Leibniz Universität Hannover
Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur

Leibniz
Universität
Hannover

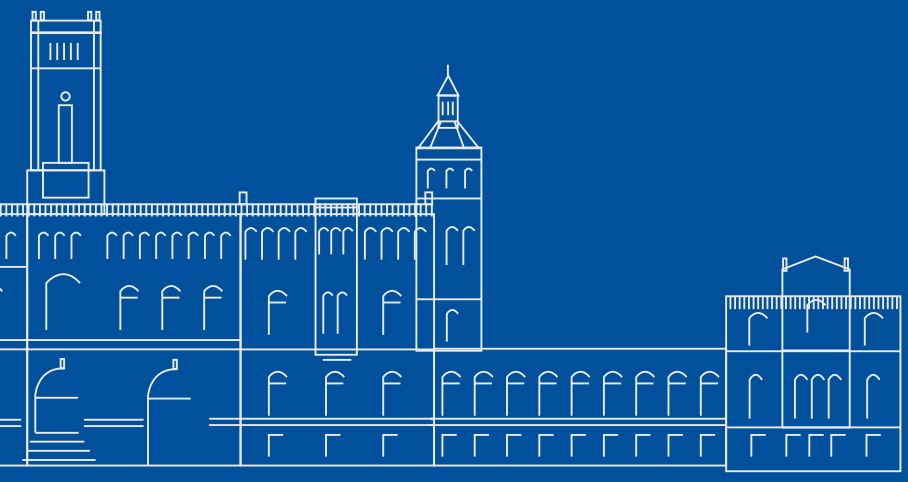
Illia Ostapyshyn

Enhancing Energy Efficiency with Advanced DRAM Management in Linux

Master's Thesis

November 1, 2024

Please cite as:
Illia Ostapyshyn, "Enhancing Energy Efficiency with Advanced DRAM Management in Linux"
Master's Thesis, Leibniz Universität Hannover, Institut für Systems Engineering, November 2024.



Leibniz Universität Hannover
Institut für Systems Engineering
Fachgebiet System und Rechnerarchitektur
Appelstr. 4 · 30167 Hannover · Germany

www.sra.uni-hannover.de

Enhancing Energy Efficiency with Advanced DRAM Management in Linux

Masterarbeit im Fach Technische Informatik

vorgelegt von

Illia Ostapyshyn

angefertigt am

Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur

Fakultät für Elektrotechnik und Informatik
Leibniz Universität Hannover

Erstprüfer: **Prof. Dr.-Ing. habil. Daniel Lohmann**
Zweitprüfer: **Prof. Dr.-Ing. Christian Dietrich**
Betreuer: **Alexander Halbuer, M.Sc.**

Beginn der Arbeit: **19. April 2024**
Abgabe der Arbeit: **1. November 2024**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Illia Ostapyshyn)
Hannover, 1. November 2024

ABSTRACT

In contrast to the early days of computing, memory is no longer a scarce resource. The capacity of dynamic random-access memory (DRAM) continues to successfully scale with ever-growing requirements of the applications. Modern server systems can house terabytes of DRAM, yet in the large-scale data centers, the average memory utilization does not exceed 70 percent. Especially in such systems, memory is a major contributor to the overall power consumption, prompting the question of whether it is possible to deactivate unused memory to conserve energy. However, neither hardware nor software offer sufficient support to realize these energy savings and millions of systems continue to expend energy on maintaining unused memory.

On the hardware side, the existing power-saving modes are only applicable at the large granularities of ranks and channels (>8 GiB). This is slowly changing with the LPDDR5 standard introducing the PARC feature and bringing the power management granularity to the sub-rank level of around 1 GiB. On the software side, contemporary operating systems are still designed around the notion of memory scarcity. They primarily manage memory in 4 KiB pages and operate under the assumption that unused memory is a wasted resource. Over time, the available memory is filled with file cache and used memory inevitably becomes scattered across all memory devices. Consequently, it becomes impossible to find unused contiguous segments of memory for deactivation in any modern system with considerable uptime. In a nutshell, the possibility of memory power management contradicts the foundational assumptions of the conventional memory management.

This work approaches the lack of power-saving mechanisms from the systems software perspective. It demonstrates that also Linux suffers from poor memory management, as the memory quickly becomes unsuitable for deactivation and remains in this state even if the utilization declines. To tackle this issue, the thesis proposes a novel compaction mechanism designed with DRAM power management in mind. Applied to real-world workloads, it successfully increases the amount of unused memory segments and reduces the power consumption of a desktop system under heavy load by up to 19.1 mW. These savings scale quickly when applied to numerous systems with overprovisioned memory worldwide. Ultimately, this work is the first to reveal that it actually pays off to actively reorganize memory contents with the goal of energy savings: the energy invested in a single compaction procedure is recovered in just under one minute.

KURZFASSUNG

Im Gegensatz zu den frühen Tagen der Rechnertechnik ist Speicher heute keine knappe Ressource mehr. Dynamic Random Access Memory (DRAM) skaliert erfolgreich mit den ständig wachsenden Anforderungen der Anwendungen. Heutige Serversysteme können Terabytes an DRAM enthalten, doch in den großen Rechenzentren liegt die durchschnittliche Speicherauslastung unter 70 Prozent. Gerade in solchen Systemen trägt der Speicher wesentlich zum gesamten Stromverbrauch bei. Dies wirft die Frage auf, ob es möglich ist, ungenutzten Speicher zu deaktivieren, um Energie zu sparen. Allerdings bieten weder Hardware noch Software ausreichende Unterstützung, um diese Energieeinsparungen zu realisieren. Somit verschwenden Millionen von Systemen weiterhin Energie für ungenutzten Speicher.

Auf der Hardwareseite sind die vorhandenen Stromsparmechanismen nur auf die große Granularität der Ranks und Kanäle (>8 GiB) anwendbar. Dies ändert sich mit dem LPDDR5-Standard, der die PARC-Funktion einführt und die Granularität der Energieverwaltung auf die Sub-Rank-Ebene von etwa 1 GiB reduziert. Auf der Softwareseite sind die heutigen Betriebssysteme immer noch auf Speicherknappheit ausgelegt. Sie verwalten den Speicher hauptsächlich in 4-KiB-Seiten und betrachten ungenutzten Speicher als eine verschwendete Ressource. Im Laufe der Zeit wird der verfügbare Speicher mit dem Datei-Cache gefüllt und der verwendete Speicher wird unvermeidlich über alle Speicherbausteine verstreut. Folglich wird es in einem modernen System mit langer Betriebszeit unmöglich, komplett ungenutzte Speichersegmente zu finden, um sie zu deaktivieren. Schlussendlich steht die Möglichkeit der Speicher-Energieverwaltung im Widerspruch zu den grundlegenden Annahmen der konventionellen Speicherverwaltung.

In dieser Arbeit werden fehlende Stromsparmechanismen aus der Perspektive der Systemsoftware betrachtet. Sie zeigt, dass die Speicherverwaltung auch in Linux suboptimal ist. Der Speicher nimmt schnell einen Zustand an, wo jegliche Deaktivierung verhindert wird, und behält ihn bei, selbst wenn die Auslastung sinkt. Um das Problem zu lösen wird in dieser Arbeit ein neuartiger Mechanismus zur Speicherkompaktifizierung speziell für DRAM-Energieverwaltung entwickelt. Angewandt auf reale Arbeitslasten erhöht er erfolgreich die Menge der ungenutzten Speichersegmente und reduziert den Stromverbrauch eines Desktop-Systems unter schwerer Last um bis zu 19,1 mW. Diese Einsparungen skalieren schnell, wenn der Mechanismus auf eine Vielzahl von Systemen mit überprovisioniertem Speicher weltweit angewendet wird. Diese Arbeit zeigt zum ersten Mal, dass es sich lohnt, Speicherinhalt zur Energieeinsparung aktiv zu reorganisieren. Die in einem einzigen Kompaktifizierungsvorgang investierte Energie wird in weniger als einer Minute zurückgewonnen.

CONTENTS

1 INTRODUCTION

In the early days of computing, memory was a scarce resource. The main memory was typically implemented as a magnetic-core memory with magnetic ring cores threaded onto wires that can be read or written by applying current [1]. Although the magnetic-core memory provided adequate speeds for the processors of that time, it was costly to produce, limiting its maximum capacity. This limitation led to the introduction of two-tiered storage systems, which divided the memory into fast, low-capacity *primary* core storage and a slow, higher-capacity *secondary* drum storage. The secondary storage enabled execution of programs with working sets larger than what the primary storage could offer. However, this new capability came at the cost of significantly complicating algorithms. Accessing secondary storage required special routines and the logic to load data from the secondary into the primary storage had to be incorporated directly into the algorithms.

This changed with introduction of *virtual memory* [2, 3]. The idea of the virtual memory is to abstract the complete storage resources of the system in a single address space. The secondary storage can be mapped into this address space and accessed by programs as if it were primary storage, without complicating their logic. This abstraction is supported by the operating system, which now transparently implements the exchanging routines. It loads data from the secondary into the primary storage on demand and evicts it when the primary storage becomes full.

Inadvertently, virtual memory—specifically its first [3] and most common implementation, *paging*—solved[1] another problem: *external fragmentation*. External fragmentation occurs when memory is allocated in chunks of different sizes. Over time, as these chunks are allocated and freed, a once large and contiguous area of free memory inevitably becomes divided into smaller, non-contiguous pieces. Eventually, an allocation request might fail as there is no contiguous free chunk of the required size in the memory, even though the total free memory—if combined—would be sufficient to fulfill the request.

The technique of paging solves the problem of external fragmentation by defining a base unit of memory management, the *page*. A page is typically 4 KiB in size. The virtual memory provides a translation mechanism, which allows every page in the virtual address space to be mapped to a contiguous *page frame* in the *physical address space*. As a consequence, the exact locations of the page frames in the physical memory become irrelevant. Even if scattered arbitrarily throughout the memory *physically*, the translation mechanism is able to present them as a contiguous region *virtually*. This greatly simplifies the memory management of the system.

The elegance of paging has led to the 4 KiB page becoming a standard for sharing memory between the OS, applications, and even external devices. Without the need to manage memory chunks of different sizes, the OS memory management has evolved with two assumptions: *"all page frames are equal"* and *"only used memory is good memory."* With no concern for fragmentation, the OS lacks initiative to keep memory unallocated and trades it for performance by filling it with the cache of the secondary storage (also known as *page cache*). The page cache remains in memory until the memory shortage occurs, at which point it can be easily evicted.

[1] Or rather transformed it into the problem of internal fragmentation, which is not as severe.

The return of external fragmentation

The first challenge to these assumptions came as soon as the early 1990s, when the original Pentium processor introduced the Page Size Extension (PSE) feature. PSE extended the virtual memory with support of larger 4 MiB pages that can coexist with the regular 4 KiB pages [4]. This feature persisted, and modern systems now support 2 MiB *huge* and 1 GiB *giant* page frames. The usage of these larger page frames puts less pressure on the translation mechanism and can potentially improve application performance. However, their introduction brings back the problem of external fragmentation, giving the OS a new reason to have free contiguous blocks of size larger than 4 KiB at its disposal. Consequently, there is a substantial body of research analyzing benefits of huge pages and proposing mechanisms for their management [5–11].

By the end of 1970s, the magnetic core memories were phased out in favor of their more efficient semiconductor counterparts [12]. The primary memory today is implemented as *dynamic random-access memory (DRAM)*, which has excelled in scaling its density to meet the ever-increasing demands of applications. This memory is not a scare resource anymore: the price per gigabyte approaches the value of $1 [13]. Meanwhile, modern server CPUs support up to 24 DRAM modules and such modules can reach 256 GiB in size, yielding 6 TiB of memory in total [14]. These factors make it economically feasible to equip the system with extra capacity, in case it is occasionally required for a demanding task.

Nevertheless, the substantial amounts of DRAM in use come with significant running costs. As each new generation doubles the device capacity, the energy consumption rises correspondingly [15]. Many studies attribute over 30 percent of the total energy consumption in data centers to DRAM [16–18], and this figure is expected to steadily increase. Meanwhile, the average memory utilization remains as low as 40 – 70 percent [19–21]. To handle usage spikes, memory in these systems is often overprovisioned, leading to excessive capacity that goes unused most of the time. A significant share of DRAM's power consumption is independent of its usage: up to 20 percent are required just to retain the stored data [15]. While it is theoretically possible to deactivate parts of DRAM not in use, neither software nor hardware provide adequate support for this functionality. Consequently, a substantial amount of power is wasted on maintaining memory that is not actively used.

The missed opportunity for energy savings can be traced back to outdated assumptions that shaped the memory management of the operating systems as we know it today. Modern DRAM devices still offer very few power-saving techniques, applicable only at granularities constrained to specific internal levels of DRAM organization. These granularities are significantly larger than the 4 KiB page frames managed by the operating system. This reintroduces the problem of fragmentation once again: for DRAM deactivation, large contiguous memory segments must be free of useful data. Operating systems are largely blind to fragmentation (*"all page frames are equal"*), resulting in data being scattered throughout the entire physical memory and thus all memory devices. Furthermore, as the system is running, the page cache continuously grows in size (*"only used memory is good memory"*). Consequently, in a modern system with considerable uptime, finding an unoccupied DRAM segment that can be deactivated becomes impossible.

Contributions of this work

While the underlying reasons for the insufficient support of DRAM power management are deeply rooted in the foundational aspects of contemporary memory management, this work does not aim to redesign it from the ground up; such task would out of scope for a master's thesis. Instead, it proposes a mechanism to revert the damage done by suboptimal memory management. This is achieved by keeping track of the DRAM segments in the system and providing a mechanism to proactively

clear them from used memory. This implements a technique known as *compaction*, where multiple partially used memory chunks are merged into one by migrating their memory contents.

Unlike the existing compaction approaches [11, 7, 8], the mechanism proposed in this thesis is designed specifically with DRAM power management in mind. It supports large granularities of over 1 GiB that are necessary for existing DRAM power-saving modes. Moreover, it incorporates a cost-benefit model, that allows it to assess whether the energy spent on cleaning memory segments will be offset their deactivation. To facilitate this assessment, the mechanism scores each segment based on the effort required for its reclamation. This scoring also enables strategic selection of segments for merging that yields the best gains for the minimum effort. Finally, it shrinks the amount of the page cache in the system as part of its operation: since the page cache continually expands and fills all available memory, a mechanism for its earlier eviction is essential for effective DRAM power management.

The thesis consists of five chapters. Following this introduction (Chapter 1), Chapter 2 provides the theoretical foundation necessary to understand the architecture of the proposed mechanism for managing physical memory. Chapter 3 provides a detailed description of the mechanism, starting with a conceptual overview and followed by a description of its implementation in the Linux kernel. In Chapter 4, the mechanism is evaluated in terms of potential energy savings and the performance impact. Finally, Chapter 5 concludes this work and discusses potential areas for future improvement.

2 FUNDAMENTALS

This chapter provides the theoretical foundation necessary for understanding this thesis. It opens with an overview of DRAM technology in Section 2.1, introducing the internals of DRAM chips and discussing their refresh mechanisms and power-saving modes. The second half of the chapter, Section 2.2, deals with the operating system's memory management, focusing specifically on the intricacies of the Linux memory subsystem. It discusses disk caching, memory reclamation, and memory migration features within Linux. Finally, Section 2.3 closes the chapter with an overview of existing research that intersects memory management with power management.

2.1 Dynamic Random-Access Memory

If we examine the components of a general-purpose computer, we will discover a spectrum of different memory technologies. These technologies can generally be graded based on two key qualities that tend to be inversely related: the access time and the price per bit. The latter economic quality also translates to another important characteristic, capacity. As the price per bit decreases, it becomes feasible to use devices with larger capacities. As a consequence, the memory devices in the system range from slow high-capacity memory, such as mechanically rotating hard drives, to fast low-capacity memory, like CPU caches using static random-access memory (SRAM) cells.

This section delves into the technology that serves as a bridge between the two extrema of the speed-capacity spectrum: the *main memory* based on DRAM technology. The designation "main memory" comes from its role in the system—it holds the working set of all applications. The DRAM capacity has managed to scale with ever-increasing memory requirements of applications. For instance, DDR5 modules up to 128 GiB are commercially available [22], with the standard allowing up to 256 GiB per module [23]. Multiple such modules can coexist in a system.

However, the scaling success story of DRAM capacity does not extend to its latency. In the early 1990s, the performance gap between CPUs and DRAM began to widen significantly [24]. This necessitated two drastic changes. Firstly, CPU caches had to grow in size to conceal the high latency of DRAM accesses [24]. Secondly, the DRAM interface started evolving to accommodate high bandwidth demands of larger CPU caches [25]. Unlike early asynchronous DRAM, contemporary devices implement a synchronous (SDRAM) interface, which offers high predictability and enables high throughput via pipelining [26].

Table 2.1 provides an overview of latency and bandwidth of different SDRAM generations. A DDR4 SDRAM takes about 13 ns for a memory access in the best case and up to three times as much in the worst case [27]. Given a 3 GHz CPU clock, this translates to 40 – 120 clock cycles per memory access. For a multicore CPU generating at least one memory reference per core per cycle, such latency is impractical. The CPU cache allows SDRAM to compensate its high latency with its high bandwidth: up to 25.6 GiB/s per channel for DDR4, or over 8 B per clock cycle at 3 GHz. A real system will require much smaller bandwidth due to data and code locality.

Year	Standard	Latency [ns]	Module bandwidth [GiB/s]
2002	DDR1 [28]	15 – 45	3.2
2006	DDR2 [29]	10 – 30	6.4
2010	DDR3 [30]	13 – 39	10.6
2016	DDR4 [31]	13 – 39	21.3

Table 2.1 – Latencies and bandwidth of different generations of DRAM memory [32]. While the latency has remained constant since over 20 years, the bandwidth doubles with each generation.

2.1.1 Standardization

A major contributor to DRAM's success is its interoperability. The dual in-line memory modules (DIMMs) are the most widespread consumer form of DRAM. These modules, which are just PCBs with DRAM chips and a standard connector, can be purchased off the shelf and installed into a computer's memory slot to improve its performance. This plug-and-play experience is made possible by standardization.

For SDRAM, the standard-governing body is JEDEC Solid-State Technology Association. JEDEC defines three main categories of double data rate (DDR) standards, each tailored towards specific application:

- Main DDR [33] targets laptops, desktops, and servers. The devices in this category come in two form-factors: as a chip soldered onto a PCB or as a replaceable DIMM module.
- Low-power LPDDR [34] devices are found in smartphones, cars, and other embedded applications and include additional power-saving features. Unlike the main DDR standard, LPDDR does not define a DIMM form-factor and these chips are always soldered directly onto the PCB.
- Graphics GDDR [35] standard targets high data bandwidth applications like GPUs and High-Bandwidth Memory (HBM).

While the high level of standardization enables easy interoperability, it is also believed to obstruct innovation [27]. New standards are released once every 5 to 8 years and the process of their ratification is not transparent. It is likely highly influenced by the major DRAM market leaders: Samsung, SK Hynix, and Micron, which collectively control 96.5 percent of the market [36]. The internal workings of DRAM chips remain highly confidential and the body of research on microarchitecture improvements is largely based on speculations [27].

2.1.2 Information Storage

At the core of the DRAM technology lies a cell consisting of a single capacitor and an access transistor. The bit of information is stored in the cell's capacitor in the form of the electric charge. These cells are arranged in a two-dimensional array known as *mat* with horizontal *wordlines* and vertical *bitlines*. Figure 2.1 demonstrates the structure of this array. The wordlines control the gates of the transistors of the whole row. When the wordline is pulled high, the transistor's drain-source channel electrically connects each capacitor to its respective bitline. This results in several peculiarities.

As the bitline is a long conductor that spans across numerous rows of the mat array, it exhibits a high capacitance. When the bitline is connected to the cell's capacitor—a process known as *row opening*—the charge equalization takes place, resulting in a small voltage change on the bitline.

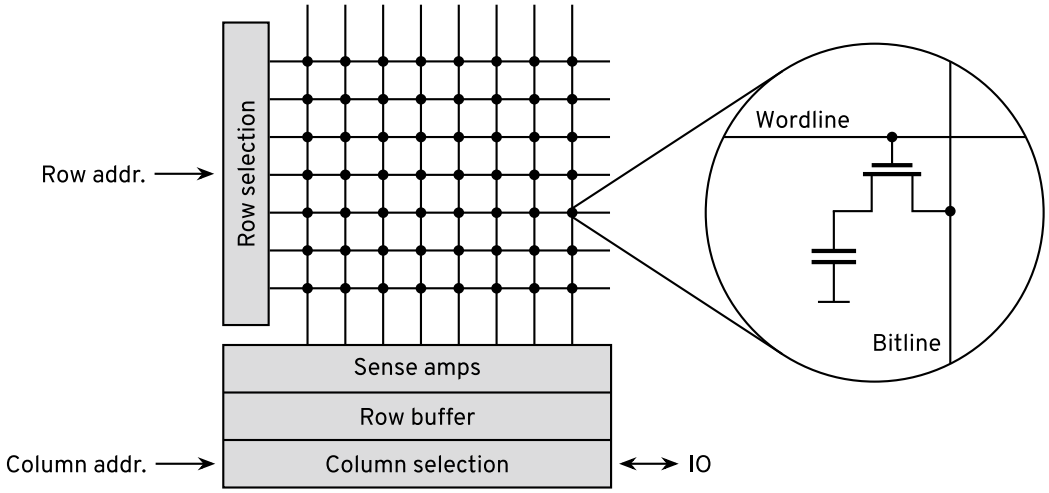


Figure 2.1 – The DRAM cells consisting of a transistor and a capacitor arranged in a two-dimensional *mat* array [26]. The array is spanned by horizontal wordlines and vertical bitlines.

With a typical 1:6 ratio of cell to bitline capacitance [38], the voltage change on the bitline is under 15 percent.

To interpret this small voltage change, a mat contains a *sense amplifier*. Before the row is opened, the sense amplifier *precharges* the bitline to a voltage level that is midway between logical 0 and logical 1. Doing this allows the sense amplifier to compare the new voltage level of each bitline with the well-defined precharge voltage after the row is opened. The result of this comparison for each bit of row is stored in the *row buffer*.

Note that the charge equalization destroys the information in form of the charge in the capacitor. The capacitor (dis)charges approximately to the precharge voltage when the row is opened. To avoid data loss, the DRAM chip must write the contents of the row buffer back into the cells before another row is accessed. This is achieved by charging each bitline to the voltage representing the

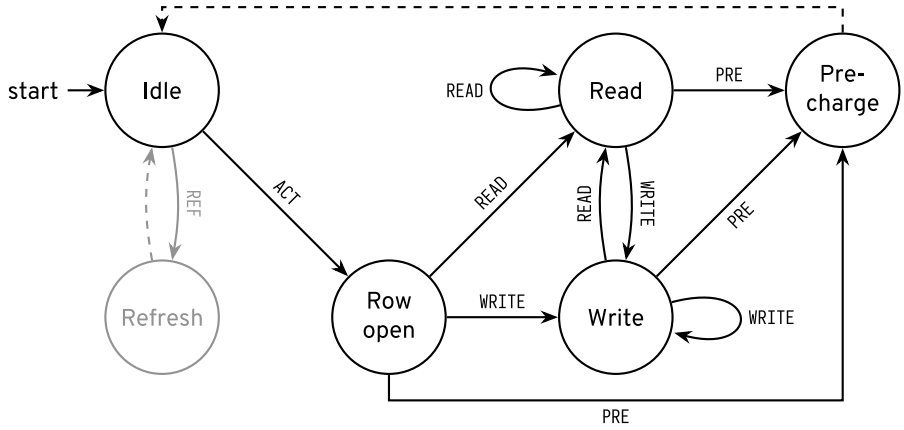


Figure 2.2 – Simplified state diagram of the chip in the SDRAM protocol [37]. Accessing a cell requires loading its whole row into the row buffer using `ACT`, allowing subsequent `READ` and `WRITE` commands to operate on the contents of the row buffer. The `PRE` command writes the row-buffer contents back into the cells and precharges the bitlines for the next `ACT`.

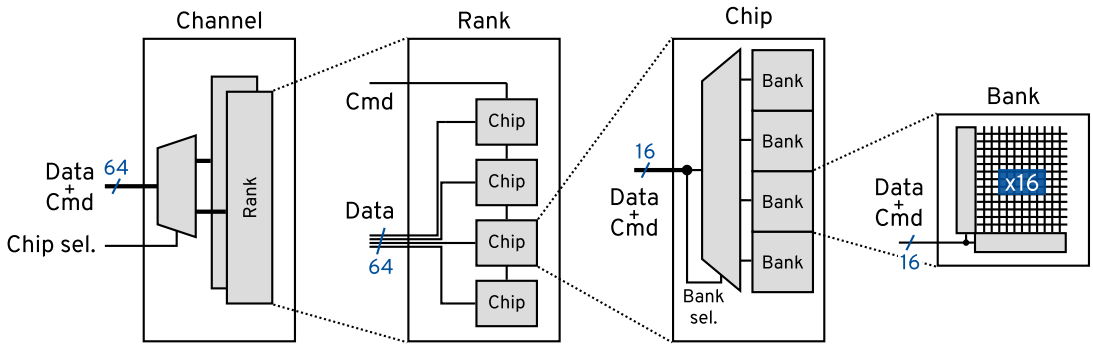


Figure 2.3 – The hierarchy of the DRAM memory system. In this example, a single channel contains two ranks. Each rank contains four chips, each with four banks inside.

bit stored in the row buffer and activating the wordline. Furthermore, due to the imperfections of the physical implementation, the charge in the capacitors leaks over time. Accordingly, retaining the data requires reading and rewriting (*refreshing*) each row periodically. Section 2.1.4 discusses the specifics of the refresh mechanism in detail.

The SDRAM command interface reflects these characteristics of the mat's operation. Figure 2.2 represents a simplified state diagram of data accesses in SDRAM. In the idle state, the bitlines remain precharged. The `ACT` command initiates the access by selecting a row and transferring it into the row buffer. The individual bits of the open row are then accessed by providing the column address to the `WRITE` and `READ` commands. Before accessing a different row, the protocol demands issuing of the `PRE` command. This writes the contents of the row buffer back into the cells and precharges the bitlines for the next `ACT`. Additionally, the refresh of the mat can be requested from the idle state by sending the `REF` command.

2.1.3 Hierarchy

A hierarchical design is essential to enable high capacity and high throughput characteristic to SDRAM. Figure 2.3 provides an overview of the memory system hierarchy with the signal routing. This hierarchy can be divided into two categories: *horizontal* levels that broaden the memory system's data width, and *vertical* levels that expand its address space. The mat level, introduced in the previous section, typically falls under the horizontal category. During a memory access, the row and column addresses of the respective commands pinpoint a specific cell in the mat, yielding a single bit of information. However, a single read operation from a DRAM chip can produce more than one bit of data, depending on its *column width*. The SDRAM column width is represented using notation x*N* and generally ranges from x4 to x16 [26]. The broad data bus is achieved by stacking multiple mats and addressing them simultaneously, with each mat contributing one bit of data.

The second, vertical, hierarchy level within a chip is the bank level. *Banks* are memory arrays that are addressed independently of each other, with the *bank address* being part of the command. Their independence comes with a performance bonus: it introduces *bank-level parallelism* into the chip's operation [39]. While a single bank is busy processing a command, another bank can be addressed for the next memory access. By interleaving bank accesses through clever address mapping [40–42] or strategic data partitioning [43, 44], the memory system's throughput can be improved significantly. The maximal amount of banks in a chip is defined by the standard, ranging from only four banks in the initial DDR standard [28], to 16 banks in the latest LPDDR5 [34] and

to 32 in DDR5 [33]. DDR4, DDR5, and LPDDR5 further optimize performance by splitting banks into groups and relaxing the timing requirements for memory accesses to different bank groups.

Multiple memory chips that share the command/address bus and the chip select signal collectively form a *rank*. The data pins of the chips within a rank are combined to form a wider data bus. An example of a DRAM rank would be one side of a module's PCB. In consumer modules, the width of the rank data bus is 64 bits (72 bits on RAM with error correction codes), requiring 4 x16, 8 x8, or 16 x4 DRAM chips [45].

The memory controller connects the CPU to the DRAM subsystem and arbitrates CPU's memory accesses. It can have multiple independent channels (recent AMD EPYC processors feature 12 channels [14]), with one or more ranks in each. The chip select signal, shared by all chips within a rank, is used to activate a specific rank in a multi-rank channel. Similar to banks, ranks are independent of each other and introduce a level of parallelism into the memory subsystem [46, 47]. In contrast to bank-level parallelism, *rank-level parallelism* has another trade-off dimension to it. Discrete chips on a PCB require long interconnects transferring data at a high rate. Many ranks in a single channel represent an increased load on the shared transmission lines (e.g., command/address buses connecting to each chip), affecting the maximum operational frequency [26]. *Registered DIMMs (RDIMMs)* mitigate this issue by buffering the signals and reducing the electrical load on the memory controller, allowing more modules per channel.

2.1.4 Refresh Mechanisms

As established in Section 2.1.2, the DRAM cells require periodical refresh to retain data due to inherent leakage currents in the cell. For this, the DRAM standard mandates that each cell must be refreshed once in the *refresh window* $t_{\mathrm{REFW}} = 64\,\mathrm{ms}$. For temperatures over 85 °C, the window is halved to 32 ms. In fact, the majority of DRAM cells hold their contents over much longer time spans [48]. For instance, Baek and associates [49] successfully reduce the refresh rate to 512 ms by identifying weak cells and removing up to 0.1 percent of respective page frames from the operating system's page-frame pool. Nevertheless, the refresh window figure has been chosen conservatively to account for the highest leakage currents occurring due to manufacturing process variations. Furthermore, the cells have also been observed to change their retention period over time [50].

During normal operation, the refresh cycle is managed through the *Auto-Refresh (AR)* mechanism. As the state diagram in Figure 2.2 illustrates, the memory controller issues the `REF` command to initiate the refresh process. Refreshing the whole chip at once is costly, so the refresh window is split into 8192 *refresh intervals*, each lasting $t_{\mathrm{REFI}} = 7.8\,\mathrm{us}$. To ensure data retention, the memory controller must issue at least eight refresh commands within the $8 \cdot t_{\mathrm{REFI}}$ period [31]. Technically, a single refresh command refreshes $N_{\mathrm{rows}}/8192$ rows in a round-robin order. The SDRAM device contains an internal counter of refreshed rows that advances with each refresh command [15].

The duration of the refresh command, also known as t_{RFC}, depends on the device's memory density. For DDR4, it ranges from 160 ns with 2 Gib devices to 550 ns with 16 Gib devices [31]. Thus, the whole rank is not available for $t_{\mathrm{RFC}}/t_{\mathrm{REFI}}$ (7 percent for 550 ns) of the time and the memory accesses have to stall. Consequently, refresh comes with a performance penalty and its real cost can be even higher due to the requirement to precharge all banks before refresh and the subsequent row reactivation after refresh.

Selective refresh with existing hardware

Until recently, none of the SDRAM standards provided a possibility to selectively refresh portions of memory during normal operation. One way to work around this limitation, is to stop issuing AR

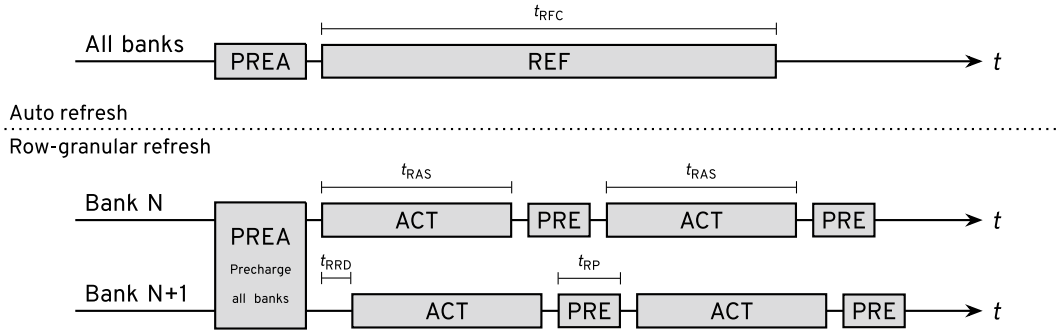


Figure 2.4 – Commands issued during Auto-Refresh and row-granular refresh [51] and their timing constraints.

Symbol	Value [ns]	Name	Description
t_{RAS}	32	Row Address Strobe	Duration of the ACT command
t_{RP}	13.75	Row Precharge	Duration of the PRE command
t_{RRD}	4.9	Row-to-Row Delay	Minimum gap between ACTs to different banks
t_{FAW}	21	Four Activation Window	Timespan in which no more than four activations are allowed to limit peak current

Table 2.2 – Timing parameters of Micron MT40A2G8VA-062E at 3200 MT/s [52].

commands altogether and simulate it by activating the row and then immediately precharging the bank, as shown in Figure 2.4. This has the effect equivalent to a refresh: reading the row into the row buffer and then writing it back again, without the IO transfer overhead.

However, the comparison of REF and ACT+PRE latencies shows the disadvantage of this approach. Consider a DDR4 16 Gib x8 SDRAM device **Micron MT40A2G8VA-062E** [52] operating at the highest data rate. It contains 2^{17} rows, each 1 KiB wide. Thus, a single refresh command handles $2^{17}/8192 = 16$ rows in all 16 banks at a time and takes $t_{RFC} = 350$ ns to complete. On the other hand, to perform the equivalent ACT+PRE sequence, the memory controller must respect the timing parameters shown in Figure 2.4 and Table 2.2. Even ignoring the four activation window t_{FAW} this requires twice as much time:

$$\left(N_{banks} - 1\right) \cdot t_{RRD} + \frac{N_{rows}}{8192} \cdot \left(t_{RAS} + t_{RP}\right) = 805.5\,\text{ns} \tag{2.1}$$

Although explicit row opening achieves selective refresh on the granularity in the ballpark of memory paging (explained in Section 2.2), it appears exceedingly inefficient. This hints that the internal implementation of the Auto-Refresh is highly optimized and the optimized mechanisms are not exposed through the SDRAM interface—at least when operating within the specification.

Mathew *et al.* [51] investigate this experimentally and discover that by reducing the SDRAM timing parameters (i.e., violating the specification) the overhead of row-granular refresh can be reduced without the loss of reliability. The DDR3 device under test has 16384 rows and thus refreshes two rows per refresh command, which takes 262.5 ns to complete. Two equivalent ACT+PRE issued for each bank in parallel (2 rows × 8 banks = 16 commands in total) require 292.5 ns according to the specification. With relaxed timing, the duration of the manual refresh reduces to 146.25 ns, 44 percent less than Auto-Refresh. The energy figures are promising as well: while the optimized

row-granular refresh still uses more energy than a full AR, it is enough to omit refresh for just 10 percent of the memory to break even. Due to the out-of-spec operation, the success of this technique may vary from device to device and relies on low amount of rows.

LPDDR2+ standards provide a *per-bank refresh* command REFpb. Initially, this command refreshed a single bank in a round-robin manner with an internal bank counter. In LPDDR4 and later, the command encoding reserved a few bits to specify the bank address, allowing the memory controller to schedule bank refreshes in an arbitrary order. Similarly, DDR5 gained a *same-bank refresh* command REFsb, allowing refresh of bank groups in any order. However, the standard forbids issuing this kind of refresh command to the same bank (or bank group in DDR5) unless all other banks (bank groups) have been refreshed, making selective refresh on the bank granularity using these commands illegal. Perhaps also here energy and performance improvements can be achieved by violating the standard. To the best of my knowledge, no works have attempted this yet.

Finally, the LPDDR5 [34] standard introduces Partial-Array Refresh Control (PARC), a mechanism previously described in a 2017 Qualcomm patent [53]. Both documents advertise the new feature as a way to reduce refresh power consumption. PARC splits banks into eight segments and uses an 8-bit bitmap register to mask the refresh operation within the specific segment of all banks, providing a way to reduce refresh power consumption during normal operation applicable on the 1/8 rank granularity.

Proposed selective refresh mechanisms

The first suggested but never actually implemented technique for selective refresh of DRAM, *Selective Refresh Architecture (SRA)* [54], dates back to 1998. It proposes introducing a bitmap that allows toggling refresh per row. Cui *at al.* [55] survey hardware- and software-based refresh reduction techniques and use the results to design their *DTail* mechanism. Extending SRA, their proposal features a 4-bit register for each row that can be used to either disable refresh altogether or to extend the refresh window beyond the standard 64 ms. These mechanisms come with significant memory and chip area overhead and are thus unlikely to be adopted by future standard versions.

Seeking to modify the hardware as little as possible, *Flexible auto-refresh (REFLEX)* [56] utilizes the fact that DRAM chips include an internal counter of recently refreshed rows that is advanced on each refresh command. The authors propose exporting this counter to the memory controller along with a *"dummy refresh"* command that increments the counter without performing any actual refresh operations. In a similar vein, Jafri and colleagues propose *Partial Array Auto-Refresh (PAAR)* [57], a refresh reduction technique in two variations: (1) the bank-granular version with minimal hardware modifications resembling PARC and (2) a version with a register containing an address range that will be refreshed on auto refresh. Their primary contribution is the algorithm to bypass refresh of accessed rows in an application with a predictable access pattern (e.g., a convolutional neural network). Indeed, as memory accesses have a side effect of refreshing the row, such rows need not be refreshed if accessed frequently. Utilizing this fact, *Smart Refresh* [58] introduces counters into the memory controller to track recent accesses and skip the refresh accordingly.

2.1.5 Power-Saving Modes

SDRAM devices feature several power-saving modes that can be entered during periods of inactivity. The *Self-Refresh (SR)* mode disables most of the clock circuitry and the external IO. In this mode, data is retained by internal timers triggering refresh without the intervention of the memory controller. To reduce the standby current further, data retention can be disabled for specific memory regions by setting the *Partial-Array Self-Refresh (PASR)* register before entering Self-Refresh. Notably, the same segment mask is utilized for both PASR and PARC in LPDDR5. Although the PASR feature is

mandatory in all LPDDR standards, it was optional in DDR2 and DDR3. The DDR4 standard does not include PASR and it is officially deprecated in DDR5 due to security concerns [59], which suggests that its low-power sibling PARC is unlikely to appear in the future DDR standards.

When examining power-saving modes and in particular the hierarchy levels they apply to, it is important to consider the *on-die termination (ODT)*. The memory subsystem includes long off-chip interconnects as traces on a PCB. When transferring signals at a high rate, the interconnect has to be analyzed as a transmission line. Any nonuniformities within the impedance of the transmission line cause reflections and distort the signal, affecting the maximum frequency [60]. To combat this challenge, the DRAM chips include a configurable (34 to 240 Ω or Hi-Z when off in DDR4) impedance on the die. While ranks can independently enter any power-saving modes, some of the modes (including Self-Refresh) disable ODT. This makes their application at this level impractical: by disabling the termination of a single rank, the whole channel suffers from resulting reflections and the transfer rates plummet.

Another power-saving mode disabling ODT is the DDR4's *Maximum Power Saving Mode (MPSM)*. In principle, MPSM is a variation of Self-Refresh without any refresh activity. Thankfully, DDR5 introduces a variation of MPSM that retains ODT and is applicable at the rank level. Moreover, all DDR and LPDDR standards support idle *Power-Down* mode that is entered when the *clock enable (CKE)* signal is asserted low. This mode keeps the ODT and exhibits very quick entry and exit latencies (7.5 ns). As a result, it can be exited periodically to send refresh commands and retain data. In fact, some standards limit the maximum duration of Power-Down to the refresh interval [34]. Table 2.3 presents a summary of power-saving modes along with entry/exit latencies and estimated potential power savings for an exemplary DDR5 memory system featuring eight 16 Gib **Micron MT60B4G4** devices in a rank.

2.2 Operating Systems

Modern computers of any scale—from smartphones to supercomputers, and even some embedded systems—rely on *virtual memory*. Virtual memory allows abstracting the address space of the main memory of the system into one or several *virtual address spaces*. The modern implementations of virtual memory use *paging*. With paging, the physical address space is divided into equally-sized page frames (typically 4 KiB). In the exact same way, the virtual address space is divided into pages.

Power mode	Granularity	Power saving [mW/GiB]		Latency	DDR 1 2 3 4 5	LPDDR 1 2 3 4 5
Full ch. power off	Channel	82.8	(100 %)	>25 ms	☑ ☑ ☑ ☑ ☑	☑ ☑ ☑ ☑ ☑
Self Refresh	Channel	0.53	(0.6 %)	640 ns	☑ ☑ ☑ ☑ ☑	☑ ☑ ☑ ☑ ☑
MPSM	Channel	22.95	(27.7 %)	640 ns	☑ ☑ ☑ ☑ ☑	□ □ □ □ □
MPSM (DDR5)	Rank (16 GiB)	15.97	(19.1 %)	21.5 ns	□ □ □ □ ☑	□ □ □ □ □
Power-Down	Rank (16 GiB)	14.6	(15.6 %)	7.5 ns	☑ ☑ ☑ ☑ ☑	☑ ☑ ☑ ☑ ☑
PASR	1/8 Rank (2 GiB)	0.53 – 22.95	(0.6 – 27.7 %)	640 ns	□ ☑ ☑ □ □	☑ ☑ ☑ ☑ ☑
PARC	1/8 Rank (2 GiB)	0 – 9.19	(0 – 11.1 %)	14 ns	□ □ □ □ □	□ □ □ □ ☑

Table 2.3 – Power-saving modes of all standards with their granularity, estimated power savings, and latency for a 16 Gib Micron MT60B4G4 [61].

Using a translation mechanism, like a page table, the dedicated hardware *Memory Management Unit (MMU)* maps the virtual pages to physical page frames transparently on each memory access.

The details of the translation mechanism are not essential here; instead, we outline its key characteristics. Firstly, the translation mapping is **dynamic**: it can be modified at runtime or completely swapped out for another one. By switching the mapping on program scheduling, the OS provides each application with its own virtual address space. Incidentally, this results in **isolation**, as the applications can only access memory that has a mapping in their virtual address space. This guarantees that programs do not interfere with each other's data unless explicitly allowed via a shared memory mechanism.

Secondly, virtual memory effectively **conceals fragmentation** of physical memory. Even if the application's physical page frames are scattered throughout the main memory, the translation mapping allows the OS to present them at an arbitrary virtual memory location in any order, or even as a contiguous memory region. And lastly, the operating systems use virtual memory to implement other **virtualization** techniques. For example, this allows the OS to defer memory allocations until they are actually needed or to transparently present other storage media as memory. These techniques are implemented by providing a *page-fault handler*, a routine that gets called when the process accesses unmapped memory.

2.2.1 Page-Frame Allocator

To manage physical page frames, the operating systems utilize a *page-frame allocator*. Its task is to manage the pool of unused and available page frames and to provide them to OS subsystems or applications on demand. Once the requesting thread finishes using the page frame and *frees* it, the allocator adds the released page frame back to its reserves. A common allocator design, also employed by the Linux kernel, is the *buddy allocator* [62, 63].

The buddy allocator maintains several lists of free contiguous naturally-aligned blocks, one list for each block size. The block sizes are limited to power-of-two multiples (*orders*) of the basic unit of memory management: a single page frame. Thus, there are separate lists for contiguous blocks of sizes 4 KiB (2^0 frames), 8 KiB (2^1 frames), 16 KiB (2^2 frames), and so on. In Linux, the maximum tracked order is 10 or $2^{10} \cdot 4\,\text{KiB} = 4\,\text{MiB}$ in size.

When a contiguous memory block of the specific order is requested, the buddy allocator first checks the corresponding list for an available block. If the list contains a free block of the requested size, it is removed from the list and returned to the caller. Should the list contain no such block, the allocator proceeds to check the list for the next higher order. This ascending process continues until a free block larger than the requested size is found. Once a suitable block is located, it is isolated from the list and the *splitting* begins.

Having isolated the block larger than the requested size, the allocator splits it into two equally-sized *buddies*. One of the buddies is placed on the next lower list corresponding to its halved size and the other buddy is used for further splitting. If the block is still bigger than requested, the process is recursively repeated until it reaches the desired size. At that point, the memory block is returned to the caller and all its former buddies reside on the respective lists. The *merging* is the reverse operation of splitting. When a block is freed, the respective list is checked for its buddy. If the buddy is present (i.e., together they form a naturally-aligned contiguous block), they are melted together into a single next-order block. The lists are recursively ascended until no melting is possible and the resulting block is stored on the list.

Figure 2.5 illustrates the working principle of the buddy allocator by example. In this case, a zero-order page frame is requested. Queries on zero-, first-, and second-order lists return no suitable

blocks ❶, only the third-order list returns a block consisting of eight page frames ❷. The block is split into two four-frame blocks, with one half put back on the second-order list ❸, and the other half used for further splitting. The process is repeated for the four-frame block ❹ and the two-frame block ❺. The result is a single page frame, which is returned to the caller ❻.

Pageblocks

By maintaining page frames as contiguously as possible, the buddy allocator proves to be effective against *external fragmentation*: scattering of blocks throughout the physical address space [64]. However, based on the observation that single page frames are much more frequently requested than higher-order blocks, Linux optimizes such allocations by employing *per-CPU page-frame caches* [65]. Each such cache contains some page frames that are instantly returned for page-frame requests by the local CPU core. In Linux 5.13, the cache was extended to other common orders. Unfortunately, this optimization eliminates the defragmenting character of the buddy allocator by delaying merging operations and complicating defragmentation heuristics [66, 10].

Huge frames or order 9 (2 MiB) blocks have a particularly high value for the memory management. Along with 4 KiB page frames, they can be used for virtual address mappings on all major architectures [67, 68]. Such mappings put less pressure on the address translation mechanism and exhibit better performance [5]. To counteract the effect of the per-CPU caches and attain huge frames, the Linux kernel introduced *pageblocks*. In a nutshell, the pageblock mechanism assigns a *migrate type* to each huge frame in the system. The migrate type categories indicate the mobility of the memory contained in the huge frame:

- `MIGRATE_RECLAIMABLE` if it can be freed quickly,
- `MIGRATE_MOVABLE` if it can be moved,
- `MIGRATE_UNMOVABLE` for immovable memory, like kernel allocations,
- and several others reserved for special cases.

The Linux buddy allocator maintains separate free-block lists for each order and for each migrate type. Keeping track of this information and separating the free lists enables the Linux kernel to place new allocations (which indicate their planned mobility) in the respective pageblocks, grouping allocations with the same mobility together. This grouping makes active defragmentation described in Section 2.2.3 more effective.

Nodes and Zones

The Linux kernel memory management supports Non-Uniform Memory Access (NUMA). NUMA systems feature different access latencies to different regions of memory, depending on the CPU core that performs the access. For example, the CPU may access its local memory quickly, but it

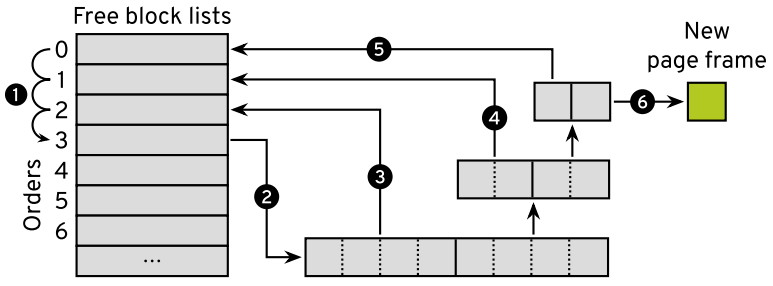


Figure 2.5 – Example allocation of order 0 with the buddy allocator. The algorithm finds no suitable blocks in lists for orders 0, 1, and 2. The order 3 block is split thrice to acquire a single page frame.

also might access memory of another CPU with additional performance penalty. In such systems it is desirable to pin applications and their memory to a particular CPU and its local memory, where it benefits from fast accesses. To implement this, Linux splits memory into *nodes*.

The nodes are split once again into *zones*. Each zone hosts its own allocator instance with its own free lists and bookkeeping. This division enables allocations with special physical address requirements. For example, `ZONE_DMA`/`ZONE_DMA32` contain the first 16 MiB/4 GiB of memory and are used to allocate memory shared with devices that have that kind of addressing constraint. The rest of allocations fall into `ZONE_NORMAL`. The optional `ZONE_MOVABLE` zone, which is disabled by default, splits the node into another region reserved for movable allocations, making compaction (Section 2.2.3) even more effective. Immovable kernel allocations are not allowed in the movable zone and are directed to other zones. On the other hand, movable allocations may fallback to other zones if the movable zone is full [69].

2.2.2 Disk Caching

The concept of virtual memory was initially developed in response to the limited memory resources available in the early days of computing [1]. With low amount of memory at hand, the programmers who needed to use more memory than the primary, directly-addressable storage could provide had to design their algorithms accordingly. This involved tedious swapping of sections of program data in the primary storage with sections located in the secondary storage (e.g., disk). Wishing to relieve the algorithm designers of this burden, the OS and hardware designers came up with a technique that makes accesses to the secondary storage transparent [3].

While the primary storage no longer poses a significant constraint for most program workloads, virtual memory is still used to abstract secondary storage. When a user program accesses a file on a disk for the first time, the OS loads the file contents into RAM and adds it to the *page cache*. Subsequent file accesses then utilize this cached copy, allowing for faster access times. The cache pages can also be mapped into the program's virtual memory mapping, virtualizing disk contents as a regular memory region. If the program modifies the file, the changes are not propagated to the disk immediately. The *writeback* is deferred in case the data is modified again in the near future. The page is said to be *dirty* if it contains unwritten content. Furthermore, even after the program has terminated or finished its file operations, the associated pages remain in the page cache in case the file is opened again.

This design is based on two assumptions: (1) disk accesses are prohibitively slow, and (2) free memory is wasted memory. Indeed, the access latency for hard drives typically falls within the millisecond range [70, 71]. Incurring such latency on each file access would render the system unusable. However, the advent of SSDs, which offer random-access latencies in microseconds [72] challenges this assumption. Moreover, unlike hard drives, SSDs can service multiple parallel requests efficiently, resulting in much higher throughput. In fact, the current Linux memory management subsystem, originally designed for HDDs, has been identified as too slow to fully utilize the high bandwidth offered by modern SSDs [73]. The second assumption that unused memory is a *stranded asset* [74] is reinforced by the fact that none of the power-saving modes described in Section 2.1.5 are utilized on consumer systems. This is partly due to the lack of support from operating systems and memory controllers for these features.

Page-frame reclamation

With the memory usage growing constantly due to disk caching, the memory reserves eventually deplete. Thus, the OS requires a mechanism to evict some of the pages to make room for new allocations. The Linux page-frame reclamation algorithm (PFRA) is commonly referred to as the LRU

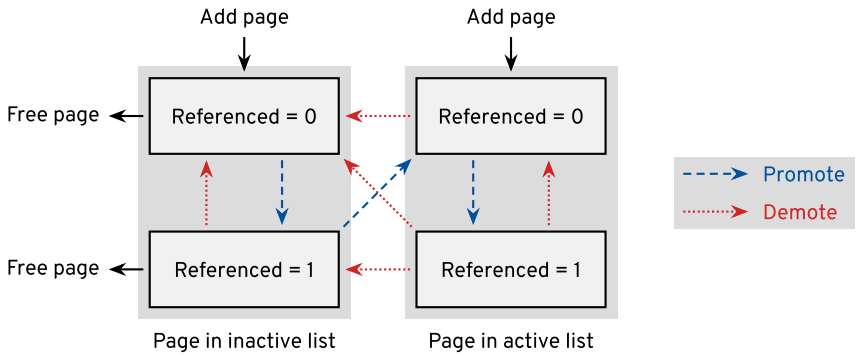


Figure 2.6 – The states assigned to pages within the Linux page-frame reclamation algorithm.

(least recently used) mechanism. However, this is not strictly true, as Linux does not maintain an LRU-ordered list of page frames [75]. Instead, Linux implements a simplified version of the 2Q algorithm [76].

The algorithm differentiates between two page types: anonymous (program memory not associated with a file) and file (page cache). For each page type, the algorithm maintains two lists per page type: the *active* and the *inactive* list. Under memory pressure, the lists are iterated and all the pages are checked for references. To check the page for references, the kernel utilizes the *reverse mapping*, which returns all virtual address spaces that map the page. In the mapping structures of the virtual address space, the MMU provides the *accessed* bit that is set transparently by hardware when a memory access to the page is detected.

By checking and resetting the accessed bit in each mapping, the kernel can determine whether the page has been accessed by the user process since the last scan. This information is then used to either promote the page to the active state, or to demote it to the inactive state. When the page is already inactive and has not been referenced, it is freed. To add some inertia to this mechanism, the page frame descriptor (`struct page`) stores another *referenced* bit, representing whether the page has been classified as accessed on the last scan. Figure 2.6 demonstrates the states that the page can assume and the possible transitions.

2.2.3 Page Migration

The internal Linux kernel interface provides functions for memory migration. These functions can be used to move contents of a set of source page frames into another set of destination page frames. More importantly, the migration functions also take care of all the mappings referencing the source page frames, allowing migration of pages that may be concurrently used by applications. Apart from the NUMA support, the two major subsystems making use of the migration infrastructure are *compaction* and *hotplugging*.

Compaction

As briefly mentioned in Section 2.2.1, contiguous naturally-aligned blocks of physical memory are a valuable resource to the memory management subsystem of the OS, as they can be used for huge 2 MiB frames. Moreover, as the kernel primarily uses a fixed identity mapping of the physical memory, all its internal allocations have to be contiguous. As shown in Figure 2.7, over 90 percent of the kernel allocations fall within the 16 – 32 KiB range and allocations as big as the maximum size of 2^{10} page frames or 4 MiB are not uncommon [66].

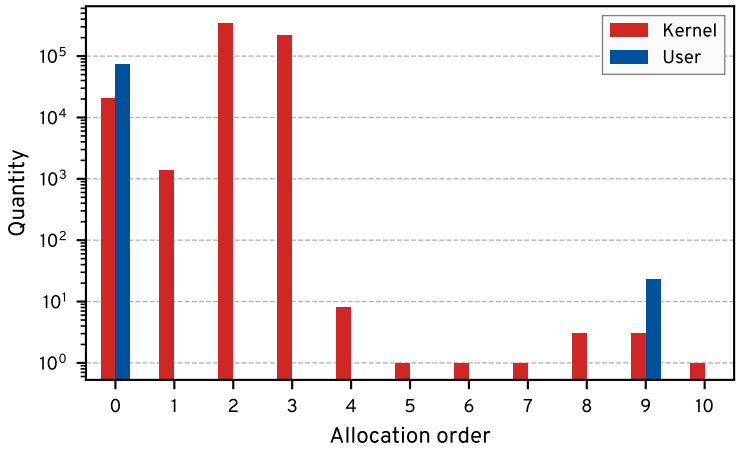


Figure 2.7 – Requested allocation sizes (logarithmic) during system startup and a 120-second Memcached benchmark [66].

To defragment the memory and gain such contiguous blocks, Linux employs active compaction. The compaction can be triggered either manually by the user, proactively by the kernel, or when an allocation of a large size fails [77, 78]. Figure 2.8 demonstrates the algorithm by a simple example. At the beginning, two iteration cursors are placed: one at the beginning of the zone and the other at the end of the zone. Then, iterating through the pages at the beginning of the zone, the algorithm accumulates movable pages that it encounters in the *movable list* ❶. As optimization, compaction checks the migrate type assigned to the pageblock (4 MiB) and skips it completely if it indicates immovable pages. Meanwhile, the second part of the algorithm works from the end of the zone, moving toward lower addresses, and adding free page frames to the *free list* ❷. Eventually the two cursors meet or enough movable pages have been collected to satisfy the failed allocation. At that point, the migration routine is called to migrate all the movable pages into the accumulated free page frames ❸.

The presence of kernel pages makes compaction less effective. Unlike user pages, which can be relocated by adjusting the mapping structures of the processes, kernel pages are generally immovable. This immobility renders the entire block unreclaimable to the defragmentation algorithm. Although the Linux kernel does have a mechanism for movable kernel pages [79], it requires explicit

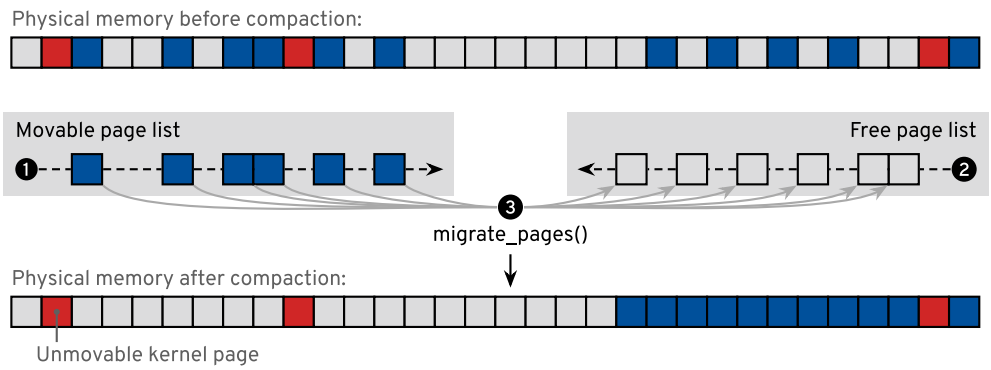


Figure 2.8 – The Linux compaction algorithm. The algorithm collects movable pages from the beginning of the zone and migrates them into free page frames at the end of the zone.

support from the subsystems. As of Linux 6.1, only two drivers (z3fold and zsmalloc) provide the necessary callbacks to relocate their pages.

Memory Hotplugging

Linux supports *memory hotplugging* [80], enabling removal of memory from the system at runtime. To avoid system crashes or data corruption, the administrator must ensure that no data currently in use by the OS or user applications resides on the memory device (e.g., a DIMM) before physically disconnecting it from the system. The hotplugging infrastructure facilitates this, providing an interface to *offline* specified memory regions.

Technically, memory hotplugging divides the memory node into contiguous blocks of fixed size depending on build-time configuration. On x86-64, these blocks are typically 128 MiB in size. Each block is presented as a separate device in `/sys/devices/system/memory`. If the user requests offlining by writing into the respective file, the kernel first iterates through the pageblocks (see Section 2.2.1) in the range and sets their migrate type to one of special types, `MIGRATE_ISOLATE`. As each migrate type has its own set of free lists within the buddy allocator, this has the effect that concurrently freed pages within the range end up on the `MIGRATE_ISOLATE` lists. These lists are never used by the buddy allocator.

Having isolated the pageblocks, the mechanism iterates through the individual page frames in the range. If the page frame is in use, it is migrated to a page frame outside the offlining range. In this case, the source page frame automatically ends up on the isolated lists of the buddy allocator after it is freed by the migration routine. The pages within the range that are already free are manually moved to the isolated lists. Finally, with all pages being removed from the allocator's pool, memory hotplugging adjusts the bookkeeping structures and returns. Memory *onlining* performs the reverse operation: it populates the allocator's free lists with the new page frames and sets all the pageblocks within the range to `MIGRATE_MOVABLE`.

Just like with compaction, the presence of an immovable kernel page within the range makes its offlining impossible. Because of this reason, the Linux documentation suggests enabling `ZONE_MOVABLE` (which is typically disabled) to increase the likelihood of successful offlining [80].

2.3 Related Work

As shown in Section 2.1.4, numerous works have raised the subject of energy-efficient memory management and proposed novel hardware mechanisms. However, a gap remains in the collaboration between the OS and the hardware to address this issue effectively. This section provides an insight into how different studies have attempted to tackle this challenge.

OS-based DRAM power management

ESKIMO [81] is the first work to propose disabling the DRAM refresh for unused memory regions. They base their work on *SRA* [54], a proposal to introduce a bitmap for toggling refresh on a per-row basis. To track unused memory regions, the authors modify the C standard library functions `malloc()` and `free()`. They simulate their approach using the DRAMsim simulator [82] and observe up to 86 percent energy savings.

Baek and colleagues [49] recognize that the OS already possesses the necessary information at row-level granularity. Consequently, they integrate their mechanism into the Linux kernel and evaluate it on single-board ARM computers, specifically the BeagleBoard [83] and PandaBoard. To circumvent the lack of selective refresh in existing hardware, they simulate row-granular refresh by (1) disabling the hardware Auto-Refresh and (2) introducing a software real-time thread

that refreshes used rows by performing reads on them. Given the inefficiency of this arrangement, they do not provide energy measurements. Instead, they demonstrate that selective refresh at a row granularity can reduce refresh operations by as much as 93.8 percent on average, and that the OS support at such fine granularities is trivial. Additionally, they propose to mask page frames containing weak DRAM cells in the OS, allowing for reduced refresh rates and thus demonstrating performance improvements.

GreenDIMM [84] proposes a deep power-down state in the DRAM at sub-array (128 – 512 MiB) granularity that comes with Linux support. The OS power manager monitors the memory utilization and employs memory hotplugging to clear randomly selected blocks before transferring them into the power-down state. Having no block selection policy, this technique does not account for the trade-off between the cost of offlining blocks and the potential energy savings achieved.

Fragmentation and its costs

There is an extensive body of research dedicated to maximizing the benefits provided by huge pages [9, 6–8, 85, 86, 11]. Much like with DRAM power management, the OS is faced with a dilemma whether it is worth investing processor cycles to construct a huge-page mapping *before* the future performance benefits can be known. *MEGA* [11] is a compaction mechanism that, unlike built-in Linux compaction, monitors both the occupancy levels of huge frames and the age of the pages within them. This information is then used to make a cost-benefit analysis before compacting. In a similar vein, Mansi *et al.* [87] introduce a policy into the Linux virtual memory management that employs an empirically-based cost-benefit model to determine whether the performance benefits of a huge-page mapping outweigh the associated costs.

While blocks larger than huge frames may not be as widely discussed, there are still some studies that emphasize their importance. On mobile systems, external devices such as the video camera may suddenly demand large contiguous blocks. To meet these requests, the prevalent approach is to reserve this memory in advance, resulting in resource underutilization. Seeking to make the reserved memory useful, the *rental memory* approach [88, 89] restricts the pages allocated within these reserved blocks to pages that can be quickly and easily evicted, such as page cache. Essentially, the device's memory is temporarily borrowed by the kernel to store quickly reclaimable data.

Alverti and associates [90] seek to alleviate address translation overhead in virtualized environments by employing large contiguous blocks. Their approach, termed as *contiguity-aware paging*, is a modification to demand paging that attempts to use physically contiguous pages for virtually contiguous mappings (VMAs). They have noted that this technique has a side effect of slowing down fragmentation development by physically grouping pages with similar lifetimes. The same effect is observed by Kim *et. al* [91], who implement memory management on mobile devices that groups pages with the same deallocation time in contiguous regions.

3 ARCHITECTURE

This chapter describes the implementation of the thesis. It starts by introducing the theoretical concepts behind the mechanism in Section 3.1. Then, it moves on to the technical details of the practical implementation in Section 3.2. Finally, Section 3.3 outlines the limitations of the implementation and proposes solutions to address them in the future.

3.1 Concept

When it comes to managing DRAM power, there is a substantial gap between software and hardware. As illustrated in Table 2.3, the power-saving modes supported by contemporary DRAM chips are restricted to coarse granularities exceeding 1 GiB. The mode with the finest granularity, PARC was introduced only recently in LPDDR5. Although systems incorporating LPDDR5 memory are now commonly available, current CPU memory controllers still do not provide software access to PARC. Not surprisingly, my extensive search has failed to yield any examples of PARC being applied in practice.

On the other side of the gap, operating systems lack any infrastructure that would facilitate usage of DRAM power-saving modes with coarse granularities effectively and the previous attempts to build such infrastructure were ultimately abandoned [92]. This situation presents a classic chicken-and-egg problem: hardware developers are reluctant to incorporate these features into DRAM and memory controllers without existing software support, yet such support is hindered by the absence of corresponding hardware features. The objective of this work is to tackle this issue from the software perspective by developing a comprehensive framework to support both existing and emerging DRAM power-saving modes within the operating system.

For brevity, I refer to DRAM segments capable of entering power-saving modes as *slices* and the process of transitioning them into a low-power state as *offlining*. More precisely, the required OS support would entail (1) the identification of unused slices, (2) communication with the memory controller to offline them, and (3) the possibility to clear partially used slices with the goal of saving energy. The first two requirements are a matter of introducing additional bookkeeping into the OS memory management and providing hardware-specific drivers. However, effectively implementing the third requirement is more complex; it necessitates evaluating the costs and benefits of the operation before proceeding.

Depending on the physical-to-DRAM address mapping, the slices may either be fully contiguous or consist of smaller contiguous segments distributed throughout the physical address space in a regular pattern. As a long as each slice maps to a subset of the physical memory consisting of complete page frames, there is a potential for energy savings. Given that the address mapping is determined solely by the memory controller, this work assumes a configuration that maps every slice to a single contiguous region in physical memory. It is reasonable to expect that if future memory controllers provide software access to slice power management, they will offer a (configurable) address mapping that makes this feature practical. Nevertheless, the presented concepts are just as well applicable to mappings with noncontiguous slices.

3.1.1 Understanding Fragmentation

Organizing data to retain large contiguous blocks of memory free is referred to as *defragmentation*. Most defragmentation efforts in the systems software research are directed towards supporting huge pages, both via passive *fragmentation avoidance* [90, 91, 9] strategies and active *compaction* [11, 7, 8] mechanisms. However, the passive allocation strategies that attempt to keep large contiguous blocks of memory free can at most *delay* the fragmentation. Eventually, the memory becomes cluttered and no allocation strategy can revert this process. To see why this is the case, it is important to understand the source of external fragmentation in main memory. It occurs when pages with different lifetimes get intermixed. As the short-lived pages get deallocated, holes in rather allocated regions appear. In an attempt to reduce fragmentation, the allocator fills these gaps with newly allocated—potentially long-lived—pages. If the memory utilization sinks, the short-lived pages eventually cease to exist but the long-lived pages remain scattered throughout the memory, rendering large contiguous blocks unusable for slice offlining or huge page-frame allocation.

An ideal allocation strategy clusters pages with similar lifetimes together. This approach would lead to large contiguous blocks of free memory following their nearly simultaneous deallocation. However, allocators lack information about the lifetimes of pages because neither user programs nor kernel subsystems provide such data at allocation. Assuming that programs do not interact with the outside world (e.g., user), retrieving this information generally requires solving the halting problem, as the last possible time at which an anonymous page is deallocated is the program termination. Without this information, the page-frame allocators can only work with heuristics, inevitably intermixing pages with different lifetimes to some extent. There is no free lunch in defragmentation: eventually it is up to compaction to actively revert the clutter via page migration.

3.1.2 Optimal Compaction

To achieve optimal compaction, the target block size must be known ahead of time. For instance, the Linux kernel compaction tends to invest excessive effort for the given gain in free blocks of the desired size. Its size-agnostic design enables simple implementation without requiring additional bookkeeping. The simplicity alleviates the extra cost, which stays within acceptable bounds for relatively fine granularities of 2 MiB. However, for the coarse RAM slice granularities, this approach is impractical.

Specifying the target size allows the compaction mechanism to analyze the memory utilization beforehand and calculate a score for each slice using a metric that reflects the effort required to reclaim it. Based on these scores, the algorithm can make informed decisions regarding two variables of memory migration: the source and the target. Aiming to maximize the benefit relative to the cost, the algorithm selects slices that require the least effort as the migration source. Conversely, the optimal target for memory migration are the *nonfull* slices that would require the most effort to free, as they are least likely to become the migration source. This approach results in the maximum fragmentation reduction for a given block size [61].

Page cache

In the context of compaction for DRAM power-saving, the page cache represents a double-edged sword. On the one hand, its pages remain allocated until they are evicted due to memory pressure. Thus, they are the prime example of the long-lived pages that fill the holes and get thereby scattered through the memory, causing fragmentation. On the other hand, unused (i.e., not mapped into applications) and clean (i.e., containing no unwritten changes) cache pages are easy to remove: they require no migration and can be simply discarded. Nevertheless, this removal comes with a

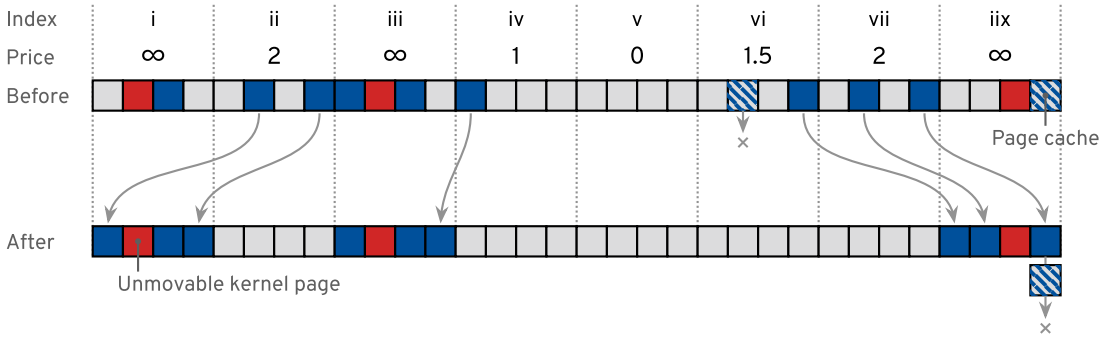


Figure 3.1 – The optimal compaction algorithm with special handling of cache pages. The memory is split into 4-page slices and each slice is assigned a reclamation price. The algorithm moves pages from the cheapest slices into the most expensive, freeing unused cache pages in the process.

caveat: if these pages are accessed again later, the removal incurred delayed costs associated with reloading the data from the backing storage. For slice offlining, however, shrinking the page cache is a necessity as it eventually fills the whole memory and prevents potential energy savings.

This distinction between different page types also complicates construction of the slice metric for optimal compaction. The hidden costs of page cache removal cannot be known ahead of time. Moreover, the cache pages can be freed in the target slice and thus become potential migration destinations, increasing compaction effectiveness. However, the single reclamation effort metric is no longer sufficient to optimally determine the target migration slice. The algorithm also needs to consider the cost and benefit of allocating pages in the destination slice. For instance, the destination slice containing n free page frames should be preferred to a slice containing n freeable cache pages: it provides equal benefit for less cost.

Figure 3.1 illustrates the algorithm on the example arrangement used in Figure 2.8 previously. The physical memory is divided into four-page slices and each slice is assigned a reclamation price. The price is equal to the amount of movable pages in the slice, while a freeable cache page counts as half a page. Slices that contain unmovable pages are assigned infinite price. The compaction reclaims four slices in total, requiring six migrations and two frees in the process. It also includes special handling of the page cache: the unused cache page in slice **vi** is freed instead of being migrated. Similarly, the cache page in the last slice **iix** is freed to accommodate a movable page from slice **vii**.

3.2 Implementation

The described algorithm is implemented in the Linux kernel version 6.1. Hereinafter, this implementation of the algorithm is called *Ramslice*. Overall, the changes span over 28 source files, all of which are in the memory management subsystem. The modifications in existing files are nonintrusive, minimal, and do not exceed few lines. The heart of the algorithm is self-contained in its own source file, which spans just over 1200 (+200 for the respective header file) significant lines of code.

Symbol	Name	Description
N_{free}	`RSI_FREE`	Amount of free page frames
N_{lru}	`RSI_V_LRU`	Amount of pages in all LRU lists (userspace pages)
N_{anon}	`RSI_ANON_MAPPED`	Amount of anonymous pages
N_{filemap}	`RSI_FILE_MAPPED`	Amount of file pages mapped into userspace
N_{fileall}	`RSI_FILE`	Total amount of file pages
N_{maps}	`RSI_MAPPINGS`	Total sum of userspace mappings for all pages

Table 3.1 – Per-slice statistics tracked by Ramslice. These counters are used to calculate the amount of freeable, movable, and unmovable pages in the slice and determine its reclamation price.

3.2.1 Data Structures

The first and foremost requirement for the effective slice offlining and compaction is defining the target slice size. In Ramslice, the slice size is a boot-time parameter passed via the kernel command line. This point in time is chosen to accommodate the second requirement: maintaining the necessary bookkeeping. The bookkeeping structures must be initialized before the page-frame allocator becomes active. In Linux, early system initialization employs the specialized *memblock* allocator [93], which handles contiguous regions rather than individual page frames. Only after the memory management is setup, Linux switches to the buddy allocator. In order to maintain consistent statistics, the Ramslice mechanism must be ready by that time.

Ramslice maintains its own state per each instance of the page-frame allocator and thus extends the kernel's data structure representing a memory zone. The core component of this state is a fixed-size array of `struct ramslice` structures that contain per-slice statistics about allocated pages. To avoid unnecessary locking and allow updates in any context, the counters are atomically updated on respective events. Table 3.1 provides an overview of the tracked values. Section 3.2.5 describes how these values are used to determine the amount of pages of each type: freeable, movable, and unmovable.

Another crucial component of Ramslice is the *slice reserve*, likewise stored in the zone structure. During compaction, the slice reserve contains all nonfull slices in the zone, ordered by the reclamation effort called the *price*. Section 3.2.5 describes the computation of the reclamation price in detail. Technically, the slice reserve is a *red-black tree* with $O(\log n)$ search and insertion. Accessing the most expensive slice is a frequent operation, so the reserve caches it for $O(1)$ access time. The tree is not maintained constantly; doing so would require recalculating the price on each update of the per-slice counters. Instead, it is constructed from scratch when compaction is requested by the user.

3.2.2 Slice Isolation

With the data structures established, we can proceed with the technical details of Ramslice compaction. The slice offlining, which can be requested through the user interface (described in Section 3.2.6), is an improved version of the existing memory hotplugging mechanism in Linux. Like with memory hotplugging, the offlined memory must remain isolated from the allocator to prevent instant reuse. Algorithm 3.1 demonstrates the offlining logic, implemented in the function `ramslice_offline`.

Inputs: (*variable*: Description)
zone: Allocator state; *LRU*: PFRA state; *slice reserve*: Slices ordered by price;
slice: Slice to be offlined; *budget*: Maximum allowed offlining cost

1 **abort if** not enough free(able) space in *slice reserve* for *slice* pages — Memory is already compacted
2 *price* ← p'(slice, slice reserve) — Final price adjustment (Section 3.2.5)
3 **abort if** *price* > *budget* — Cost exceeds benefits
4 **if** *slice* is not eligible for offlining — Spans across multiple zones, contains memory holes, etc.
5 ⌞ poison *slice* and **abort** — Reflect eligibility in future price calculations
6 disable *LRU* and *zone* caches — Ensure consistent state
7 **for each** *pageblock* **in** *slice*:
8 ⎸ migrate type of *pageblock* ← `MIGRATE_ISOLATE` — Future frees will end up on isolated list automatically
9 ⎸ **with lock** on *zone* — Already free frames have to be moved there manually
10 ⎸ ⌞ move **each** *free page frame* **in** *pageblock* to isolated list of *zone*
11 *movable list* ← ∅
12 **for each** *page* **in** *slice*:
13 ⎸ **if** *page* is unmapped, clean, and belongs to a file: — Page is freeable
14 ⎸ ⌞ free *page*
15 ⎸ **else if** *page* is on *LRU* lists: — Page is movable
16 ⎸ ⌞ move *page* from *LRU* lists to *movable list*
17 ⎸ ⌞ **else abort** — Page is immovable
18 **until** *movable list* = ∅:
19 ⎸ *source page* ← pop from *movable list*
20 ⎸ *target page frame* ← Algorithm 3.2 — Fetch migration destination
21 ⎸ ⌞ migrate *source page* to *target page frame* — Adjust mappings and copy contents

Algorithm 3.1 – The slice offlining algorithm corresponding to `ramslice_offline` function in the source code. The significant changes from the memory hotplugging `offline_pages` function are highlighted in green.

Sanity checks

Firstly (lines 1 – 5), the offlining mechanism verifies whether the given slice is a valid and viable offlining target. This involves confirming that the free and freeable space in the *slice reserve* is sufficient to accommodate all movable pages of the passed *slice*. If the result is negative, then the memory is already compacted to the maximum. Subsequently, the slice reclamation *price* is adjusted according to the contents of the *slice reserve* and compared with the expected benefit *budget*. If the *price* exceeds the *budget*, offlining aborts. The final check verifies that the *slice* is fully contained in its *zone* and does not span over any architectural memory holes. Inter-zone slices are not supported as they do not appear in most memory configurations and would complicate the algorithm significantly. If the *slice* cannot be offlined due to one of these reasons, it is permanently marked as *poisoned* in its flags. This signalizes for the future price calculations that the *slice* is unreclaimable, making it a destination—rather than source—candidate.

Free page isolation

After determining that the operation is viable for the given slice, the function disables (line 6) two caches: (1) the global LRU cache designed to speed up the addition of pages to the page-frame reclamation algorithm (PFRA), and (2) the per-CPU caches of the buddy allocator in the zone. This

ensures that a consistent view of both buddy lists and LRU lists is maintained even with concurrent allocations and frees.

The next step (lines 7 – 10), implemented in `start_isolate_page_range`, is to isolate all the free page frames in the slice from the buddy allocator. As with memory hotplugging, this is done by iterating over the pageblocks and setting their migrate type to `MIGRATE_ISOLATE`. As a reminder, each of the migrate types has its own free-block lists in the allocator's state. `MIGRATE_ISOLATE` is a special type that is never used to satisfy allocations—the pages residing on it are effectively removed from the allocator. Thus, setting the migration type to `MIGRATE_ISOLATE` results in future deallocations being automatically directed into the isolated lists. The page frames that are already free, however, need to be manually moved into these lists (line 10).

Movable page migration

At this point, all the free page frames reside on the isolated lists and therefore cannot be allocated anymore. The only active page frames still remaining in the slice range are the allocated ones. To remove them from the range, they have to be migrated somewhere else. For this, the algorithm iterates (lines 12 – 17) over all the pages in the range, skipping unallocated page frames. For each *page*, the following distinction is made. If the *page* belongs to the page cache and is neither used by any process nor contains unwritten changes, it is directly freed. The implications of this are discussed in Section 3.3. The second case applies if the *page* cannot be freed directly but resides on the LRU lists of the PFRA. All userspace pages normally reside on these lists and they are movable: the *page* is removed from the respective LRU list and stored on the local *movable list*. Otherwise, the *page* must be immovable and the offlining is aborted. This is different from hotplugging, where the mechanism repeatedly tries to migrate the pages in hope that immovable pages get deallocated.

While it is not strictly required to isolate the page from the PFRA while migrating it, it frees the list head contained in the page descriptor (`struct page`). The list head within the page descriptor is reused for many purposes: buddy-block lists, LRU lists, per-CPU cache lists, and more. Removing the page from LRU and using the list head to store the page on the local list is a repeating pattern in the Linux kernel code. Here, it is used to store the page on the local *movable list*.

Finally, all the movable pages are accumulated on the *movable list*. The algorithm passes (lines 18 – 21) the *movable list* to the `migrate_pages` function as the list of source pages. Along with the source pages, the interface of the migration function accepts a callback that will be used to allocate the destination page frames. It then copies the contents of the *source page* over, adjusts the userspace mappings to point to the new *target page frame*, and frees the *source page*. The passed allocation callback is where the crucial difference from memory hotplugging lies. For memory hotplugging the destination is irrelevant, so the migration function is passed a generic page-frame allocation callback. Instead, Ramslice passes its own callback that allocates the target page frames from expensive slices, implementing the optimal compaction algorithm described at the beginning of the chapter.

3.2.3 Migration Target Search

The second part of the offlining mechanism shown in Algorithm 3.2 implements the callback used to allocate target page frames for Algorithm 3.1. The algorithm has an internal state that persists between invocations. This state holds (1) the location where the last invocation of the algorithm stopped (*PFN* for *page-frame number*, initially zero) and (2) a reserve of free pages (*free list*, initially empty). On each call, if there are pages in the *free list*, the fast path (line 27) simply returns one. Otherwise, the algorithm starts the search for free page frames (lines 2 – 26).

Inputs: (*variable*: Description)
zone: Allocator state; *LRU*: PFRA state; *slice reserve*: Slices ordered by price;
amount: Amount of movable pages isolated by Algorithm 3.1
Persistent state:
PFN ← 0: Cached scan location; *free list* ← ∅: Reserve of isolated free page frames
Outputs: a single *free page frame*

```
1  if free list = ∅:                                              Scan if no pages in reserve
2      cache-page list ← ∅, buddy-block list ← ∅
3      slice ← most expensive slice in slice reserve              Fetch good migration target
4      if PFN is not in slice:
5          PFN ← lowest PFN of slice                              Iterate over page frames in the slice
6      with lock on zone                                          Lock against concurrent allocs and frees
7          while PFN is in slice:
8              break if amount pages are accumulated
9              break if zone lock is contended                    Release the lock for some time
10             if page frame at PFN is allocated:
11                 page ← get page at PFN
12                 if page is unmapped, clean, and belongs to a file:   Save cache page for later free
13                     move page from LRU lists to cache-page list
14                 PFN ← PFN + 1
15             else:                                              Page frame belongs to a free block
16                 buddy block ← get buddy block at PFN
17                 move buddy block from zone list to buddy-block list
18                 PFN ← PFN + size of buddy block                Skip the whole buddy block
19     if PFN is not in slice:                                    Slice is fully scanned
20         remove slice from slice reserve                        Next call will get a new slice
21     for each page in cache-page list:
22         evict page from page cache                             Prevent further usage as cache
23         page frame ← reset page                                Reset page descriptor and zero page contents
24         add page frame to free list
25     for each buddy block in buddy-block list:
26         add each page frame in buddy block to free list        Split into 2^order page frames
27 free page frame ← pop from free list
```

Algorithm 3.2 – The free page scan corresponding to `ramslice_scan` function in the source code. The algorithm iterates over expensive destination slices and accumulates free page frames as migration targets. Red snippets are relevant to page-cache freeing and blue snippets to buddy-block isolation.

Destination slice scan

The search begins by retrieving (line 3) the *slice* that requires the most reclamation effort from the *slice reserve* tree. If the saved *PFN* lies within the *slice*, then the last invocation ended prematurely (i.e., before reaching the end of the *slice*) and the scan is continued from that position. Otherwise, the *PFN* is reset (line 5) to the *slice*'s beginning. There are two potential reasons for an early return from the scan. Firstly, the scan finishes if enough pages have been accumulated to satisfy all future allocations for the *movable list* in Algorithm 3.1. Secondly, the scan is periodically interrupted to alleviate the contention on the *zone* lock, which is required to remove free blocks from the buddy allocator. With per-CPU pages disabled (Algorithm 3.1, line 6), all allocations and frees in the *zone*

attempt to acquire this lock, making it highly contended. To allow these operations to proceed, the free page scan terminates after iterating over 128 page frames.

The scan iterates towards higher addresses and checks the state of the page frame at the current address *PFN*. If it is allocated, then the algorithm checks whether it is freeable with the already familiar check for unmapped and clean file pages. If the check is positive (lines 11 – 13), the *page* is removed from the LRU lists and stored on the local *cache-page list*. Otherwise, not freeable allocated pages are skipped. Free buddy blocks (lines 16 – 17) are another target for the algorithm: they are likewise removed from their buddy lists in *zone* and stored in the local *buddy-block list*. Eventually the scan terminates. If the reason for termination is that the *PFN* cursor has left the *slice*, the *slice* has been fully scanned and is removed (line 20) from the *slice reserve*. Next invocation of the algorithm will fetch a new slice.

Page-frame preparation

Finally, the algorithm consumes both the *cache-page list* (lines 21 – 24) and the *buddy-block list* (lines 25 – 26) by converting their members into free page frames for the *free list*. The file pages are invalidated in the page cache, ensuring that they will not be used for future file accesses. The *page* is then reset to the pristine state: its page descriptor is reinitialized and the contents are zeroed. The resulting clean page frame is put on the *free list*. The free blocks from the *buddy-block list* require splitting, as they likely consist of multiple page frames. Their individual page frames are iterated and added to the *free list*.

The buddy blocks are split in full, which can mean overprovisioning of the target page frames. In the worst case, the Algorithm 3.1 isolates a single movable page on the *movable list* and the Algorithm 3.2 encounters a single order 10 free block, resulting in 1024 page frames on the *free list*. Since the destination slices are expensive to reclaim and thus almost full, such an extreme case is unlikely. Moreover, this overprovisioning is part of the design: a single compaction procedure will offline multiple slices, invoking Algorithm 3.1 repeatedly. The movable pages of the subsequent slices are not yet on the *movable list*, but the contents of the *free list* will be used to satisfy these migrations as well. Nonetheless, after the compaction finishes, the page frames on the *free list* must be released back to the allocator. This step also resets the persistent *PFN* cursor.

3.2.4 Slice Onlining

When offlining aggressively with a high budget, the algorithm eventually achieves the optimal case: all used page frames are condensed into few slices and the rest are isolated from the allocator. If the memory utilization rises, the allocator has no reserve to satisfy these new allocations. To accommodate this case, Ramslice must *online* slices on time by releasing the isolated page frames back to the allocator and setting the respective pageblocks to `MIGRATE_MOVABLE`: the reverse of free page isolation in Algorithm 3.1 (lines 7 – 10). The onlining can be either requested manually (Section 3.2.6) or happens automatically on memory pressure. For the latter, Ramslice hooks the entry functions of the PFRA: `out_of_memory` and `shrink_node`. These hooks check if any slices are offline, online one of them, and return from the PFRA before any userspace pages are freed.

3.2.5 Price Model

Constructing the slice reserve tree requires scoring each slice based on its reclamation effort called the price. Ramslice employs a simple linear regression model that uses the per-slice statistic items from Table 3.1 to predict the expected run time (in CPU cycles) of the offlining operation. The

Symbol	Name	Reflects duration of	Lines	Alg.	Value
p_{slice}	`isolate_slice`	Empty slice isolation (size-independent part)	1 – 6	3.1	695
p_{page}	`isolate_page`	Empty slice isolation per 1024 page frames	7 – 10	3.1	303
p_{drop}	`drop_page`	Freeing single freeable file page	13 – 14	3.1	39
p_{move}	`migrate_page`	Copying page contents to the new page frame	21	3.1	100
p_{map}	`migrate_mapping`	Adjusting one mapping during migration	21	3.1	29
$p_{\text{a,free}}$	`alloc_buddy`	Isolating a free page frame as a migration target	25 – 26	3.2	1
$p_{\text{a,cache}}$	`alloc_cache`	Converting a cache page into a migration target	21 – 24	3.2	25
p_∞	`unreclaimable`	Constant offset added to unreclaimable slices	n/a	n/a	10^9

Table 3.2 – Parameters of the price model. The third and fourth columns indicate the algorithm and the relevant lines for which the parameter approximates the duration. The last column indicates the default value in dimensionless units.

model uses eight parameters listed in Table 3.2 to calculate the expected effort. The methodology used to determine the relevant parameters and their default values is described in Chapter 4.

The first two parameters p_{slice} and p_{page} do not depend on the slice contents and are merely used to approximate the cost of isolating an empty slice, corresponding to lines 1 – 10 in Algorithm 3.1. The first parameter is taken as constant and the second parameter is used to scale the slice size N_{pages}. Together, they form the proportion of the price p_0 independent on the content of the slice:

$$p_0 = p_{\text{slice}} + p_{\text{page}} \cdot \frac{N_{\text{pages}}}{1024} \tag{3.1}$$

The next three parameters depend on the slice contents. To be precise, they map three values to the duration incurred by them: the amount of (1) freeable pages, (2) movable pages, and (3) the total mappings of the slice's pages. Except for the amount of mappings, these values are not directly present in the statistics (Table 3.1) but can be calculated from them. The count of freeable file pages is the amount of total file pages with the mapped file pages subtracted. The amount of movable pages is just the sum of anonymous and mapped file pages. Thus, the slice-dependent price proportion $p_1(\boldsymbol{s})$ for slice $\boldsymbol{s}$ is:

$$\begin{aligned} N_{\text{drop}}(\boldsymbol{s}) &= N_{\text{fileall}}(\boldsymbol{s}) - N_{\text{filemap}}(\boldsymbol{s}) \\ N_{\text{move}}(\boldsymbol{s}) &= N_{\text{filemap}}(\boldsymbol{s}) + N_{\text{anon}}(\boldsymbol{s}) \\ p_1(\boldsymbol{s}) &= p_{\text{drop}} N_{\text{drop}}(\boldsymbol{s}) + p_{\text{move}} N_{\text{move}}(\boldsymbol{s}) + p_{\text{maps}} N_{\text{maps}}(\boldsymbol{s}) \end{aligned} \tag{3.2}$$

Lastly, the *unreclaimable offset* p_∞ is added to the price if it contains unmovable pages (i.e., the amount of allocated pages exceeds userspace pages in LRU lists) or is marked as poisoned. This ensures that the slices which cannot be offlined are always selected by Algorithm 3.2 as migration target slices. The full price $p(\boldsymbol{s})$ used to order the slices in the reserve tree is:

$$p(\boldsymbol{s}) = \begin{cases} p_0 + p_1(\boldsymbol{s}) + p_\infty & \text{if } N_{\text{pages}} - N_{\text{free}}(\boldsymbol{s}) > N_{\text{lru}}(\boldsymbol{s}) \vee \boldsymbol{s} \in \mathscr{P} \\ p_0 + p_1(\boldsymbol{s}) & \text{otherwise} \end{cases} \tag{3.3}$$

where $\mathscr{P}$ is the set of poisoned slices.

However, during slice isolation, the price is adjusted one more time (Algorithm 3.1, line 2). Having already constructed the slice reserve $\boldsymbol{R}$, it is possible to determine the amount of free page frames $N_{\text{a,free}}(\boldsymbol{s}, \boldsymbol{R})$ and unmapped cache pages $N_{\text{a,cache}}(\boldsymbol{s}, \boldsymbol{R})$ in the target slices that will be isolated for offlining as part of the Algorithm 3.2. These values are then scaled with $p_{\text{a,free}}$ and $p_{\text{a,cache}}$, respectively. This yields the final slice- and reserve-dependent price of the operation $p'(\boldsymbol{s}, \boldsymbol{R})$ that is compared with the allowed budget before proceeding or aborting:

$$p'(\boldsymbol{s}, \boldsymbol{R}) = p(\boldsymbol{s}) + p_{\text{a,free}} N_{\text{a,free}}(\boldsymbol{s}, \boldsymbol{R}) + p_{\text{a,cache}} N_{\text{a,cache}}(\boldsymbol{s}, \boldsymbol{R}) \tag{3.4}$$

3.2.6 User Interface

Like many Linux kernel subsystems, Ramslice exposes its functionality in the `sysfs` filesystem tree. The `kernel/debug/ramslice` directory hosts a directory tree representing all memory nodes and their zones (e.g., `ramslice/node0/Normal` for `ZONE_NORMAL` on the first node) in the system. The zone directory contains the files listed in Table 3.3. Most of these files reveal the statistics maintained by Ramslice, either per slice or globally.

The `cmd` file facilitates requests for compaction by allowing users to write commands into it. Its interface resembles the usage of a command-line program: the first word is the requested command, followed by its arguments. The most useful operation, `offline_batch`, performs batch compaction until the budget runs out or memory is fully compacted. It accepts two arguments: (1) the maximum offlining budget and (2) the limit of slices offlined per call. Both of these arguments accept `inf` value to indicate unlimited budget or amount of slices. By default, the budget is cumulative: the price of the offlined slice is subtracted from the budget for the next slice. The optional `--per-slice` flag causes the budget to remain constant for each slice. For example, writing `offline_batch --per-slice 50000 inf` into `cmd` offlines all slices whose price does not exceed 50 000.

The other commands are mostly useful for development purposes. For instance, `offline_index` and `online_index` work on the slice with the specific index (obtained by reading from `all`). Unlike batch compaction, these commands do not automatically construct the slice reserve and release the free list (Algorithm 3.2). For this reason, `reinit` and `release` commands with no arguments are provided. Finally, `offline_free` accepts no arguments and offlines all empty slices. Similarly, `online_all` onlines all slices that are offline, restoring the default system state. Another debugging and profiling feature is enabled by writing into `dummy_mode` file in the root `ramslice` directory. This mode essentially disables the price system: the offlined slices are always migrated into another randomly selected empty slice. This mode will be useful for profiling the mechanism in the next chapter.

File name	Purpose	Read	Write
`nr_slices/nr_empty/nr_offline`	Amount of total/empty/offline slices in the zone	☑	☐
`all`	Snapshot of statistics counters for all slices	☑	☐
`reserve`	Snapshot of the slice reserve tree	☑	☐
`summary`	Per-slice statistics summed over all slices in the zone	☑	☐
`stats`	Statistics of performed operations (write resets)	☑	☑
`cmd`	Interface for operation requests	☐	☑
`result`	Result of the last `cmd` operation	☑	☐

Table 3.3 – Per-zone files exposing the Ramslice functionality in `sysfs`.

3.3 Discussion

Skipping ahead, Chapter 4 demonstrates that the implemented mechanism provides efficient compaction and fulfills its design goals. However, the practical implementation has some limitations in the three aspects covered below.

Slice size and contiguity limitations

Conceptually, the optimal compaction algorithm described in the beginning of the chapter does not limit the slice size or their contiguity in any way. However, the Ramslice implementation in the Linux kernel does have constraints in these aspects. Building upon memory hotplugging, Ramslice uses pageblock infrastructure to isolate free pages in slices from the page-frame allocator during offlining (Algorithm 3.1). As a mechanism with the primary purpose of supporting huge pages, pageblocks are fixed to 2 MiB in size.

Thus, the slice size in Linux is limited to multiples of 2 MiB. Similarly, noncontiguous slices can be supported as long as they consist of contiguous 2 MiB blocks. Removing the dependency on pageblocks might overcome these limitations at the cost of additional complexity. On the other hand, it is reasonable to expect that if future systems support slice offlining in hardware, the address translation will map them onto contiguous regions.

Cost of page cache removal

Currently, the Ramslice mechanism frees every unused file-cache page that it encounters (both in Algorithm 3.1 and in Algorithm 3.2) without considering the delayed cost of this decision. This behavior is indeed suboptimal and its negative consequence can be observed in one of the benchmarks of Chapter 4. Ideally, the compaction mechanism should predict whether a page will be needed in the near future. If the future usage is likely, the page should be migrated rather than freed.

However, Linux lacks infrastructure for such decisions. At the first glance, the page-frame reclamation algorithm seems to fulfill the exact purpose: classify pages based on their usefulness to remove less useful ones on memory pressure. In practice, two reasons make the Linux PFRA unsuitable. Firstly, it only executes when the memory pressure is high. In the idle case, the LRU lists are not maintained and the page states (Figure 2.6) are outdated. Moreover, if the PFRA were to run without memory pressure (e.g., manually triggered by Ramslice), it would instantly categorize unmapped file pages as inactive, leading to their eviction. The PFRA is designed to free as much memory as quickly as possible and the unused file pages are the natural candidates for “quick and dirty” results.

However, there might be an efficient way to implement a new mechanism to predict usage of unmapped file pages. The classical approach to prediction used to implement all sorts of caches is examining the past and extrapolating it into the future. Tracking page accesses like done by PFRA for this purpose does not scale well: it requires sampling the accessed bit from the applications’ page tables using the reverse mappings. However, this is not strictly necessary in the Ramslice scenario. As the cache pages in question are unmapped, the only way they have been accessed in the recent past is via the regular system calls: `open`, `read`, `write`, and `close`. If they became unmapped recently, this change happened via either `munmap` or `exit` system calls. All these system calls transfer the control flow to the kernel anyway, which could be used to implement tracking of regularly used pages with minimal overhead.

Huge-page support

In their current state, neither Algorithm 3.1 nor Algorithm 3.2 support huge pages adequately. If offlining encounters an allocated transparent huge page (THP) within the slice, it is split into reg-

ular page frames. The same is impossible for explicitly-requested hugetlbfs pages, so the page is migrated into an arbitrary huge page frame returned by the page-frame allocator as a fallback. The allocator's placement strategy for the migrated huge page is bound to be suboptimal.

Supporting huge pages properly would require finding a suitable locations in the destination slices to host them. If the destination slices on the expensive end of the price spectrum do not contain free huge frames, Ramslice should not attempt to offline the slice containing the huge page. Making this decision could be facilitated by introducing counters for (1) allocated huge pages and (2) free huge frames into the slice statistics.

Furthermore, the huge-page support uncovers the general problem of competing defragmentation granularities. Supporting multiple target block sizes introduces another level of trade-offs into the compaction mechanism. For example, Linux attempts to use huge pages to improve application performance transparently, but the presence of huge pages obstructs the efforts of Ramslice to create contiguous blocks of bigger size. In this case, Ramslice has to decide between abandoning the slice or splitting the huge page and potentially hurting the application performance.

4 ANALYSIS

This chapter presents the analysis of the Ramslice mechanism described in the previous chapter. It starts with a description of the system setup and developed tools in Section 4.1. Then, the mechanism is profiled to develop a comprehensive cost model in Section 4.2. Finally, Ramslice is applied to real-world workloads to examine potential energy benefits and its effect on the latency of server applications in Sections 4.3 and 4.4.

4.1 Setup and Methodology

All measurements presented in this chapter are performed on a Lenovo ThinkCentre M75t Gen 2 machine. It is equipped with an AMD Ryzen 7 PRO 5750G CPU with a base frequency of 3.8 GHz. When running demanding tasks, the AMD Turbo Core feature can temporarily increase the frequency up to 4.7 GHz. The CPU features eight cores, and each core comprises two logical threads. For the operating system, the machine is running Debian 12 with the custom Linux 6.1 kernel that includes Ramslice modifications described in Chapter 3.

On the memory side, the system has two DRAM channels, each featuring a single two-rank 16 GiB DDR4 DIMM running at 3200 MT/s. This configuration provides 32 GiB of main memory in total. To increase the offlining success, the Linux memory subsystem is configured to provide separate allocator zones for movable (`ZONE_MOVABLE`) and immovable (`ZONE_NORMAL`) memory. While the movable zone still contains unmovable pages that host its page descriptors, they are static and concentrated at the end of the zone. Linux reserves the lower 4 GiB for DMA and immovable allocations. Consequently, the movable zone occupies the remaining 28 GiB of the physical memory.

4.1.1 Artificial Fragmentation

Some measurements rely on consistent behavior across runs. The key to reproducibility when profiling the mechanism is a deterministic workload. Listing 4.1 shows the Python program used to create an artificial fragmentation pattern in the physical memory that represents a consistent workload for the compaction mechanism.

The program creates three memory mappings of a specified size. One mapping (`vmafile`) is backed by a large temporary file and the two other mappings are anonymous (`vmafree`, `vmaanon`). Due to Linux's demand paging, the memory for these new mappings is not allocated immediately. To force the allocations, the program proceeds with a loop that iterates through each page of all three mappings. For each page, a random boolean with a mapping-specific bias decides whether to access the page, causing its allocation, or to skip it and leave it unallocated. From the perspective of the page-frame allocator, this loop translates to a random sequence of allocations for `vmafile`, `vmaanon`, and `vmafree`. This results in an interleaved arrangement of the pages belonging to three different mappings within the physical memory. Although the exact sequence of page allocations is random, the workload maintains a consistent distribution of mappings at the slice level.

```python
def fragment_memory(size, free_probability, anon_probability, file_probability, anon_mapcount=1):
    vmafree = mmap(size, flags=MAP_PRIVATE | MAP_ANONYMOUS)        # Create first anon. mapping
    vmaanon = mmap(size, flags=MAP_PRIVATE | MAP_ANONYMOUS)        # Create second anon. mapping
    vmafile = mmap(size, flags=MAP_PRIVATE, file=mktempfile(size)) # Create file mapping

    for i in range(0, file.size(), PAGE_SIZE):
        if random() < file_probability:
            _ = vmafile[i] # access page to load file into cache
        if random() < free_probability:
            vmafree[i] = 1 # access page to have it allocated
        if random() < anon_probability:
            vmaanon[i] = 1

    munmap(vmafile) # Unmap file pages      -> they remain in memory as freeable
    munmap(vmafree) # Unmap anonymous pages -> their page frames become free

    # Reach required amount of mappings for the anonymous part
    for i in range(anon_mapcount-1):
        if fork() == 0:
            break

    # The physical memory contains a mix of free, movable, and freeable pages of required size
    while True: sleep(1)
```

Listing 4.1 – The Python program used to create the consistent artificial workload for profiling. The result of its execution is an mix of free, freeable, and movable pages in the physical memory.

Finally, the pages of each mapping are converted to their respective target types. The first anonymous mapping (`vmafree`) is freed, freeing up the page frames where its pages previously resided. Likewise, the file mapping mapping is freed as well. Unlike with anonymous memory, the underlying pages stay in memory as page cache in anticipation of future accesses. The other anonymous mapping `vmaanon` remains mapped, but its mapping count is subsequently elevated to `anon_mapcount`. To achieve this, the program calls the `fork` system call `anon_mapcount-1` times. As the final step, the program and all its children enter indefinite sleep to ensure that anonymous pages stay in memory until they are explicitly terminated.

The result of running this program is a mixture of free, freeable, and movable pages in the memory. While the exact sequence of the page types in the memory is random, it maintains a consistent proportion of free, freeable, and movable pages on average when inspected at the granularity of slices. The exact proportions can be controlled by specifying the probabilities for each mapping. For instance, setting `anon_probability` to zero and the other two to 100 percent yields an equal mix of freeable and free pages and no anonymous pages.

4.2 Mechanism Profiling

Before applying the mechanism to real-world workloads to reveal its effectiveness, it is necessary to derive a comprehensive price model that enables the cost-benefit assessment. The approach pursued by this thesis is to analyze the runtime behavior of the algorithm and derive a mathematical model that approximates the time or the energy required for the compaction.

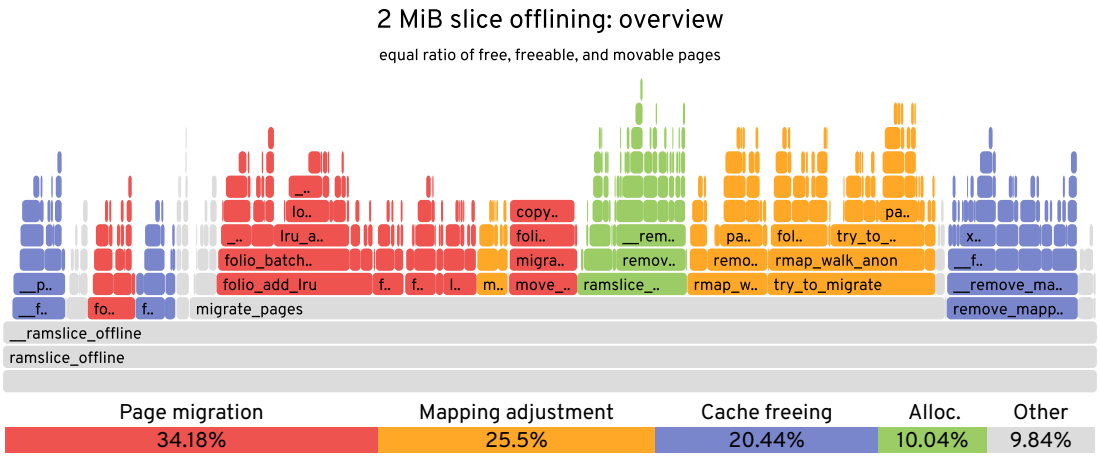


Figure 4.1 – Flame graph of 2 MiB slice offlining (`ramslice_offline`, Algorithm 3.1) with the equal amount of free, freeable, and movable pages. The relevant parts of the algorithm handling page migration, mapping adjustment, cache freeing, and allocation are highlighted, illustrating how much they contribute to the total runtime.

4.2.1 Identifying the Variables

The first step to create a model from measurements is to identify the independent variables. The algorithm is traced using the Linux `perf` [94] tool to examine in which parts the most of the execution time is spent. The tool samples the execution of the compaction with a frequency of 4000 Hz and writes the call stack of each sample into a file. The recorded call stacks are then visualized using the `FlameGraph` [95] tool. In the flame graph, each rectangle represents a function, where functions on top of each other represent nested functions calls. The width of each rectangle represents the amount of samples and thus the relative time spent in the function.

Figure 4.1 shows the flame graph for the offlining of 11 496 slices. The slice size is set to the minimal size of 2 MiB and the slices contain equal amount of free, freeable, and movable pages on average. This distribution is accomplished using the artificial fragmentation workload described earlier. In the graph, parts of the algorithm are highlighted as follows: blue for freeing freeable pages, red for migrating movable pages, and amber for adjusting their mappings. Additionally, the search for migration target page frames (Algorithm 3.2) is shown in green.

As expected, handling of the movable pages consumes the most time. Along with mapping adjustments and searches for migration targets, which are necessary for every movable page, this accounts for 69.72 percent of the total time. In contrast, freeing unused page cache takes only about 20.44 percent of the total time, or roughly a third of the time spent on movable pages. All movable pages in this measurement have the mapping count of one, meaning the amber portion of the algorithm would increase with a higher mapping count. Finally, about 10 percent of the time is spent in functions that cannot be attributed to any specific page type. The time spent in these functions is either constant (disabling the per-core LRU and allocator caches) or depends on the slice size (like free page-frame isolation).

Figure 4.2 zooms into the green portion of the first flame graph, showing the runtime behavior of the migration target search in Algorithm 3.2. The number of free and freeable pages in the destination slices is equal, which facilitates comparison of their respective contributions to the total runtime. The Algorithm 3.2 frees every freeable page it encounters. In this measurement, this process

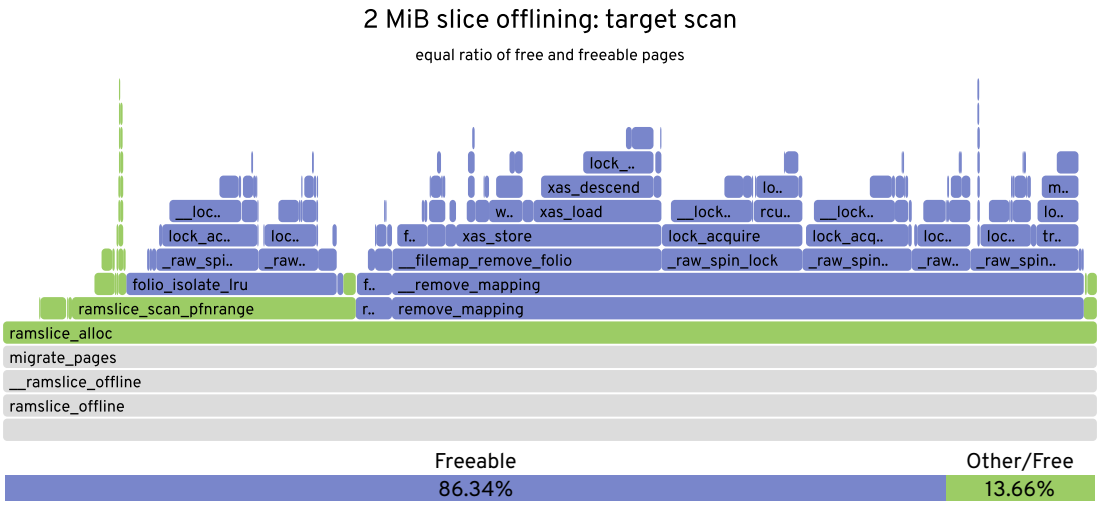


Figure 4.2 – Flame graph of the migration target search (`ramslice_alloc`, Algorithm 3.2) with the equal amount of free and freeable pages in the scanned slices. The portion of responsible for acquisition of freeable pages dominates the total runtime and is highlighted in blue.

accounts for over 86 percent of the total runtime. This emphasizes the importance of considering the cost of page cache deallocation, especially when searching for migration targets, as isolating a free buddy block is significantly faster.

4.2.2 Determining the Parameters

The flame graphs indicate that the runtime of the compaction process is primarily influenced by three factors: the number of freeable and movable pages, and the total number of mappings for these movable pages. To gain better understanding of how these factors impact the final runtime, the artificial fragmentation workload program was enhanced with a new feature. This feature allows specifying a probability range, rather than a fixed probability, for accessing specific mappings in the allocation loop. The probability is varied as the program iterates over the pages of the mappings, creating a *fragmentation gradient* in memory. This means that the slices in the resulting workload vary in their contents. For example, the slices in the gradient can range from containing no movable pages to consisting solely of movable pages, and the states in between.

For the following experiments, the slice size is increased to 64 MiB. The larger size ensures more consistent results; the random irregularities in the artificial fragmentation workload disappear when zooming out by increasing the slice size. To take page allocation costs out of the equation, the dummy mode (Section 3.2.6) was used: the target page frames are preallocated before the offlining begins. Using the new fragmentation gradient feature, five different workloads are generated. The first one contains slices that range from empty to full and consist only of freeable page cache. Technically, each page in this workload has zero mappings. The remaining four workloads include only movable pages, with the mapping count doubling for each subsequent workload.

Figure 4.3 (left) shows the scatter plot of compaction time against the fill level of the slice for each of these experiments. It reveals a clear linear relationship between the number of pages in a slice and the total compaction duration. By calculating the slope for each workload, it is possible to determine the duration incurred per single page. Figure 4.3 (right) plots this per-page duration

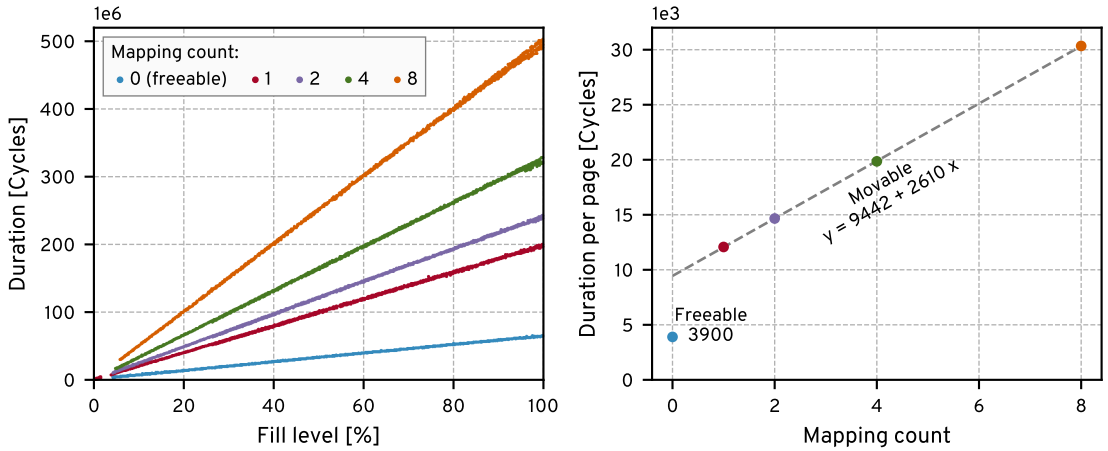


Figure 4.3 – Duration of the compaction depending on the amount of pages in it and their mapping counts. The experiment with mapping count set to zero contains only freeable pages. The plot on the right illustrates the slope of each line on the left.

for each experiment. On average, it takes 3900 clock cycles to free a single freeable page. For the movable pages, it takes 12 074, 14 661, 19 855, and 30 342 clock cycles for 1, 2, 4, and 8 mappings, respectively. Also these numbers demonstrate a clear linear dependency of the migration duration on the amount of mappings, with an increase of 2610 clock cycles per each additional mapping. The crossing of this line with the y-axis at 9443 clock cycles yields the duration of the migration without the mapping-induced part.

The similar approach is used to determine the parameters that depend on the size of the slice, but not on its contents. In this experiment, the system is rebooted with eight different slice sizes ranging from 512 to 4096 page frames in increments of 512. Then, all the empty slices in the freshly booted system are offlined (i.e., isolated from the buddy allocator). Figure 4.4 depicts the duration

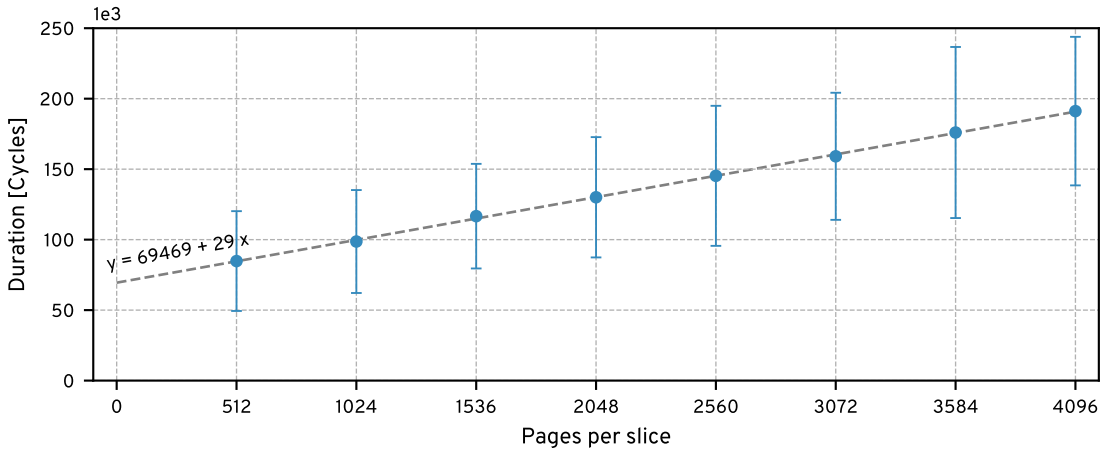


Figure 4.4 – Duration of offlining an empty slice depending on the slice size. The dots indicate the average and the error bars represent the standard deviation. The average offlining duration scales linearly with the slice size.

Benchmark	Algorithm / Lines	μ	σ
Isolating a buddy block	Alg. 3.2 / 15 – 17	55.3	27.2
Splitting an order 0 buddy block[2]	Alg. 3.2 / 25 – 26	67.5	159.8
Splitting an order 1 buddy block	Alg. 3.2 / 25 – 26	87.1	34.0
Splitting an order 2 buddy block	Alg. 3.2 / 25 – 26	147.8	40.8
Isolating cache page from LRU	Alg. 3.2 / 10 – 13	805.0	890.6
Evicting page from page cache	Alg. 3.2 / 21 – 24	2409.8	2699.4

Table 4.1 – Results of microbenchmarks in the migration target search in clock cycles.

of offlining an empty slice as a function of its size. Although the measurements show a considerable spread, the average durations still form a straight line. The function's slope is 29 clock cycles per page frame and the extrapolated constant duration is 69 469 clock cycles.

The final missing piece needed to construct a comprehensive model of compaction duration is the time required for target allocation. Profiling this aspect externally is challenging without significantly modifying the algorithm, as it always prefers the most expensive slice. Instead, the relevant parts of the algorithm were instrumented with time measurements. The results in Table 4.1 indicate that acquiring a free page frame from the buddy allocator takes around 55 + 67.5 = 122.5 clock cycles in the worst case of a zero-order block. As the order of the buddy block increases, the respective time to acquire a single page frame decreases. For example, splitting a second-order block yields four page frames in 147.8 clock cycles, or 36.95 clock cycles per page frame. Isolating a freeable page from the LRU lists requires 805 clock cycles on average, while evicting it from the page cache to use it as the migration target takes approximately 2409.8 clock cycles, totaling to 3214.8 cycles per single freeable page.

4.2.3 Developing the Model

The results from profiling the mechanism yield sufficient information to model the compaction cost. In the context of power management, the price unit represents the energy needed to perform the compaction. However, this work adopts the simplified assumption that CPU time equates to energy. As such, the price unit is defined as 100 clock cycles, denoted with a symbol "¤"[3] for convenience.

Converting the results of the measurements in the previous sections from clock cycles to budget units in ¤ yields the parameters in the column **A** of Table 4.2. The model's accuracy is assessed using a random artificial workload featuring a gradient of free, movable, and freeable pages with various mapping counts. The duration of offlining each slice is recorded along with relevant variables. The slice size is constant for all recorded samples and equals to 64 MiB.

Figure 4.5 (A) presents the results for the first model. The top diagram plots the estimated price of each sample against the actual price. The darker spots represent a higher concentration of samples for those specific combinations. The closer the samples are to the diagonal red line, the more precise the model is. The bottom plot is as a histogram of deviations from the actual duration, as seen when looking along the diagonal red line of the top plot. In summary, the model shows a mean squared error (MSE) value of 528 M¤^{2} and 99 percent of deviations are under 92 K¤ (9.2 million

[2]Although splitting a zero-order block is not necessary in theory, the splitting function adjusts the page-frame descriptor to indicate that the page-frame does not belong to a buddy block.

[3]¤ is a generic currency sign introduced as the placeholder for national currencies in the first attempts to internationalize the ASCII standard [96].

Sym.	Description	Experiment	Model parameters [¤]		
			A	B	C
p_{slice}	Empty slice isolation (size-independent part)	Fig. 4.4	695	695	6092
p_{page}	Empty slice isolation per 1024 page frames	Fig. 4.4	303	303	0
p_{drop}	Freeing single freeable file page	Fig. 4.3	39	39	39
p_{move}	Copying page contents to the new page frame	Fig. 4.3	94	100	5791
p_{map}	Adjusting one mapping during migration	Fig. 4.3	26	29	27
$p_{\text{a,free}}$	Converting free frame into migration target	Tab. 4.1	1	1	−5694
$p_{\text{a,cache}}$	Converting a cache page into a migration target	Tab. 4.1	32	25	−5659
Mean squared error [M¤^{2}]			528	263	210
99th percentile of deviations [ms]			2.37	1.79	1.68

Table 4.2 – The parameters for each of the three analyzed price models. Model **A** is based on raw measurements from mechanism profiling. Model **B** is the model **A** with manually tweaked parameters to achieve better accuracy. The final model **C** is derived using the linear least squares method.

clock cycles or 2.37 ms @ 3.8 GHz, which is 7.6 percent of the maximal duration of 31.1 ms in the analyzed dataset). The overwhelming majority of estimations are lower than the actual duration.

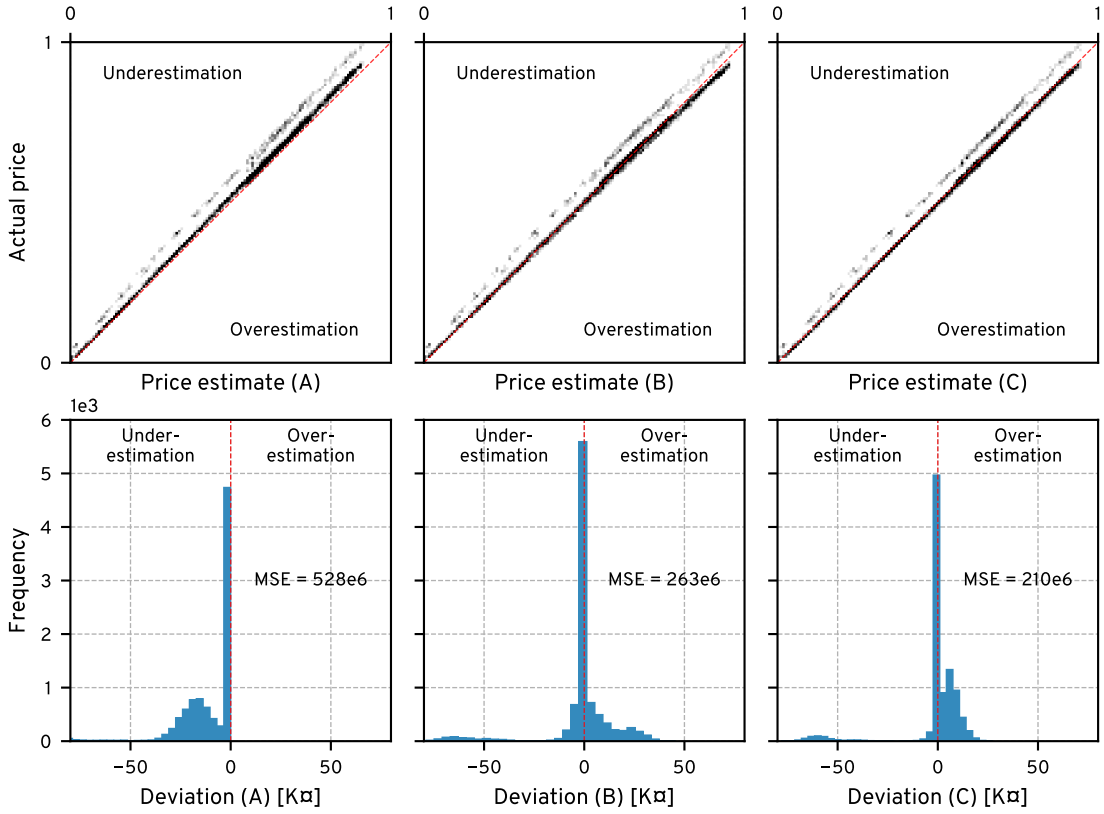
Next, the first model is manually tweaked by examining the outliers and varying the parameters to achieve better results. Specifically, p_{move} is increased from 94 to 100 ¤, p_{map} is increased to 29 ¤, and $p_{\text{a,cache}}$ is reduced to 25 ¤. The adjustments result in the second model **B**, whose accuracy is depicted in Figure 4.5 (B). It exhibits improved accuracy with no significant bias towards under- or overestimation and halves the initial MSE to 263 M¤^{2}. The 99 percent of estimation errors fall under 68.6 K¤ or 1.79 ms. This model is used for all further experiments in this chapter.

The third approach completely eliminates the need for direct measurements by relying on a purely statistical method to calculate the coefficients. This approach involves formulating a minimization problem that allows the construction of an equation, the solution of which provides the optimal coefficients. Model **C** uses the *linear least squares* method [97] to derive these parameters.

$$\boldsymbol{A}\vec{\boldsymbol{x}} = \vec{\boldsymbol{b}}$$

$$\overbrace{\begin{pmatrix} N_{\text{drop}}(s_1) & N_{\text{move}}(s_1) & N_{\text{maps}}(s_1) & N_{\text{a,free}}(s_1) & N_{\text{a,cache}}(s_1) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ N_{\text{drop}}(s_n) & N_{\text{move}}(s_n) & N_{\text{maps}}(s_n) & N_{\text{a,free}}(s_n) & N_{\text{a,cache}}(s_n) \end{pmatrix}}^{\boldsymbol{A}} \overbrace{\begin{pmatrix} p_{\text{drop}} \\ p_{\text{move}} \\ p_{\text{map}} \\ p_{\text{a,free}} \\ p_{\text{a,cache}} \end{pmatrix}}^{\vec{\boldsymbol{x}}} = \overbrace{\begin{pmatrix} t_1 \\ \vdots \\ t_n \end{pmatrix}}^{\vec{\boldsymbol{b}}} \tag{4.1}$$

To begin with, the price model can be expressed using the linear equation (4.1). $\boldsymbol{A}$ is the matrix containing the values of independent variables for each slice and $\vec{\boldsymbol{b}}$ is the vector of the respective offlining durations. Then, $\vec{\boldsymbol{x}}$ contains the model's coefficients. Both $\boldsymbol{A}$ and $\vec{\boldsymbol{b}}$ are easy to obtain without instrumenting the kernel code. While the equation may not have an exact solution, it can be solved approximately using minimization techniques. With linear least squares, the value of $\vec{\boldsymbol{x}}$ that minimizes the squared error $||\boldsymbol{A}\vec{\boldsymbol{x}} - \vec{\boldsymbol{b}}||^2$ is determined by solving $\boldsymbol{A}^{\text{T}}\boldsymbol{A}\vec{\boldsymbol{x}} = \boldsymbol{A}^{\text{T}}\vec{\boldsymbol{b}}$. The resulting coefficients of $\vec{\boldsymbol{x}}$ are presented in column **C** of Table 4.2. This model yields the best accuracy with the MSE of 210 M¤^{2} or 1.68 ms deviation on the 99th percentile.



(A) Raw measurements **(B)** Tweaked measurements **(C)** Linear least squares

Figure 4.5 – The accuracy of each price model as a two-dimensional density plot of estimations vs. the actual price and the histogram of deviations. The estimations falling along the red dashed line match the actual value.

However, there are several important considerations when using this approach. Firstly, it does not yield the p_{page} coefficient as the dataset used when minimizing included only 64 MiB slices. As a result, p_{slice} encompasses both the constant and size-dependent components of runtime for this specific slice size. Constructing a dataset that includes multiple slice sizes would require multiple reboots. Also, with large slices and limited memory, it becomes challenging to provide a versatile workload. For instance, with 1 GiB slices, there are only 28 slices in total in the movable zone, thereby necessitating many iterations to comprehensively cover all scenarios.

Secondly, while the parameters are intended to reflect the runtime durations of specific parts of the algorithm, some values exhibit anomalies. Notably, the value of p_{move} is exorbitantly high and $p_{\text{a,cache}}$ and $p_{\text{a,free}}$ are negative, which is nonsensical in terms of time durations. This can be explained by the fact that these variables are not fully independent: each move requires an allocation (cache or free) and these terms compensate each other during price calculations. For the model to be even more effective, it requires a more careful selection and consideration of truly independent variables.

4.2.4 Discussion

The costs of freeing freeable unmapped page-cache pages are considerably lower than migrating a movable page; $p_{\text{drop}} = 39\,¤$ is less than a third of $p_{\text{move}} + p_{\text{map}} = 129\,¤$. Additionally, freeing the page cache enhances the compaction effectiveness by reducing the overall memory that needs to be accommodated and thus decreasing the number of online slices after compaction. However, there are severe downsides to freeing all freeable pages during the migration target search, as the discrepancy between freeing the page cache and isolating an already free page frame is even more pronounced, exceeding a factor of 20. Implementing a heuristic to skip useful cache pages during the search could optimize this process. However, this requires a mechanism to identify such pages (discussed in Section 3.3), which is currently absent in Linux. Moreover, the second price function could be introduced to select target slices based not only on offlining effort, but also on target acquisition effort.

Model derivation on new systems

When deploying Ramslice on a new system in the end-user setting, it is unrealistic to perform the kernel profiling described in Section 4.2.2. Moreover, each system might behave differently and exhibit a different relationship between parameters. Ideally, the process of deriving the model should be automated. This could be achieved by either adjusting the parameters on the fly or by providing a program that manages this task during deployment. The user would execute the training program to transparently develop and install the model into the system's configuration. The model derivation process could utilize artificial fragmentation workloads and statistical methods, such as least squares, to determine the appropriate parameters.

4.3 Case Studies

With the price model established, the mechanism can now be applied to real-world workloads to explore potential energy savings. As the memory controller of the Ryzen 7 PRO 5750G CPU does not expose any DRAM power-saving mechanisms, this section simulates a hypothetical scenario assuming presence of the novel PARC feature (Table 2.3). The hypothetical memory system comprises two channels, each hosting two ranks of eight parallel 8 Gib LPDDR5 devices. This arrangement results in 8 Gib × 8 (chips) × 2 (ranks) × 2 (channels) = 32 GiB of memory in total with PARC applicable at the granularity of 1 GiB. Thus, the slice size is set to 1 GiB.

Three different workloads are selected to analyze energy savings in case refresh of unused regions is disabled via PARC. For each of the workloads, the following experiments were conducted:

baseline	The run without any compaction
period 30s drop cache	All unused page cache is freed[4] every 30 seconds
period 30s budget ∞	Ramslice compaction with unlimited budget every 30 seconds
period 30s budget x	Ramslice compaction with budget x every seconds

Before each experiment, the main memory is fully defragmented and thereby restored to a consistent clean state by invoking Ramslice compaction with unlimited budget. These case studies focus solely on energy savings and do not consider the total durations of the workloads. The workloads are highly dynamic and last between 30 to 60 minutes. Any potential fluctuations in their duration that are within the magnitude of several seconds are essentially random.

[4] `echo 1 > /proc/sys/vm/drop_caches` [77]

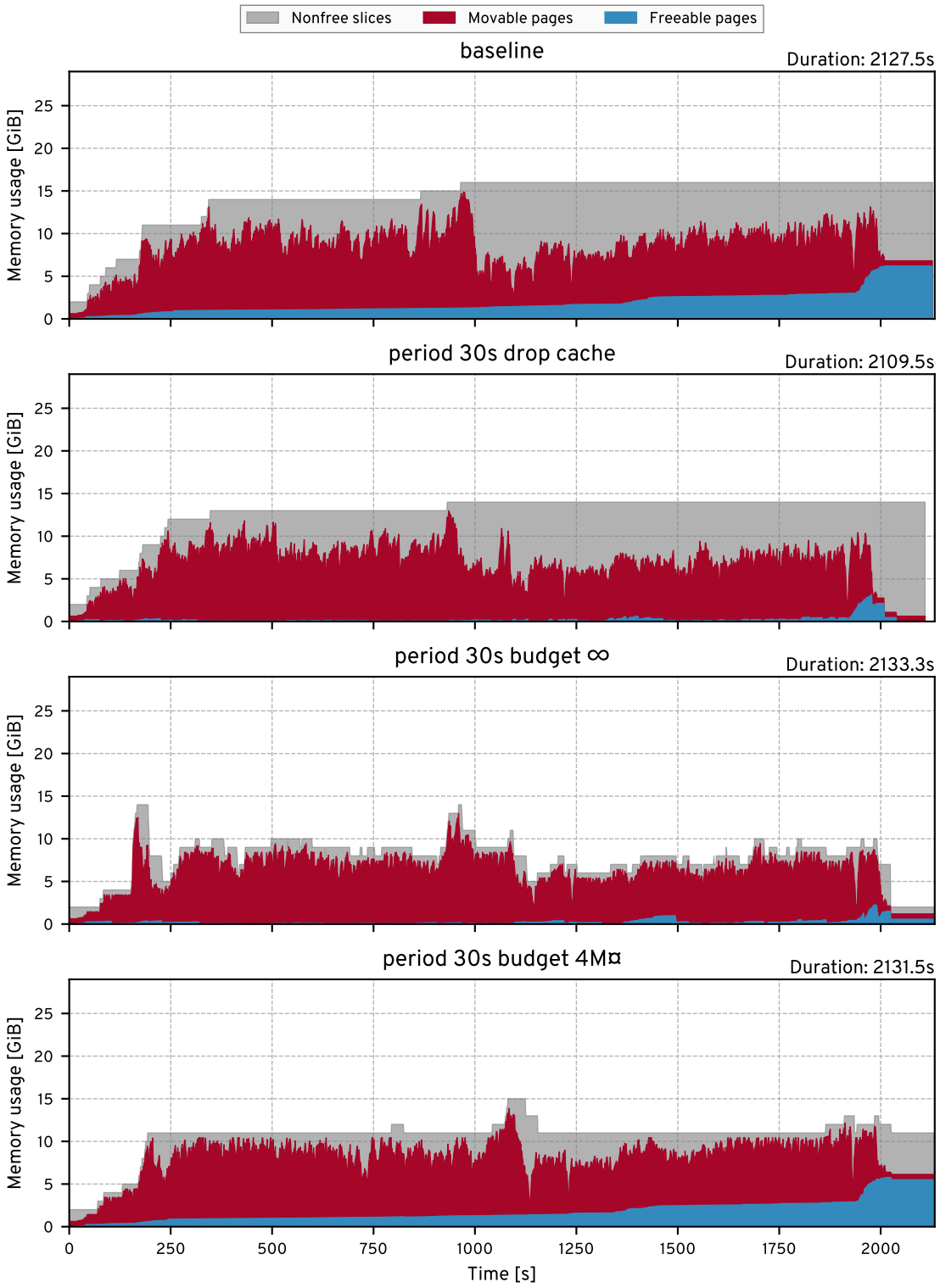


Figure 4.6 – Memory utilization over time during LLVM compilation for each of the experiments. The blue and red areas represent the amount of movable and freeable pages, respectively. The gray area indicates the amount of 1 GiB slices containing allocated pages.

4.3.1 LLVM Compilation

A suitable workload for analyzing Ramslice requires dynamic memory usage. Compilation is a good candidate, as it involves spawning numerous parallel processes, each consuming varying amounts of memory and exhibiting different lifetimes. Additionally, compilation performs many file accesses and thus uses page cache extensively. These factors combined result in quick fragmentation. Consequently, the first analyzed workload is the compilation of the *LLVM* toolchain [98]. On the used machine, LLVM compilation with 16 parallel processes takes approximately 40 minutes to complete. During this process, the memory usage peaks at 14.9 GiB and the compilation loads 6.2 GiB of files into the page cache.

The memory usage during the baseline run without any compaction is shown in the top diagram of Figure 4.6. In this figure, the red and blue areas indicate the amount of movable and freeable pages in the memory at each point in time. The gray background illustrates the amount of slices that contain at least one allocated page. These slices contain used memory, and therefore cannot be offlined for energy savings. The course of this curve highlights the fragmentation issue very vividly. Each time the memory utilization reaches a new peak, that portion of memory remains unavailable for offlining until the end of the experiment. This occurs due to long-lived pages being allocated within the respective slices, causing them to persist there for extensive amounts of time. For instance, all 16 previously used slices still contain allocated pages after the compilation has successfully terminated in the baseline run.

Dropping the cache periodically barely improves this situation. While the amount of fragmented slices reduces to 14, the general tendency remains: once a slice is used, it remains populated. This is different in the third graph that demonstrates the advantage of the Ramslice approach. When compacting with unlimited budget, the page cache is dropped just as aggressively as in the **drop cache** experiment. However, this time, the amount of nonempty slices closely follows the memory utilization. This is because the long-living pages are either freed or migrated into other slices when the memory utilization sinks.

To determine the optimal budget for offlining, it is essential to consider energy consumption across the workload for each experiment. The net energy is composed of two opposing components. On the one hand, energy can be saved by employing PARC. To calculate the respective savings, the amount of empty slices in each experiment is integrated over time, yielding a GiB·s figure. This value is then multiplied by the estimated normalized PARC savings of 9.19 mW/GiB from Table 2.3 to calculate the potentially saved energy in joules.

On the other hand, power is consumed while reclaiming the slices for offlining, either via Ramslice or by dropping the cache. For this system, the power usage is around 10 W when idle and increases to 20 W during the compaction process. These values are measured using the built-in power measurement capabilities of the Emerson MPH2 [99] power distribution unit that powers the machine. To estimate the energy invested into slice reclamation, the total accumulated duration of the Ramslice invocations across the workload is calculated. Then, this value is multiplied by 10 W to approximate for the additional power consumption due to compaction.

Figure 4.7 (top) displays the two energy components for each experiment accumulated over its whole duration. The orange bars represent the energy invested in compaction and the green bars reflect the energy saved by offlining slices using PARC. The gray portions of the savings bars illustrate the theoretical limit of energy savings, achieved by maintaining fully compact memory with no freeable pages. It is worth noting that the baseline figure for energy savings with no reclamation measures presents an optimistic best-case scenario. This is because the memory is fully compacted before each experiment, essentially yielding a system state equivalent to fresh boot. In a real-world system with realistic uptime, the whole memory would likely be fragmented, resulting

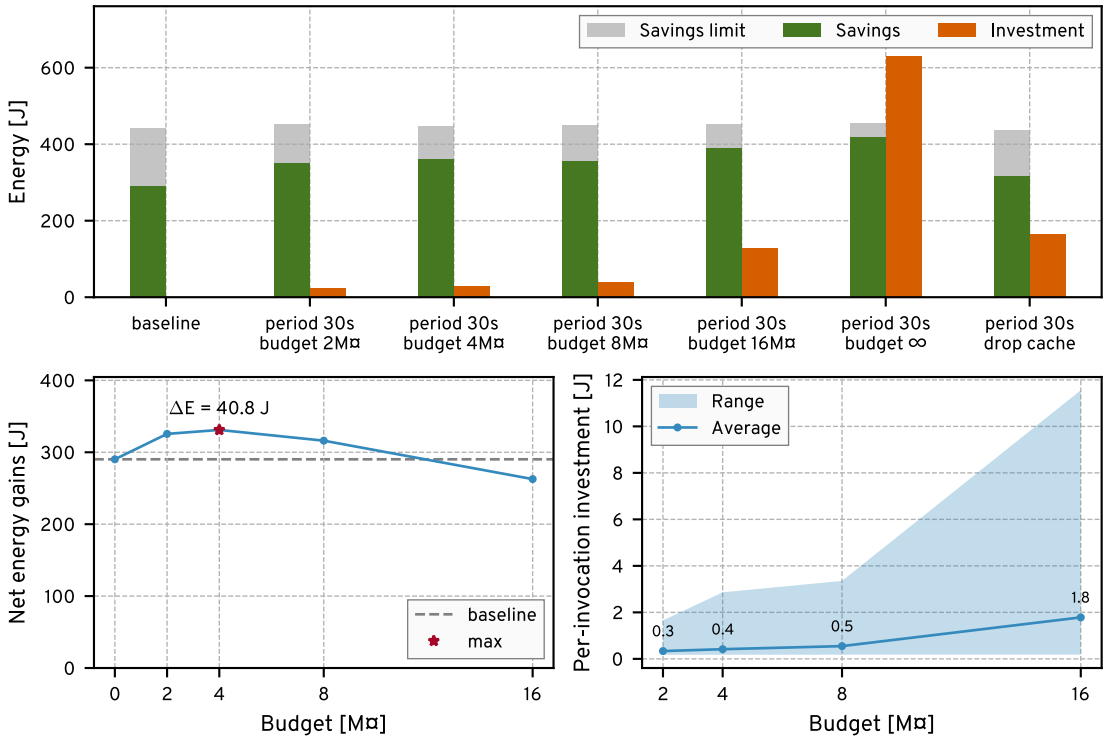


Figure 4.7 – Energy balance for LLVM compilation in each of the experiments. The top plot shows the energy savings and their theoretical limit due to PARC and the invested energy to reclaim slices. The bottom plots illustrate the net energy after subtracting investment from savings and the energy investment per single invocation of Ramslice.

in minimal amount of slices available for offlining right at the start. In general, the results demonstrate that—apart from the **drop cache** scenario—additional investments into compaction result in higher energy savings. However, the dependency is not linear: while the increase from 8 M¤ to 16 M¤ almost quadruples the invested energy, the increase in savings remains minor.

The net energy gain due to compaction is revealed when subtracting the energy investment from the energy savings. Figure 4.7 (bottom left) illustrates the net energy balance over the whole experiment as a function of the compaction budget. The point with zero budget represents the baseline run with no compaction. For all experiments in this case, the energy balance is positive. The effectiveness of the approach is revealed by comparing it with the optimistic baseline case. With budgets ranging from 2 to 8 M¤, energy savings exceed those of the baseline case, indicating that it is beneficial to invest energy in compaction. That is, the invested energy is not only recovered, but also provides for additional net energy gains. Specifically, with the budget of 4 M¤, a total of 40.8 J is saved for the compilation run. Averaged across the whole workload, this amounts to 19.1 mW. As illustrated in the bottom right plot of Figure 4.7, when using 4 M¤, 0.4 J are consumed per single compaction invocation. This quantity shows high variation, as many compaction procedures are interrupted early with remaining budget if the next cheapest slice in the reserve has a high cost.

The memory usage of the experiment with 4 M¤ compaction budget is shown in the bottom plot of Figure 4.6. The curve delimiting the gray area illustrates why this budget is effective: apart from the peak at around 1000 second mark, the amount of nonempty slices remains mostly constant. Thus, low amount of redundant offlining operations are performed with this budget. In contrast,

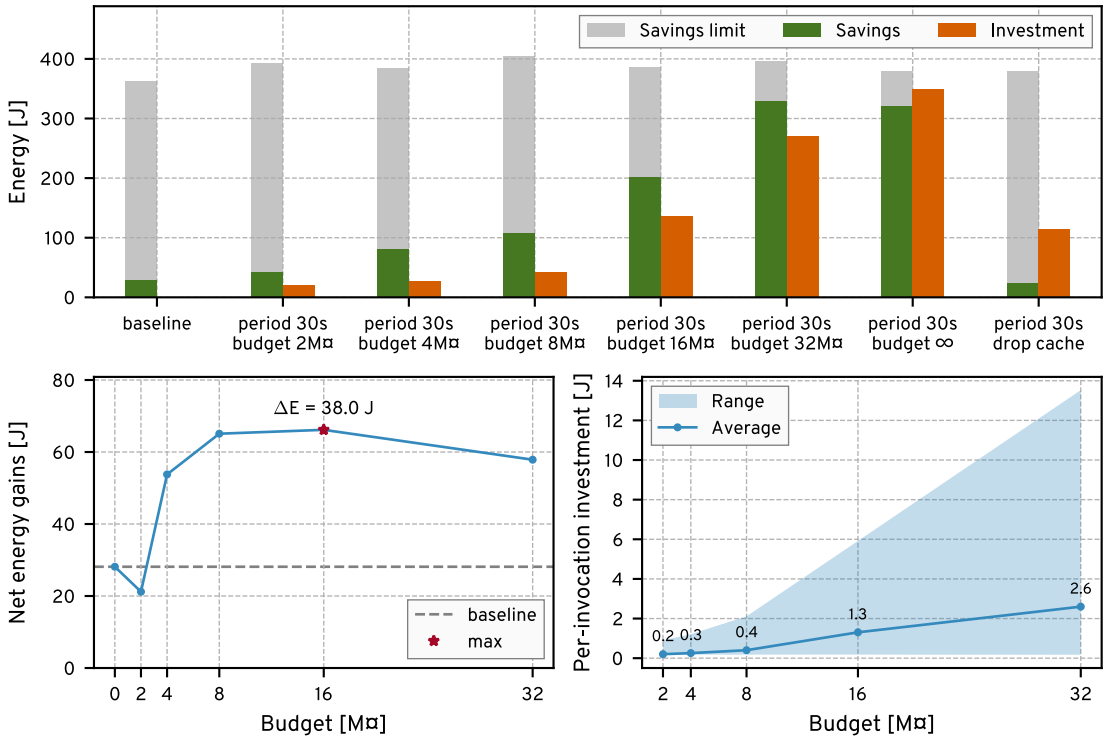


Figure 4.8 – Energy balance for the TPC-H benchmark in each of the experiments.

excessive budgets lead to offlining too many slices, necessitating the reactivation of other slices when new memory allocations occur—without the chance of recovering the invested costs.

4.3.2 TPC-H Queries

The second workload is the business-decision support benchmark *TPC-H* [100] integrated into an in-process SQL database *DuckDB* [101]. The TPC-H data set with the scaling factor of 100 is 26 GB in size and contains sample customer data. The benchmark defines 22 complex queries that can be performed on this data set. These queries are used to answer hypothetical business questions, like shipping priority, revenue forecast, etc. Each query shows a different memory utilization pattern, which makes them interesting for Ramslice analysis. To simulate a real-world scenario, the workload consists of 100 randomly selected queries with 30 second pauses in between.

Figure 4.8 and Figure 4.9 show the energy and memory usage for the experiments with this workload. In contrast to LLVM compilation, the memory utilization is higher: it reaches 28 GiB at its peak, with 8.4 GiB of total page cache being allocated. Just like in the LLVM case, the baseline run and the run with periodic page-cache dropping show the tendency to develop fragmentation quickly. In both runs, over 25 slices remain populated until the experiment concludes after a short memory utilization peak of 28 GiB at the 100th second. Using compaction with unlimited budget, the amount of online slices quickly falls after the peaks and follows the memory utilization. Limiting the budget to 16 M¤, which is the optimal budget for this workload, adds some inertia to the course of the gray curve. For instance, after the peak at the 300 second mark, six invocations of offlining are required before the amount of online slices stabilizes at 21.

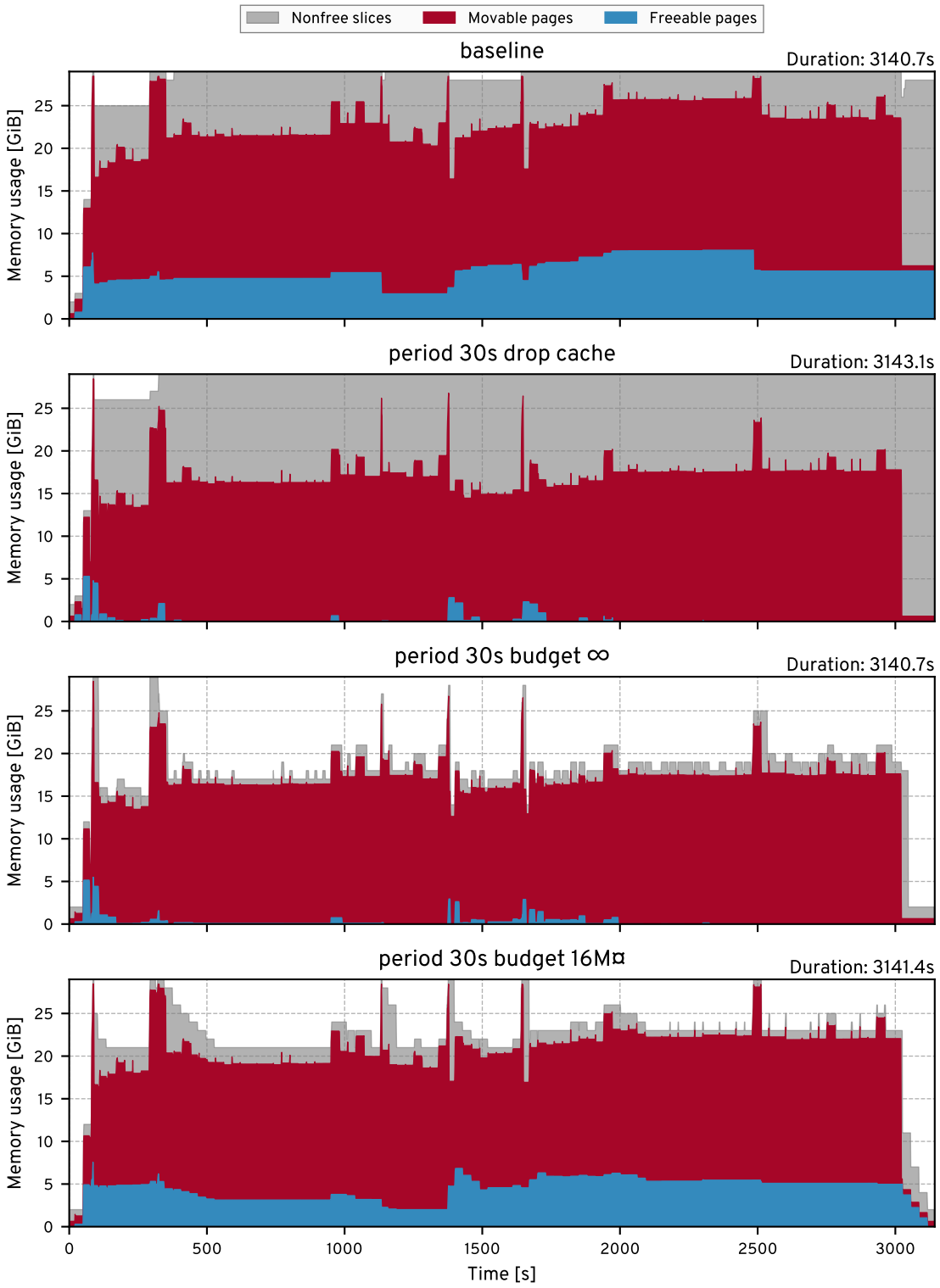


Figure 4.9 – Memory utilization over time during the TPC-H benchmark for each of the experiments.

On the energy side, there are differences to the previous LLVM workload. The saved energy across all approaches is lower, explained by the fact that the memory utilization is closer to the total zone size of 28 GiB. In the baseline case with no active slice reclamation, only 28 J can be saved. When compacting with unlimited budget, this figure reaches 320 J, with a high investment of approximately 350 J. The theoretical savings maximum when maintaining fully compact memory without freeable pages fluctuates across experiments, ranging from 360 to 400 J. These fluctuations can be attributed to different experiment durations and the general dynamic nature of the workload.

The best net energy gains are achieved when compacting with a budget within the 8 – 16 M¤ range. They reach 39 J or 12.4 mW across the workload. Overall, this workload is particularly suited for Ramslice, as the net energy exceeds the baseline for all budgets over 2 M¤. Only when compacting with a low budget of 2 M¤, the gains are lower than the baseline. The average energy investment per Ramslice invocation is similar to the LLVM case across all budgets.

4.3.3 OpenStreetMap Import

The final workload is importing the *OpenStreetMap* data [102, 103] for North America into an *PostgreSQL* [104] relational database. The import is done using an open source *OSM2PGSQL* tool, which also processes geometries of geographical objects. This process is both CPU and memory intensive, and uses page cache extensively. The dump of the geographical data for North America is 15 GB in size. This is the biggest region that can be successfully imported without OSM2PGSQL running out of memory and crashing. The total page cache volume loaded by the import is 25 GiB and the maximum amount of movable memory reaches 18 GiB.

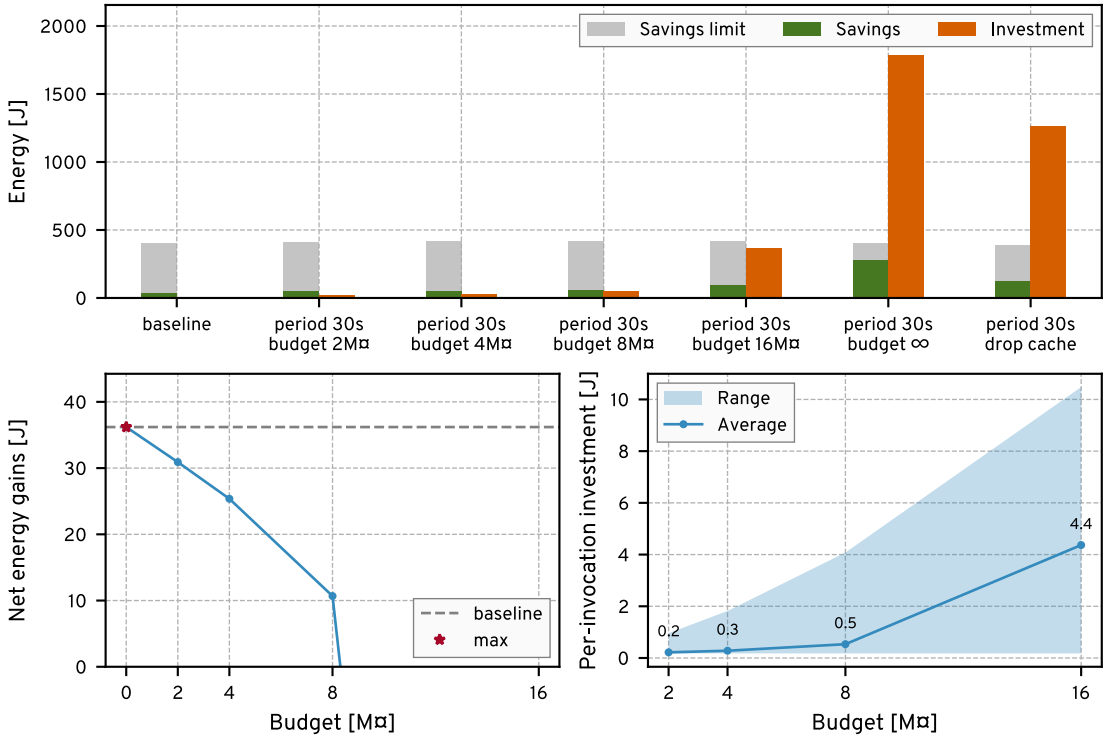


Figure 4.10 – Energy balance for importing OpenStreetMap data into a PostgreSQL database.

This workload reveals a pathological case for Ramslice. As seen in Figure 4.10 (bottom left), none of the budgets manage to achieve better energy efficiency than the baseline case, which itself saves only 36.2 J. Furthermore, increasing the budget leads only to minimal increases in cumulative offline slices. Setting the budget to infinity results in an energy investment of over 1.7 kJ. The resulting savings are only 273 J out of theoretically possible 403.9 J.

Figure 4.11 sheds some light on the reason for the failure to achieve energy gains. When compacting with a limited budget of 8 M¤, the memory utilization is similar to the baseline experiment. Essentially, no slices are offlined, which could mean that the budget is either insufficient to offline even the cheapest slice or that slices are reactivated simultaneously with offlining. The former scenario is unlikely, as the investment per invocation is consistent with the two previous studies, suggesting that significant work is indeed being carried out.

The memory utilization when compacting with unlimited budget supports the latter explanation. Every time a substantial portion of the page cache is freed, an equivalent amount is promptly reloaded into the memory. These fluctuations reach the extreme extent between the 1700th and the 2500th seconds of the experiment, where the volume of the file cache oscillates rapidly between 5 and 25 GiB. The same oscillations can be observed when periodically dropping the cache without migration. This behavior suggests that the page cache evicted during compaction is soon required again, leading to its reloading into memory. In this way, the energy investment results in even more energy being wasted on loading file contents into the memory again.

4.3.4 Discussion

The results of applying Ramslice vary across the three workloads. Both LLVM and TPC-H benefit from the additional energy investment to compact the memory. The energy expenditure not only pays itself off, but also results in about 40 J of net energy gain, translating to an average power reduction of over 12 mW. Projected over eight hours, this amounts to more than 340 J—the energy required to melt 1 g of ice. When scaled to numerous server machines, which often have memory overprovisioned for handling request bursts yet remain underutilized most of the time, the potential savings in energy and electricity costs could be substantial.

Both TPC-H and LLVM generally profit from freeing the unused page cache, as it reduces the overall memory utilization and consequently reduces the amount of online slices. However, the OSM2PGSQL workload presents a scenario where freeing the page cache not only fails to offline any slices, but potentially comes with additional performance penalty. As a result, this leads solely to energy losses.

In general, for different workloads different budgets lead to the best energy gains. However, in both successful cases they fall into the 4 – 16 M¤ range. The optimal budget may also depend on the specific hardware and system configuration. For all three workloads, one compaction run with 8 M¤ budget translated into approximately 0.485 J of energy investment on average. As a rough feeling for this budget, 52.8 GiB·s (calculated as $\frac{0.485\,\mathrm{J}}{9.19\,\mathrm{mW/GiB}}$) have to be offlined with PARC to recover this energy. In simpler terms, a single 1 GiB slice must remain in a power-down state for just under one minute.

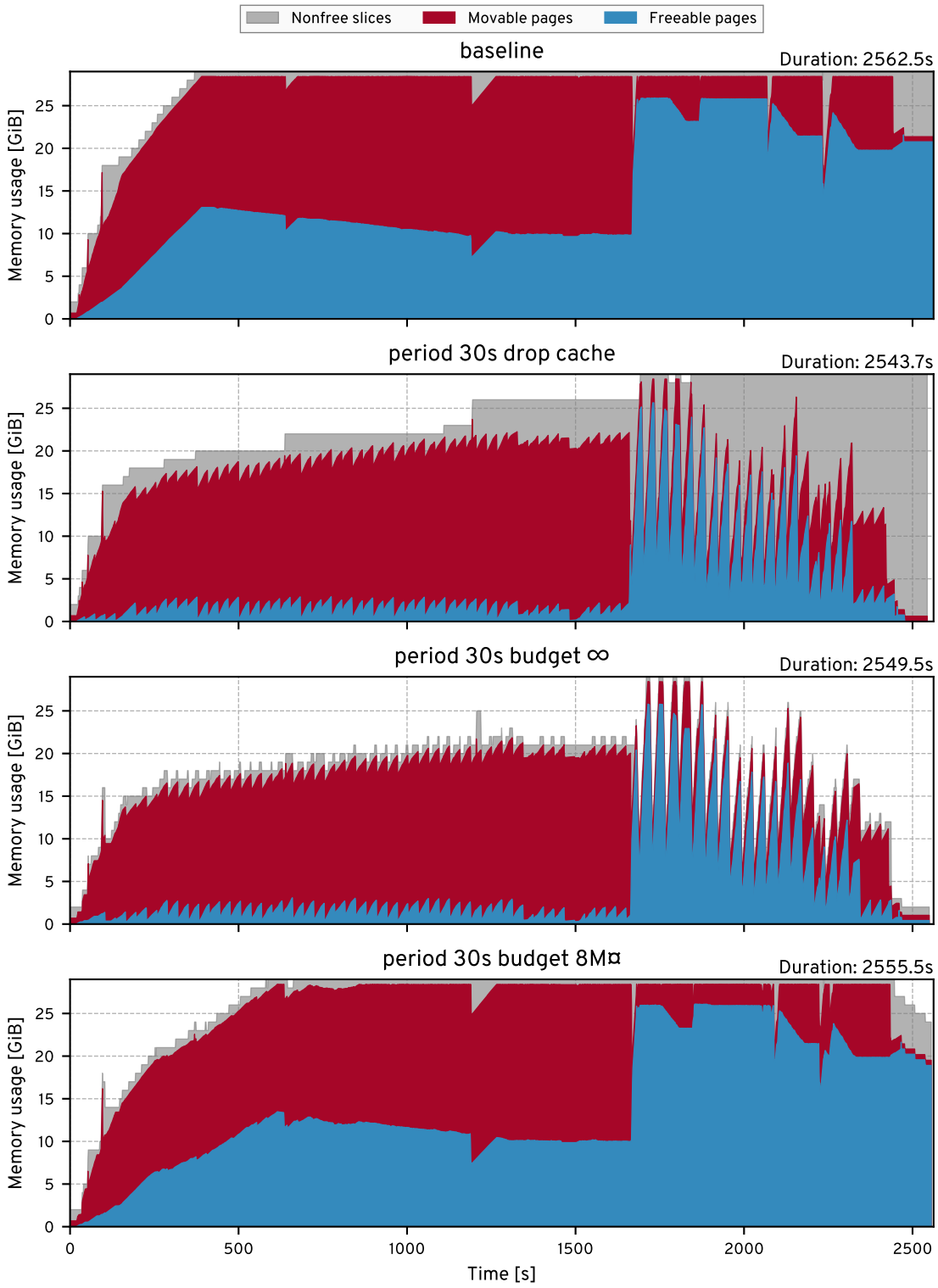


Figure 4.11 – Memory utilization over time while importing OpenStreetMap data into a PostgreSQL database.

4.4 Effect on Latency

The final benchmark aims to examine the impact of the compaction process on other applications running within the system. For instance, periodic compaction could affect server latency, potentially resulting in degraded service and consequential financial costs. For this benchmark, Redis [105] in-memory database has been selected.

The goal of the setup is to trigger the migration of the pages belonging to Redis while it is actively serving requests and assess the impact of it on the client-side latency. To provoke the migration of Redis' pages, they have to be distributed throughout the memory in a fragmentation pattern. For this purpose, the artificial fragmentation program from Section 4.1 is employed once again. This time, it generates a workload comprising 10 GiB of free and 10 GiB of anonymous pages mixed randomly. Subsequently, the Redis server is populated with 10 GiB of data. As a consequence, Linux satisfies allocations coming from Redis using the free page frames that are intermixed with the anonymous pages of the artificial fragmentation. After the key-value store population is complete, the artificial fragmentation program is terminated and its anonymous pages are freed. This results in a 50:50 mix of Redis' anonymous pages and free page frames, which belonged to the fragmentation program before its termination.

After populating the database and distributing its pages throughout the memory, the *Memtier benchmark* [106] running on the same machine is used to perform and benchmark GET requests to the Redis server. The benchmarking program is extended to output individual samples of all requests into a file, enabling analysis of the latency over time. Simultaneously, compaction limited to one slice with unlimited budget is triggered every five seconds. Thus, a each invocation migrates approximately 512 MiB of pages belonging to Redis from one 1 GiB slice.

Figure 4.12 (top) demonstrates the running average, as well as the minimum and the maximum of the recorded latency over time. The regions with red background along the x-axis highlight the time during which Ramslice is active. The average latency curve indicates a moderate increase during each compaction event, rising from approximately 3.7 to 4.5 ms or 21.6 percent. The maximum latency shows the same tendency: during the first compaction run at the 2 second mark it rises by 20 percent, from 7.5 to 9 ms.

Causes of latency impact

There are multiple ways in which the application's performance can be affected by compacting its memory. As already established, if it performs file operations and the respective page cache gets freed, additional latency incurred on the subsequent file accesses. However, Redis is an in-memory database and does not perform any file operations to serve the requests, so this scenario is not applicable.

Another potential cause of increased latency is page faults. During migration, the page-table entries of the page in user application are temporarily replaced with placeholder migration entries. If the application happens to access a page that is currently being migrated, the MMU fails to resolve the translation and initiates a page fault, transferring the control flow to the OS. In Linux, the page fault handler waits until the migration is completed before proceeding with program's execution. Also this is not the case in the Redis experiment. With a 10 GiB working set, the probability of Redis accessing a page that is currently being migrated is extremely low. In fact, the tracing of the page faults with `perf` [94] in this experiment records zero page faults during compaction.

Finally, the third potential reason for increased latency during Redis execution is the *TLB flushing*. Performing page-table walks on each memory access is too costly, as it requires another four to five memory accesses to traverse the page tables on modern architectures. To mitigate this overhead, the MMU incorporates the *translation lookaside buffer (TLB)*, a cache that contains recent transla-

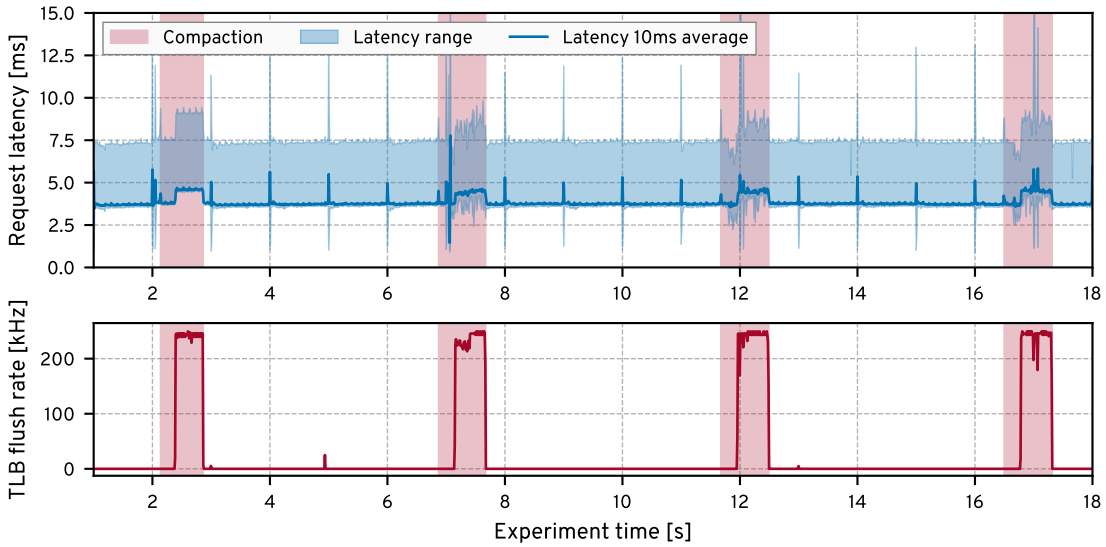


Figure 4.12 – Latencies of GET requests to a Redis in-memory database and the TLB flushing rates on the CPU cores where Redis is running. The red vertical stripes indicates the time during which the compaction process is active. The latency spikes at the beginning of each second correspond to the residual 1 Hz tick interrupt in Linux [107].

tions. If the virtual address is present in the TLB on a memory access, the corresponding physical address is used, eliminating the need for page-table walks.

Unlike the CPU caches, the TLB must be explicitly managed. If the OS modifies the translation mapping or its access rights, the applications continue using the outdated cached translation as long as it remains in the TLB. Consequently, to prevent the applications from accessing invalid memory, entries containing old translations must be explicitly *flushed* from the TLB. Notably, the flushing operation is local to the current CPU core. If the outdated entries also reside on a different CPU core, the OS issues a software interrupt to it. The interrupt pauses the execution of the user application, flushes the respective entries from the local TLBs, and notifies the OS about completion.

As shown in the bottom plot of Figure 4.12, this is precisely the cause of the latency increase during compaction in the Redis experiment. The plot illustrates the amount of TLB flushes recorded using `perf` on the CPU core where Redis is running. When migrating the pages, the TLB flushing rate reaches over $200 \cdot 10^6$ flushes per second. The flushing begin aligns with the latency surge, but does not start immediately with the compaction process. This can be explained by looking at Algorithm 3.1: at first, it isolates free page frames from the buddy allocator, frees freeable pages, and collects the movable pages in the movable list. Only after this step—which takes here about one fourth of the total compaction time—the actual migration and the TLB flushing begins.

4.4.1 Discussion

The compaction of memory in order to increase energy savings not only demands investing CPU time but can also impact the performance of applications running within the system. The primary and most probable cause of this is the TLB flushing: a process evicting the outdated translation mappings from the MMU's translation cache. This step is essential because migration cannot pro-

ceed with copying the contents from the source page to the destination page until it is confirmed that the application will not make any modifications to the source page.

Currently, this flushing occurs for each page separately, necessitating a separate interrupt for each migration. The performance impact could be reduced by batching multiple migrations and flushing multiple entries collectively in a single interrupt. The mechanism for this is already present in Linux. However, it is not used by page the migration infrastructure as it would complicate the logic for an operation that is rarely performed.

In addition to implementing batched TLB flushing, there are other potential improvements that can be made to the migration infrastructure. Currently, the page table entries of the pages being migrated are removed completely and cause page faults for any kind of accesses. However, concurrent reads from the migrated page are not inherently problematic; only concurrent writes pose a risk of a race condition leading to inconsistent page contents. As an alternative to complete entry removal, the entry could be set to read-only before migration and switched to point to the new page after migration. In this way, the page-fault penalty would only happen for writes to the page under migration. However, this optimization is only viable for read-only pages. If the page is writable, a single migration would require two TLB flushes: one to degrade the access rights and another after modifying the entry to point to the new page frame. Perhaps in combination with TLB batching this optimization also becomes feasible for writable pages.

5 CONCLUSION

As outlined in Chapter 2, there are a number of technical possibilities to deactivate portions of DRAM that do not contain any useful data, offering energy savings and even performance improvements. However, effective utilization of these possibilities necessitates close cooperation between hardware and software. On the hardware side, the SDRAM chips, whose functional scope is strictly defined by the JEDEC standards, must provide an interface to their internal power management. Furthermore, these interfaces must be passed through to the software by the intermediate layers in between, particularly the memory controller and the CPU. There is a growing interest for these features within the industry, likely motivated by the efforts to maximize battery life of mobile devices. Notably, the latest LPDDR5 standard [34] introduces the Partial-Array Refresh Control (PARC) feature. With PARC, the granularity of the segments that can be deactivated is brought down to 1 GiB for the first time since introduction of SDRAM. However, no CPUs on the market currently provide software access to this feature.

This situation is unsurprising given that software support for DRAM power management is nearly nonexistent. While the bare minimum OS support, entailing identification of the unused memory regions, is trivial to implement, it would be quickly rendered futile by the outdated assumptions underpinning the design of the contemporary operating systems. Specifically, with the way the physical memory is managed today, it inevitably becomes (1) filled with unused file cache and (2) cluttered by external fragmentation. Therefore, the adequate OS support for DRAM power management requires more than basic bookkeeping: it must also keep the file cache at bay and offer a mechanism for active memory defragmentation via migration. Both facilities must be aware of costs and benefits of their operation, enabling analysis of whether the invested energy or CPU time is likely to be recovered.

As a response to these challenges, this thesis approaches the lack of DRAM power management support from the perspective of system software. Its primary contribution is an algorithm for active memory defragmentation that simultaneously reduces the amount of file cache during its operation. The algorithm is cost-aware and is theoretically capable of achieving optimal defragmentation with minimal effort. To test the concept in practice, it has been implemented in the Linux kernel 6.1 under the name Ramslice. The evaluation profiles the costs of the mechanism's operations to develop the price model and applies it to three real-world workload to assess the potential energy savings. Finally, it investigates the impact of memory migration on the latency of a server application, discusses the underlying causes, and proposes improvements to the Linux memory migration infrastructure.

The three workloads used for the energy analysis simulate a hypothetical scenario in which contiguous 1 GiB memory regions can be disabled using PARC to conserve energy. The experiments demonstrate that the energy expended by Ramslice to reorganize the memory contents is not only recuperated but also results in additional net energy savings. For instance, the energy consumption of the LLVM compilation process is reduced by 40.8 J through periodic compaction with Ramslice. This translates to a reduction in average power consumption of 19.1 mW. In another benchmark involving business-related queries on a customer database, the average power consumption drops by 12.4 mW.

Ultimately, this work challenges the outdated assumptions that (1) unused memory is a wasted resource and (2) all page frames are equal. It demonstrates that from the operating system's perspective, it is worthwhile to invest effort in evicting the file cache and reorganizing the physical memory to disable portions of DRAM. The invested energy is recovered even in highly dynamic workloads such as the compilation of a large software project. These energy saving figures scale quickly when applied to numerous server machines, which tend to be overprovisioned in their memory resources [21]. The respective changes to the Linux kernel are nonintrusive and maintain acceptable complexity. Hopefully, these findings will motivate manufacturers to accelerate the introduction of novel DRAM power-saving features into general-purpose customer hardware.

5.1 Future Work

There is a significant potential for future improvements in the Ramslice implementation, highlighting the extensiveness of this research area. One of its primary drawbacks is its rudimentary handling of page cache. To maximize the effectiveness of the compaction, Ramslice evicts every unused cache page that it encounters. While this strategy maximizes the amount of unused DRAM segments immediately after compaction, it fails to consider the hidden costs of the page-cache eviction, which occur when programs access the evicted files in the near future. These costs are evident in the OSM2PGSQL benchmark, where Ramslice fails to achieve energy savings, as it consumes energy and time evicting pages that are immediately reloaded into memory.

The Linux memory migration infrastructure offers opportunities for enhancement as well. Currently, it flushes the TLB of all CPU cores individually for each migrated page, significantly affecting the performance of other applications within the system. In the Redis benchmark, this results in an over 20 percent increase in latency. To alleviate this overhead, the migration mechanism can be redesigned to batch migrations and TLB flushes, thereby reducing the amount of interrupts on remote cores. Moreover, it can be enhanced to cleverly handle read-only pages, preventing unnecessary page faults when a page under migration is accessed.

While the underlying concept has no limitations regarding the size and contiguity of the managed DRAM segments, the Linux implementation is constrained to contiguous memory blocks that are integer multiples of 2 MiB. This limitation stems from usage of the pageblock infrastructure in Linux that is used to implement memory isolation. Additionally, the slice size is determined at boot time and remains fixed throughout the system's runtime. An improved solution for DRAM power management would allow for segments of arbitrary size and contiguity, with the potential to adjust them dynamically at runtime. However, without the hardware support on the market, the feasibility of the additional technical complexity to support this remains unclear.

The currently developed price model is derived from profiling the algorithm within the kernel. To enable deployment of Ramslice in the end-user settings, a way to derive the price model transparently without user interaction is necessary. As shown in Chapter 4, simple statistical methods have proven effective in determining the model parameters, requiring minimal to no alterations to the kernel. Further investigation into this direction can help identify the most effective strategy for statistical modeling.

Finally, it is worthwhile to explore other suited problem areas where Ramslice compaction could prove useful. For instance in Linux, allocating giant 1 GiB pages is typically only reliable during the boot process; subsequent attempts are possible but often unsuccessful. Ramslice can facilitate the acquisition of such pages by invoking its compaction process. Furthermore, devices such as video camera on mobile systems may require large (>512 MiB [89]) contiguous chunks of memory for DMA. Also here, Ramslice can be employed to efficiently obtain a free chunk of memory on demand.

LIST OF ACRONYMS

AR	Auto-Refresh
COW	Copy-On-Write
DDR	double data rate
DIMM	dual in-line memory module
DRAM	dynamic random-access memory
HBM	High-Bandwidth Memory
LRU	least recently used
MMU	Memory Management Unit
MPSM	Maximum Power Saving Mode
MSE	mean squared error
NUMA	Non-Uniform Memory Access
ODT	on-die termination
PARC	Partial-Array Refresh Control
PASR	Partial-Array Self-Refresh
PFN	page-frame number
PFRA	page-frame reclamation algorithm
SDRAM	synchronous dynamic random-access memory
SR	Self-Refresh
SRAM	static random-access memory
SSD	solid state drive
THP	transparent huge page
TLB	translation lookaside buffer

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

REFERENCES

1. Denning, P. J.: Before Memory Was Virtual, http://denninginstitute.com/pjd/PUBS/bvm.pdf, (1996)

2. Jessen, E.: Die Entwicklung des virtuellen Speichers. Informatik-Spektrum. 216–219 (1996). https://doi.org/10.1007/s002870050034

3. Kilburn, T., Edwards, D. B. G., Lanigan, M. J., Sumner, F. H.: One-Level Storage System. IRE Transactions on Electronic Computers. 223–235 (1962). https://doi.org/10.1109/TEC.1962.5219356

4. Shanley, T., MindShare, C., Inc.: Pentium Pro and Pentium II system architecture (2nd ed.). Addison-Wesley Longman Publishing Co., Inc., USA (1998)

5. Gorman, M., Healy, P.: Performance characteristics of explicit superpage support. In: Proceedings of the 2010 International Conference on Computer Architecture. pp. 293–310. Springer-Verlag, Saint-Malo, France (2010). https://doi.org/10.1007/978-3-642-24322-6_24

6. Gorman, M., Whitcroft, A.: The What, The Why and the Where To of Anti-Fragmentation, https://www.kernel.org/doc/ols/2006/ols2006v1-pages-369-384.pdf, (2006)

7. Gorman, M., Whitcroft, A.: Supporting the Allocation of Large Contiguous Regions of Memory, https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=0eb664c1e593753796fd40ca29de060d7a93f1ec, (2007)

8. Gorman, M., Healy, P.: Supporting superpage allocation without additional hardware support. In: Proceedings of the 7th International Symposium on Memory Management. pp. 41–50. Association for Computing Machinery, Tucson, AZ, USA (2008). https://doi.org/10.1145/1375634.1375641

9. Navarro, J., Iyer, S., Cox, A.: Practical, Transparent Operating System Support for Superpages. In: 5th Symposium on Operating Systems Design and Implementation (OSDI 02). USENIX Association, Boston, MA (2002)

10. Gorman, M., Whitcroft, A.: The what, the why and the where to of anti-fragmentation. In: Ottawa Linux Symposium. pp. 369–384 (2006)

11. Michailidis, T., Delis, A., Roussopoulos, M.: MEGA: overcoming traditional problems with OS huge page management. In: Proceedings of the 12th ACM International Conference on Systems and Storage. pp. 121–131. Association for Computing Machinery, Haifa, Israel (2019). https://doi.org/10.1145/3319647.3325839

12. Stafford, W.: How DRAM changed the world, https://www.micron.com/about/blog/memory/dram/how-dram-changed-the-world, last accessed 2024/10/30

13. Patel, D.: The Memory Wall: Past, Present, and Future of DRAM, https://www.semianalysis.com/p/the-memory-wall, last accessed 2024/10/30

14. AMD EPYC™ 9004 and 8004 Series Processors, https://www.amd.com/content/dam/amd/en/documents/products/embedded/epyc/epyc-embedded-9004-and-8004-series-product-brief.pdf, last accessed 2024/10/09

15. Bhati, I., Chang, M.-T., Chishti, Z., Lu, S.-L., Jacob, B.: DRAM Refresh Mechanisms, Penalties, and Trade-Offs. IEEE Transactions on Computers. 65, 108–121 (2016). https://doi.org/10.1109/TC.2015.2417540

16. Maruf, H. A., Wang, H., Dhanotia, A., Weiner, J., Agarwal, N., Bhattacharya, P., Petersen, C., Chowdhury, M., Kanaujia, S., Chauhan, P.: TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3. pp. 742–755. Association for Computing Machinery, Vancouver, BC, Canada (2023). https://doi.org/10.1145/3582016.3582063

17. Kumar, K., Doshi, K., Dimitrov, M., Lu, Y.-H.: Memory energy management for an enterprise decision support system. In: IEEE/ACM International Symposium on Low Power Electronics and Design. pp. 277–282 (2011). https://doi.org/10.1109/ISLPED.2011.5993649

18. Meisner, D., Gold, B. T., Wenisch, T. F.: PowerNap: eliminating server idle power. In: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 205–216. Association for Computing Machinery, Washington, DC, USA (2009). https://doi.org/10.1145/1508244.1508269

19. Lu, C., Ye, K., Xu, G., Xu, C.-Z., Bai, T.: Imbalance in the cloud: An analysis on Alibaba cluster trace. In: 2017 IEEE International Conference on Big Data (Big Data). pp. 2884–2892 (2017). https://doi.org/10.1109/BigData.2017.8258257

20. Tirmazi, M., Barker, A., Deng, N., Haque, M. E., Qin, Z. G., Hand, S., Harchol-Balter, M., Wilkes, J.: Borg: the next generation. In: Proceedings of the Fifteenth European Conference on Computer Systems. Association for Computing Machinery, Heraklion, Greece (2020). https://doi.org/10.1145/3342195.3387517

21. Wang, Y., Luo, B., Shen, Y.: Efficient Memory Overcommitment for I/O Passthrough Enabled VMs via Fine-grained Page Meta-data Management. In: 2023 USENIX Annual Technical Conference (USENIX ATC 23). pp. 769–783. USENIX Association, Boston, MA (2023)

22. High-capacity DDR5 solution from Micron's leading-edge 1β (1-beta) technology, https://www.micron.com/content/dam/micron/global/public/documents/products/product-flyer/128gb-ddr5-rdimm-product-brief.pdf, last accessed 2024/10/15

23. DDR5 vs DDR4 DRAM – All the Advantages & Design Challenges, https://www.rambus.com/blogs/get-ready-for-ddr5-dimm-chipsets/

24. Drepper, U.: What Every Programmer Should Know About Memory, https://people.freebsd.org/~lstewart/articles/cpumemory.pdf, (2007)

25. Cuppu, V., Jacob, B., Davis, B., Mudge, T.: A performance comparison of contemporary DRAM architectures. In: Proceedings of the 26th Annual International Symposium on Computer Architecture. pp. 222–233. IEEE Computer Society, Atlanta, Georgia, USA (1999). https://doi.org/10.1109/ISCA.1999.765953

26. Jacob, B., Ng, S., Wang, D.: Memory Systems: Cache, DRAM, Disk. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2007)
27. Balasubramonian, R., Natalie, E. J., Martonosi, M.: Innovations in the Memory System. Morgan & Claypool Publishers (2019)
28. Double Data Rate (DDR) SDRAM Specification, (2000)
29. DDR2 SDRAM Specification, (2009)
30. DDR3 SDRAM Standard, (2012)
31. DDR4 SDRAM, (2021)
32. Hennessy, J. L., Patterson, D. A.: Computer Architecture, Sixth Edition: A Quantitative Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2017)
33. DDR5 SDRAM, (2024)
34. Low Power Double Data Rate (LPDDR) 5/5X, (2023)
35. Graphics Double Data Rate 7 SGRAM Standard (GDDR7), (2024)
36. DRAM manufacturers revenue share worldwide from 2011 to 2024, by quarter, https://www.statista.com/statistics/271726/global-market-share-held-by-dram-chip-vendors-since-2010/
37. Bulić, P.: Understanding Computer Organization. Springer International Publishing (2024)
38. Seshadri, V., Hsieh, K., Boroum, A., Lee, D., Kozuch, M. A., Mutlu, O., Gibbons, P. B., Mowry, T. C.: Fast Bulk Bitwise AND and OR in DRAM. IEEE Computer Architecture Letters. 14, 127–131 (2015). https://doi.org/10.1109/LCA.2015.2434872
39. Mutlu, O., Moscibroda, T.: Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. SIGARCH Comput. Archit. News. 36, 63–74 (2008). https://doi.org/10.1145/1394608.1382128
40. Kaseridis, D., Stuecheli, J., John, L. K.: Minimalist open-page: a DRAM page-mode scheduling policy for the many-core era. In: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture. pp. 24–35. Association for Computing Machinery, Porto Alegre, Brazil (2011). https://doi.org/10.1145/2155620.2155624
41. Ghasempour, M., Jaleel, A., Garside, J. D., Luján, M.: DReAM: Dynamic Re-arrangement of Address Mapping to Improve the Performance of DRAMs. In: Proceedings of the Second International Symposium on Memory Systems. pp. 362–373. Association for Computing Machinery, Alexandria, VA, USA (2016). https://doi.org/10.1145/2989081.2989102
42. Hillenbrand, M., Bellosa, F.: Putting the OS in control of DRAM with mapping aliases. In: Proceedings of the 10th ACM International Systems and Storage Conference. Association for Computing Machinery, Haifa, Israel (2017). https://doi.org/10.1145/3078468.3078487
43. Yun, H., Mancuso, R., Wu, Z.-P., Pellizzoni, R.: PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In: 2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 155–166 (2014). https://doi.org/10.1109/RTAS.2014.6925999

44. Liu, L., Cui, Z., Li, Y., Bao, Y., Chen, M., Wu, C.: BPM/BPM+: Software-based dynamic memory partitioning mechanisms for mitigating DRAM bank-/channel-level interferences in multicore systems. ACM Trans. Archit. Code Optim. 11, (2014). https://doi.org/10.1145/2579672

45. DRAM Memory Module Rank Calculation, https://www.digikey.de/en/pdf/v/viking-technology/dram-memory-module-rank-calculation

46. Shin, W., Jang, J., Choi, J., Suh, J., Kwon, Y., Moon, Y., Kim, L.-S.: Rank-Level Parallelism in DRAM. IEEE Transactions on Computers. 66, 1274–1280 (2017). https://doi.org/10.1109/TC.2017.2654339

47. Muralidhara, S. P., Subramanian, L., Mutlu, O., Kandemir, M., Moscibroda, T.: Reducing memory interference in multicore systems via application-aware memory channel partitioning. In: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture. pp. 374–385. Association for Computing Machinery, Porto Alegre, Brazil (2011). https://doi.org/10.1145/2155620.2155664

48. Khan, S., Lee, D., Kim, Y., Alameldeen, A. R., Wilkerson, C., Mutlu, O.: The efficacy of error mitigation techniques for DRAM retention failures: a comparative experimental study. SIGMETRICS Perform. Eval. Rev. 42, 519–532 (2014). https://doi.org/10.1145/2637364.2592000

49. Baek, S., Cho, S., Melhem, R.: Refresh Now and Then. IEEE Transactions on Computers. 63, 3114–3126 (2014). https://doi.org/10.1109/TC.2013.164

50. Yaney, D., Lu, C., Kohler, R., Kelly, M., Nelson, J.: A meta-stable leakage phenomenon in DRAM charge storage - Variable hold time. In: 1987 International Electron Devices Meeting. pp. 336–339 (1987). https://doi.org/10.1109/IEDM.1987.191425

51. Mathew, D. M., Zulian, É. F., Jung, M., Kraft, K., Weis, C., Jacob, B., Wehn, N.: Using run-time reverse-engineering to optimize DRAM refresh. In: Proceedings of the International Symposium on Memory Systems. pp. 115–124. Association for Computing Machinery, Alexandria, Virginia (2017). https://doi.org/10.1145/3132402.3132419

52. DDR4 SDRAM: MT40A4G4, MT40A2G8, MT40A1G16, https://www.mouser.com/datasheet/2/671/16Gb_DDR4_SDRAM-1578714.pdf

53. Qualcomm Inc.: Partial Refresh Technique to Save Memory Refresh Power, https://patents.google.com/patent/US10332582B2, last accessed 2024/10/12

54. Ohsawa, T., Kai, K., Murakami, K.: Optimizing the DRAM refresh count for merged DRAM/logic LSIs. In: Proceedings of the 1998 International Symposium on Low Power Electronics and Design. pp. 82–87. Association for Computing Machinery, Monterey, California, USA (1998). https://doi.org/10.1145/280756.280792

55. Cui, Z., McKee, S. A., Zha, Z., Bao, Y., Chen, M.: DTail: a flexible approach to DRAM refresh management. In: Proceedings of the 28th ACM International Conference on Supercomputing. pp. 43–52. Association for Computing Machinery, Munich, Germany (2014). https://doi.org/10.1145/2597652.2597663

56. Bhati, I., Chishti, Z., Lu, S.-L., Jacob, B.: Flexible auto-refresh: Enabling scalable and energy-efficient DRAM refresh reductions. In: 2015 ACM/IEEE 42nd Annual International Symposium

on Computer Architecture (ISCA). pp. 235–246 (2015). https://doi.org/10.1145/2749469.2750408

57. Jafri, S. M. A. H., Hassan, H., Hemani, A., Mutlu, O.: Refresh Triggered Computation: Improving the Energy Efficiency of Convolutional Neural Network Accelerators. ACM Trans. Archit. Code Optim. 18, (2021). https://doi.org/10.1145/3417708

58. Ghosh, M., Lee, H.-H. S.: Smart Refresh: An Enhanced Memory Controller Design for Reducing Energy in Conventional and 3D Die-Stacked DRAMs. In: 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007). pp. 134–145 (2007). https://doi.org/10.1109/MICRO.2007.13

59. JEDEC Updates JESD79-5C DDR5 SDRAM Standard: Elevating Performance and Security for Next-Gen Technologies, https://www.jedec.org/news/pressreleases/jedec-updates-jesd79-5c-ddr5-sdram-standard-elevating-performance-and-security, last accessed 2024/10/13

60. Oklobdzija, V. G., Krishnamurthy, R. K.: High-Performance Energy-Efficient Microprocessor Design. Springer New York (2006)

61. Halbuer, A., Ostapyshyn, I., Steiner, L., Wrenger, L., Jung, M., Dietrich, C., Lohmann, D.: The New Costs of Physical Memory Fragmentation. In: Proceedings of the 2nd Workshop on Disruptive Memory Systems (2024)

62. Tanenbaum, A. S.: Modern Operating Systems. Prentice Hall PTR (2007)

63. Knuth, D. E.: The Art of Computer Programming, Vol. 1: Fundamental Algorithms. Addison-Wesley, Reading, MA, USA (1997)

64. Bhattacharjee, A., Lustig, D.: Architectural and Operating System Support for Virtual Memory. Springer (2017)

65. Bovet, D., Cesati, M.: Understanding The Linux Kernel. Oreilly & Associates Inc (2005)

66. Wrenger, L., Rommel, F., Halbuer, A., Dietrich, C., Lohmann, D.: LLFree: Scalable and Optionally-Persistent Page-Frame Allocation. In: 2023 USENIX Annual Technical Conference (USENIX ATC 23). pp. 897–914. USENIX Association, Boston, MA (2023)

67. Arm Limited: Arm® Architecture Reference Manual for A-Profile Architecture, (2022)

68. Intel Corporation: Intel® 64 and IA-32 Architectures Software Developer's Manual, (2021)

69. Gorman, M.: Create optional ZONE_MOVABLE to partition memory between movable and non-movable pages v2, https://lwn.net/Articles/224255/, last accessed 2024/10/15

70. Nedev, S., Kamenov, V.: HDD performance research. In: Proc. 8th International Scientific Conference Computer Science, Greece, Kavala. pp. 106–111 (2018)

71. Lui, G.: Hard Drive Performance Over the Years, https://goughlui.com/the-hard-disk-corner/hard-drive-performance-over-the-years/, last accessed 2024/10/15

72. Smith, L.: Samsung 870 EVO SSD Review, https://www.storagereview.com/review/samsung-870-evo-ssd-review, last accessed 2024/10/15

73. Leis, V., Alhomssi, A., Ziegler, T., Loeck, Y., Dietrich, C.: Virtual-Memory Assisted Buffer Management. Proc. ACM Manag. Data. 1, (2023). https://doi.org/10.1145/3588687

74. Li, H., Berger, D. S., Hsu, L., Ernst, D., Zardoshti, P., Novakovic, S., Shah, M., Rajadnya, S., Lee, S., Agarwal, I., Hill, M. D., Fontoura, M., Bianchini, R.: Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. pp. 574–587. Association for Computing Machinery, Vancouver, BC, Canada (2023). https://doi.org/10.1145/3575693.3578835

75. Gorman, M.: Understanding the Linux Virtual Memory Manager. Prentice Hall (2004)

76. Shasha, D., Johnson, T.: 2q: A low overhead high performance buffer management replacement algoritm. In: Proc. 20th Int. Conf. Very Large Databases. pp. 439–450 (1994)

77. The Linux kernel contributors: Documentation for /proc/sys/vm/, https://www.kernel.org/doc/Documentation/admin-guide/sysctl/vm.rst, (2024)

78. Corbet, J.: Memory compaction, https://lwn.net/Articles/368869/, last accessed 2024/10/15

79. Corbet, J.: Making kernel pages movable, https://lwn.net/Articles/650917/, last accessed 2024/10/25

80. The Linux kernel contributors: Memory Hot(Un)Plug, https://www.kernel.org/doc/Documentation/admin-guide/mm/memory-hotplug.rst, (2024)

81. Isen, C., John, L.: ESKIMO - energy savings using semantic knowledge of inconsequential memory occupancy for DRAM subsystem. In: 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). pp. 337–346 (2009)

82. Wang, D., Ganesh, B., Tuaycharoen, N., Baynes, K., Jaleel, A., Jacob, B.: DRAMsim: a memory system simulator. SIGARCH Comput. Archit. News. 33, 100–107 (2005). https://doi.org/10.1145/1105734.1105748

83. BeagleBoard - open hardware computers for makers, educators and professionals, https://www.beagleboard.org/, last accessed 2024/10/13

84. Lee, S., Kang, K.-D., Lee, H., Park, H., Son, Y., Kim, N. S., Kim, D.: GreenDIMM: OS-assisted DRAM Power Management for DRAM with a Sub-array Granularity Power-Down State. In: MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture. pp. 131–142. Association for Computing Machinery, Virtual Event, Greece (2021). https://doi.org/10.1145/3466752.3480089

85. Kwon, Y., Yu, H., Peter, S., Rossbach, C. J., Witchel, E.: Coordinated and Efficient Huge Page Management with Ingens. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). pp. 705–721. USENIX Association, Savannah, GA (2016)

86. Panwar, A., Prasad, A., Gopinath, K.: Making Huge Pages Actually Useful. SIGPLAN Not. 53, 679–692 (2018). https://doi.org/10.1145/3296957.3173203

87. Mansi, M., Tabatabai, B., Swift, M. M.: CBMM: Financial Advice for Kernel Memory Managers. In: 2022 USENIX Annual Technical Conference (USENIX ATC 22). pp. 593–608. USENIX Association, Carlsbad, CA (2022)

88. Jeong, J., Kim, H., Hwang, J., Lee, J., Maeng, S.: Rigorous rental memory management for embedded systems. ACM Trans. Embed. Comput. Syst. 12, (2013). https://doi.org/10.1145/2435227.2435239

89. Jeong, J., Kim, H., Hwang, J., Lee, J., Maeng, S.: DaaC: device-reserved memory as an eviction-based file cache. In: Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems. pp. 191–200. Association for Computing Machinery, Tampere, Finland (2012). https://doi.org/10.1145/2380403.2380439

90. Alverti, C., Psomadakis, S., Karakostas, V., Gandhi, J., Nikas, K., Goumas, G., Koziris, N.: Enhancing and exploiting contiguity for fast memory virtualization. In: Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture. pp. 515–528. IEEE Press, Virtual Event (2020). https://doi.org/10.1109/ISCA45697.2020.00050

91. Kim, S.-H., Kwon, S., Kim, J.-S., Jeong, J.: Controlling physical memory fragmentation in mobile systems. In: Proceedings of the 2015 International Symposium on Memory Management. pp. 1–14. Association for Computing Machinery, Portland, OR, USA (2015). https://doi.org/10.1145/2754169.2754179

92. Coquelin, M.: [RFCv1 0/6] PASR: Partial Array Self-Refresh Framework, https://lore.kernel.org/all/1327930436-10263-1-git-send-email-maxime.coquelin@stericsson.com/, last accessed 2024/10/18

93. The Linux kernel contributors: Boot time memory management, https://www.kernel.org/doc/Documentation/core-api/boot-time-mm.rst, (2024)

94. perf: Linux profiling with performance counters, https://perfwiki.github.io/main/, last accessed 2024/10/27

95. Gregg, B.: Flame Graphs, https://www.brendangregg.com/flamegraphs.html, last accessed 2024/10/27

96. Czyborra, R.: Good ole' ASCII, http://czyborra.com/charsets/iso646.html

97. Bevington, P., Robinson, D.: Data Reduction and Error Analysis for the Physical Sciences. McGraw-Hill (1992)

98. The LLVM Compiler Infrastructure, https://www.llvm.org/, last accessed 2024/10/27

99. MPH2™ Managed Rack PDU, https://www.vertiv.com/4a7754/globalassets/products/critical-power/power-distribution/mph2-managed-rack-pdu-data-sheet.pdf, last accessed 2024/10/28

100. TPC-H Homepage, https://www.tpc.org/tpch/, last accessed 2024/10/27

101. TPC-H Extension – DuckDB, https://duckdb.org/docs/extensions/tpch.html, last accessed 2024/10/27

102. OpenStreetMap, https://www.openstreetmap.org/about, last accessed 2024/10/27

103. Geofabrik Download Server: North America, https://download.geofabrik.de/north-america.html, last accessed 2024/10/27

104. PostgreSQL: The World's Most Advanced Open Source Relational Database, https://www.postgresql.org/, last accessed 2024/10/27

105. Redis - The Real-time Data Platform, https://redis.io/, last accessed 2024/10/27

106. NoSQL Redis and Memcache traffic generation and benchmarking tool, https://github.com/RedisLabs/memtier_benchmark, last accessed 2024/10/27

107. The Linux kernel contributors: NO_HZ: Reducing Scheduling-Clock Ticks, https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt, (2024)