.2.

In the "should have" category, the first requirement is reducing energy usage. Unfortunately, we do not fulfil this requirement. Section 6.4 contains considerations as to why we do not achieve the results we expected from our benchmarks. Additionally, Section 6.3 contains considerations as to how we have made our benchmarks. The other "should have" requirement, of implementing flags to enable, disable, or change the parameters for the cold zone in the kernel, is discussed in Section 6.1.5.

The first "could have" requirement is changing the size of cold memory in the kernel during run time. We have not implemented this, as the zones are defined at compile time in the kernel. The next "could have" requirement is automatically allocating and moving data between the zones in the kernel. This is not imple-

mented, although it could reduce energy usage without modifying programs to take advantage of the modified kernel. Section 7.2 further elaborates on this. The last "could have" requirement is support for different architectures in the kernel. We have chosen to only support x86_64. Supporting more architectures is discussed in Section 6.1.2.

To round off the discussion, we reflect on the relevance and context of the project in Section 6.5.

6.1 The Cold Zone

We choose to use the zone infrastructure that already exists in the Linux kernel. This allows us to make an arbitrary split in physical memory, separating regularly allocated memory from cold allocated memory, without adding new data structures and with minimal changes to the memory allocation algorithm.

6.1.1 Process of Implementing the Cold Zone

Creating and utilising a new zone in the Linux kernel is a simpler process than expected. Our implementation of the cold zone is described in Section 4.1.1 on page 29. Zones are represented by the `zone` data structure, and a zone is created for each value in the `zone_type` enum. Introducing the cold zone is a matter of adding it to the enum, assigning it a maximum page frame number, and limiting the size of the normal zone. To allocate to the zone, a GFP flag associated with the zone must be added. The code in the kernel related to allocating memory in a zone works automatically with the new cold zone. To allocate memory, the kernel uses a macro that iterates from the requested zone to the lower zones, attempting to allocate in the highest possible zone. Here the cold zone is automatically included in these iterations, if the request for memory is a request for cold memory. As with allocations to any zone, we can fall through to lower zones to complete an allocation meant for the cold zone. Thus, the kernel will iterate through the normal zone, and lower zones if necessary, to complete the memory request to the cold zone. Most other code associated with the cold zone is related to debugging or statistics in the OS.

6.1.2 Cold Zone in Relation to Other Approaches

As described in Section 3.2 on page 22, there are other possible ways of implementing a separation in memory. One is adding a region data structure under each zone to split hot and cold memory. Another is allowing each page frame to determine whether it is hot or cold based on its address. The approach of introducing regions requires a new data structure to be implemented. In the case of embedding information on whether a page is hot or cold in page frames, this can be done by comparing the start address of a page frame to a boundary address. In both cases the new implementations would require changes to the memory allocation algorithm. With the region data structure, the kernel has to manage more than one buddy allocator per zone and choose the buddy allocator that fits the memory allocation request. For the page frame approach, it is necessary to retry the memory allocation algorithm until a page matching the allocation type is found.

An additional zone adds little in terms of run time overhead, as it is only one extra iteration when iterating through zones during memory allocation. If the user requests memory in a zone lower than the cold zone, no additional iteration is run. Adding a layer of regions would result in a similar performance when allocating memory. This is since each region would require a separate buddy allocator that can be used to find a specific type of page, just as each zone has its own buddy allocator. If we were to separate hot and cold memory with the page frames, it would decrease the performance when allocating memory. For instance, if there are few hot pages and many cold pages, the system has to iterate through many cold page frames before finding a usable hot page.

The definition of the cold zone, such as the size of it, is defined when compiling the kernel. The cold zone can not be enabled or disabled at run time and its size is not adjustable at run time either. This may discourage users from utilising it as it cannot change in reaction to the workload on the system. Even with a constant workload it may be a tedious task to find the best size for the cold zone as adjustments require recompilation of the kernel. It might be possible to resize the cold zone, if functionality is added to the kernel that can move memory from the buddy allocator of one zone to that of another zone. For the approach that uses regions, that are independent from zones, similar functionality is necessary to resize regions. However, the page frame approach requires fewer changes to support this, as changing the boundary address might be sufficient to move the

split between hot and cold memory. In any case, it is necessary to move the data that reside in the memory area, which has changed from one type of memory to another. Moving data, when changing the size of the parts of memory, is an extensive task.

While the implementation of the cold zone is straightforward, it does contain a few lines of architecture-specific code. We make our implementation for the x86_64 architecture. As this architecture supports a certain set of memory zones, it contains a specific `zone_sizes_init` function to set the sizes of the zones. As described in Section 4.1.1 on page 29, we modify this function to limit the normal zone in size and set a size for the cold zone. To support architectures such as ARM64 or RISC-V, similar changes should be made to their implementation. Some architectures do not have a `zone_sizes_init` function and therefore further investigation has to be made into how to support the cold zone for these architectures.

The other approaches might be less architecture specific than the cold zone. This is due to the other approaches not requiring changes to the zone sizes, which is the only architecture-specific code we use in the implementation of the cold zone. Thus, unless the two other approaches have other requirements that must be set for specific architectures, they are less architecture specific.

Seeing how well our implementation with the zones fit into the kernel code, the zones seem to be a good way of implementing cold memory. The two other approaches do have some advantages over the zone approach, such as easing portability and dynamically resizing the cold memory. However, as these requirements do not have high priority in this thesis, the simplicity and smaller overhead from the zones are preferable.

6.1.3 Optionally Overflowing Hot Memory to Cold Zone

Our modification is a big change to how memory is made available to processes of the system. Regular programs that have not been made to take advantage of the possibility of allocating in cold memory will, by default, allocate all their data in hot memory. This has the consequence that if few or no processes utilise the cold zone, the cold memory will be unused while the hot memory can be filled to the point where swapping data to secondary storage is necessary. To combat this scenario we have made the option to enable `NON_STRICT_EEM`. As described in Section 4.1.3 on page 32, this setting will attempt to separate memory by

allocating regular memory requests in hot memory, but it will allocate memory requests for the normal zone in the cold zone, if the normal zone is under high memory pressure.

The setting has its pros and cons: It can potentially improve performance over the regular setting of strictly separating memory. If a machine has 8 GiB of hot memory and 8 GiB of cold memory and running processes need 12 GiB of memory, the strict separation can cause the OS to swap a lot of data between hot memory and disk. This is a disadvantage over simply allocating data in the cold zone. If, however, the policy is enabled and it allocates hot data in the cold zone, the memory from the cold zone may be kept hot by the data that was overflowed to the cold zone. This can negate any possible memory power savings if the data is hot and its owner process does not deallocate it. Moreover, the act of allocating hot data in the cold zone is slower than normal. This is due to the overflowing happening in the slow path of the memory allocation algorithm. Once allocated, the memory is as fast to access as any other memory, given that the cold memory is kept in higher power states.

6.1.4 Size of Cold Zone

The split between hot and cold memory can be made at any address with a granularity of a GiB. This is not necessarily as fine granularity as the user of the system may want. The split could also be given as a multiple of the size of a page frame in memory. On x86_64 bit machines, the pages are 4 KiB in size. Determining the best page frame number for splitting memory would be a complex task for the user to do, especially given that page frame sizes varies across architectures. Instead, giving the separation address in GiB is relatable to the information that can be obtained about the memory layout of a computer.

We do not have a lower limit to the start address of the cold zone. If a user chooses to set the split at address 0 GiB, the normal zone will not contain any data. If, however, the DMA32 zone exists, the OS allocations to the normal zone will fall through to this zone. If only the DMA zone with a maximum size of 16 MiB, is enabled the system may become unusable as there is not enough memory to contain user processes or even the Linux kernel. This issue can be resolved by enforcing a minimum split address or enforcing the `NON_STRICT_EEM` memory allocation setting for addresses under a certain boundary. We have not spent time to solve this issue, though, as we determine that the default split

at 8 GiB indicates that higher memory addresses should be used. Besides, we provide some help in the documentation in Kconfig.

6.1.5 User Customisation

The Linux kernel has many options that can be enabled or disabled at compile time. To maintain this modularity, we have added three compilation options to control the cold zone in the modified kernel; one to enable the cold zone, another to allow overflowing to the cold zone under high memory pressure, and lastly an option to set the address of the boundary that separates hot and cold memory. Additionally, it is implemented to support the command line tool, menuconfig, that the Linux kernel is already using, to ensure that it works as any other compilation option in the kernel.

Allowing users to enable, disable, or modify the kernel is important for the usability. Though, it is required that the user has knowledge of memory to navigate the options. The user should have an understanding of what memory interleaving is and how to disable it or take other measures to allow the kernel to allocate memory on specific hardware. Also, the user should have an understanding of the memory space of the machine to set the boundary address separating hot and cold memory. To help the user navigate this, we added information to the compile options menu in the kernel.

6.1.6 Linux Community Guidelines

Currently, zones exist in the kernel to ensure compatibility between the physical memory space and legacy devices that can utilise only a subset of available addresses [12, pp. 299–301]. An exception does exist in the form of `ZONE_MOVABLE`. This zone exists to enable the memory hotplug feature for some memory and to allow easier migrations of data from the `ZONE_MOVABLE` zone [6]. The cold zone does not exist to ensure compatibility with hardware devices, as it spans over addresses that could already be used in the normal zone on 64 bit systems. Instead, the cold zone acts to separate physical memory into parts that can be utilised for different purposes - in this case for cold data. As special case zones, such as `ZONE_MOVABLE` are allowed in the Linux kernel, using a zone to manage cold data is not a problem for merging our changes into the kernel.

Also, in order to be widely accessible, support for more architectures should be implemented. If more architectures can be supported, showing that the cold

zone is not only a specialised use case for x86_64 machines, our implementation may stand a better chance of acceptance in the Linux community.

The system call, mmap, is able to take several flags in Linux that are not specified in any POSIX specification [17]. As there exist several Linux specific flags, we determine that an additional `MAP_COLD` flag will not be a problem to integrate in the kernel.

Adding the user customisation to let the user enable and disable our changes, is crucial. It is important to the Linux community that the kernel is modular and it is possible to disable features from the kernel [29].

6.2 The System Call

To interface with the changes we have made in the kernel, we modify the existing mmap system call. Allocating in the new cold memory zone is done by adding the cold zone flag to the mmap allocation. An advantage of this solution is that the kernel does not check for the cold zone bit in the mmap flags, if the cold zone is disabled. Thus, using mmap with the cold zone is backwards compatible with the original Linux kernel, allowing programmers to use the programs that support the cold zone on any kernel.

In Section 3.3 on page 24, different possible interfaces for the modified kernel are considered. The mmap interface is the simplest of the interfaces, and is useful as it is possible to implement the remaining interfaces using the modified mmap system call. The other interfaces, for example malloc, are also relevant as they are easier to use for programmers, and are further discussed as future work in Section 7.2.1.

Modifying the mmap system call requires few lines of code. We need to pass the flag from the mmap flags to the flags in a virtual memory area. One issue is that currently only bits above the 32nd bit in the bitmask are free. We are, thus, forced to use a higher bit meaning that the system call does not work on 32-bit architectures. One way of changing this, is to change the type containing the flag from a long to a "long long" type, such that a minimum of 64-bit is available.

Once the flag is passed to the virtual memory area, we need to ensure that the physical memory allocation algorithm uses the flag. To support this, we have

added a check in the memory allocation path to allocate virtual memory areas that have the cold flag in the cold zone. The simplicity of changing the mmap system call stems from the design choice of using a zone to represent cold data in the kernel. With a different approach, such as having hot and cold page frames or split zones into a hot and a cold region, it would require more changes to the memory allocation algorithm to support the cold data. The page frame approach would require a check to see if the page frames found in the memory allocation algorithm is the same type as the request. Additionally, it needs to retry if it was the wrong type, potentially attempting to allocate pages many times before it is successful. Splitting zones into a hot and a cold region requires a check to determine where in the zone to allocate the data.

6.3 Benchmarks

To test the modified kernel, we use two benchmarks. The first is a benchmark called STREAM, which we use to test the memory layout of the system. The other benchmark is the EAM, implemented by us. It is made to match the benchmark used to test DimmStore in [4].

6.3.1 STREAM

The STREAM benchmark is advantageous for us to use, as it can show the memory layout of our machine indirectly by measuring the memory bandwidth. As we do not have the option of disabling memory interleaving in the BIOS, we rely on the memory layout to single out a DIMM. To allocate the memory dynamically, the STREAM benchmark has to be changed, as it is allocated statically as default. To do this, we use the modified mmap system call we have made. The benchmark is not affected by this, as the memory is allocated before the benchmark begins measurements.

For the test we used a data size of 13.4 GiB, which is large enough to use most of the hot or cold memory space without the risk of swapping compromising our results. In the benchmark, it is important that the data size is large enough that the tests can be measured accurately by the timer granularity of the computer. This is fulfilled by the test size we use.

The STREAM benchmark runs 10 iterations of the tests per default. Tables 5.3.2, 5.3.1, 5.3.4, and 5.3.3 all show the smallest and the largest time usage

for these measurements. All of these values are less than 1% different from the average. Thus, more iterations are not necessary, as the measurements are precise.

6.3.2 The EAM

The EAM is made to see if we can save power in the memory of the test machine. The benchmark is an in-memory database that handles read and write operations on a large amount of data. This is a realistic workload for a server, as many servers manage data and handle requests. While we strive to follow the implementation used in DimmStore, there are differences that we will discuss with their impact on the benchmark.

While we have used the same parameters to control how pages are moved between the hot and the cold memory area, we have not implemented a mechanism that automatically moves unused pages to the cold zone. In DimmStore a number of threads evict cold data every 1 ms, if their system region (corresponding to our hot memory) is under high memory pressure. We do not move pages to the cold area until the hot area is full. This slightly increases the latency when handling a request that requires memory to be moved. This change slightly skews the results compared to DimmStore, although it does not make a difference when comparing our own results to each other.

Another significant difference is the hardware used to run the benchmark. DimmStore uses a server with eight DIMMs placed in eight memory channels, where we use a desktop computer with three DIMMs placed in two channels. Additionally, DimmStore has disabled interleaving, allowing them to use two DIMMs for the system region, where the remaining six potentially are in low power state. We only have the possibility of putting one DIMM in a low power state, as the other two are interleaved. Thus, both the total power consumption of memory and the potential power savings are larger in DimmStore. However, if the single DIMM goes into low power state, we should see a power reduction in the modified kernel compared to the standard Linux kernel.

The last difference is how the energy measurements are taken during the benchmark. We use RAPL to measure the energy usage of the EAM, where DimmStore uses hardware to measure the energy usage of the memory. RAPL has a good accuracy, meaning that we can reliably compare our own results with each

other [30, 31]. Thus, we can compare the relative power usage in our results to the relative power usage in DimmStore.

While implementing the EAM, multiple parameters are introduced that influence the performance. Most of these parameters are set to match those used in DimmStore to allow comparing our results to theirs. Changing the parameters will likely lead to slightly different results, but we expect the results to be similar.

Each benchmark is run at least five minutes to reduce the measurement noise, such as system interrupts. The initialisation and warm-up phases make the workload more realistic, as a database normally is a long-lived process, where initialisation is unimportant compared to the performance of activities such as inserting or reading rows of data.

6.4 Results

By running our benchmarks, we have obtained the results seen in Section 5.3. First, we interpret the results of the STREAM benchmark, then the results for the EAM.

6.4.1 STREAM

The goal of the STREAM benchmark is to determine, whether the memory layout behaves as we expect. The results show that flex mode interleaving is used on the machine. This is apparent, as the cold zone memory has a significantly lower bandwidth, at approximately 60% of the hot zone bandwidth, seen in Tables 5.3.2 and 5.3.1. Additionally, if we change the memory layout, such that the entire memory space is interleaved, this difference is no longer seen. With that memory layout, the memory allocated to hot and cold zones have the same bandwidth, as seen in Tables 5.3.4 and 5.3.3. We account the small differences in run time between the two to measurement uncertainty. Thus, we conclude that the memory layout is as we expect, with the hot zone consisting of the two 8 GiB DIMMs, and the cold zone consisting of the single 16 GiB DIMM. Additionally, we see that the kernel is capable of separating the physical memory in hot and cold zones.

6.4.2 The EAM

While we get results that match our expectations in the STREAM benchmark, the results from the EAM benchmarks are a different story. We expect the modified kernel to use less memory power at the cost of a small performance overhead. Figure 5.3.2 and 5.3.3 show that we do not reduce the memory power consumption when scaling the workload nor when scaling the size of the workload. In both graphs we see that the results from the three different settings yield close to the same memory power consumption.

For the workload graph, the memory power consumption of all three kernels rises proportionally to the workload. The workloads below 120 ktps have almost the same power consumption for all settings. From 150 ktps we see the different settings differ slightly from one another. The kernel utilising hot and cold data is slightly smaller than or equal to the other two settings. Though, the power usage differences are so small that they might stem from measurement uncertainties.

For the tests with a varying database size there is again, little difference between the three kernels. The pattern of the power consumption in relation to the database size is very similar between the kernels, even down to the rise and fall of the power consumption for every other increase of database size. Only at 14 GiB does the overflowing hot zone distinguish itself by continuing to increase its power consumption for the second database size increment in a row. It does, however, fall back into the pattern of the others after the 16 GiB database size. Thus, the updates to the kernel do not show a general pattern of higher memory efficiency depending on the database size, in the EAM benchmark.

This is in stark contrast to the results from DimmStore [4], where they save up to 50% memory power on the smallest database. Thus, either the kernel modifications, the EAM, or the hardware setup behaves drastically different in our benchmark than it does in DimmStore. The first possibility is that our implementations in the kernel do not act as we intended. However, as the STREAM benchmark confirms the modified kernel's control of the physical memory, it is not the kernel that causes our results to be different from the results of DimmStore.

The second difference is the EAM implementation. Specifically, the way of evicting data from the hot to the cold zone. In DimmStore, multiple threads run an algorithm that checks the LRU lists of data in their system region. If the

memory pressure in the system region exceeds a limit, then the thread will evict 64 KiB of data from the LRU list to the cold region of memory. The threads do this while the database remains available for requests. In our EAM, we evict data to the cold area if there is no room to allocate memory in the hot area. This is a blocking operation, potentially raising the latency for requests. While the EAM has this performance bottleneck, caused by its single-threaded nature, the memory power consumption should not be affected greatly by whether a worker thread evicts data to the cold memory or if the main thread does so. In general, our implementation is not equivalent to DimmStore as it is completely different code bases. However, we have made the EAM following the basic principles of DimmStore, and we use similar parameters for data management. Thus, we determine that the differences between the EAM and DimmStore are not significant enough to be the cause of the substantial difference in power savings between them.

The last difference is the hardware used for the benchmark. DimmStore is run on a server, where our benchmark is run on a desktop computer. Therefore, there might be a difference in the memory controller, which is the hardware that controls when memory goes into low power states. In our benchmark it seems from the power usage that the single 16 GiB DIMM is not put into low power state, whenever the system only uses the two 8 GiB DIMMs. This might be due to the strategy the memory controller uses to put DIMMs into low power states.

While we do not get the power reduction we had hoped for, the performance overhead of separating the data is smaller than expected. In DimmStore they have a performance overhead of 2%. Our benchmark has an overhead of 0.33% when separating the hot and cold data in the performance benchmark. Additionally, if the kernel is misused by only allocating to the hot zone, we can limit the performance overhead to 0.37%, if the overflow option is enabled in the kernel. Our overhead is significantly lower than expected, to the point of barely affecting the run time in our performance run. The overhead might increase once the memory hardware correctly enters low power states more of the time to save memory power consumption.

6.5 Relevance

The aim of this project is to allow programmers to save power in memory by modifying the OS kernel to manage memory energy efficiently. The potential power savings are high on systems with large amounts of memory, as described in [4] and [15].

In the benchmarks we have run on the desktop machine, we do not see the same results as are achieved in [4], even though we are following their test setup. The most significant difference is the hardware we use to run the tests, where they use a server machine. It seems that the server machine has a better control over memory interleaving and the power states in the DIMMs than the desktop computer. Thus, using our modified kernel is mostly relevant in a server context. As servers consume a lot of energy globally in data centres [2], it is highly relevant to make server-specific improvements to the Linux kernel to achieve power savings.

When developing means to save power on a server, Linux is a good choice to work with, as Linux is the most used OS kernel on servers [5]. Additionally, as x86 based processors continue to be popular in data centres [20], the focus on the x86_64 architecture is relevant.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The goal of this thesis is to research how the Linux kernel can be extended to support more energy efficient memory management. The main contribution is that we have modified the Linux kernel to allow the memory hardware more time in the power saving state. To do this, the memory is separated into two parts. The two parts are hot and cold memory, where hot memory consists of data that is accessed frequently, and cold memory consists of data that is infrequently accessed. Given that the hardware of the machine supports it, the DIMMs that contain cold data can enter low power states more often and, thereby, decrease the power consumption of memory.

Our main change to the kernel is the introduction of the cold zone. The cold zone takes up some of the memory that is normally reserved for the normal zone on a x86_64 architecture machine. The zone integrates well with the existing infrastructure for physical memory allocation in the kernel such that normal memory allocations are separated from the cold memory. No functionality in the memory system, such as swapping to the disk during high memory pressure, has been altered or removed. For processes that allocate cold memory, it is

transparent whether the memory request actually ends up being allocated in hot or cold memory.

We provide the system user with several options to adjust how our modified kernel behaves in terms of memory separation. These options can be adjusted when compiling the kernel. The user can even fully exclude our modifications, when compiling the Linux kernel, if the user so wishes.

To take advantage of the memory separation in the kernel, we make it possible to request memory allocations in cold memory via an additional flag that is to be passed to the mmap system call. The addition is backwards compatible, such that the usage of the flag on kernels without hot and cold memory separation has no effect and, thus, works as regular memory requests.

We test the modified kernel to ensure that hot memory and cold memory correspond to separate DIMMs in memory hardware. This ensures that the two types of memory in the kernel are completely separated in the hardware, allowing the memory hardware with cold data more time in low power states. To perform this test, we use the benchmarking tool STREAM, modified to utilise memory allocated with our mmap system call. We obtain results that show that we are able to affect memory interleaving in the way we expect by the placement of the memory DIMMs. We also see that we are able to allocate data in different parts of physical memory as intended, with our kernel modifications.

Additionally, a proof of concept in-memory database, that separates hot and cold data with our updated mmap system call, is implemented. This is to test how the power usage and the run time performance are affected in the modified kernel. In these tests, we do not observe any reduction in the memory power usage. We attribute this to the memory power management strategy of the memory controller in our test machine. In a machine that allows memory to enter low power states when unused, the energy consumption of main memory can be reduced with our modified kernel. The tests also reveal that the run time overhead of managing the memory separation is $0.3\% - 0.4\%$, which is significantly smaller than the 2% that we have seen in the existing literature. While the run time is excellent, the overhead might increase when the memory has to transition more between the power states.

7.2 Future Work

In this thesis, we have laid the foundation to support energy efficient memory with the Linux kernel. This section suggests multiple areas to expand on this, such as making the modified kernel more user friendly, optimising performance, and investigate more use cases.

7.2.1 Other Interfaces

As mentioned in Section 3.3 on page 24, there are multiple options for creating interfaces to ease the interaction between the programmer and the memory management required to save power. We have implemented the mmap system call, which is one of the lowest level ways of managing memory in a program. Thus, to make the modified kernel more user friendly to the programmers developing programs to run on it, other interfaces are relevant.

Malloc

Malloc is easier to use than mmap for memory allocation, as it keeps track of the size of the different allocations, such that the programmer just needs to use the `free` function to deallocate. Malloc also has some strategies to reduce the cost of allocating memory. First of all, Malloc uses both the mmap and sbrk system calls. The sbrk system call is good to allocate a small amount of data, in extension of your current data, where mmap is better when allocating larger or separate data areas. Secondly, Malloc sometimes allocates more memory than needed or waits with releasing the freed memory to the kernel. This allows Malloc to handle small memory allocation requests, without having to perform a context switch to the OS with a system call.

Runtime Environments

To completely remove the memory allocation and management from the user, a runtime environment interface to the modified kernel can be made. One example of this is to modify the .NET core JIT-compiler, such that the memory management of the generational garbage collector is performed in an efficient way. Modifying a runtime environment to work with the modified kernel, allows all programs developed for that platform to save memory. Furthermore, programs would not even have to be ported to be used energy efficiently with this approach, as they merely must be run in a different runtime environment.

Automatic Management of Hot and Cold Data in the Kernel

The last approach to ease the memory management for programmers is to built the memory management of hot and cold data directly into the kernel. This way, it is not necessary to change anything about the programs running on the system.

To allow the kernel to automatically divide the memory into hot and cold, we need to know which data is used frequently and which is used infrequently. Fortunately, the kernel is already keeping track of which memory pages are used infrequently, as these can be swapped to disk under high memory pressure. While this seems promising, the issue with the LRU list of the kernel is that the list contains pages from all zones. This is an issue, as pages from, for instance, the DMA zone should not be moved to the cold zone. In general, it is important to be careful when moving memory around, as not all pieces of memory are allowed to be moved. Additionally, all of the memory, that is already in the cold zone, will most likely be a part of the LRU list, as this should be some of the least accessed data. Thus, the LRU list will most likely be filled with data that is not possible to evict to the cold zone, meaning that using it for this purpose, while a seemingly good idea, most likely will not work, as it covers all zones. An alternative to this is to implement a LRU list for each zone, which can be used to evict data from a given zone, if this is allowed for the zone.

Now that we have discussed how to evict hot data to the cold zone, we also need to be able to move back data from the cold zone to the hot zone, if it becomes more frequently accessed. This is difficult, as the kernel does not keep track of how many times a page of memory is accessed. Our initial idea is to use a TLB miss on the page, to move the page back to the hot area with some probability. However, this is not possible in the x86 architecture, as TLB misses are hardware managed [32, 33]. Thus, it is necessary to find some other strategy for moving hot pages away from the cold zone.

While solving the problems of moving data to and from the cold zone, it is also important to consider how often the kernel moves data, as moving the data too much will create a large overhead. It is important that the overhead from moving data around is minimal and any increase of CPU energy consumption should be met with a bigger decrease in memory energy consumption. Thus, while this is a clever solution that requires very little of the programmers, it

is difficult to make a good implementation in the Linux kernel to manage the memory in a way that allows the hardware to save energy.

7.2.2 NUMA

Early in the project, we choose to delimit the project from the NUMA architecture, which is mostly used on servers. Thus, making the modified kernel capable of saving power in memory on the NUMA architecture is another option to expand on this project.

NUMA is an architecture, where the computer has multiple CPUs that access different parts of memory at different speeds. Thus, the goal with NUMA is to make the CPUs mostly access the memory that are close to them, rather than the memory that is far away [12, pp. 297–298]. This is managed in the kernel by dividing the memory into nodes. Each node has a zone structure, the same way as a non-NUMA system. In fact, when disabling NUMA, the Linux kernel views the system the same way as a NUMA system that only has one node. Thus, since we have implemented an extra zone, supporting NUMA requires that we add this zone to each node. This way, each node has hot and cold memory to store hot and cold data.

7.2.3 Dynamic Settings

Currently the size of the cold zone and the policy of allowing memory overflows to the cold zone are set at compile time. Work could be done to allow these parameters to be set when booting the kernel via kernel boot parameters. This would allow the user to e.g. set the size of the cold zone when rebooting their machine rather than having to recompile the kernel for each change. Some modifications would be needed to the `zone_sizes_init` function, as it currently uses constants to determine the maximum sizes for the normal zone. Likewise, a kernel parameter could be used to determine whether memory should be allowed to overflow to the cold zone. This would require a run time check for the parameter instead of currently only including the code, depending on the parameters set at compile time. We determine that an extra if-check is insignificant, as this code is run in the slow path of the memory allocation algorithm. Changing the option of overflowing to the cold zone can even be set when the system is running, as it only requires changing a variable. This

is significantly harder for regulating the cold zone size, as it would require the system to redistribute the memory space between the zones.

7.2.4 Portability

While the x86 architecture is the dominating architecture in the world of servers [20] it may not stay that way for ever. To accommodate changes in the server market, more architectures could be supported, as the absence of the possibility to save power in memory, which may act as an additional obstacle in transitions to different server architectures. For architectures with a zone structure, as x86_64, this should be similar changes to those of this thesis. Though, architectures with different memory representations than zones might require more comprehensive changes.

Bibliography

[1] Rui Pereira et al. "Energy Efficiency across Programming Languages". In: (2017). DOI: 10.1145/3136014.3136031. URL: https://doi.org/10.1145/3136014.3136031.

[2] George Kamiya. *Data Centres and Data Transmission Networks*. 2020. URL: https://www.iea.org/reports/data-centres-and-data-transmission-networks (visited on 25/03/2021).

[3] Raja Appuswamy, Matthaios Olma and Anastasia Ailamaki. "Scaling the memory power wall with dram-aware data management". In: *Proceedings of the 11th International Workshop on Data Management on New Hardware*. 2015, pp. 1–9.

[4] Alexey Karyakin and Kenneth Salem. "DimmStore: Memory Power Optimization for Database Systems". In: *Proc. VLDB Endow.* 12.11 (July 2019), pp. 1499–1512. ISSN: 2150-8097. DOI: 10.14778/3342263.33422629. URL: https://doi.org/10.14778/3342263.33422629.

[5] Steven J. Vaughan-Nichols. *Can the Internet exist without Linux?* Oct. 2015. URL: https://www.zdnet.com/article/can-the-internet-exist-without-linux/ (visited on 31/05/2021).

[6] Linus Torvalds. *Linux 5.11*. 2021. URL: https://github.com/torvalds/linux/releases/tag/v5.11 (visited on 05/03/2021).

[7] Alan Mycroft. *Programming Language Design and Analysis motivated by Hardware Evolution*. Aug. 2007. URL: https://www.cl.cam.ac.uk/~am21/papers/sas07slides.pdf (visited on 25/03/2021).

[8] Ulrich Drepper. "What Every Programmer Should Know About Memory". In: 2007.

[9] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. 0.80. Arpaci-Dusseau Books, 2014.

[10] Jean-Pierre Lozi et al. "The Linux Scheduler: A Decade of Wasted Cores". In: *Proceedings of the Eleventh European Conference on Computer Systems*. EuroSys '16. London, United Kingdom: Association for Computing Machinery, 2016. ISBN: 9781450342407. URL: https://doi.org/10.1145/2901318.2901326.

[11] Adrien Mahieux. *Memory Management: From Silicon to Algorithm*. 2018. URL: https://www.slideshare.net/Saruspete/memory-management-112860641 (visited on 25/03/2021).

[12] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. 3rd ed. O'Reilly Media, Inc., 2006. ISBN: 0596005652.

[13] Jonathan Corbet. *Memory power management*. 2011. URL: https://lwn.net/Articles/446493/ (visited on 26/03/2021).

[14] Bruce Jacob, Spencer W. Ng and David T. Wang. *Memory Systems: Cache, DRAM, Disk*. 1st ed. Denise E.M. Penrose, 2008. ISBN: 9780123797513.

[15] Mark Gottscho. "ViPZonE: Exploiting DRAM Power Variability for Energy Savings in Linux x86-64". In: (Mar. 2014). DOI: 10.13140/2.1.4932.3204.

[16] John D. McCalpin. "Memory Bandwidth and Machine Balance in Current High Performance Computers". In: *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), pp. 19–25.

[17] Michael Kerrisk. *mmap(2) — Linux manual page*. Mar. 2021. URL: https://man7.org/linux/man-pages/man2/mmap.2.html (visited on 09/04/2021).

[18] Jonathan Corbet. *Memory compaction*. Jan. 2010. URL: https://lwn.net/Articles/368869/ (visited on 23/04/2021).

[19] Dai Clegg and Richard Barker. *Case Method Fast-Track: A Rad Approach*. 1st ed. Addison-Wesley, 1994. ISBN: 9780201624328.

[20] Timothy Prickett Morgan. *x86 Servers Dominate The Datacenter - For Now*. June 2015. URL: https://www.nextplatform.com/2015/06/04/x86-servers-dominate-the-datacenter-for-now/ (visited on 12/04/2021).

[21] Matthew Flatt. *Dynamic memory allocation*. Oct. 2018. URL: https://my.eng.utah.edu/~cs4400/malloc.pdf (visited on 23/04/2021).

[22] Microsoft. *Fundamentals of garbage collection*. Nov. 2019. URL: https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals (visited on 23/04/2021).

[23] Ubuntu. *ReleaseNotes*. Feb. 2021. URL: https://wiki.ubuntu.com/FocalFossa/ReleaseNotes (visited on 24/05/2021).

[24] Tom Bauer. *Dual channel mode for DDR, DDR2, DDR3 and DDR4*. 2019. URL: https://www.compuram.de/blog/en/single-dual-and-multi-channel-memory-modes/ (visited on 07/06/2021).

[25] Brian F. Cooper et al. "Benchmarking Cloud Serving Systems with YCSB". In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 143–154. ISBN: 9781450300360. DOI: 10.1145/1807128.1807152.

[26] Kenneth J. Christensen. *Zipfian*. Nov. 2003. URL: https://www.csee.usf.edu/~kchriste/tools/genzipf.c (visited on 20/05/2021).

[27] Masoud Kazemi. *How to generate Zipf distributed numbers efficiently?* Jan. 2018. URL: https://stackoverflow.com/questions/9983239/how-to-generate-zipf-distributed-numbers-efficiently (visited on 20/05/2021).

[28] Casper Susgaard Nielsen et al. "The Influence of Programming Paradigms on Energy Consumption". In: (2021). URL: https://projekter.aau.dk/projekter/da/studentthesis/the-influence-of-programming-paradigms-on-energy-consumption(145e96f3-b48b-443b-9b74-39c7e4aa9a85).html.

[29] kernel.org. *Working with the kernel development community*. URL: https://www.kernel.org/doc/html/v4.14/process/index.html (visited on 07/06/2021).

[30] Kashif Nizam Khan et al. "RAPL in Action: Experiences in Using RAPL for Power Measurements". In: *ACM Trans. Model. Perform. Eval. Comput. Syst.* 3.2 (Mar. 2018). ISSN: 2376-3639. DOI: 10.1145/3177754. URL: https://doi.org/10.1145/3177754.

[31] T. Ilsche et al. "Power measurements for compute nodes: Improving sampling rates, granularity and accuracy". In: *2015 Sixth International Green and Sustainable Computing Conference (IGSC)*. 2015, pp. 1–8.

[32] Benedict Brown et al. *Computer Organization and Design*. 2020. URL: https://www.seas.upenn.edu/~cis371/current/slides/12_virtual_memory.pdf (visited on 02/06/2021).

[33] Adarsh Patil. "TLB and Pagewalk Performance in Multicore Architectures with Large Die-Stacked DRAM Cache". In: *CoRR* abs/2002.01073 (2020). arXiv: 2002.01073. URL: https://arxiv.org/abs/2002.01073.

Appendix

A Complete Results

name	duration(s)	pkg(j)	dram(j)	dram(W)	temp(C)
030 ktps	3000.0	37353	3107	1.034	36.0
060 ktps	1500.0	20698	1596	1.064	38
090 ktps	1000.0	15134	1097	1.097	37.5
120 ktps	750.0	11599	846.3	1.128	42.5
150 ktps	600.0	9681	697.6	1.163	43.5
180 ktps	500.0	8383	598.0	1.196	40.5
210 ktps	427.9	7389	520.3	1.216	41
240 ktps	374.9	6471	464.5	1.239	42
270 ktps	332.9	5969	425.9	1.279	42.5
300 ktps	300.0	5394	395.7	1.319	41.5
08 GiB	300.0	4376	325.7	1.086	38.5
10 GiB	300.0	4584	323.6	1.079	39
12 GiB	300.0	4537	326.3	1.088	39
14 GiB	300.0	4247	325.4	1.085	38.5
16 GiB	300.0	4532	330.6	1.102	40.5
18 GiB	300.0	4489	334.9	1.089	40.0
20 GiB	300.0	4517	329.5	1.098	38.5
22 GiB	300.0	4487	327.7	1.092	38
24 GiB	300.0	4331	330.4	1.101	40
26 GiB	300.0	4612	328.1	1.094	40

Table A.1: The raw results from the tests with the original kernel

name	duration(ms)	pkg(µj)	dram(µj)	dram(W)	temp(C)
030 ktps	3000.0	37350	3102	1.034	37.5
060 ktps	1500.0	20870	1594	1.063	39
090 ktps	1000.0	14780	1094	1.094	40
120 ktps	750.0	11650	847.4	1.130	40.5
150 ktps	600.0	9447	691.7	1.153	40
180 ktps	500.0	8186	590.8	1.182	42
210 ktps	427.9	7178	518.7	1.212	42
240 ktps	374.9	6098	465.5	1.241	42
270 ktps	332.9	5919	425.1	1.277	43
300 ktps	300.0	5323	393.0	1.310	42.5
08 GiB	300.0	4412	326.7	1.089	40.5
10 GiB	300.0	4350	324.5	1.082	40
12 GiB	300.0	4477	327.0	1.090	40
14 GiB	300.0	4424	325.4	1.085	43
16 GiB	300.0	4506	329.5	1.098	40
18 GiB	300.0	4432	325.5	1.085	41
20 GiB	300.0	4429	328.8	1.096	40.5
22 GiB	300.0	4443	327.6	1.092	42
24 GiB	300.0	4481	331.9	1.106	42.5
26 GiB	300.0	4538	328.4	1.095	40.5

Table A.2: The raw results from the tests with the hot/cold kernel

name	duration(s)	pkg(j)	dram(j)	dram(W)	temp(C)
030 ktps	3000	36511	3099	1.033	38
060 ktps	1500	20501	1592	1.062	38.5
090 ktps	1000	14805	1094	1.094	42.5
120 ktps	750.0	11580	843.7	1.125	40
150 ktps	599.9	9331	695.9	1.160	44.5
180 ktps	500.0	7976	594.9	1.190	43
210 ktps	427.9	6997	523.4	1.223	41
240 ktps	374.9	6124	469.0	1.251	42.5
270 ktps	332.9	5745	425.5	1.278	41
300 ktps	300.0	5150	395.5	1.318	42
08 GiB	300.0	4171	326.2	1.087	41
10 GiB	300.0	4324	324.1	1.080	41
12 GiB	300.0	4228	325.9	1.086	40.5
14 GiB	300.0	4242	327.3	1.091	40
16 GiB	300.0	4456	329.8	1.099	40.5
18 GiB	300.0	4420	326.4	1.088	41.5
20 GiB	300.0	4463	327.8	1.093	39
22 GiB	300.0	4441	327.1	1.090	39
24 GiB	300.0	4487	330.3	1.101	39
26 GiB	300.0	4475	328.4	1.095	39

Table A.3: The raw results from the tests with the non-strict kernel that allows overflow of data from hot memory to cold memory