SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Improving Memory Management in the Linux Kernel for NUMA Architectures

Clément Gachod

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Improving Memory Management in the Linux Kernel for NUMA Architectures

Verbesserung der Speicherverwaltung im Linux-Kernel für NUMA-Architekturen

Author:	Clément Gachod
Supervisor:	Prof. Dr. Pramod Bhatotia
Advisor:	Jean-Pierre Lozi, PhD
Submission Date:	15.07.2023

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.07.2023 Clément Gachod

Acknowledgments

I would like to express my gratitude towards Jean-Pierre Lozi and Julia Lawall for the opportunity to conduct my research at INRIA. Their unwavering support, insightful guidance, and profound expertise have been instrumental in achieving the results of this thesis. I am immensely grateful for their contributions and mentorship, which have been critical to the success of my research.

I am sincerely thankful to Prof. Pramod Bhatotia for giving me the chance to write my Master's Thesis at the Chair of Distributed Systems & Operating Systems of the Technical University of Munich.

Abstract

Non-Uniform Memory Access (NUMA) architectures offer significant performance optimization potential for multicore systems by allowing processors faster access to local memory compared to non-local memory. However, effective memory management strategies are crucial to maximize these performance gains. The Linux kernel, widely utilized across diverse computing environments, incorporates mechanisms to manage NUMA-specific memory, yet there remain unresolved issues and opportunities for further optimization. This thesis investigates the performance anomalies observed in multicore benchmarks on NUMA hardware when the Linux kernel's automatic NUMA balancing is activated. Through a comprehensive research methodology, including exploratory experiments, detailed profiling, and the development of a proof of concept, the study aims to identify and address underlying inefficiencies.

Key findings reveal significant performance slowdowns due to false sharing of transparent huge pages, where pages allocated on one node are accessed by CPUs on other nodes, resulting in up to 30% performance degradation. A kernel code change proposed as a proof of concept to address this issue demonstrated a 9% performance improvement on the NAS benchmark CG. These promising results not only highlight the remaining potential for performance improvements in NUMA systems but also provide a foundation for future enhancements in memory management within the Linux kernel.

Abstract

Non-Uniform Memory Access (NUMA)-Architekturen bieten ein erhebliches Leistungsoptimierungspotenzial für Multicore-Systeme, da sie den Prozessoren einen schnelleren Zugriff auf den lokalen Speicher als auf den nichtlokalen Speicher ermöglichen. Um diese Leistungsgewinne zu maximieren, sind jedoch effektive Strategien zur Speicherverwaltung entscheidend. Der Linux-Kernel, der in verschiedenen Computerumgebungen weit verbreitet ist, enthält Mechanismen zur Verwaltung von NUMA-spezifischem Speicher, doch gibt es noch ungelöste Probleme und Möglichkeiten zur weiteren Optimierung. Diese Arbeit untersucht die Leistungsanomalien, die in Multicore-Benchmarks auf NUMA-Hardware beobachtet werden, wenn das automatische NUMA-Balancing des Linux-Kernels aktiviert ist. Durch eine umfassende Forschungsmethodik, einschließlich Sondierungsexperimenten, detaillierter Profilerstellung und der Entwicklung eines Konzeptnachweises, zielt die Studie darauf ab, die zugrunde liegenden Ineffizienzen zu identifizieren und zu beheben.

Die wichtigsten Ergebnisse zeigen erhebliche Leistungseinbußen aufgrund der falschen gemeinsamen Nutzung transparenter großer Seiten, bei der auf einem Knoten zugewiesene Seiten von CPUs auf anderen Knoten aufgerufen werden, was zu einer Leistungsverschlechterung von bis zu 30% führt. Eine Kernel-Code-Änderung, die als Proof-of-Concept vorgeschlagen wurde, um dieses Problem zu beheben, zeigte eine Leistungsverbesserung von 9% beim NAS-Benchmark CG. Diese vielversprechenden Ergebnisse zeigen nicht nur das verbleibende Potenzial für Leistungsverbesserungen in NUMA-Systemen auf, sondern bilden auch eine Grundlage für künftige Verbesserungen der Speicherverwaltung im Linux-Kernel.

Contents

1 Introduction

Non-Uniform Memory Access (NUMA) architectures have been a significant focus of research and development due to their potential to optimize performance in multicore systems. The NUMA architecture allows processors to access local memory faster than non-local memory, necessitating efficient memory management strategies to maximize performance gains. The Linux kernel, a prevalent operating system kernel in diverse computing environments, has incorporated various mechanisms to handle NUMA-specific memory management. However, despite more than a decade since the peak of research into NUMA optimizations, there remain unresolved issues and opportunities for performance improvements within these mechanisms.

While substantial advancements have been made in optimizing the Linux kernel for NUMA architectures, certain aspects remain elusive, with potential performance enhancements yet to be fully realized. Much of the existing codebase utilizes basic heuristics that could likely be outperformed by more sophisticated, fine-tuned management techniques. This thesis investigates these potential improvements, particularly within the context of the Linux kernel's automatic NUMA balancing mechanism, which, despite its intent to optimize memory access patterns, sometimes results in lower-than-expected performance in multicore benchmarks.

The most notable research efforts in NUMA optimization have concentrated on mitigating imbalances in memory access, where one node becomes a bottleneck due to excessive requests from other nodes [8]. Key studies have also analyzed the impact of different hardware configurations on performance, and more recently, efforts have been directed towards task grouping on cores to prevent idle periods and consequent frequency decreases [23]. These approaches, while foundational, indicate that there is still room for exploration, especially with evolving hardware capabilities and emerging workload patterns.

Despite the foundational work in NUMA optimization, there has been a noticeable decline in the research community's focus on this area in recent years. Specifically, the automatic NUMA balancing mechanism in the Linux kernel has not been extensively covered in recent research. This gap highlights an opportunity to revisit and potentially refine these mechanisms to address any underlying inefficiencies that may still exist.

The primary objective of this thesis is to investigate why certain multicore benchmarks on NUMA hardware exhibit lower-than-expected performance, particularly when Linux's automatic NUMA balancing is activated. The research aims to uncover any underlying issues contributing to these performance discrepancies and explore potential solutions to mitigate them.

The research methodology encompasses three key steps:

1. **Exploratory Experiments** : Initial experiments were conducted to identify performance anomalies and interesting behaviors in NUMA systems.
2. **Profiling** : Following the exploratory phase, a deeper analysis was performed using correlation analysis on hardware events to pinpoint the causes of performance issues. This phase also involved sampling events to identify memory access patterns.
3. **Proof of Concept** : Based on the findings, a kernel patch was designed to address the identified issues, specifically focusing on the management of transparent huge pages.

An automated testing environment was created to facilitate the experiments, leveraging Intel Performance Counter Monitor (PCM) via the perf Linux tool to record hardware events. The analysis of the collected data was conducted using Jupyter Notebooks, enabling detailed examination and visualization of performance metrics. The proof of concept involved modifying the existing automatic NUMA balancing mechanism to optimize the handling of transparent huge pages. This modification aimed to reduce false sharing and improve overall memory access efficiency.

The experiments and profiling phase revealed a significant issue related to the false sharing of transparent huge pages. These pages, allocated on one node but utilized by CPUs on different nodes, led to performance slowdowns of up to 30%. Implementing the proposed kernel patch resulted in a performance improvement of up to 8% on the NAS benchmark suite. Further optimizations yielded an additional 1% improvement, demonstrating the potential for performance gains through targeted adjustments.

This research provides foundational work towards addressing the issue of false sharing of transparent huge pages within the context of automatic NUMA balancing. The insights and solutions proposed in this thesis contribute to the ongoing efforts to optimize memory management in NUMA architectures, potentially guiding future enhancements in the Linux kernel and other operating systems.

Key contributions :

- **An automated testing environment** which facilitates the efficient execution of exploratory experiments and profiling activities. This code base leverages Intel Performance Counter Monitor (PCM) via perf to systematically collect and record hardware events, enabling precise measurement and analysis of NUMA-related performance metrics.
- **A framework for data analysis** utilizing Jupyter Notebooks. This framework allows for detailed examination and visualization of performance data, enabling the identification of patterns and correlations that underpin the observed performance issues in NUMA systems. It helped us dissect complex performance behaviors and derive meaningful insights.
- **A proof of concept kernel patch** aimed at optimizing the management of transparent huge pages within the automatic NUMA balancing mechanism. This patch addresses the issue of false sharing, in an attempt to improve memory access efficiency and overall system performance. The implementation demonstrates a tangible performance increase, validating the effectiveness of the proposed solution.

2 Background

2.1 Non-Uniform Memory Access

2.1.1 Overview

Non-Uniform Memory Access (NUMA) is a computer memory architecture used in multiprocessor systems where the time it takes for a processor to access a particular region of memory depends on where each of the two is located on the machine. This architecture can roughly be summarized as having a distributed memory structure within a single computer. In a NUMA system, the hardware resources - including processors, memory, and I/O buses, are split into several physically disjoint spaces known as nodes. These nodes are then connected together using what is called a system interconnect, allowing access to memory located on any node from any other node. In this arrangement, a processor can access the memory located on its node faster than the memory located on other nodes, respectively referred to as local memory and remote memory. The system interconnect can take the form of a crossbar or point to point links, and can be of arbitrary complexity. The following properties are thus not uncommon :

- **Asymmetry** : it can be faster to access data on node A from node B, than accessing data on node B from node A
- **Non-Transitivity** : it can be faster to access data on node A from node B via node C, than by using the link between node A and B

Before NUMA, the most common type of computer architecture was Uniform Memory Access (UMA), in which all the processors share a common memory space and thus have equal access times to all regions of the memory. It is a far simpler design to deal with, especially in the context of programming, which is most likely the reason it was so popular. However, as computers were already struggling with the von Neuman bottleneck, adding more processors in parallel made things even worse.

The von Neumann bottleneck is a term invented by John Backus in 1978 to describe a context in which processors ran many times faster than the memory they were accessing, causing them to stall and waste cycles waiting for memory. He writes : “in

its simplest form a von Neumann computer has three parts: a central processing unit (or CPU), a store, and a connecting tube that can transmit a single word between the CPU and the store (and send an address to the store). I propose to call this tube the von Neumann bottleneck. The task of a program is to change the contents of the store in some major way; when one considers that this task must be accomplished entirely by pumping single words back and forth through the von Neumann bottleneck, the reason for its name becomes clear". [3]

With the advent of multiprocessing systems, there was even more pressure put on the memory, and with only a single bus to handle all of these accesses in the UMA design, it was inevitable that it would cause a bottleneck. Initially, large CPU caches and sophisticated cache management algorithms helped mitigate the performance impact by reducing main memory access. But as operating systems, applications, and data sets continued growing rapidly in size, the caches were eventually overwhelmed. NUMA architectures emerged as an innovative solution, providing a distributed memory design that could finally scale memory access performance as more processors were added. By giving each CPU its own local memory pool, NUMA avoided the heavy contention for shared memory pathways in UMA systems. For workloads with sufficiently distributed data, NUMA could provide performance scalability roughly linear to the number of processors or memory nodes in the system. This drove the adoption of NUMA in servers, HPC systems, high-end workstations, and other areas where data parallelism and scalable memory performance were critical. [22]

Nowadays, NUMA is widely used in servers, high-performance computing clusters, and other systems that need to scale memory access beyond what is possible with a shared UMA design. Major processor vendors like AMD and Intel now build NUMA capabilities into their server and workstation CPUs and chipsets. Nodes are now interconnected using specialized high-bandwidth system interconnects like AMD's HyperTransport [7], Intel's QuickPath Interconnect [17], SGI's NUMALink [16], or other similar technologies. This allows processors to access and share the full system memory address space, while keeping the performance penalty of remote wire delays below 30% [5] in most cases.

2.1.2 Cache Coherency

In computer architecture, a cache is a specialized, high-speed memory device that serves as temporary storage for frequently accessed data or instructions [39]. It allows the processor to access data at a much higher rate than it would if it was communicating directly with the main memory, and as such it is one way to solve the von Neumann

bottleneck mentioned above. The concept of cache is predicated on the principles of locality of reference, which observe that computer programs often access the same sets of data or data located near each other repeatedly over short periods. By storing these data points in a faster-access memory cache, the system reduces the number of time-consuming memory access cycles to slower main memory, speeding up data retrieval and increasing overall system performance.

To further exploit the potential of caches to improve computer performance, an effective method called multi-level caching [38] consists in stacking them in a hierarchical manner. In most of today's architectures, 3 levels are being used : L1, L2 and L3. Each level represents a trade-off between speed and size. The L1 cache is the fastest but also the smallest in terms of how many bytes of memory it can hold, and is usually further divided into an instruction cache and a data cache. The L2 and L3 caches on the other hand are unified, meaning they store both instructions and data, and are increasingly larger and slower. When a processor needs to fetch data or instructions, it first checks the L1 cache. If the requested item is found (referred to as a cache hit), it can be rapidly retrieved and processed. If the item is not found however (referred to as a cache miss), the processor proceeds to check the next levels of cache (L2, then L3 if available). If the item is still not found in any cache level, it must be fetched from the main memory, incurring a significant delay.

There exist various policies and algorithms to decide which data or instructions to keep in the cache and which ones to evict in order to make space for incoming items: Least Recently Used (LRU) [20, 30], Pseudo-LRU, and random replacement are all examples of common replacement policies. In modern processors, caches are integrated directly into the processor chip, and these replacement algorithms are implemented in hardware for faster execution. The size and organization of these caches vary depending on the processor architecture and design objectives, which aim to balance performance, power consumption, and cost. By leveraging the principles of temporal and spatial locality—the tendency for programs to access the same or nearby data and instructions repeatedly—caches can significantly enhance overall system performance. This improvement is achieved by reducing the number of slow main memory accesses, thereby allowing the processor to operate more efficiently.

A fundamental concept when it comes to caches that are part of a multiprocessor system is **coherency**. If caches were completely independent, a situation could arise in which two caches both have a modified version of a particular piece of data. When the time comes to write that data back to main memory, only one of the versions

will actually be persisted and the other will be lost. This is where cache coherency mechanisms come in : they make sure all the caches in the machine maintain a consistent view of the memory, preventing processors from accessing stale or inconsistent data. Although some NUMA (Non-Uniform Memory Access) systems operate without cache coherency mechanisms [18], programming such systems within the standard von Neumann architecture model is prohibitively complex. That is why most NUMA systems today come with a cache coherency protocol in a design called ccNUMA, for cache coherent Non-Uniform Memory Access.

Cache coherency in a NUMA architecture is challenging to achieve efficiently due to the significant overhead associated with maintaining consistency across distributed caches [29], particularly as the number of processors scales. Protocols such as MESIF (Modified, Exclusive, Shared, Invalid, Forward) address this challenge by reducing the communication overhead required to maintain coherency, thereby optimizing data transfers between caches. Similarly, the Scalable Coherent Interface (SCI) [1], an IEEE standard, employs a directory-based approach to manage cache states and ensure consistent memory views across processors. Still, when multiple processors frequently access the same memory location, the continuous synchronization and communication between their caches can lead to increased latency and performance bottlenecks [11, 12]. These issues are particularly pronounced in applications with high contention for shared data.

2.1.3 Performance considerations

In NUMA systems, optimal performance hinges on the strategic placement of tasks and memory, as accessing local memory within the same node is faster than accessing memory from remote nodes. Therefore, it is crucial for both application code and operating system kernels to incorporate sophisticated algorithms designed to optimize task scheduling and memory allocation. These mechanisms aim to maximize local memory accesses and minimize cross-node traffic, which can degrade system performance due to increased latency. By effectively managing memory and task placement, operating systems can leverage the inherent advantages of the NUMA architecture to enhance application efficiency and overall computational speed.

However, locality of memory accesses is not the only factor playing a role in the performance of applications running on NUMA hardware. Indeed, the following have been identified as relevant as well :

- **Memory congestion** : “Massive data traffic creates congestion in memory controller queues and on interconnects. When this happens, memory access latencies

can become as large as 1000 cycles, from a normal latency of only around 200. Such a dramatic increase in latencies can slow down data-intensive applications by more than a factor of three". [8] This is particularly pronounced in systems where memory-intensive applications generate substantial data traffic across nodes. Such congestion increases memory access latencies, which can severely slow down applications. Effective management of this congestion involves advanced memory placement strategies that go beyond simple locality optimization, aiming instead to manage overall traffic flow within the system to prevent hotspots and balance load across nodes.

- **Interconnect asymmetry** : "the asymmetry of the interconnect in modern NUMA systems drastically impacts performance". [24] Different nodes may be connected by links of varying bandwidths and latencies, leading to non-uniform communication speeds across the system. For example, some links might be faster in one direction than the other, or some might be shared between more nodes, increasing traffic and potentially leading to congestion. Such disparities in interconnect architecture mean that the physical placement of threads and memory across nodes becomes crucial, as the bandwidth available between nodes can vary significantly. This complexity necessitates advanced placement algorithms that can intelligently distribute processes and memory to optimize communication paths and minimize latency, thus enhancing overall system performance.

- **General unpredictability** : The performance of different placements of virtual cores onto physical cores in NUMA systems can be highly unpredictable. This unpredictability stems from the complex interactions between hardware resources, like CPU caches and memory, which can vary significantly between different systems. For example, a placement strategy that works well on one type of hardware may perform poorly on another due to differences in memory hierarchy or interconnect architecture. Additionally, the optimal placement is not only hard to predict, but also difficult to achieve, especially when attempting to balance the number of used nodes with performance goals. [14] This complexity makes it challenging to determine the best-performing placement in advance, often requiring dynamic testing and adaptation to the specific characteristics of each system and workload.

2.2 The Linux Kernel

Description
The Linux kernel is the central component of a Linux operating system, acting as the core interface between the computer's hardware and its processes. It ensures efficient resource management and mediates between hardware components - such as memory, CPUs, and input/output devices - and the software applications that operate on them. The kernel operates primarily in "kernel space", an exclusive area where it performs critical functions invisible to users, such as memory allocation and process management. Users interact with the system through "user space", where applications function and communicate with the kernel via a system call interface.

The kernel has multiple key responsibilities, including memory management, which tracks and allocates memory usage; process management, which schedules process access to the CPU; handling device drivers, serving as a bridge between hardware and software processes; and managing system calls and security, processing service requests from various applications. This structure ensures that if a user-level process fails, the impact is contained, preventing widespread system damage, unlike a kernel-level failure which can lead to complete system crashes.

A Linux machine can be conceptualized as having three layers: the hardware, which includes the physical components like the CPU and RAM; the Linux kernel, which resides in memory and directs the hardware operations; and user processes, which include all the running applications managed by the kernel. These processes also engage in inter-process communication (IPC) as facilitated by the kernel. The separation between kernel mode, which has unrestricted hardware access, and user mode, which is limited to interfacing through the system call interface, plays a crucial role in system security and stability, allowing for operations like live patching without downtime.

History
Linux kernel, initiated by Linus Torvalds in 1991, has seen continuous development and expansion over the decades [25]. Started as a simple hobby project for 386 AT clones, the early versions were foundational, with version 0.01 in 1991 and a significant jump to version 0.11 by December 1991 which was self-hosted. By 1994, version 1.0.0 marked Linux's capability to operate as a stable system on single-processor i386 machines. The kernel evolved with the release of version 2.0 in 1996, introducing support for symmetric multiprocessing (SMP), thereby enhancing its utility for more complex, multi-processor environments. The 2.6 series, launched in 2003, broadened this scope significantly by improving scalability [6] and supporting a broader range of devices.

This series also integrated the Native POSIX Thread Library and SELinux, enhancing both functionality and security. The transition to the 3.x series started in 2011 with version 3.0, focusing more on system performance and security enhancements. By 2015, the kernel reached version 4.0, which introduced live kernel patching, allowing users to update systems without rebooting. With each of these versions, Linux has consistently expanded its support for hardware, improved its file systems, and enhanced security protocols to address the growing demands of modern computing infrastructure. After that, the Linux kernel advanced further into the 5.x series, which began with version 5.0 released in 2019. This series continued to extend support for newer hardware, particularly modern GPUs and CPUs, reflecting Linux's commitment to adapting to technological advancements.

More recently, version 5.0 and subsequent updates brought significant performance enhancements for both desktop and server environments, alongside improvements in file system capabilities with continued updates to ext4 and Btrfs among others. The 5.x series also introduced important updates to network handling and security measures, such as the inclusion of WireGuard, a simpler and more efficient VPN protocol, in version 5.6. Enhancements in scheduler and memory management ensured better handling of high-load scenarios and complex computational tasks, crucial for enterprise-level operations.

Finally, the 6.x series of the Linux kernel, initiated with version 6.0 in October 2022, represents significant strides in hardware support, performance, and security enhancements. This series introduced robust features such as kernel modules in Rust with version 6.1, stable Intel Arc drivers in 6.2, and extensive performance improvements across CPUs like Intel Xeon and AMD Ryzen in 6.0. Noteworthy developments include the integration of Rust for better reliability, **the EEVDF scheduler in 6.6 replacing the older CFS scheduler** for optimized process handling, and significant filesystem advances such as the introduction of Bcachefs in 6.7 and continuous enhancements to Btrfs and Ext4. Each version aimed to enhance compatibility with modern technology, notably with ongoing support for Intel's Meteor Lake and AMD's technologies across several releases, culminating in specialized improvements like faster boot times and better support for 4K displays in version 6.9. The evolution throughout the 6.x series underscores Linux's commitment to adapting to cutting-edge technological trends while bolstering system efficiency and security.

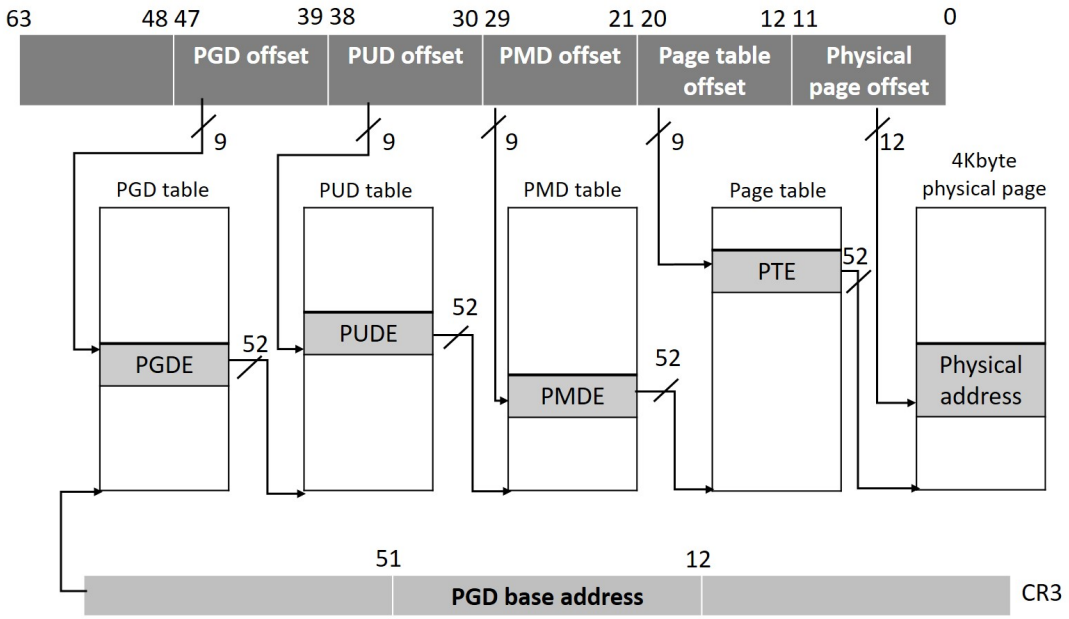

Figure 2.1: Page Table layout in the Linux kernel for x86_64 architecture. [4]

2.2.1 Memory Management

This paragraph will briefly introduce how memory management works in the Linux Kernel, as some of the concepts will be used in the coming section about automatic NUMA balancing. As described in the above paragraph, one of the key responsibilities of the Linux Kernel is to be the interface between applications and the physical memory of the computer. This is what is referred to as memory management. For various reasons that we are not going to discuss here, the Linux Kernel - as well as some hardware components - use an intermediary memory address space called virtual memory [9]. Because of that, the process of accessing memory from a process is somewhat complex, and here are the main steps involved [32] :

1. **Virtual Address Generation** : When a program executes, it generates virtual addresses. These addresses are split into multiple parts, each part targeting a different level of the page table structure in systems using multi-level page tables like Linux. The parts typically include a segment pointing to the Page Global Directory (PGD), another pointing to the Page Middle Directory (PMD), and a third pointing to the Page Table Entry (PTE), with the final part of the address representing the offset within the actual physical page.

2. **Page Table Walk**, see Figure 2.1 :
    - The process starts when the CPU needs to translate a virtual address into a physical address. This translation begins at the highest level of the page table (PGD), which contains a pointer to the next level (PMD).
    - The PGD index derived from the virtual address leads to a PMD. The PMD further points to a page table that contains PTEs.
    - Each PTE then points to a physical page frame, with the final offset specified in the virtual address pointing to the specific byte in the physical page.
    - If any level of the page table does not have a valid entry (indicating the data is not in memory), a page fault is generated, prompting the OS to fetch the required data from disk and update the page tables accordingly.
3. **Translation Lookaside Buffer (TLB)** :
    - To speed up the translation process, CPUs use a TLB, which is a special cache that stores recent translations of virtual addresses to physical addresses. Before performing a page table walk, the CPU first checks the TLB to see if the translation for the virtual address is already available, significantly speeding up memory access.
    - If the translation is found in the TLB (a TLB hit), the physical address is immediately available, and the page table walk is skipped. If the translation is not in the TLB (a TLB miss), a page table walk is necessary.
    - After a successful page table walk, the new translation is added to the TLB. If the TLB is full, it may use a replacement algorithm (like least recently used) to make space for the new entry.
4. **Memory Access** : Once the physical address is determined (either via the TLB or a page table walk), the CPU can access the physical memory at that address.
5. **Page Replacement** : If physical memory is full when a page fault occurs, the OS must choose a page to evict, typically using a page replacement algorithm (like LRU, FIFO, etc.). The evicted page may be written back to disk if it has been modified.
6. **Handling Page Faults** : During a page table walk, if a required page is not found in memory (indicated by an invalid or absent entry in the page table), a page fault occurs. The operating system then needs to load the missing page from disk into physical memory, update the page table to reflect the new mapping, and then restart the instruction that caused the page fault.

This entire mechanism ensures that virtual memory provides an effective abstraction for managing the system's RAM, allowing multiple processes to run concurrently without directly managing physical memory. This architecture not only maximizes the use of physical memory but also isolates processes, improving security and stability.

2.2.2 Automatic NUMA balancing

Even though there had already been various improvements to support NUMA hardware better as early as 2003 in the version 2.5 of the Linux kernel [10], some Linux Kernel developers set out to work on better NUMA scheduling in 2012 [40]. Initially, two distinct approaches emerged to address the performance challenges associated with NUMA's memory and process locality. On one hand, Peter Zijlstra introduced the sched/numa patch set which emphasized a "lazy migration" mechanism for moving pages close to the processing node and introduced a "home node" concept aimed at minimizing the migration of processes across NUMA nodes. Conversely, Andrea Arcangeli's AutoNUMA patch set focused on actively migrating pages based on usage patterns identified through a tracking mechanism that analyzed page faults. [2] The disagreement between these approaches highlighted differing philosophies regarding overhead and scheduler interactions.

Despite these early efforts, no clear consensus emerged, leading to the involvement of other developers like Mel Gorman. Gorman introduced a new patch series titled "Foundation for automatic NUMA balancing," which aimed to lay down a basic infrastructure that allowed for easier experimentation with different NUMA placement policies. His approach, initially using a simple policy humorously named "MORON" (Migrate On Reference Of `pte_numa` Node), was designed to be a foundational layer upon which more sophisticated policies could be built. This infrastructure was crucial as it facilitated further enhancements and provided a standardized framework for ongoing NUMA-related developments. Gorman advocated for his patch by demonstrating its practicality and applicability across a wide range of computing scenarios. [26]

By 2013, Gorman's NUMA balancing infrastructure was merged into the Linux 3.8 kernel. This merging marked a pivotal moment in the kernel's development, setting the stage for more refined balancing mechanisms. Gorman's subsequent patches continued to evolve the NUMA balancing capabilities of the kernel. In particular, the concept of NUMA groups was introduced to manage cases where multiple processes share memory. This feature was designed to optimize the placement of processes that access common sets of pages, enhancing the efficiency of memory usage across shared applications.

Functionally, automatic NUMA balancing is articulated around 3 primary operations:

1. **Memory Scanning** : This task involves a periodic update of page table entries to a protected status (`prot_numa`), which blocks any process from accessing these pages while still permitting kernel access. When a process attempts to access a protected page, it incurs a page fault. This fault is classified by the kernel as a "NUMA hinting fault." Subsequently, the kernel captures detailed information related to the NUMA context of the attempted access, such as the accessing node and CPU. It also updates relevant statistics. After logging this information, the kernel restores access to the page table entry and returns control to the application.

2. **Page Migration**: While handling a NUMA hinting fault, an evaluation is made to determine if moving the page to a different node would benefit the application's performance. This decision is based on the statistics gathered during the memory scanning phase. If migration is deemed advantageous, the page is transferred to the designated "target node."

3. **Task Migration**: Beyond page relocation, automatic NUMA balancing includes a procedure for determining the optimal node for the currently running task, identified typically as the node where the most page faults occur. This decision is complex and aims to reposition the process to a node that maximizes efficiency. However, task migration presents challenges, particularly as it may conflict with the decisions of Linux's built-in scheduler, making it arguably the most intricate component of the NUMA balancing strategy.

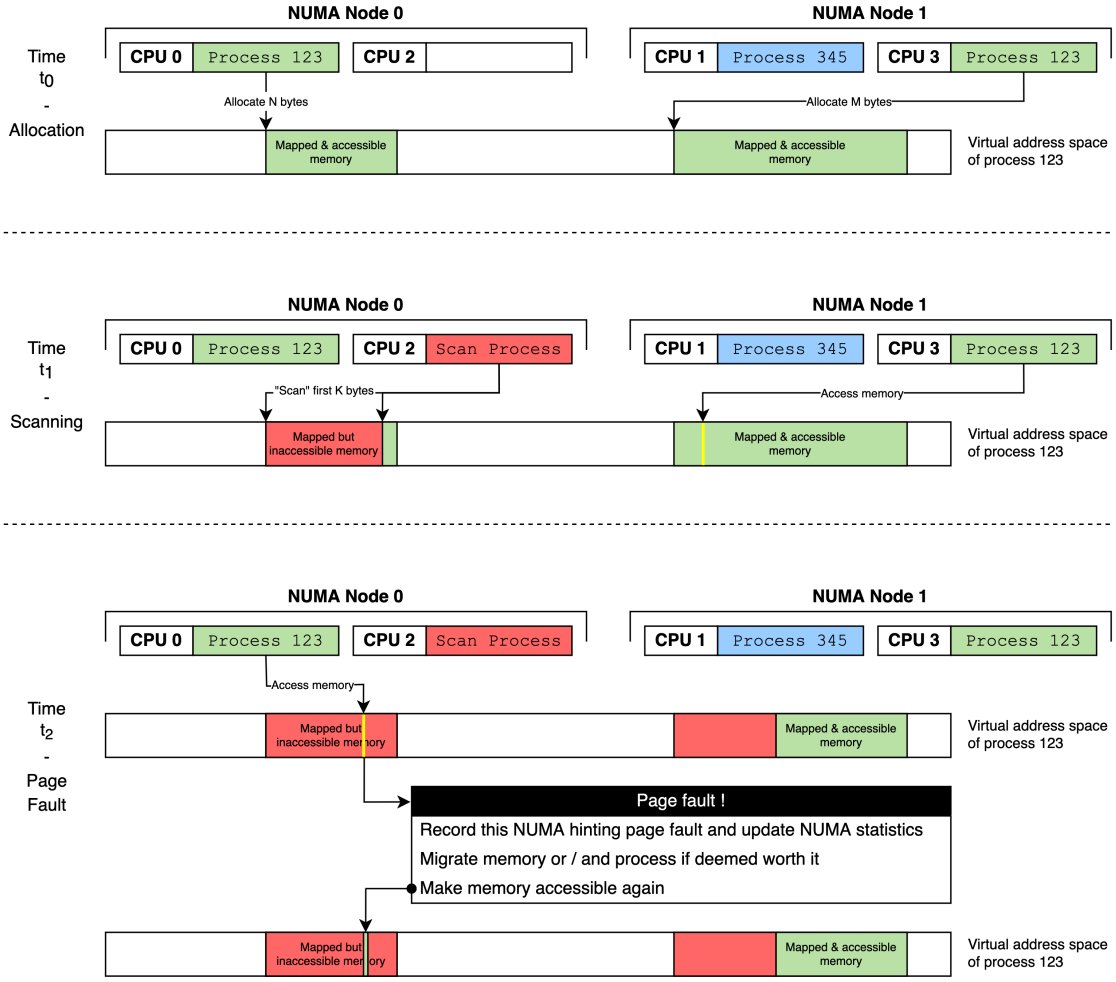

Figure 2.2: Diagram showing how the memory scanning process from automatic NUMA balancing works

2.2.3 Transparent Huge Pages

Transparent Huge Pages (THP) is a feature in the Linux operating system designed to enhance the performance of memory-intensive applications by utilizing larger memory pages, known as huge pages, instead of the standard smaller pages. Currently, THP supports anonymous memory mappings and tmpfs/shmem, with potential expansion to other filesystems. [42]

The performance improvements offered by THP are primarily due to two key factors. The first factor, though less significant, involves reducing the number of page faults during initial memory access. Normally, accessing new memory regions incurs a page fault, where the operating system must map the required memory page. With huge pages, a single page fault can map a large 2MB memory region, reducing the frequency of kernel entry/exit operations by a factor of 512 compared to standard pages of size 4KB. However, this factor has a downside: larger clear-page and copy-page operations during page faults can introduce latency. This impact is generally minimal, as it only occurs during the first access to each memory region.

The second, more impactful factor relates to TLB performance. The TLB caches the mappings from virtual memory addresses to physical memory addresses. Huge pages improve TLB efficiency in two ways:

1. **Faster TLB Miss Resolution** : When a TLB miss occurs (i.e., the TLB does not have the required mapping), the system must perform a more time-consuming lookup. With huge pages, this lookup is faster because the system can handle larger memory regions more efficiently. This is particularly beneficial in virtualized environments with nested page tables, where both the host (KVM) and the guest operating systems can use huge pages to streamline memory access. Even if only one of these systems uses huge pages, the TLB miss resolution process is still expedited.

2. **Reduced TLB Miss Frequency** : Each TLB entry can map a much larger amount of memory when huge pages are used. For example, a single 2MB huge page requires only one TLB entry, compared to 512 entries for 4KB pages. This significantly reduces the number of TLB misses, as fewer entries are needed to cover the same amount of memory. The reduction in TLB misses translates to fewer performance penalties, as the system spends less time handling memory lookups and more time executing application code.

Modern implementations of THP, referred to as multi-size THP (mTHP), introduce additional flexibility by allowing memory allocations in various larger block sizes [27]. This approach helps to mitigate latency spikes typically associated with larger memory pages, while still reducing the overall number of page faults. THP can be turned on for the whole system or just for specific applications. A background process called khugepaged helps by combining smaller pages into huge pages over time. Unlike older methods that required reserving huge pages ahead of time (which could waste memory), THP uses available memory more efficiently.

In summary, THP provides a robust mechanism for improving the efficiency of memory management and enhancing application performance, making it a valuable feature for systems running memory-intensive workloads.

2.3 Observation and Profiling Tools

2.3.1 Intel Performance Counter Monitor

Intel Performance Counter Monitor (PCM) operates by leveraging the built-in Performance Monitoring Units (PMUs) within Intel processors. These PMUs are specialized hardware components designed to track various low-level events that occur within the CPU during its operation. The PMUs include a set of counters that can be programmed to monitor specific types of activities, such as instructions executed, cache hits and misses, memory accesses, and data traffic on interconnects like the Intel® QuickPath Interconnect (QPI) [17].

To support PCM, Intel processors have been extended with both core and uncore PMUs. Core PMUs monitor events occurring within individual processor cores, such as instruction execution and cache performance. Uncore PMUs, on the other hand, monitor activities in parts of the processor that are shared among cores, such as the integrated memory controller and QPI links. This distinction allows PCM to provide a comprehensive view of both core-specific and system-wide performance metrics.

The hardware implementation involves configuring these PMUs to capture specific events of interest. This is done through a set of control registers that define which events are monitored and how the counters should operate. Once configured, the PMUs continuously track the specified events, updating their counters in real-time. These counters can be read by software at any time to gather performance data.

Intel processors also include support for advanced features like Intel® Turbo Boost and Intel® Hyper-Threading Technology, which introduce additional complexities in performance monitoring. PCM takes these features into account by providing detailed metrics that reflect the dynamic behavior of the processor under varying workloads. Overall, the hardware extension for PCM includes:

- **Performance Monitoring Units (PMUs)**: Core and uncore PMUs with configurable event counters.
- **Control Registers**: Registers to configure the PMUs and select specific events to monitor.
- **Data Registers**: Counters that store the accumulated counts of monitored events.

By integrating these hardware extensions, Intel processors can provide detailed and accurate performance metrics which we have relied on heavily in our study.

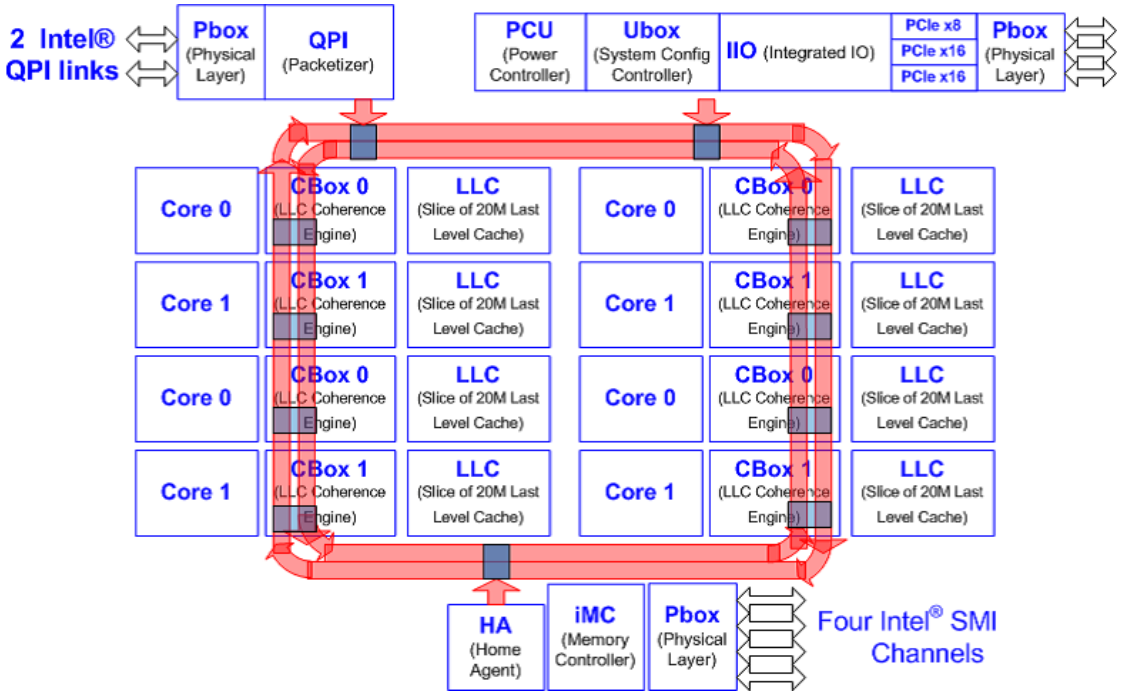

Figure 2.3: Intel® Xeon® E5 series block diagram. The Xeon E5 series processor's uncore has multiple 'boxes' similar to the Xeon E7 processor. Intel PCM v2.0 supports Intel® QPI and memory metrics for the new processor. [19]

2.3.2 FTrace and Trace-cmd

Ftrace, or Function Tracer, is an internal tracing framework within the Linux kernel designed to assist developers and system designers in understanding the internal workings of the kernel [13]. Primarily used for debugging and performance analysis, Ftrace provides insights into the kernel's operations outside the user-space, such as identifying latency issues and analyzing preemption and scheduling activities. At its core, Ftrace offers a suite of tracing utilities, including function tracing, event tracing, and latency tracing, which collectively help in pinpointing and diagnosing issues within the kernel.

Event tracing is particularly noteworthy as it leverages static event points scattered throughout the kernel, allowing users to monitor specific kernel events by enabling them through the `tracefs` file system. Ftrace's functionality is deeply integrated with the `tracefs` file system, where control files and output files are located, facilitating user interaction with the tracing mechanisms. These files, such as `available_tracers`, `current_tracer`, `tracing_on`, `trace`, and `trace_pipe`, allow users to configure tracers, manage tracing activities, and retrieve trace outputs. The trace data, typically in microseconds, is crucial for understanding the kernel's behavior in real-time or through detailed logs. By providing a granular view of the kernel's operations, Ftrace is indispensable for diagnosing complex issues and optimizing system performance.

`trace-cmd` is a command-line tool that acts as a wrapper for the Ftrace functionality [41], simplifying the process of kernel tracing for users. It provides a more user-friendly interface for managing Ftrace operations, allowing users to start, stop, and manage traces without directly interacting with the `tracefs` file system. `trace-cmd` abstracts the complexity of Ftrace, offering commands to record, extract, and view trace data efficiently. It supports various options and commands to filter specific events, manage trace buffers, and configure tracing parameters, making it a useful tool for developers who need to perform detailed kernel analysis without diving into the lower-level intricacies of Ftrace. By leveraging `trace-cmd`, users can quickly set up and execute tracing sessions, capture detailed performance data, and analyze the kernel's behavior with greater ease and accuracy. This tool enhances the accessibility and usability of Ftrace, enabling more effective troubleshooting and performance optimization in kernel development.

2.3.3 Perf

`perf` is a versatile performance analysis tool in the Linux environment designed to monitor and measure various hardware and software events. It operates primarily

through two modes: counting and sampling [35]. In counting mode, `perf` aggregates the occurrences of specified events during process execution, such as cache misses, branch mispredictions, and CPU cycles, presenting these statistics upon completion. `perf` can handle multiple events concurrently through time multiplexing when the number of events exceeds the available hardware counters. This method ensures that each event is periodically given access to the monitoring hardware, although it introduces estimation errors as events are not monitored continuously. The collected data is scaled to provide an estimate of what the event count would have been if measured throughout the entire run.

In sampling mode, `perf` leverages event-based sampling, recording a sample when the counter overflows. This mode captures the state of the program, including the instruction pointer, at the moment of interruption. However, due to interrupt handling delays, the recorded instruction pointer may not precisely indicate where the counter overflowed, requiring careful interpretation. `perf`'s default event for sampling is cycle counting, but it supports a broad range of hardware and software events, offering detailed insights into system and application performance. This functionality makes `perf` an invaluable tool for diagnosing performance bottlenecks, optimizing code, and understanding system behavior under various workloads.

3 Overview

3.1 System Overview

3.1.1 Hardware

In the context of our research, it was crucial to conduct all tests and benchmarks on physical hardware rather than virtual machines. The abstraction layer in virtual machines, governed by the hypervisor, generally isolates programs from hardware constraints, which could obscure any NUMA related behaviors critical to our study. Fortunately, Inria is part of the Grid'5000 network which provided us with exactly this ability.

Grid'5000 is a distributed research infrastructure designed to support experimental computer science, particularly focusing on areas such as parallel and distributed computing, Cloud, HPC, Big Data, and AI. This testbed enables a wide array of scientific explorations thanks to its provision of significant computational resources : 15,000 processing cores across 800 machines - also known as compute nodes - that are organized into 31 homogeneous clusters across multiple geographical locations, called sites. These clusters are equipped with a variety of advanced technologies such as Persistent Memory (PMEM), Graphics Processing Units (GPUs), Solid State Drives (SSDs), Non-Volatile Memory express (NVMe), and diverse networking technologies including 10G and 25G Ethernet, Infiniband, and Omni-Path. One of the core strengths of Grid'5000 lies in its highly reconfigurable nature, allowing researchers complete control over their software stacks through bare-metal deployments and network isolation capabilities. This flexibility is crucial for conducting reproducible and traceable experiments under varied and controlled conditions. Moreover, it offers sophisticated monitoring and measurement tools that collect detailed traces of networking and power consumption, thus providing deep insights into experimental behaviors and outcomes.

Grid'5000 operates on a reservation-based system where users can book computational resources as needed. The primary interface for making reservations is through a frontend machine specific to each site, enabling users to allocate nodes and clusters on this site according to their project needs. Users specify the type and quantity of

resources they need, such as processors or entire nodes, for a determined time period. This system allows for precise scheduling, ensuring that resources are available for exclusive use during the reserved times. Several reservation modes are available, among which is “deployment”. It grants root access on selected nodes, enabling the installation and configuration of any required software stack down to the operating system.

Of all the available machines, we have mostly used the ones from the following clusters :

Name of the cluster	**dahu**
Number of machines	32
Manufacturing year	2017
Number of CPUs (1 NUMA node per CPU)	2
CPU model	Intel Xeon Gold 6130
Number of cores per CPU	16
Architecture	x86_64
Main memory	192 GiB of RAM

Table 3.1: Specifications of the dahu cluster

Name of the cluster	**yeti**
Number of machines	4
Manufacturing year	2017
Number of CPUs (1 NUMA node per CPU)	4
CPU model	Intel Xeon Gold 6130
Number of cores per CPU	16
Architecture	x86_64
Main memory	768 GiB of RAM

Table 3.2: Specifications of the yeti cluster

3.1.2 Software

Although there was little doubt around the fact that we were going to use a Linux operating system to carry out our work, we still had to choose a distribution for it. Indeed, Linux itself is not a monolithic entity but rather a family of operating systems underpinned by the Linux kernel, distributed in various formats known as distributions.

Each distribution bundles the Linux kernel with a collection of software applications and management tools, offering different configurations and user experiences. While there are numerous Linux distributions, each with its unique features and configurations, their core functionalities are quite similar since they are all built around the Linux kernel. For our experimental setup, the specific choice of Linux distribution was less crucial since our interest lied in the kernel. Still, we had to pay attention to the following:

- The distribution might include extraneous services that could potentially operate concurrently with our experiments, thereby contaminating our results with unforeseen interactions or resource drain.
- The distribution might come with an old or modified version of the Linux kernel

To address those concerns, we opted for Debian 12, a distribution known for its stability and minimalism. To further align the operating system with our experimental needs and mitigate any kernel-related variables, we recompiled the latest version of the Linux kernel to replace the default one provided with Debian. This approach allowed us to maintain a consistent and controlled environment for our studies.

Regarding the benchmarking tools used, our main go-to was the NAS Parallel Benchmarks (NPB) [28]. It is a suite of programs developed to assess the performance capabilities of parallel supercomputing systems. Originally derived from computational fluid dynamics applications, the benchmarks consist of both kernels and pseudo-applications designed to mimic various computational patterns relevant to scientific computing. The benchmarks are especially valuable for evaluating the effects of NUMA on system performance because they include a range of memory access patterns and communication strategies. This variation in memory and communication complexity makes NPB particularly well suited for studying how different hardware architectures, particularly those with NUMA configurations, handle diverse computational loads. In our study, we utilized a significant subset of these benchmarks to investigate performance variations under NUMA effects, leveraging both the original benchmarks and their extended versions in NPB 3.4.3, which incorporate additional problem classes and programming models. This comprehensive approach allowed us to effectively measure and analyze the impact of NUMA on parallel computational efficiency.

3.2 Goals

First of all, the core work on NUMA balancing as described in the Background section is now over a decade old. While this may appear insignificant, it is crucial to consider that the kernel has undergone extensive modifications during this period. Not only has it substantially increased in complexity, but has also seen significant modifications, notably in its scheduling algorithm. With the release of Linux Kernel 6.6, the Earliest Eligible Virtual Deadline First (EEVDF) scheduler has replaced the Completely Fair Scheduler (CFS) as the default, marking a pivotal shift in process scheduling. In a similar fashion, hardware platform have evolved substantially over the last decade. Consequently, an auxiliary aim of this study is to evaluate the resilience of the NUMA balancing mechanisms through time. Specifically, we wanted to assess whether the NUMA balancing infrastructure remains effective and relevant within the context of the NAS benchmarks.

Moreover, the focus of this study emerged from initial observations suggesting that Linux might not perform optimally on NUMA architectures under certain conditions when executing a variety of unrelated tests. Leveraging the NAS benchmarks and the specified hardware setup, this study was structured around the following goals, in that order :

1. The primary objective was to **validate or refute** the preliminary suspicion that the **performance of multicore applications on NUMA systems under Linux might occasionally deteriorate or fail to benefit from NUMA balancing mechanisms**. This involves a thorough assessment to assert if there are specific scenarios where NUMA balancing either does not significantly enhance performance or potentially degrades it.

2. The second objective was to **dive into the workings of the NUMA balancing code to comprehend its complexities and details**. This exploration aims to determine whether the nuances of its implementation could explain the performance behaviors observed during the preliminary tests.

3. The third objective focused on **identifying the root causes of any inefficiencies** detected through these investigations. This involved not only pinpointing the issues but also, if necessary, conducting further experiments to explore these inefficiencies more deeply.

4. Finally, this study aimed to **develop and propose a strategy to address these inefficiencies**. The plan would include designing solutions and implementing them to the extent feasible within the constraints of the available time frame.

3.3 Challenges

3.3.1 Configurability

The first obstacle we encountered was the sheer complexity of the Linux Kernel's configuration system. The Linux Kernel is renowned for its extensive customizability, which is reflected in its vast array of configuration parameters. They are so numerous and intricate in fact that the kernel compilation process includes an interactive tool - accessible via the command make menuconfig — specifically designed to facilitate their management. This complexity led to two primary complications :

- **Selection of Configuration Parameters** : It was not easy figuring out which configurations to test for. Given the time constraints of our research project, we chose to focus on the default configuration provided by our Linux distribution. However, it is worth noting that even with an extended research timeline, comprehensively covering even half of the possible configuration permutations would have been a formidable task. This highlights the vast scope of potential kernel configurations and the difficulties inherent in conducting exhaustive testing.
- **Code Complexity and Readability** : The second, and arguably more problematic, issue was the impact of these numerous configuration parameters on the readability of the kernel code. It is common for functions within the kernel to invoke different functions based on one or several configuration parameters. This leads to increasingly complex branching, which significantly complicates the task of understanding the Linux kernel's code structure at a deeper level. This branching complexity escalates rapidly, making it exceptionally challenging to construct a coherent, low-level representation of the Linux kernel's code structure. Consequently, this impediment significantly complicates efforts to analyze and understand the kernel's inner workings at a granular level.

3.3.2 Observability

Another challenging aspect of kernel development is its observability. Due to its foundational position in the software stack, the kernel presents unique observational challenges that are not typically encountered in higher-level software development.

- Firstly, certain kernel functions are deeply integrated with the hardware, complicating their monitoring. This is the case of memory management for example, where in most cases the table walk mentioned earlier is entirely carried out in hardware. In such scenarios, direct observation is hindered unless the hardware is specifically designed with built-in instrumentation capabilities. Even with some

monitoring tools available, there remains an inherent limitation in the precision with which hardware operations can be observed, a constraint not applicable to pure software environments.

- Additionally, the process of obtaining a crash report in kernel development differs considerably from that in traditional application development. A kernel crash usually precipitates a complete system failure, necessitating a system reboot and resulting in the loss of any unsaved data. This complicates the process of diagnosing the cause of the crash, which is a critical step in debugging. However, the kdump tool provides a valuable mechanism for capturing information about the crash. It utilizes the kexec system call to boot into a secondary kernel without needing to reboot the system, thereby preserving the memory state of the primary kernel at the time of the crash for subsequent analysis. This capability significantly aids in understanding and addressing kernel crashes, albeit within the constraints imposed by the intrinsic characteristics of kernel-level operations.

3.3.3 Time

The last major hurdle encountered in this study was managing time effectively. Specifically, compiling the Linux kernel proved to be a highly time-intensive task. On the more powerful systems, compilation required approximately 8 minutes, while on slower ones it could take as long as 15 minutes. Subsequent recompilations after modifications were somewhat quicker than the initial compilation, yet rarely under the 2 minutes mark. Furthermore, in scenarios involving a kernel crash, significant time had to be allocated for the following processes:

- Restarting the system to recover from the crash.
- Recompiling the kernel with necessary adjustments.
- Rebooting the system to apply the changes.

On top of that, there were occasional instances where the kernel unexpectedly needed to be recompiled from scratch following a system reboot, without any discernible cause. These factors collectively posed substantial obstacles to achieving meaningful progress within the six-month duration of the study.

4 Design

4.1 Exploratory Experiments

In the initial phase of our study, we aimed to validate our hypothesis regarding potential performance issues in a NUMA context. Our objective was to transition from an intuitive suspicion to empirical evidence through systematic measurements. To achieve this, we designed a series of experiments using the NAS benchmark suite for the reasons mentioned in Subsection 3.1.2. We ran those tests using `trace-cmd` in order to record both their runtime as well as scheduling events happening during the run, in case we would want to analyze them later on, and we varied the following parameters:

- The application within the NAS benchmark suite
- The size of the data used by the application
- The machine used
- The thread pinning configuration

4.1.1 Pinning Configurations

For context, a thread is a basic unit of CPU utilization that forms part of a process - or application. Essentially, a thread is a sequence of programmed instructions that can be managed independently by a scheduler, which is part of the operating system. This allows multiple threads to run concurrently within a single process, facilitating more efficient use of computing resources by enabling parallel execution of tasks. Thread pinning, also known as CPU affinity, is the practice of assigning specific threads to specific CPU cores. The primary goal of thread pinning is to increase the performance of an application by reducing the overhead caused by context switching between cores and optimizing cache usage on the CPU. By keeping a thread on a designated core, data used by the thread remains in the closest cache, thus reducing memory access times and improving the efficiency of the process execution.

In our study, we set the pinning configurations via OpenMP, a widely-used multiprocessing library used by the NAS benchmark applications. To cover various cases, we tested the following pinning configurations, named arbitrarily :

- **sockorder** : Fill one NUMA node at a time. The first thread of the application is pinned to the first core of the first NUMA node, the second thread to the second core of the first NUMA node, and so forth, until all cores of the first NUMA node are occupied. Subsequently, the process continues with the second NUMA node, followed by the third, and so on. The name is derived from "within sockets, in order".
- **sockets** : Similar to the sockorder configuration, this method assigns threads to specific NUMA nodes. However, the threads are allowed to move freely among the cores of the NUMA node to which they have been assigned. This is a standard OpenMP pinning configuration. [31]
- **sequential** : Round-robin pinning strategy. The first thread of the application is pinned to the first core of the first NUMA node, the second thread to the first core of the second NUMA node, continuing in this pattern until the first core of all NUMA nodes is filled. The process is then repeated for the second core of each NUMA node, and so on. This configuration is named "sequential" because it ends up pinning cores in order as they are already numbered in a round-robin fashion.
- **none** : No explicit pinning. The operating system is allowed to place and move threads on cores dynamically during the execution of the application.

4.1.2 Testing Automation

As we began our experimental process, it became evident that a structured approach to conducting and managing the experiments was necessary. This realization stemmed from two primary considerations:

- The potentially extended duration of the tests necessitated a reliable method for initiating and automatically saving results without manual intervention.
- The use of `trace-cmd` meant there was an individual result file for each test run, requiring a systematic file organization strategy to manage the large volume of data efficiently. Doing this by hand would have been time consuming.
- There was heavy bookkeeping work needed to keep track of all the configuration parameters for each of the test results.

Our initial solution was the development of a Bash script to automate the execution of test runs. However, as the complexity of our requirements increased, we encountered several limitations:

- The growing size of the Bash codebase became unwieldy
- The limited flexibility of Bash made code reuse challenging
- Bash proved inadequate for managing the increasing complexity of our experiments

To address these issues, we migrated our automation framework to Python, which resolved the aforementioned challenges. However, this transition introduced new concerns:

- The execution of terminal commands in Python can be cumbersome
- Frequent switching between Python scripts and the actual experiments could potentially impact cache performance

To mitigate these concerns and improve usability without interrupting the experimental process, we implemented an abstraction layer above the Python scripts. This layer compiled all desired commands into temporary storage before executing them as a batch. This approach allowed us to maintain the benefits of Python while minimizing interruptions to the test application runs, thereby preserving cache states and ensuring more consistent experimental conditions. Also, it was a lot more convenient to use than having to use the interface between Python and the terminal every time.

4.2 Profiling

Following the initial stage of our investigation, we decided to concentrate our efforts on the CG application from the NAS benchmark suite, with data size C. Our objective in this more focused phase was to identify the underlying factors contributing to the application's performance variations across different contexts. To interact effectively with data and dynamically generate visual representations, we used Jupyter Notebooks. A Jupyter Notebook is an interactive web tool that allows for the execution of live code, visualization, and explanatory text all within a single document. This capability is especially beneficial for data analysis as it enables real-time experimentation and iteration, making it easier to derive insights from complex data sets.

Due to time constraints, we limited our focus to the dahu cluster in which the machines have 2 NUMA nodes instead of 4 for the yeti cluster. This simpler configuration was helpful for data analysis, in particular when it comes to studying remote accesses. Indeed, a remote memory access in a 2 node system necessarily means it was fetched from the other node, whereas it could be fetched from any of the other 3 in a 4 node system.

4.2.1 False Cache Sharing

One hypothesis for the performance degradation observed in certain OpenMP pinning configurations was the occurrence of false cache sharing. False cache sharing occurs when threads on different cores modify different variables that happen to reside on the same cache line. Even though the threads are accessing different variables, the entire cache line containing these variables is marked as modified, causing the cache coherence protocol to invalidate and update the cache line across cores. This results in significant overhead and degraded performance, as the same cache line is frequently moved between caches.

To test this hypothesis, we modified the Fortran code of the CG benchmark to space out the elements of its arrays in memory. This adjustment aimed to ensure that array elements accessed by different threads would not reside on the same cache line, thereby mitigating the effects of false cache sharing. We conducted a series of experiments with varying spacing values to ensure coverage beyond the typical 64 bytes cache size of the *dahu* machines. The specific spacing values tested were:

- 64 bytes
- 96 bytes
- 128 bytes
- 256 bytes

4.2.2 Correlation

For this analysis, we made use of the performance counters introduced earlier via the `perf-stat` tool, a sub-program of `perf` that simply prints out the total number of occurrences for each specified event. Importantly, we recorded these counts per CPU to avoid aggregate data masking specific CPU behaviors that could significantly impact overall runtime. For example, in scenarios involving barrier synchronization, the duration of the longest task primarily dictates the program's completion time,

while tasks that complete earlier merely idle until the longest task finishes. In our comprehensive data collection effort, we monitored a broad spectrum of performance counters, aiming to identify any potentially relevant metrics. Among the counters, we focused on:

- L3 cache misses, categorized into remote, local, load, and store misses
- L2 cache misses, similarly detailed
- General performance metrics such as total cache misses and branch operations

With the collected performance data, we constructed a correlation matrix to systematically analyze the relationships between various counters and the overall performance outcome, measured in runtime.

4.2.3 Memory Access Patterns

The objective of this phase was to conduct an in-depth analysis of the observed `mem_load_l3_miss_retired.remote_dram` events and to identify memory access patterns that might explain the performance degradation observed when threads are pinned sequentially. This required the design of:

1. An experimental setup
2. An analysis infrastructure

Experimental Setup
The experimental setup focused on sampling relevant events during a single execution of the target application. Specifically, it involved recording 1 out of N events along with all associated data, where N is called the sampling period. For instance, if an event involved a memory access, we captured the virtual and physical addresses, the CPU that performed the access, and other pertinent details. The types of events sampled included:

- Memory allocation events
- Generic read/write memory accesses
- Specific `mem_load_l3_miss_retired` events

Analysis Infrastructure
The analysis infrastructure was designed with the following objectives:

- Parsing the data file containing the event samples

- Plotting memory accesses and their characteristics

Plotting memory accesses posed a significant challenge due to the dual nature of virtual and physical memory, which complicates the analysis. Generally, performance issues tied to the hardware are more likely to be identified in the physical memory space, whereas the virtual memory space offers a clearer view of the program's operations. Still, conceptual issues related to program design are more apparent in the virtual space. Therefore, it was essential to plot both physical and virtual memory spaces.

While memory access samples include both virtual and physical addresses, memory allocation events only provide hardware addresses. To be able to represent the latter in virtual memory graphs, we developed a function to determine the virtual mapping of an allocation, shown in Figure 4.1. This function searches for the first memory access within the bounds of the allocation in the physical address space. It then calculates the starting address of the allocation in the virtual address space by subtracting the page offset and page index from the virtual address of the memory access. Finally, the virtual mapping of the allocation is determined by adding the length of the allocation to this starting address.

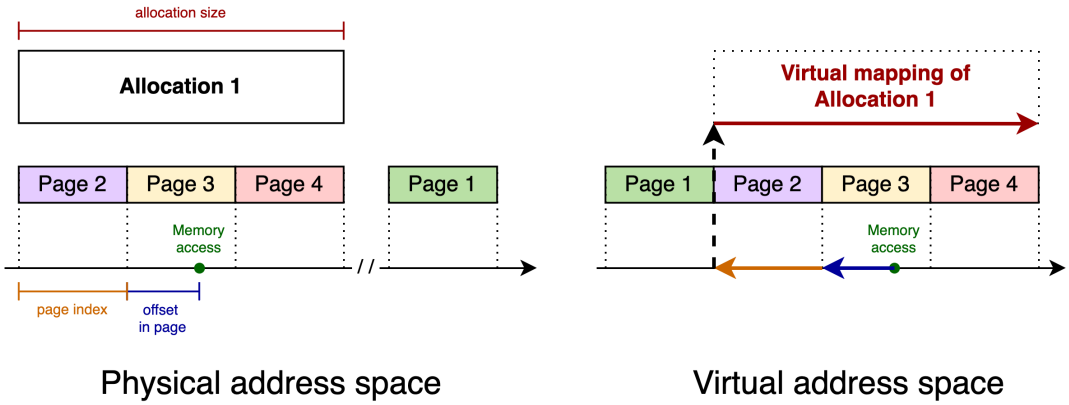

Figure 4.1: Diagram showing the construction of a virtual mapping for an allocation

This dual mapping approach facilitated a comprehensive analysis of memory accesses, enabling the identification of potential causes for the observed performance degradation.

4.3 Proof of Concept

Building on the experiments conducted thus far, our final objective was to come up with a code change in the Linux kernel that would allow splitting falsely shared huge pages and migrating the resulting pages to their respective accessing NUMA nodes. This process would occur during the kernel's handling of page faults, as a new branch of the NUMA balancing process.

The ideal procedure was as follows :

1. While handling a NUMA hinting fault on a huge page, detect that it is being accessed by different NUMA nodes.
2. Split the huge page into normal pages.
3. Identify which of the split pages are accessed exclusively by which NUMA node.
4. Migrate all exclusively accessed pages to their corresponding NUMA node.
5. Remove the NUMA balancing protections on all the pages.
6. Return control to the application.

However, following this exact procedure presented several challenges :

1. **Uncertainty in Migration Targets**: The huge page splitting mechanism can be initiated after only a few accesses, as long as these accesses demonstrate that the page is shared among multiple nodes. In fact, this rapid triggering is one of the objectives of the design. However, in scenarios where the mechanism is triggered swiftly, there is often insufficient access data collected for most of the normal pages that make up the huge page. Consequently, this lack of data makes it challenging to determine if and where these normal pages should be migrated.
2. **Performance Impact**: Conducting all the migrations simultaneously might lead to noticeable slowdowns. Thus, it could be more practical to distribute the migration process over a certain duration to minimize performance degradation.

Despite these challenges, this ideal design provided a guiding framework that helped maintain our focus and direction throughout the development process.

5 Implementation

5.1 Exploratory Experiments

5.1.1 Commands

As stated in Subsection 4.1.1, we made use of the OpenMP library to apply the pinning configuration by setting the environment variable `OMP_PLACES`. To illustrate, on a hypothetical test machine with 4 NUMA nodes and 2 cores per node, the cores would be organized as follows:

- Node 0: Cores 0 and 4
- Node 1: Cores 1 and 5
- Node 2: Cores 2 and 6
- Node 3: Cores 3 and 7

For each pinning configuration tested, the following command examples demonstrate the settings on the hypothetical machine:

- **sockorder**: `OMP_PLACES="{0}{4}{1}{5}{2}{6}{3}{7}"`
- **sockets**: `OMP_PLACES="sockets"`
- **sequential**: `OMP_PLACES="{0}{1}{2}{3}{4}{5}{6}{7}"`
- **none**: `unset OMP_PLACES`

NUMA balancing is managed via the file located at `/proc/sys/kernel/numa_balancing` — using the sysctl system detailed in Subsection 5.4.3. The commands to enable or disable it are :

- Enable: `echo 1 > /proc/sys/kernel/numa_balancing`
- Disable: `echo 0 > /proc/sys/kernel/numa_balancing`

In all the machines utilized for this study, NUMA balancing was enabled by default.

To conduct the tests, we employed the hyperfine command, which facilitates setting a number of warmup runs as well as timed runs :

```
hyperfine --warmup [n_warmups] --runs [n_runs] --export-json
      [json_output_filename] [program_to_run]
```

Typically, we performed 2 warm up runs and between 20 to 100 timed runs, depending on the duration required to complete each benchmark. This approach ensured accurate and consistent benchmarking results across different configurations and systems.

5.1.2 Launching Infrastructure

The abstraction layer utilized for executing shell commands in a deferred manner was implemented with the `CommandChain` class. This class provides a structured approach to manage and execute shell commands.

Listing 5.1: CommandChain class that implements the abstraction layer to run shell commands at once in a deferred manner

```
class CommandChain:
    def __init__(self, temp_sh_file_path: str, lines: List[str] = None):
        self.sh_file_path = temp_sh_file_path
        self.lines = lines if lines is not None else []

    def append(self, new_line: str):
        self.lines.append(new_line)

    def add(self, other: "CommandChain"):
        self.lines += other.lines

    def __add__(self, o):
        return CommandChain(self.lines + o.lines)

    def execute(self):
        with open(self.sh_file_path, 'w') as f:
            f.write("#!/bin/bash\n\n")

            for l in self.lines:
                f.write(l)
                f.write("\n")

        subprocess.run(f"chmod␣+x␣{self.sh_file_path}", shell=True)
        subprocess.run(f"{self.sh_file_path}", shell=True)
```

To avoid having to specify each configuration manually, we wrote the `grid_experiment` function. It automatically explores all combinations of provided parameters and their values, subsequently invoking a specified function to execute the test for each configuration.

Listing 5.2: grid_experiment function used to traverse all the possible configurations from a list of parameters with given values

```
# Uses a backtracking algorithm to explore all the configurations,
# and calls the run_function on each one of them
def grid_experiment(parameters: List[str], parameter_values: List[List[tuple]],
                    run_function: callable):
    total_runs = 1
    for pv in parameter_values:
        total_runs *= len(pv)
    print(f"Grid experiment with total number of configs : {total_runs}")

    curr_params: List[tuple] = []

    def arg_dict_from_curr_params():
        return {parameters[i]: curr_params[i][1] for i in range(n)}

    def name_from_curr_params():
        params_str = [curr_params[i][0] for i in range(n)]
        return "-".join(params_str)

    def recu_explore(idx: int):
        if idx == len(parameter_values):
            arg_dict = arg_dict_from_curr_params()
            arg_dict["name"] = name_from_curr_params()
            run_function(**arg_dict)
            return

        for val in parameter_values[idx]:
            curr_params.append(val)
            recu_explore(idx + 1)
            curr_params.pop()

    recu_explore(0)
```

Listing 5.3: Usage example for function grid_experiment

```
def experiment_benchmark_different_config(program_path: str, runs: int = 20,
    warmups: int = 2):
    # [...]
```

```
3
4       def run_with_params(name: str, nb: bool, ompPlaces: Optional[str]):
5           # [...]
6           command_chain = init_command_chain_with_config(nb, ompPlaces)
7           measure_command = f"hyperfine␣--warmup␣{warmups}␣--runs␣{runs}␣--export-
    json␣{filepath}␣'{program_path}'"
8           command_chain.append(measure_command)
9           command_chain.execute()
10
11      grid_experiment(
12          ["nb", "ompPlaces"],
13          [
14              [("nb-enabled", True), ("nb-disabled", False)],
15              [("sockorder", get_sockorder_omp_places()), ("sockets", "sockets"),
     ("sequential", get_sequential_omp_places()), ("none", None)]
16          ],
17          run_with_params
18      )
```

5.2 Deeper Dive into Automatic NUMA Balancing

As outlined in the Overview chapter, a detailed examination of the NUMA balancing code was conducted to gain a comprehensive understanding of its inner workings. This investigation aimed to elucidate the mechanisms behind NUMA balancing and potentially account for some of the observed performance variations associated with it.

To achieve this, explored the source code, mapping out the functions and their interactions. The results of this thorough exploration are encapsulated in the form of flow graphs, which are presented in Figure 5.1, Figure 5.2, Figure 5.3, Figure 5.4. These flow graphs provide a visual representation of the function calls within the NUMA balancing code, illustrating the flow of execution and the relationships between different components.

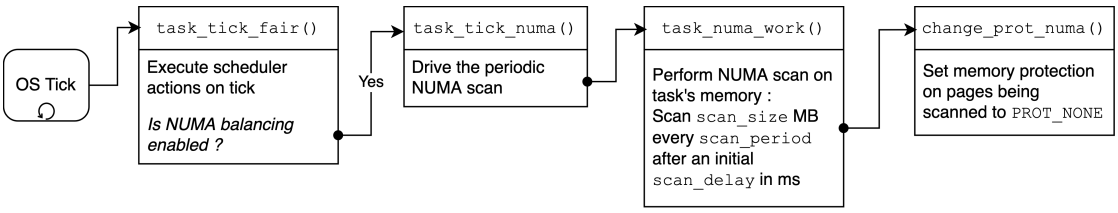

Figure 5.1: Flow chart of the memory scanning part of automatic NUMA balancing.

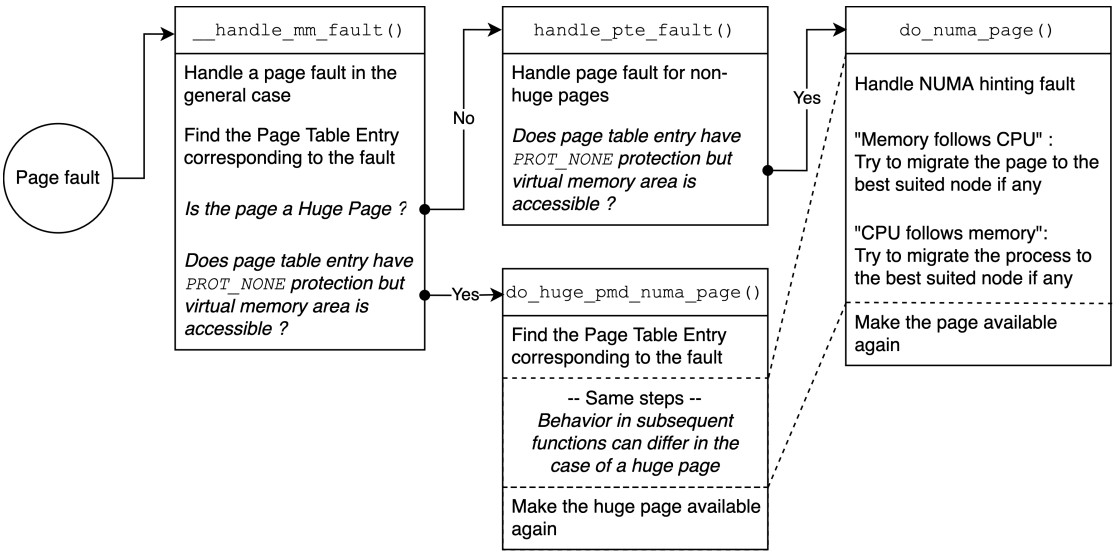

Figure 5.2: Flow chart of the page fault handling part of automatic NUMA balancing.

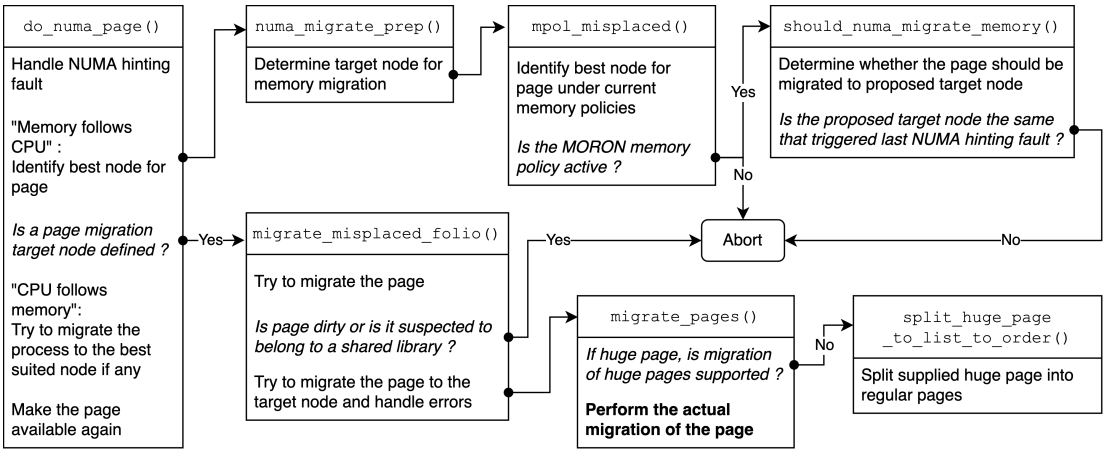

Figure 5.3: Flow chart of the page migration part of automatic NUMA balancing.

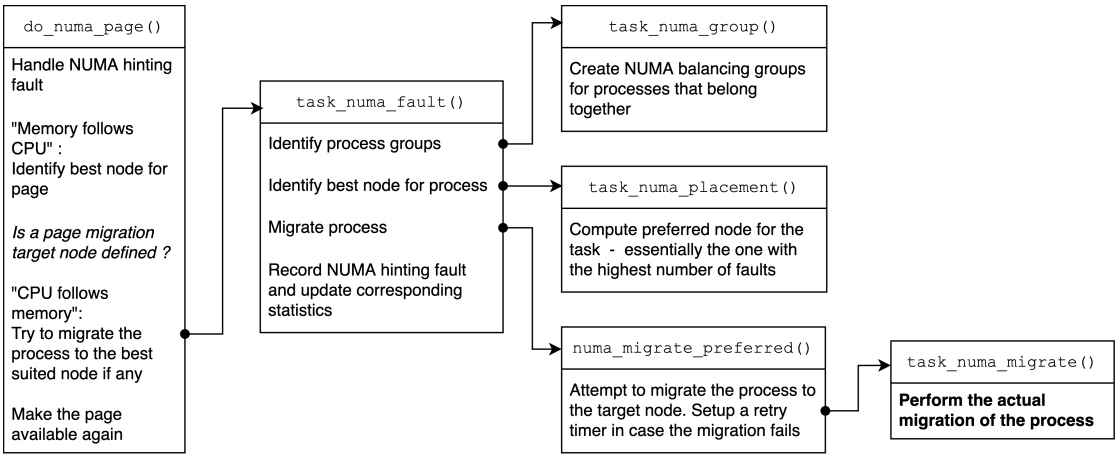

Figure 5.4: Flow chart of the process migration part of automatic NUMA balancing.

5.3 Profiling

5.3.1 False Cache Sharing

In order to space out all the array elements in the NAS benchmark CG, we went through 2 steps. The first one involved changing the size of every allocation in order to account for the desired padding. An example of this extension is shown below.

Listing 5.4: Diff of the spacing of allocations in the NAS benchmark CG

```
allocate (  &
    ![...]
-   &           x(na+2),  &
-   &           z(na+2),  &
-   &           p(na+2),  &
+   &           x(spacing_multiplier * (na+2)),  &
+   &           z(spacing_multiplier * (na+2)),  &
+   &           p(spacing_multiplier * (na+2)),  &
    &           stat = ios)
```

Then, we proceed with the main part of the change : we altered the code to use only 1 out of every N elements of every array in the computation. Highlights of the code change are as follows.

Listing 5.5: Diff of the spacing of elements in the NAS benchmark CG

```
-   p(j) = 0.0d0
+   q(j * spacing_multiplier) = 0.0d0
    ![...]
-   norm_temp1 = norm_temp1 + x(j)*z(j)
+   norm_temp1 = norm_temp1 + x(j * spacing_multiplier)*z(j *
    spacing_multiplier)
    ![...]
-   suml = suml + a(k)*p(colidx(k))
+   suml = suml + a(k)*p(colidx(k) * spacing_multiplier)
```

5.3.2 Correlation Analysis

In this analysis, our primary focus was on exploiting the performance counters related to cache misses. We hypothesized that cache misses were the most likely cause of the performance bottlenecks, or at least the most easily identifiable issue. To investigate this, we tracked the following events:

- **L3 cache misses related performance events** [37] :

  - `LLC-loads` : Number of memory load requests to the L3 cache
  - `LLC-load-misses` : Number of memory load requests that missed the L3 cache
  - `LLC-stores` : Number of memory store requests to the L3 cache
  - `LLC-store-misses` : Number of memory store requests that missed the L3 cache
  - `cycle_activity.stalls_l3_miss` : Execution stalls while L3 cache miss demand load is outstanding.
  - `mem_load_l3_miss_retired.local_dram` : Retired load instructions whose data sources missed L3 but serviced from local DRAM
  - `mem_load_l3_miss_retired.remote_dram` : Retired load instructions which data sources missed L3 but serviced from remote DRAM
  - `mem_load_l3_miss_retired.remote_fwd` : Retired load instructions whose data sources were forwarded from a remote cache
  - `mem_load_l3_miss_retired.remote_hitm` : Retired load instructions whose data sources was remote HITM

- **L2 cache misses related performance events**:
  - `l2_rqsts.all_demand_references` : Total number of memory requests to the L2 cache
  - `l2_rqsts.all_demand_miss` : Total number of memory requests that missed the L2 cache

- **General performance events** : `cache-references`, `cache-misses`, `migrations`, `context-switches`

Data was recorded using the perf command [36]. An example command used for data collection is shown below:

```
perf stat -A -a -j -e [previously_listed_events] -D 100 -o
    [output_file_path] [program_path]
```

The options are as follows:

- `-A` : do not aggregate counts across all monitored CPUs.
- `-a` : system-wide collection from all CPUs.
- `-j` : print out a JSON format output that can be used for parsing.

- `-e` : list of events to collect the count of
- `-D` : After starting the program, wait msecs before measuring. This is useful to filter out the startup phase of the program, which is often very different.

5.3.3 Memory Access Patterns

In contrast to the correlation analysis, where our goal was to count the total number of events occurring within a run, the objective here was to sample these events. Specifically, we aimed to capture both virtual and physical memory addresses associated with each event to facilitate the analysis of memory access patterns. The following events were sampled:

- **Memory Allocation events** :
  - `kmem:*` : all kernel memory events, in particular `kmem:mm_page_alloc` that indicates actual page allocation
  - `syscalls:sys_enter_mmap` : event indicating that the control flow has entered the `mmap` kernel function
- **L3 Miss Events** :
  - `mem_load_l3_miss_retired.remote_dram:P` : cf. Subsection 5.3.2
  - `mem_load_l3_miss_retired.remote_fwd:P` : cf. Subsection 5.3.2
  - `mem_load_l3_miss_retired.remote_hitm:P` : cf. Subsection 5.3.2
- **Generic CPU load and store events** :
  - `cpu/mem-loads` : Any memory load
  - `cpu/mem-stores` : Any memory store

We intentionally excluded the `mem_load_l3_miss_retired.local_dram` event from our sampling list as it did not provide sufficiently meaningful information to justify the additional file size, which presented significant challenges as detailed below.

The `P` specifier at the end of certain events enables "precise mode." In the context of memory access counters, this mode records additional details such as the associated virtual and physical memory addresses and the TLB state.

One of the primary issues encountered was the substantial increase in the size of the generated data file with higher sampling frequencies. The data size rapidly grew to several gigabytes, which posed a problem not only because of its sheer size but also because the data is produced in a compressed format. Our analysis

framework, detailed in the subsequent section, can only process uncompressed perf data files. Decompressing these large files became increasingly time-consuming, often taking several minutes. This limitation necessitated a reduction in the sampling rate to manage file size without losing critical information, particularly regarding the `mem_load_l3_miss_retired.remote_dram` events under investigation.

To address this, we found a way to operate the cpu load/store sampling at a different frequency than the rest of the events. For instance, using a sampling period of 2000 for generic load/store events and 200 for L3 miss events allowed us to balance the data file size and the capture of meaningful data. An example command is as follows:

```
perf record -e cpu/mem-loads,period=2000/P,cpu/mem-stores,period=2000/P -c 200 -e [l3_miss_events]
```

This adjustment enabled us to reduce the data file size while retaining sufficient detail about the events of interest.

Regarding the need for getting the corresponding virtual addresses for allocation events as mentioned in the Design chapter, it essentially came down to mapping a "physical page" - referred to as Page Frame Number or PFN - to a virtual page. For that purpose we implemented the following function :

Listing 5.6: find_virt_page_for_pfn function that is used to find the virtual mapping for allocations in the physical memory space

```
def find_virt_page_for_pfn(pfn: int, order: int, access_df: pd.DataFrame, page_size_order: int):
    page_size = 2 ** page_size_order
    min_phys_addr = pfn * page_size
    max_phys_addr = (pfn + 2 ** order) * page_size

    access_to_zone_df = filter_in_bounds(access_df,
                            (min_phys_addr, max_phys_addr), "phys")
    access_record = access_to_zone_df.loc[access_to_zone_df['virt'].idxmin()]
    phys_pfn = int(access_record['phys']) >> page_size_order
    virt_page = int(access_record['virt']) >> page_size_order
    return virt_page - (phys_pfn - pfn)
```

5.4 Proof of Concept

5.4.1 Folio vs Page

For better comprehension of the intricacies of the Linux kernel, we are going to detail the concept of memory folios because it is going to appear throughout the code listings that will follow.

For additional context, the order of a page in memory management refers to the power of two that represents the number of contiguous base (order-0) pages that form a larger block of memory. In other words, the order indicates the size of a compound page in terms of its constituent base pages. An order-0 page is a single, smallest unit page. It is usually 4096 bytes long but not necessarily. An order-1 page comprises 2^1 (or 2) contiguous base pages, an order-2 page comprises 2^2 (or 4) contiguous base pages, and so on. Higher-order pages are used to allocate larger contiguous blocks of memory efficiently, which is particularly useful for certain types of data structures and performance optimizations. The order helps the memory management system quickly calculate the size of the page and manage memory allocation and deallocation effectively.

A folio is a newly introduced data structure in memory management that encapsulates either an order-0 page or the head page of a compound page, effectively serving as a packaging layer for pages without adding overhead. In addition, it streamlines the management of these pages by consolidating their metadata into a unified structure. This simplification allows functions to clearly indicate they are dealing with head pages, eliminating the need for repeated checks and conversions, such as `VM_BUG_ON(PageTail(page))` and `compound_head()`. The introduction of folios reduces redundant calls to `compound_head()`, decreases execution costs, and helps developers immediately recognize that a folio represents a head page, thereby avoiding potential bugs associated with tail pages. For instance, functions like `page_mapping()` are now simplified to `folio_mapping()`, as the folio inherently implies the presence of head pages. By distinguishing between head and tail pages through the folio structure, memory management becomes more efficient and less error-prone, enhancing both performance and code clarity.

5.4.2 Address Mapping

5.4.3 Sysctl

In Linux, `sysctl` is a mechanism for dynamically modifying kernel parameters at runtime, allowing users to fine-tune system performance and behavior without requiring a system reboot. It provides an interface to configure various kernel settings that control system functionality, including network configurations, file system behavior, and virtual memory management. These parameters are represented within the `/proc/sys/` directory, where each file corresponds to a specific kernel setting. By reading from or writing to these files, system administrators can adjust the kernel's operation to better suit specific workloads or performance requirements. Kernel parameters are reset to their default values on reboot, but persistent changes can be ensured by placing configuration settings in files such as `/etc/sysctl.conf` or within the `/etc/sysctl.d/` directory. This capability makes `sysctl` an essential tool for managing and optimizing Linux system performance, offering a flexible approach to system configuration that can be easily adjusted to meet the needs of different environments and applications.

Using `sysctl` was crucial to enable toggling our code change on and off, thereby preventing crash cycles. Without `sysctl`, the code change would be perpetually active, leading to a problematic scenario: if the code change caused a crash, it would remain active upon reboot, resulting in a continuous crash cycle. To mitigate this issue, we incorporated a kernel parameter that allows the code change to be toggled. We set this parameter to 1 just before running our experiments to enable our kernel patch.

5.4.4 Challenges

Initially, we attempted to implement our huge page split by duplicating sections of existing code that performed this function. Despite referencing several examples and conducting numerous trials, we were unable to achieve success within the context of handling a NUMA-hinted fault. The only outcome of these experiments was a kernel crash. The likely causes of these issues were related to one or both of the following factors:

- Locking Patterns: The Linux kernel extensively uses locking and unlocking mechanisms, and even a single missing lock can result in a kernel crash. When dealing with multiple layers of functions calling each other, determining which objects to lock before invoking a function becomes highly complex.
- List Management: The Linux kernel handles lists in a unique manner, frequently transferring objects from one list to another. If these expected list transfers do not

occur, it can rapidly lead to a kernel crash.

Given the limited time remaining at this stage, we could not afford to dive deeper into the intricate requirements for proper locking and list management. This was especially critical considering that testing a single code change could take up to 10 minutes due to the necessity of kernel recompilation and multiple restarts, as discussed in the Overview section. Consequently, the only viable option was to identify an existing code structure that successfully splits huge pages and utilize it directly, rather than attempting to replicate the functionality from scratch.

5.4.5 First Working Prototype

Fortunately, we identified a branch within the `migrate_pages_batch` function that would split a huge page prior to migration if the system did not support huge page migration. The only thing left to do at this point was to make sure the control flow took this branch whenever it was decided that the huge page should be split.

This however was not as straightforward as expected, due to the intricate control flow within the kernel's memory management subsystem. Specifically, the need for split was determined in the `do_huge_pmd_numa_page` function, but by the time the execution reached `migrate_pages_batch` two function calls deep, relevant information regarding the necessity for page splitting was lost. Modifying the function signatures to carry this information through the call chain would typically be the straightforward solution, however such changes to the Linux kernel are generally discouraged due to the complexity and potential for introducing instability.

To circumvent this, we devised a workaround that leveraged the accessibility of specific node information within the `migrate_pages_batch` function. Notably, we manipulated the `target_node` parameter, which is pivotal in determining the migration strategy. We implemented the following changes to achieve our goal :

- **Original Behavior**: If the best node for migration is the same as the current node of the folio, the `target_node` is set to `NUMA_NO_NODE`, and no migration occurs.
- **Modified Behavior**: To enforce the splitting of huge pages, we modified the code to set the `target_node` to the current node of the folio whenever a split is desired. This effectively passes the necessary information down to `migrate_pages_batch`.

In `migrate_pages_batch`, we then adjusted the control flow to ensure that the code responsible for splitting the page is executed if the folio is designated to be migrated

to the same node. This approach allowed us to achieve the desired functionality without extensive changes to the kernel's function signatures, maintaining stability while enabling the required page migration behavior.

Listing 5.7: Diff of do_huge_pmd_numa_page for the first working prototype

```
@@ mm/huge_memory.c:1713
vm_fault_t do_huge_pmd_numa_page(struct vm_fault *vmf)
{
    struct vm_area_struct *vma = vmf->vma;
    pmd_t pmd;
    struct folio *folio;
    unsigned long haddr = vmf->address & HPAGE_PMD_MASK;
    int nid = NUMA_NO_NODE;
+   int this_cpu_nid = numa_node_id();
    int target_nid, last_cpupid = (-1 & LAST_CPUPID_MASK);
    // [...]

    folio = vm_normal_folio_pmd(vma, haddr, pmd);
    nid = folio_nid(folio);
    // [...]
+   if (static_branch_likely(&sched_nb_split_shared_hugepages)) {
+       if (!cpupid_cpu_unset(last_cpupid)
+               && cpupid_to_nid(last_cpupid) != this_cpu_nid) {
+           target_nid = nid;
+           goto migrate;
+       }
+   }

    target_nid = numa_migrate_prep(folio, vma, haddr, nid, &flags);
    if (target_nid == NUMA_NO_NODE) {
        folio_put(folio);
        goto out_map;
    }

+migrate:
    // [...]
    migrated = migrate_misplaced_folio(folio, vma, target_nid);
    // [On success goto out, on failure goto out_map]

out:
    if (nid != NUMA_NO_NODE)
        task_numa_fault(last_cpupid, nid, HPAGE_PMD_NR, flags);
    return 0;
```

```

out_map:
    // [Make the PMD accessible again and goto out]
}
```

Listing 5.8: Relevant lines of migrate_misplaced_folio, called by do_huge_pmd_numa_page

```
@@ mm/migrate.c:2558
int migrate_misplaced_folio(struct folio *folio, struct vm_area_struct *vma,
                            int node)
{
    // [Exit if folio is likely shared library, or if dirty]
    // [...]
        list_add(&folio->lru, &migratepages);
        nr_remaining = migrate_pages(&migratepages, alloc_misplaced_dst_folio,
                                     NULL, node, MIGRATE_ASYNC,
                                     MR_NUMA_MISPLACED, &nr_succeeded);
    // [...]
}
```

Listing 5.9: Relevant lines of migrate_pages, called by migrate_misplaced_folio

```
@@ mm/migrate.c:1909
int migrate_pages(struct list_head *from, new_folio_t get_new_folio,
                free_folio_t put_new_folio, unsigned long private,
                enum migrate_mode mode, int reason, unsigned int *ret_succeeded
    )
{
    // [...]
again:
        // [...]
        if (mode == MIGRATE_ASYNC)
                rc = migrate_pages_batch(&folios, get_new_folio, put_new_folio,
                                private, mode, reason, &ret_folios,
                                &split_folios, &stats,
                                NR_MAX_MIGRATE_PAGES_RETRY);
        // [...]
        if (!list_empty(&split_folios)) {
                migrate_pages_batch(&split_folios, get_new_folio,
                                put_new_folio, private, MIGRATE_ASYNC, reason,
                                &ret_folios, NULL, &stats, 1);
                list_splice_tail_init(&split_folios, &ret_folios);
        }
        if (!list_empty(from))
```

```
                goto again;
out:
    // [Count migration events and return]
}
```

Listing 5.10: Core of the the Initial Working Prototype : diff of migrate_pages_batch, called by migrate_pages

```
@@ mm/migrate.c:1621
static int migrate_pages_batch(struct list_head *from,
        new_folio_t get_new_folio, free_folio_t put_new_folio,
        unsigned long private, enum migrate_mode mode, int reason,
        struct list_head *ret_folios, struct list_head *split_folios,
        struct migrate_pages_stats *stats, int nr_pass)
{
    // [...]
    for (pass = 0; pass < nr_pass && retry; pass++) {
        list_for_each_entry_safe(folio, folio2, from, lru) {
            is_large = folio_test_large(folio);
            is_thp = is_large && folio_test_pmd_mappable(folio);
            nr_pages = folio_nr_pages(folio);
            // [...]
-           if (!thp_migration_supported() && is_thp) {
+           if ((!thp_migration_supported() && is_thp)
+                   || (is_thp && folio_nid(folio) == private)) {
                nr_failed++;
                stats->nr_thp_failed++;
                if (!try_split_folio(folio, split_folios)) {
                    stats->nr_thp_split++;
                    stats->nr_split++;
                    continue;
                }
                stats->nr_failed_pages += nr_pages;
                list_move_tail(&folio->lru, ret_folios);
                continue;
            }

            rc = migrate_folio_unmap(get_new_folio, put_new_folio,
                    private, folio, &dst, mode, reason,
                    ret_folios);
            // [...]
        }
    }
    // [Collect statistics]
```

```
move:
    // [Actually migrate the folios]
out:
    // [Cleanup the remaining folios]
}
```

Note that a large part of the code that is non essential to the understanding of our first working prototype has been hidden. This includes the various locking calls mentioned above.

Although this implementation finally allowed us to achieve the desired outcome, it had one major flaw : it forces the split pages to be migrated to the node they are already in. This involves unnecessary copying of each of the split pages as well as the creation of the associated page structs in the kernel, hindering the potential performance gains of our solution. The next part of this section describes how we addressed this issue.

5.4.6 No Same Node Page Migration

Instead of modifying the function signature as previously mentioned, a safer but more involved solution was essentially to rewrite the functions taking part in the migration process. This approach allowed us to achieve the granularity needed to avoid same node migration. Specifically, we replicated sections of `numa_migrate_prep`, `migrate_misplaced_folio`, and `do_huge_pmd_numa_page` to recreate the environment that these functions establish, and worked within that copy environment.

Listing 5.11: Diff of do_huge_pmd_numa_page to get rid of same node migration

```
@@ mm/huge_memory.c:1952
vm_fault_t do_huge_pmd_numa_page(struct vm_fault *vmf)
{
    // [...]
    folio = vm_normal_folio_pmd(vma, haddr, pmd);
    nid = folio_nid(folio);

+   if (static_branch_likely(&sched_nb_split_shared_hugepages)
+            && this_cpu_nid != nid) {
+       // [Copied block from numa_migrate_prep]
+       // Mostly copied from Block A :
+       spin_unlock(vmf->ptl);
+       int splitted = split_thp_on_page_fault(folio, vmf);
+       if (!splitted) {
+           nid = NUMA_NO_NODE;
```

```
+            vmf->ptl = pmd_lock(vma->vm_mm, vmf->pmd);
+            if (unlikely(!pmd_same(oldpmd, *vmf->pmd))) {
+                spin_unlock(vmf->ptl);
+                return 0;
+            }
+            goto out_map;
+        }
+        return 0;
+    }

    target_nid = numa_migrate_prep(folio, vma, haddr, nid, &flags);
    if (target_nid == NUMA_NO_NODE) {
        folio_put(folio);
        goto out_map;
    }

    // [...]
    migrated = migrate_misplaced_folio(folio, vma, target_nid);
    // [Block A : On success goto out, on failure goto out_map]

out:
    if (nid != NUMA_NO_NODE)
        task_numa_fault(last_cpupid, nid, HPAGE_PMD_NR, flags);
    return 0;

out_map:
    // [Make the PMD accessible again and goto out]
}
```

Listing 5.12: New function split_thp_on_page_fault, introduced to perform the split

```
@@ mm/huge_memory.c:1728
// Returns 0 on error, 1 on success
static int split_thp_on_page_fault(struct folio *folio, struct vm_fault *vmf)
{
    struct vm_area_struct *vma = vmf->vma;
    int ret = 0;

    // [Copied block from migrate_misplaced_folio. Essentially : do not
    //  migrate folio if likely to be shared library, or if dirty]

    bool is_large = folio_test_large(folio);
    bool is_thp = is_large && folio_test_pmd_mappable(folio);
    if (!is_thp)
```

```
14          goto out;
15
16      folio_lock(folio);
17      ret = !split_folio(folio);
18      folio_unlock(folio);
19
20  out:
21      folio_put(folio);
22      return ret;
23  }
```

During this process, we discovered that the huge page split procedure eliminates any pre-existing page protection, notably the `prot_numa` protection used for NUMA balancing scanning. Consequently, for the newly split pages to be moved to their appropriate node and address the performance issue, the following steps were required for each page:

1. **First NUMA scanning**: The process memory must undergo N cycles of NUMA balancing scan until the page is marked `prot_numa`.
2. **First memory Access**: A memory access must occur to trigger a NUMA hinted fault and record the accessing node.
3. **Second NUMA scanning**: The process memory must undergo another M scan cycles until the page is marked `prot_numa` again.
4. **Second memory access**: A new memory access must occur to trigger a NUMA hinted page fault, making the page suitable for migration because it has been accessed twice in a row from the same node.

Figure 5.5: Inefficient steps remaining for the page to be migrated

This process needs to occur for each page that requires migration. Although these steps happen more or less in parallel for all the pages, it can still take a significant amount of time before all pages are migrated to their appropriate node. The subsequent sections detail our approach to addressing these inefficiencies.

5.4.7 Reapplying NUMA Protections

The process of reapplying the NUMA related page protections after a transparent huge page split was notably one of the more labor-intensive tasks following the successful development of the initial prototype. The inherent complexity of the Linux kernel code contributed to multiple failures during this phase.

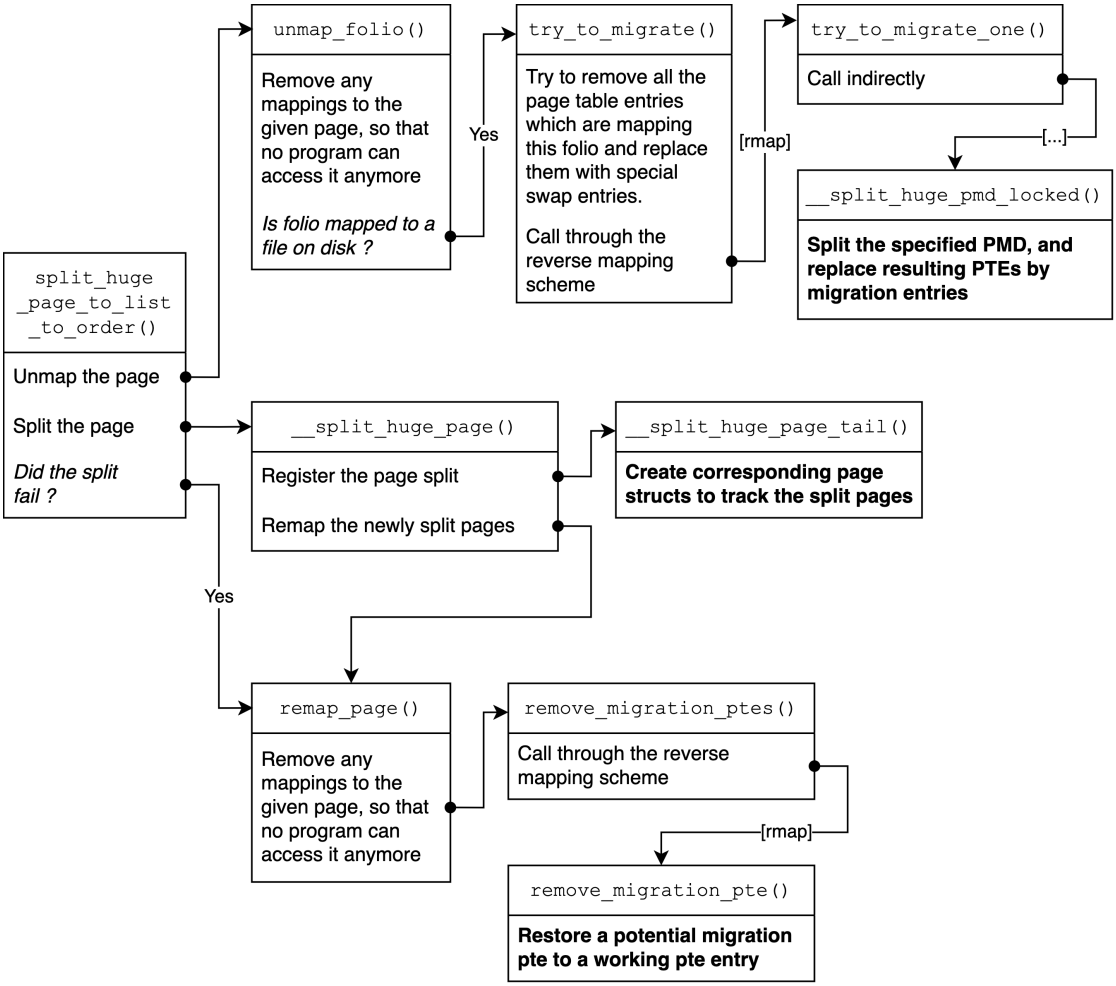

Figure 5.6: Summarized call graph of how transparent huge page splitting is implemented in the Linux kernel

Our initial strategy involved temporarily altering the Virtual Memory Area (VMA) protections at the moment of the split. In the function responsible for replacing the

migration entries, new Page Table Entry (PTE) objects are initialized with the same protection as the VMA to which they belong. We attempted to exploit this mechanism by setting the VMA to `prot_numa` before splitting the page and subsequently resetting it to its original value. However, this approach led to kernel crashes, making it unsuitable for restoring protections on the split pages.

Listing 5.13: Diff of split_thp_on_page_fault to try to reapply NUMA protections

```
@@ mm/huge_memory.c:1728
static int split_thp_on_page_fault(struct folio *folio, struct vm_fault *vmf)
{
    // [...]

    folio_lock(folio);
+   pgprot_t initial_prot = vma->vm_page_prot;
+   vma->vm_page_prot = PAGE_NONE;
    ret = !split_folio(folio);
+   vma->vm_page_prot = initial_prot;
    folio_unlock(folio);

out:
    folio_put(folio);
    return ret;
}
```

Listing 5.14: Relevant lines from remove_migration_pte. The protection initialization happens on line 17.

```
@@ mm/migrate.c:185
static bool remove_migration_pte(struct folio *folio,
                struct vm_area_struct *vma, unsigned long addr, void *old)
{
        while (page_vma_mapped_walk(&pvmw)) {
                pte_t old_pte;
                pte_t pte;
                swp_entry_t entry;
                struct page *new;
                unsigned long idx = 0;

                if (folio_test_large(folio) && !folio_test_hugetlb(folio))
                        idx = linear_page_index(vma, pvmw.address) - pvmw.pgoff
    ;
                new = folio_page(folio, idx);
        // [...]
                folio_get(folio);
```

```
17                  pte = mk_pte(new, READ_ONCE(vma->vm_page_prot));
18                  old_pte = ptep_get(pvmw.pte);
19                  entry = pte_to_swp_entry(old_pte);
20                  if (folio_test_dirty(folio) && is_migration_entry_dirty(entry))
21                          pte = pte_mkdirty(pte);
22          // [...]
23                  if (is_writable_migration_entry(entry))
24                          pte = pte_mkwrite(pte, vma);
25          // [...]
26                  update_mmu_cache(vma, pvmw.address, pvmw.pte);
27          }
28  }
```

In our second attempt, we tried setting the protection of the migration entries to `prot_numa`, such that when remapping them we could copy the existing protection and end up with protected split pages, ready to trigger a NUMA hinting fault on the next access. Unfortunately, migration entries seem to require a specific set of protections for proper kernel functionality, or at the very least, must not have the `PROT_NONE` bit set. This attempt also resulted in kernel crashes.

Ultimately, we took advantage of the kernel reverse mapping scheme, which provided a viable solution.

Listing 5.15: Diff of split_thp_on_page_fault to try to reapply NUMA protections via reverse mapping scheme

```
1  @@ mm/huge_memory.c:1728
2  static int split_thp_on_page_fault(struct folio *folio, struct vm_fault *vmf)
3  {
4     // [...]
5
6      unsigned int nr = 1 << folio_order(folio);
7      folio_lock(folio);
8      ret = !split_folio(folio);
9  +   if (ret)
10 +       make_folios_ptes_protnone(folio, nr);
11     folio_unlock(folio);
12
13 out:
14     folio_put(folio);
15     return ret;
16 }
```

Listing 5.16: New function make_folios_ptes_protnone that calls make_pte_protnone on each PTE via reverse mapping

```
@@ mm/huge_memory.c:3352
static void make_folios_ptes_protnone(struct folio *folio, unsigned long nr) {
    // Mostly copied from remap_page and remove_migration_ptes
    int i = 0;
    for (;;) {
        struct rmap_walk_control rwc = {
            .rmap_one = make_pte_protnone,
            .arg = folio,
        };
        rmap_walk_locked(folio, &rwc);
        i += folio_nr_pages(folio);
        if (i >= nr)
            break;
        folio = folio_next(folio);
    }
}
```

Listing 5.17: New function make_pte_protnone that make the given PTE prot_numa

```
@@ mm/huge_memory.c:3307
static bool make_pte_protnone(struct folio *folio,
        struct vm_area_struct *vma, unsigned long addr, void *old)
{
    // Mostly copied from remove_migration_pte
    DEFINE_FOLIO_VMA_WALK(pvmw, old, vma, addr, 0);

    while (page_vma_mapped_walk(&pvmw)) {
        pte_t pte;
        // [...]
        pte = ptep_get(pvmw.pte);
        // [...]
        pte = pte_modify(pte, PAGE_NONE);
        set_pte_at(vma->vm_mm, pvmw.address, pvmw.pte, pte);
        update_mmu_cache(vma, pvmw.address, pvmw.pte);
    }

    return true;
}
```

It is important to note that the Linux kernel's page fault mechanism mandates that the page that caused the fault — later referred to as the *faulting page* — must be left

in an accessible state for the application upon the next request, and failure to do so results in a kernel crash. To comply with this requirement, we had to **leave the page unlocked**. Although we were unable to find a sufficiently convenient method to restore the protections on all pages except the one that triggered the page fault, we chose to protect all split pages and then unlock the faulting page. This approach ensured the stability and functionality of the system.

Listing 5.18: Diff of do_huge_pmd_numa_page to unlock the split page responsible for the NUMA hinting fault

```
@@ mm/huge_memory.c:1952
vm_fault_t do_huge_pmd_numa_page(struct vm_fault *vmf)
{
    // [...]
    folio = vm_normal_folio_pmd(vma, haddr, pmd);
    nid = folio_nid(folio);

    if (static_branch_likely(&sched_nb_split_shared_hugepages)
            && this_cpu_nid != nid) {
        // [Copied block from numa_migrate_prep]
        spin_unlock(vmf->ptl);
        int splitted = split_thp_on_page_fault(folio, vmf);
+       if (splitted) {
+           handle_unlock(vmf);
        } else {
            // [...]
            goto out_map;
        }
        return 0;
    }
    // [...]
out_map:
    // [Make the PMD accessible again and goto out]
}
```

Listing 5.19: New function handle_unlock introduced to restore the original protections of the split page responsible for the NUMA hinting fault

```
@@ mm/huge_memory.c:2018
static int handle_unlock(struct vm_fault *vmf) {
    // Mostly copied from handle_pte_fault and do_numa_page
    vmf->pte = pte_offset_map_nolock(vmf->vma->vm_mm, vmf->pmd,
                        vmf->address, &vmf->ptl);
    vmf->orig_pte = ptep_get_lockless(vmf->pte);
```

```
    // [Safety checks]
    struct vm_area_struct *vma = vmf->vma;
    struct folio *folio = NULL;
    int nid = NUMA_NO_NODE;
    bool writable = false;
    pte_t pte, old_pte;

    spin_lock(vmf->ptl);
    old_pte = ptep_get(vmf->pte);
    // [More safety checks]
    pte = pte_modify(old_pte, vma->vm_page_prot);
    writable = pte_write(pte);

    if (static_branch_likely(&sched_nb_split_reapply_prot)) {
        old_pte = ptep_modify_prot_start(vma, vmf->address, vmf->pte);
        pte = pte_modify(old_pte, vma->vm_page_prot);
        pte = pte_mkyoung(pte);
        if (writable)
            pte = pte_mkwrite(pte, vma);
        ptep_modify_prot_commit(vma, vmf->address, vmf->pte, old_pte, pte);
        update_mmu_cache_range(vmf, vma, vmf->address, vmf->pte, 1);
    }

    pte_unmap_unlock(vmf->pte, vmf->ptl);
    return 0;
}
```

5.4.8 Resetting NUMA Counters

Following the elimination of the first inefficient step left to migrate pages after a split (see Figure 5.5), our objective was to remove steps 2 and 3 by enforcing page migration on the initial access. Our initial approach involved resetting the NUMA counters on the pages, leveraging the NUMA balancing code's behavior of migrating a page upon its first remote access. However, this behavior is only guaranteed for the first 4 scan cycles, beyond which instant migration is not triggered. To address this limitation, we devised an ad hoc method to mark pages for migration immediately after they are split. This marking process was integrated into the `make_folios_ptes_protnone` function, which conveniently iterates over the split pages' folios. Subsequently, we modified the `should_numa_migrate_memory` function to force it to return true whenever this migration marking is detected.

Listing 5.20: Diff of make_folios_ptes_protnone to mark the split pages ready to migrate on next NUMA hinting fault

```
@@ mm/huge_memory.c:3352
static void make_folios_ptes_protnone(struct folio *folio, unsigned long nr) {
    // Mostly copied from remap_page and remove_migration_ptes
    int i = 0;
    for (;;) {
        struct rmap_walk_control rwc = {
            .rmap_one = make_pte_protnone,
            .arg = folio,
        };
+       folio_xchg_last_cpupid(folio,
+           ((9 & LAST__CPU_MASK) << LAST__PID_SHIFT) | (-1 & LAST__PID_MASK));
        rmap_walk_locked(folio, &rwc);
        i += folio_nr_pages(folio);
        if (i >= nr)
            break;
        folio = folio_next(folio);
    }
}
```

Listing 5.21: Diff of should_numa_migrate_memory to migrate marked pages

```
@@ kernel/sched/fair.c:1807
bool should_numa_migrate_memory(struct task_struct *p, struct folio *folio,
                int src_nid, int dst_cpu)
{
    int dst_nid = cpu_to_node(dst_cpu);
    int last_cpupid, this_cpupid;
    // [...]
    this_cpupid = cpu_pid_to_cpupid(dst_cpu, current->pid);
    last_cpupid = folio_xchg_last_cpupid(folio, this_cpupid);

+   if (cpupid_pid_unset(last_cpupid) && cpupid_to_cpu(last_cpupid) == 9)
+       return true;

    /*
     * Allow first faults or private faults to migrate immediately early in
     * the lifetime of a task. The magic number 4 is based on waiting for
     * two full passes of the "multi-stage node selection" test that is
     * executed below.
     */
    if ((p->numa_preferred_nid == NUMA_NO_NODE || p->numa_scan_seq <= 4) &&
        (cpupid_pid_unset(last_cpupid) || cpupid_match_pid(p, last_cpupid)))
```

```
22          return true;
23
24      /*
25       * Multi-stage node selection is used in conjunction with a periodic
26       * migration fault to build a temporal task<->page relation. By using
27       * a two-stage filter we remove short/unlikely relations.
28       * [...]
29       * This quadric squishes small probabilities, making it less likely we
30       * act on an unlikely task<->page relation.
31       */
32      if (!cpupid_pid_unset(last_cpupid) &&
33                  cpupid_to_nid(last_cpupid) != dst_nid)
34          return false;
35
36      /* Always allow migrate on private faults */
37      if (cpupid_match_pid(p, last_cpupid))
38          return true;
39      // [...]
40  }
```

This modification is probably the most risky in terms of kernel stability, as we cannot guarantee that it will not introduce problems in unforeseen places in the code. Nevertheless, the current likelihood of a `cpu_pid` object having a CPU set but no PID remains very low, suggesting minimal immediate risk.

5.4.9 Migrating the Faulting Page

An additional optimization, which was relatively inexpensive, involved moving the split page causing the fault directly to the accessor node if remote. As previously discussed, this specific page will not be set to `prot_none` again, which means it must wait until the NUMA balancing scanning process sets it to `prot_none` again, then trigger a new NUMA hinted fault to be migrated. To streamline this process, we proposed and implemented a feature that moves the faulting page directly to the accessor node before the process resumes at the end of the page fault handling. Given our extensive migration codebase, this implementation was straightforward. It primarily involved integrating a migration component into the `handle_unlock` function. This adjustment ensures that the page is efficiently relocated to the accessor node, reducing the latency associated with multiple page fault handling steps.

Listing 5.22: Diff of do_huge_pmd_numa_page to pass the migration target node to handle_unlock if any

```
1  @@ mm/huge_memory.c:1952
```

```
vm_fault_t do_huge_pmd_numa_page(struct vm_fault *vmf)
{
    // [...]
    if (static_branch_likely(&sched_nb_split_shared_hugepages) && this_cpu_nid
    != nid) {
        // [...]
        int splitted = split_thp_on_page_fault(folio, vmf);
        if (splitted) {
-           handle_unlock(vmf);
+           handle_unlock(vmf,
+               (this_cpu_nid != nid) ? this_cpu_nid : NUMA_NO_NODE);
        }
    // [...]
}
```

Listing 5.23: Diff of handle_unlock to migrate the faulting page directly to the provided node

```
@@ mm/huge_memory.c:2018
-static int handle_unlock(struct vm_fault *vmf) {
+static int handle_unlock(struct vm_fault *vmf, int target_nid) {
    // Mostly copied from handle_pte_fault
    // [Safety checks, variable declarations, ...]
    spin_lock(vmf->ptl);
    old_pte = ptep_get(vmf->pte);
    // [More safety checks]
    pte = pte_modify(old_pte, vma->vm_page_prot);
    writable = pte_write(pte);

+   if (target_nid == NUMA_NO_NODE)
+       goto out_map;
+
+   folio = vm_normal_folio(vma, vmf->address, pte);
+   // [...]
+   if (migrate_misplaced_folio(folio, vma, target_nid))
+       return 0;
+   // [...]
+
+out_map:
    // [Make the PTE accessible again]
}
```

6 Evaluation

6.1 Exploratory Experiments

In this first part, we wanted to compare how the performance of each NAS benchmark application was impacted by automatic NUMA balancing. We also wanted to see if there was any performance difference between machines with 2 NUMA nodes and 4 NUMA nodes.

As mentioned in Subsection 4.1.1, we evaluated the performance of the following pinning configurations to understand their impact on application performance :

1. **sockorder**: Fill NUMA nodes one at the time, in the order of the cores.
2. **sockets**: Similar to the sockorder configuration, but the threads are free to move among the cores of the NUMA node to which they have been assigned.
3. **sequential**: Fill all the nodes concurrently in a round-robin fashion.
4. **none**: No pinning

Per Benchmark Analysis
Initially, we examined the performance differences between enabling and disabling NUMA balancing for each NAS benchmark (see Figure 6.1, Figure 6.2, Figure 6.3). The analysis revealed that, contrary to expectations, NUMA balancing generally resulted in performance degradation, with runtime increases exceeding 20% in some cases as shown in Figure 6.1. This trend was consistent across both 2-node and 4-node machines, with a more pronounced impact observed on 4-node machines. Notably, the only benchmarks that exhibited slight performance improvements with NUMA balancing were CG with data size C on both 2-node and 4-node machines, and UA with size C on 2-node machines only. The more significant performance impact on 4-node machines suggests that the adverse effects of NUMA balancing may escalate with an increasing number of NUMA nodes. However, due to the lack of machines with more than four nodes at our disposal, this hypothesis remains untested. It is crucial to note that these findings are specific to the NAS benchmarks and may not be representative of a broader range of computing workloads. Despite this, the results are noteworthy as they contradict the expected benefits of NUMA balancing, prompting

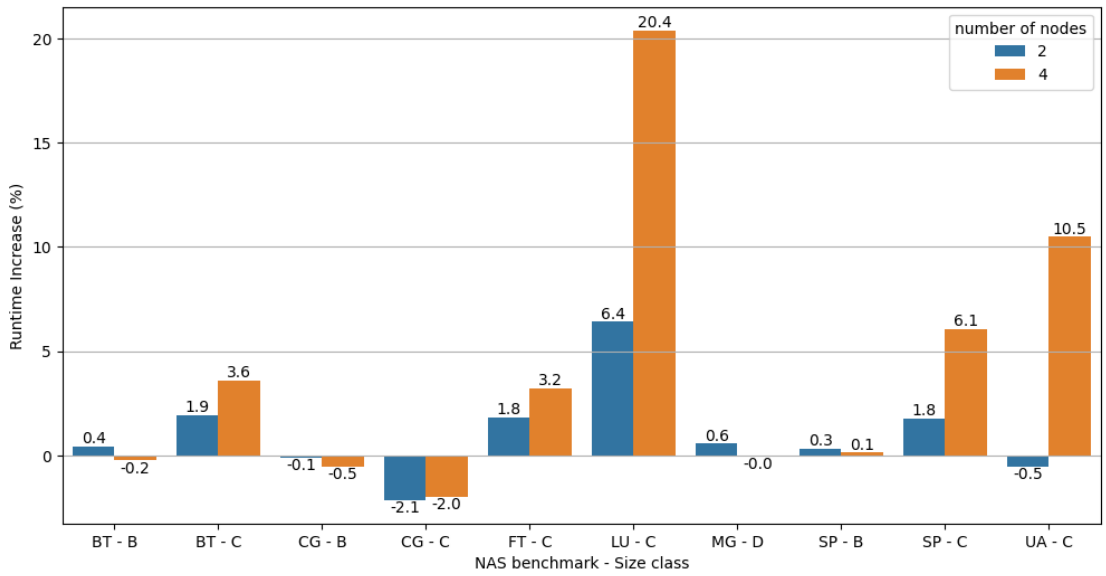

Figure 6.1: Variation of average runtime between NUMA balancing enabled and disabled, averaged over all pinning configurations. Positive values means NUMA balancing is worse.

further investigation into the underlying causes of this underperformance.

Per Pinning Configuration Analysis
Following this, we compared the performance variations caused by NUMA balancing for each pinning configuration averaging the results across all tested benchmarks. In Figure 6.4, the efficiency of NUMA balancing appeared even worse : no configuration demonstrated equal or better results with NUMA balancing enabled compared to when it was disabled. It is important to highlight that for pinning configurations other than "none," only the memory migration aspect of NUMA balancing was active, as threads were restricted to their assigned cores. These results were essentially a different viewpoint on the inability of NUMA balancing to improve performance in the NAS benchmarks in general.

Between Pinning Configurations Analysis
Finally, we shifted our focus away from NUMA balancing to examine how application performance varied between different pinning configurations with NUMA balancing disabled (see Figure 6.5, Figure 6.6, Figure 6.7). This analysis revealed a significant insight : the sequential pinning configuration mostly underperformed compared to

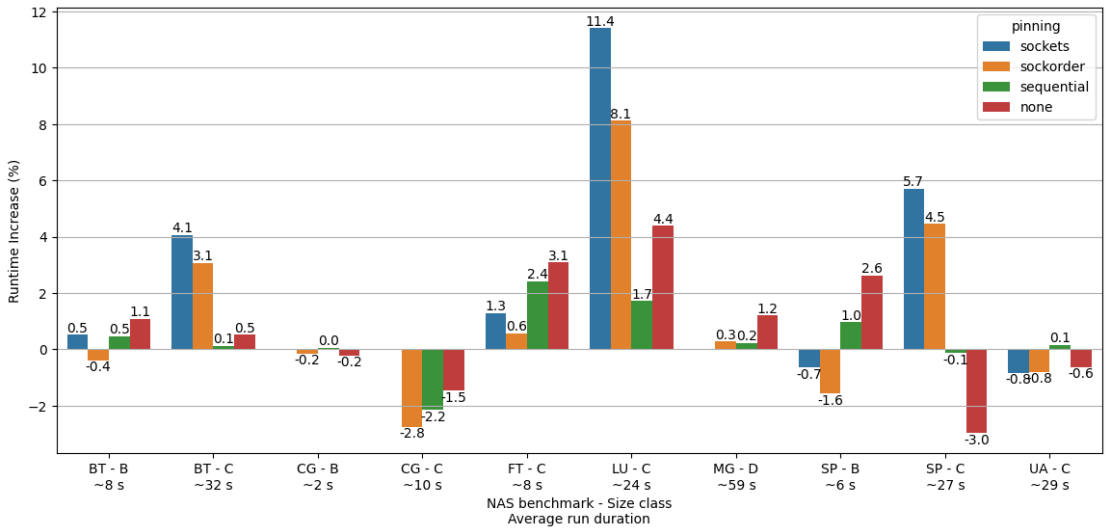

Figure 6.2: Variation of average runtime between NUMA balancing enabled and disabled, dual-node machines.

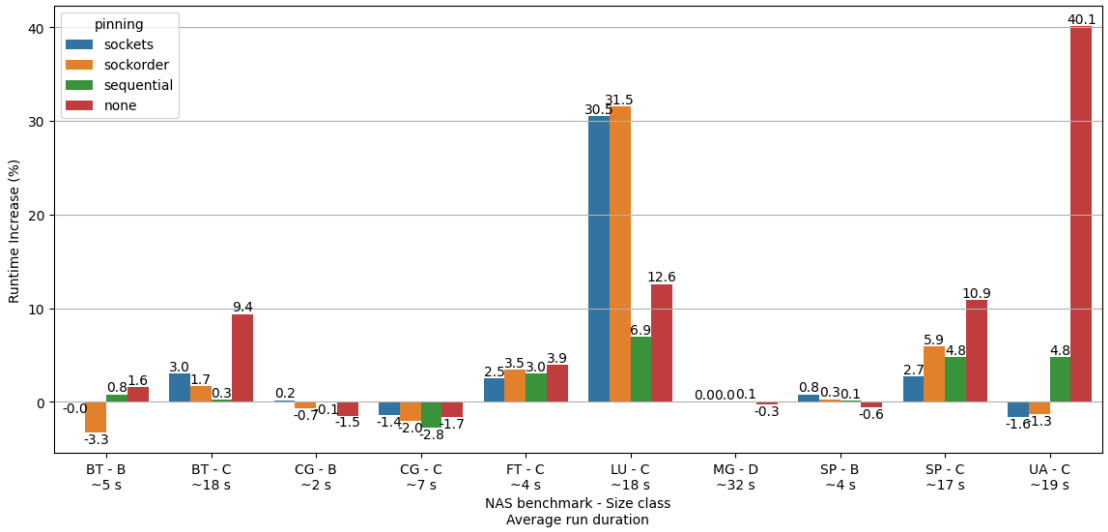

Figure 6.3: Variation of average runtime between NUMA balancing enabled and disabled, quad-node machines.

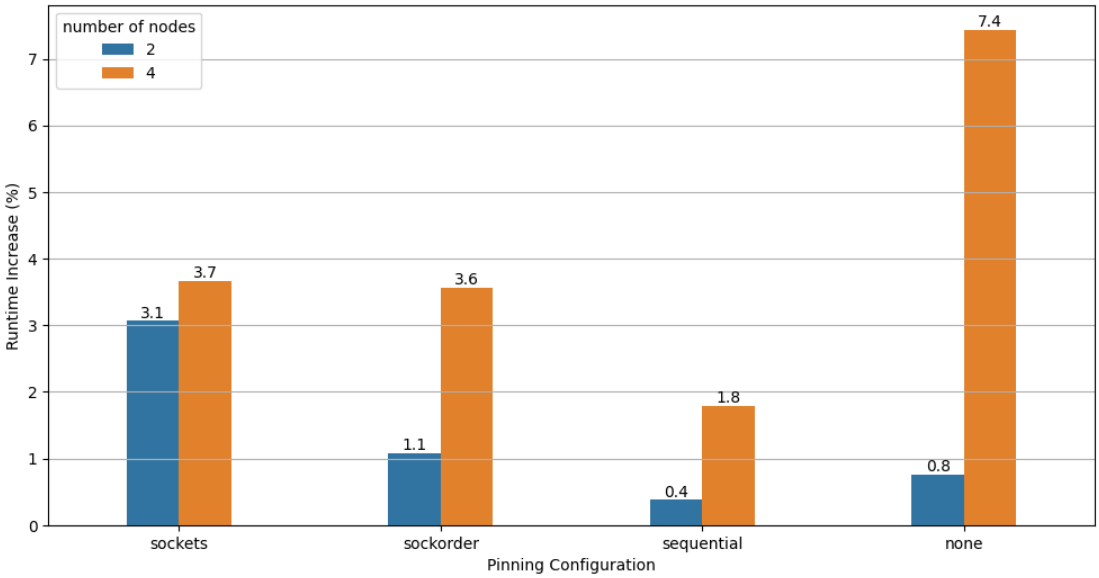

Figure 6.4: Variation of average runtime between NUMA balancing enabled and disabled, averaged across all benchmarks.

the sockorder and sockets configurations, which exhibited similar performance levels. This unexpected difference in performance motivated the second part of our study, aimed at uncovering the underlying reasons for this discrepancy which initially seemed counterintuitive.

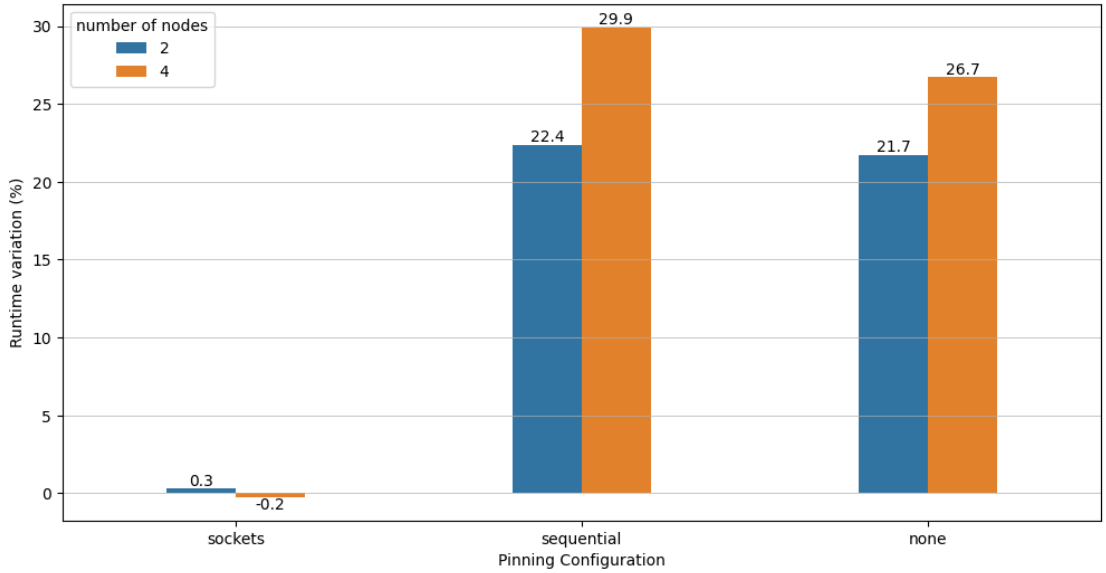

Figure 6.5: Variation of average runtime compared to pinning configuration "sockorder". NUMA balancing disabled.

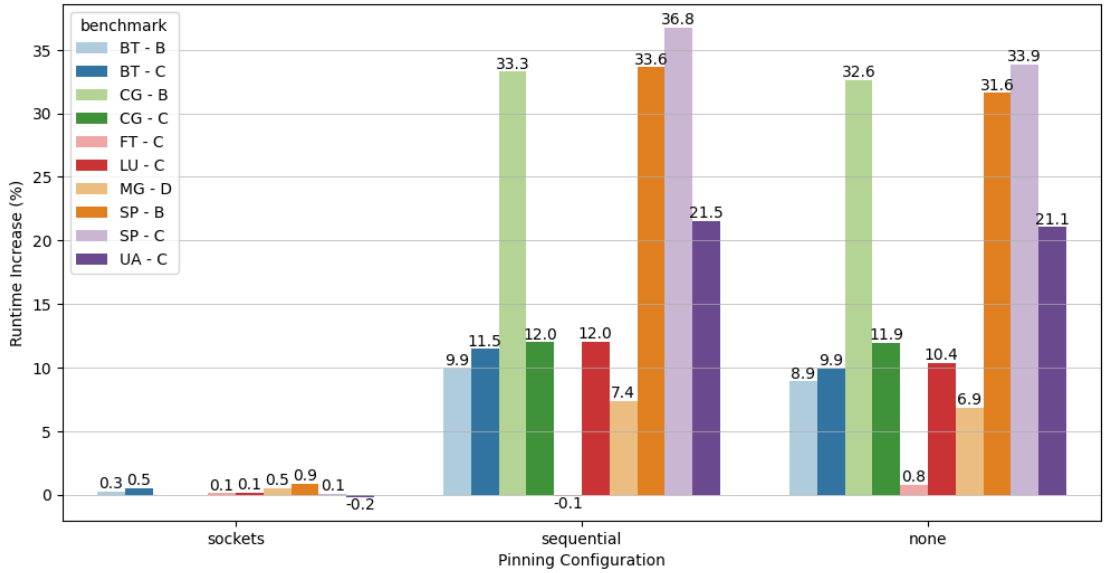

Figure 6.6: Variation of average runtime compared to pinning configuration "sockorder", dual-node machines. NUMA balancing disabled.

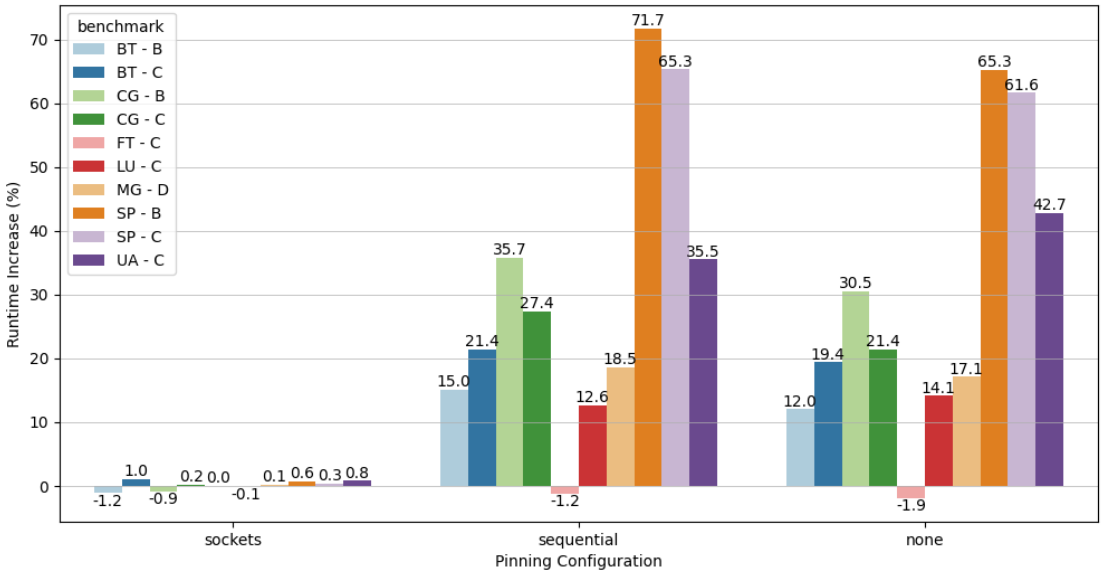

Figure 6.7: Variation of average runtime compared to pinning configuration "sockorder", quad-node machines. NUMA balancing disabled.

6.2 Profiling

6.2.1 False Cache Sharing

Figure 6.8 shows the result of the memory spacing experiments.

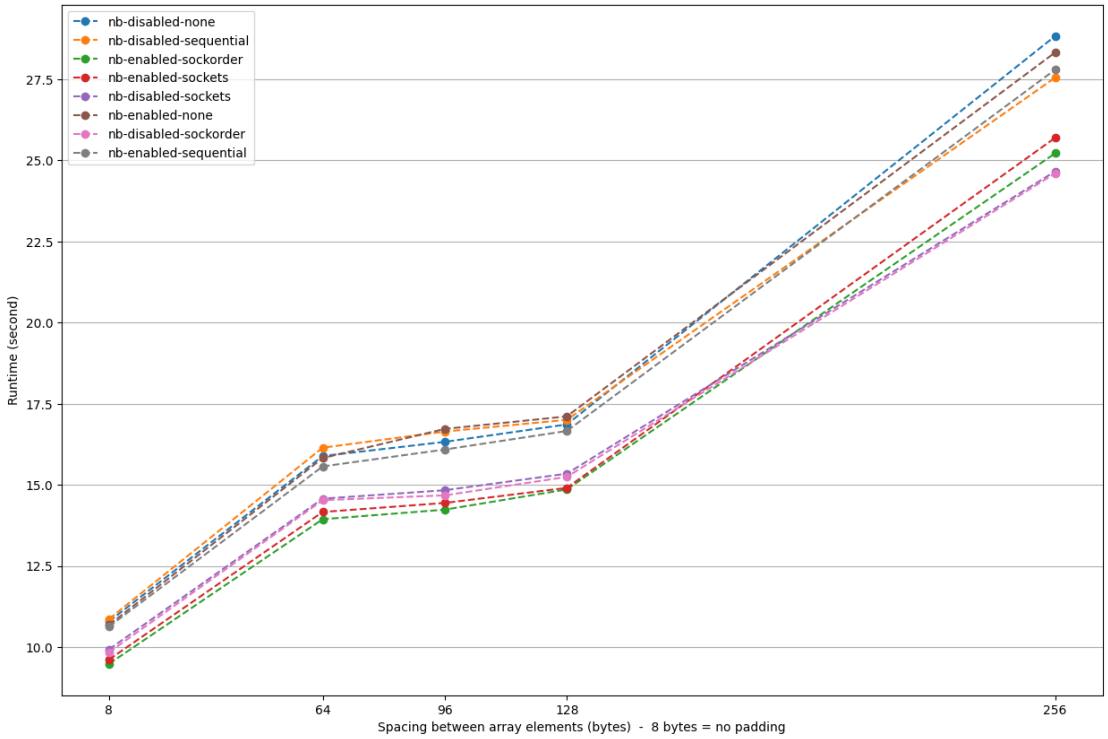

Figure 6.8: Runtime for various pinning policies at different spacing values between array elements, NAS benchmark CG with size C, varius dahus. Array elements are 8 bytes. Each column was run as one experiment.

If the issue was linked to cache sharing, we would have expected to see the two performance groups merge into one as we increased the amount of padding between the elements. However this is not what we see, meaning the performance difference is unlikely to come from false cache sharing.

6.2.2 Correlation Analysis

Given that the NAS benchmark CG was the only application whose performance improved with the activation of NUMA balancing, and considering its rather straightforward code structure, our analysis primarily concentrated on this benchmark. As

explained in Subsection 5.3.2, we derived new metrics from existing ones to facilitate more meaningful data analysis. Specifically, we calculated the following ratios:

- Metric A : $\frac{mem\_load\_l3\_miss\_retired.local\_dram}{all(mem\_load\_l3\_miss\_retired)}$
- Metric B : $\frac{mem\_load\_l3\_miss\_retired.remote\_dram}{all(mem\_load\_l3\_miss\_retired)}$
- Metric C : $\frac{mem\_load\_l3\_miss\_retired.remote\_hitm}{all(mem\_load\_l3\_miss\_retired)}$
- Metric D : $\frac{mem\_load\_l3\_miss\_retired.remote\_fwd}{all(mem\_load\_l3\_miss\_retired)}$

With $all(mem\_load\_l3\_miss\_retired) = mem\_load\_l3\_miss\_retired.local\_dram + mem\_load\_l3\_miss\_retired.remote\_hitm + mem\_load\_l3\_miss\_retired.remote\_fwd$

We examined the correlation between various event metrics and runtime for the CG benchmark with data size C. The results, illustrated in Figure 6.9, were particularly striking. Specifically, the average of Metric A across all cores exhibited an exceptionally high negative correlation coefficient of -0.987. Although it does not mean a high average of Metric A will always result in a short runtime, it suggests a strong relationship between the two and warrants further investigation. Overall, these results suggest that the performance impact of memory locality is particularly strong in situations with memory load L3 misses. This finding led us to focus our study on memory load L3 misses, as they may partly explain the poor performance observed in the sequential pinning configuration compared to the sockorder or sockets pinning configurations.

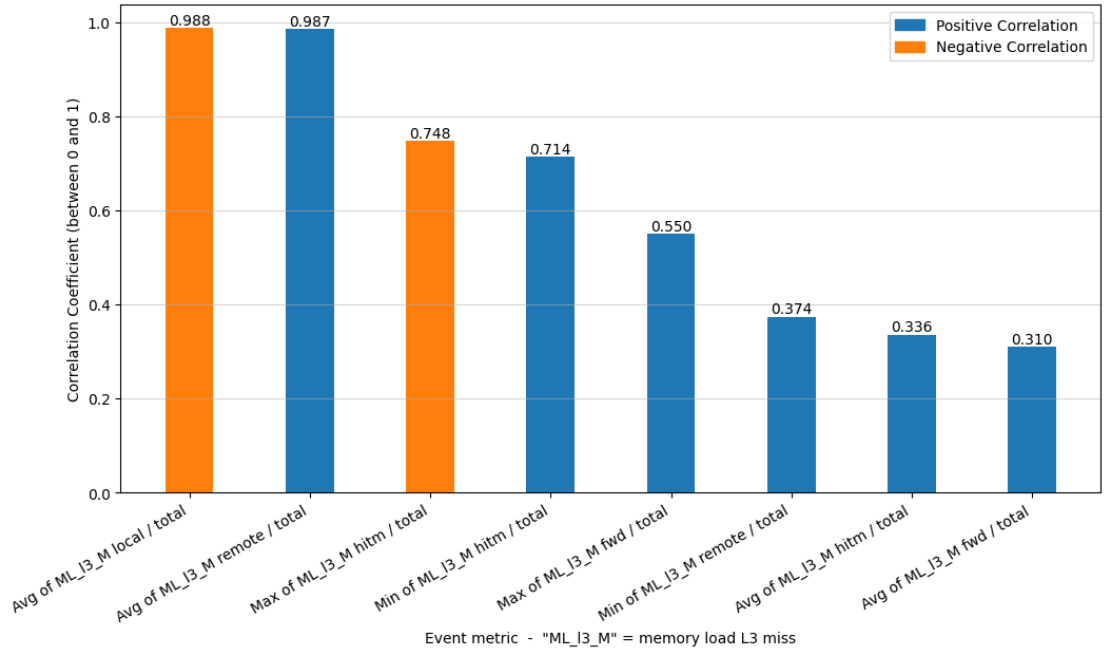

Figure 6.9: Coefficient of correlation between events metrics and runtime for NAS benchmark CG with size C, dual-node machine

6.2.3 Memory Access Patterns

After extensive adjustments to the observed memory space region, the plotted variables, and the plotting configuration, we produced the diagram presented in Figure 6.10. This diagram illustrates memory accesses and memory allocations within the virtual address space. Memory accesses are depicted as dots with the following color coding:

- Green dots indicate any type of memory load.
- Blue dots indicate any type of memory store.
- Red dots indicate remote RAM memory load L3 misses, as previously mentioned.
- Memory accesses from the same CPU are connected by a colored dotted line to facilitate the identification of zones accessed by each CPU.

Memory allocations are represented by vertical bars, with the color of each bar corresponding to the CPU responsible for the allocation.

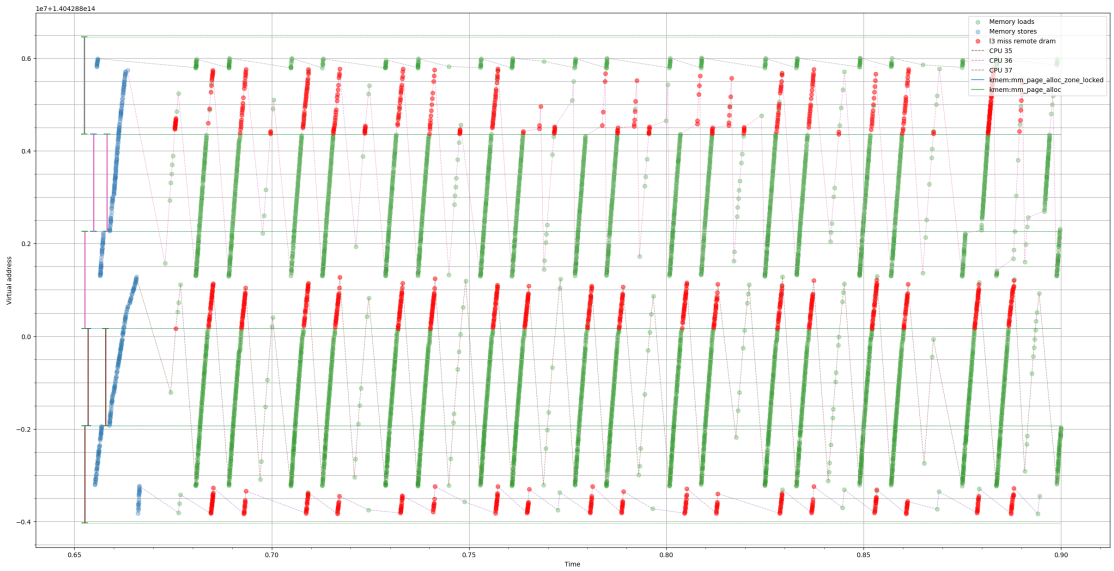

Figure 6.10: Memory accesses and memory allocation in the virtual address space through time. Vertical bars show memory allocation and are 512 pages long.

The diagram reveals that certain allocation ranges are accessed by the "next" processor, leading to numerous remote DRAM memory load L3 misses. However, in

these occurrences the two processors access distinct regions of the memory allocation, essentially creating a scenario of false page sharing of the allocation space. Upon closer examination, we observed that the allocations are 512 pages long, matching the size of a huge page. This observation led us to suggest that these allocations could be transparent huge pages. Consequently, we tested the hypothesis by deactivating transparent huge pages, which resulted in a runtime decrease of more than 10%, and confirmed that :

1. The allocations are indeed transparent huge pages.
2. The false sharing of huge pages is one cause - though not necessarily the only one - of performance degradation observed across the various pinning configurations tested.

Having identified one of the causes of the poor performance of the sequential pinning configuration compared to the others, we set out to develop a solution that would be more refined than simply deactivating transparent huge pages across the entire machine.

6.3 Proof of Concept

To assess the effectiveness of the proof of concept as a whole as well as each of the improvements discussed in the Implementation chapter, we used kernel parameters to manage the activation and deactivation of each optimization individually. This approach allowed us to systematically compare the performance impact of each modification. The results of these comparisons are presented in Figure 6.11.

The configurations evaluated are labeled using the following code :

- `nb` / `no-nb` : NUMA balancing enabled / disabled.
- `no-hp`: Transparent huge pages disabled on the machine.
- `split` : Enabled splitting of pages as described in Subsection 5.4.6. Otherwise the code change is simply not enabled.
- `reapply` : Enabled optimization of reapplying NUMA protections after the split, as described in Subsection 5.4.7.
- `migrate` : Enabled optimization of migrating the faulting page to the accessing node directly, as described in Subsection 5.4.9.

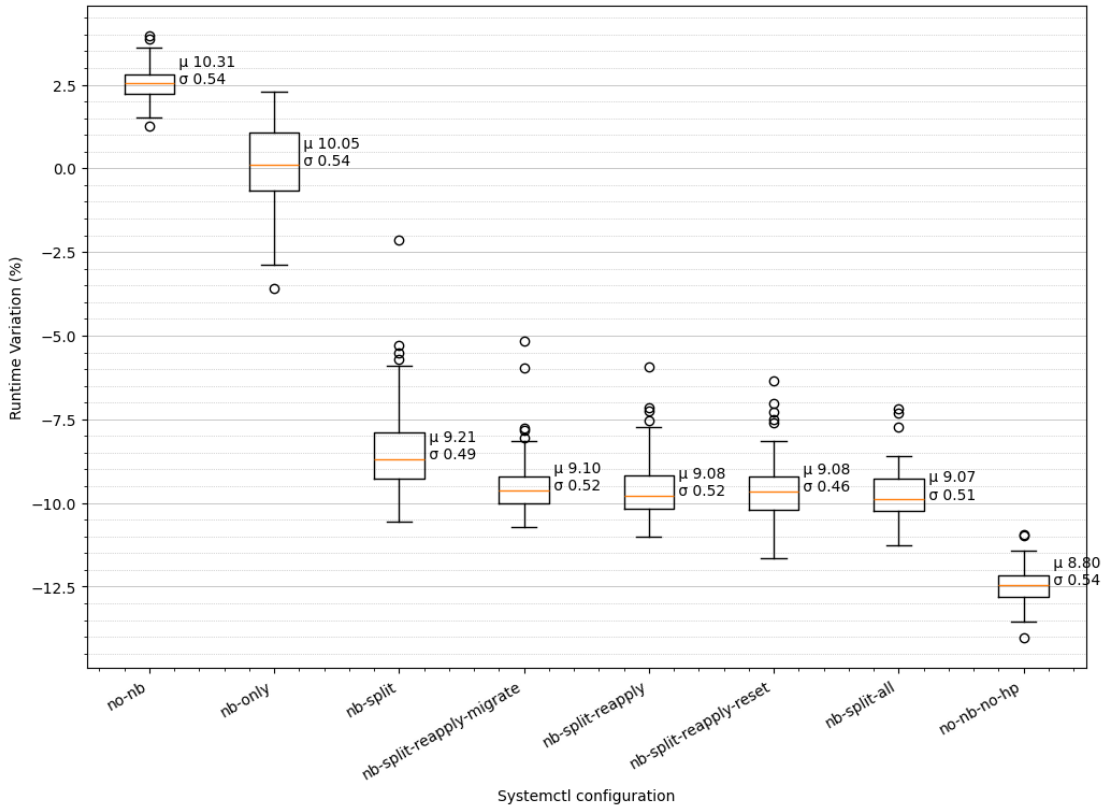

Figure 6.11: Variation of runtime compared to the configuration nb-only

- `reset` : Enabled optimization of resetting the NUMA balancing counters after split, as described in Subsection 5.4.8.
- `all` : All optimizations enabled

Immediately, we observed a **decrease in runtime by more than 8% with the refined page split mechanism** (without same node migration). This reduction is quite promising for a proof of concept. However, the other optimizations demonstrated only marginal improvements, particularly in comparison with each other. The reasons for the limited performance gains from these "optimizations" are not yet clear. We propose several hypotheses to explain this weak performance increase:

1. The scanning frequency might be either too high or too low to render the optimizations effective.
2. There may be additional optimizations that our current ones depend on, which we have not yet identified or implemented.
3. The implementation method might be obstructing their effectiveness.
4. These optimizations may not be necessary at all.

Despite these challenges, the results are promising and would deserve further exploration. It is important to note that, by design, we should not be able to surpass the performance of the configuration with transparent pages disabled. Nonetheless, there remains significant potential for improvement to approach this performance benchmark.

7 Related work

In their 2012 paper [21], Lachaize et al present a sophisticated profiler designed to aid programmers in optimizing memory access patterns on NUMA architectures. MemProf constructs temporal flows of interactions between threads and memory objects, providing detailed insights into remote memory accesses. This allows for the identification of inefficiencies and implementation of effective application-level optimizations. The profiler has demonstrated significant performance improvements in various applications, achieving gains up to 161% with minimal code modifications. Despite its potential usefulness in our research, MemProf is limited to AMD processors due to its reliance on Instruction-Based Sampling (IBS), which is not supported by the Intel hardware used in our study. This hardware incompatibility precluded its direct application in our experiments.

In 2014, Gaud et al. investigated the performance implications of using large pages in NUMA architectures [15]. The study reveals that huge pages, while intended to reduce TLB misses and page faults, can exacerbate NUMA-related issues such as poor locality and memory controller imbalances. The authors propose and evaluate Carrefour-LP, an extension of the NUMA-aware page placement algorithm Carrefour, which dynamically splits large pages to address these issues. The results show that Carrefour-LP can restore or improve performance by mitigating the adverse effects of large pages on NUMA systems. This work underscores the importance of fine-grained memory management in NUMA environments.

Despite its relevance, Carrefour-LP was designed and tested on AMD hardware also utilizing Instruction-Based Sampling (IBS), limiting its direct applicability to the Intel-based system used in our study. The other key difference with our work is that while Gaud et al. integrate their solution within the framework laid out by Carrefour, our approach integrates directly into the automatic NUMA balancing mechanism of the Linux kernel. Additionally, a decade has passed since their work, during which the kernel has become increasingly more complex, justifying a renewed effort to explore what further optimizations are possible in the current context.

Finally, in 2016 Park et al investigated the performance effects of using large pages in NUMA systems, particularly in the context of in-memory big-data processing. [33]

The authors find that large pages generally offer minimal performance gains over 4KB pages when handling sufficiently large datasets. However, smaller datasets and larger NUMA system scales tend to benefit more from large pages due to reduced page fault rates and TLB misses. The study also highlights significant performance degradation in some cases due to NUMA-related issues, such as frequent remote memory accesses and garbage collection inefficiencies. The authors propose optimizations, including low-overhead automatic NUMA balancing and advanced TLBs, to mitigate these issues. Despite the relevance of these findings, this work primarily focuses on the specific challenges of big-data workloads and does not address some of the complexities of modern Linux kernel memory management mechanisms that have evolved over the past decade.

8 Summary and conclusion

This thesis explored the intricacies of NUMA systems, and in more specifically their optimization within the Linux kernel. The research was motivated by the observation that despite the potential of NUMA architectures for enhanced performance in multicore systems, existing memory management strategies like the Linux kernel's automatic NUMA balancing mechanism sometimes fall short of delivering optimal performance. The study investigated the reasons behind the suboptimal performance observed in certain multicore benchmarks and proposed targeted solutions to address these issues.

The research methodology employed includes three key steps:

1. **Exploratory Experiments**: Initial experiments were conducted to identify performance anomalies and interesting behaviors in NUMA systems.
2. **Profiling** : A deeper analysis was performed using correlation analysis on hardware events to pinpoint the causes of performance issues. This phase also involved sampling events to identify memory access patterns.
3. **Proof of Concept** : Based on the findings, a kernel patch was designed to address the identified issues, specifically focusing on the management of transparent huge pages.

An automated testing environment leveraging Intel Performance Counter Monitor (PCM) via the perf Linux tool was created to facilitate the experiments, and Jupyter Notebooks were used for detailed examination and visualization of performance metrics.

The research uncovered a significant issue related to the false sharing of transparent huge pages, where pages allocated on one node but utilized by CPUs on different nodes led to performance slowdowns of up to 30%. This false sharing was identified as a major bottleneck in achieving optimal performance in NUMA systems under specific computing loads.

The proof of concept designed to address this issue focused on optimizing the handling of transparent huge pages within the automatic NUMA balancing mechanism.

Implementing this patch along with some optimizations resulted in a performance increase of 9% on the NAS benchmark CG with data size C. These results demonstrate the potential for non-negligible performance gains through targeted adjustments in memory management strategies.

9 Future work

9.1 Profiler

One of the most valuable yet low-effort advancements would be to extend the current analysis framework for memory access patterns into a tool designed to quickly detect false sharing of huge pages by recording program runs. While such a tool might require some time to extract and process the large volume of data produced, this duration could be reduced by lowering the sampling rate.

The existing infrastructure already constructs a detailed representation of memory allocations and accesses within the virtual address space, providing a solid foundation upon which this tool could be built. By leveraging this infrastructure, the development of a tool to identify non-overlapping accesses would require minimal additional work. However, it is important to note that the accuracy of detecting non-overlapping accesses may decrease as the sampling rate is reduced.

Despite this potential limitation, the creation of a tool for detecting false sharing could significantly enhance our ability to optimize memory management in NUMA architectures, providing quick and actionable insights into memory access patterns and helping to identify and mitigate performance bottlenecks caused by false sharing.

9.2 Proof of Concept

As seen in the Evaluation section, the results of the various optimizations we attempted are rather disappointing, highlighting the need for a deeper understanding of the underlying issues. This study could greatly benefit from a comprehensive analysis tool to track specific operations and ensure that everything proceeds as planned. Such a tool would enable us to gain better insights into the page migration process and identify opportunities for more efficient strategies.

Additionally, extending these experiments to a wider range of benchmarks and tools is absolutely necessary. The current results are limited to one NAS benchmark, and it

is crucial to determine if similar outcomes can be observed in other applications and contexts. This broader data collection would significantly enhance the validity and publishability of our findings. It is also essential to test on different hardware and architectures, as the observed page sharing issues and L3 remote misses may not be universally applicable nor be the primary cause of slowdowns across various systems.

9.3 Going Further

One promising avenue for future research is the utilization of Intel Memory Protection Keys (MPK) to enhance the detection and management of memory accesses across different NUMA nodes. Intel MPK, introduced in the Skylake architecture, is a userspace mechanism that allows for the tagging of memory pages with up to 16 distinct protection keys. This tagging facilitates the rapid adjustment of memory access rights directly from userspace without the overhead of frequent system calls, as would be required with traditional methods like `mprotect` that is being used in the automatic NUMA balancing to set pages to `prot_numa`

By making use of MPK, for example though libmpk [34], we could potentially capture the fact that a memory page is being accessed by different NUMA nodes via hardware mechanisms rather than via a costly page protection scheme. Specifically, by tagging pages with protection keys and monitoring their access rights through the PKRU register, we could directly observe and log cross-node memory accesses with minimal performance overhead. This method could provide real-time insights into memory access patterns and significantly streamline the process of identifying shared pages.

Implementing this approach could allow parts of the automatic NUMA balancing mechanism to avoid the costly operations of marking pages as `prot_numa`. Instead, it would rely on the hardware to flag shared accesses. This would not only reduce the performance penalty associated with fault handling but also enhance the granularity and accuracy of access tracking, leading to more efficient memory management strategies in NUMA environments.

List of Figures

List of Tables

Listings

Bibliography

[1] K. Alnaes, E. Kristiansen, D. Gustavson, and D. James. "Scalable Coherent Interface." In: June 1990, pp. 446–453. ISBN: 0-8186-2041-2. DOI: 10.1109/CMPEUR.1990.113656.

[2] *AutoNUMA: the other approach to NUMA scheduling [LWN.net] — lwn.net*. https://lwn.net/Articles/488709/. [Accessed 30-06-2024].

[3] J. Backus. "Can programming be liberated from the von Neumann style? a functional style and its algebra of programs." In: *Commun. ACM* 21.8 (Aug. 1978), pp. 613–641. ISSN: 0001-0782. DOI: 10.1145/359576.359579.

[4] R. Bharadwaj. *Mastering Linux Kernel Development*. Packt Publishing, 2017. ISBN: 9781785883057.

[5] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, M. F. Kaashoek, R. T. Morris, A. Pesterev, L. Stein, M. Wu, Y.-h. Dai, et al. "Corey: An Operating System for Many Cores." In: *OSDI*. Vol. 8. 2008, pp. 43–57.

[6] R. Bryant and J. Hawkes. "Linux scalability for large NUMA systems." In: *Linux Symposium*. Vol. 76. 2003.

[7] S. Cleveland, S. Swanstrom, and C. Neuts. "HyperTransport™ Technology: Simplifying System Design." In: *White Paper AMD* (2002), pp. 1–22.

[8] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. "Traffic management: a holistic approach to memory placement on NUMA systems." In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '13. Houston, Texas, USA: Association for Computing Machinery, 2013, pp. 381–394. ISBN: 9781450318709. DOI: 10.1145/2451116.2451157.

[9] P. J. Denning. "Virtual memory." In: *ACM Computing Surveys (CSUR)* 2.3 (1970), pp. 153–189.

[10] M. Dobson, P. Gaughen, M. Hohnbaum, and E. Focht. "Linux support for NUMA hardware." In: *Proc. Linux Symposium*. Vol. 2003. Citeseer. 2003, pp. 169–184.

[11] M. Dubois and F. A. Briggs. "Effects of cache coherency in multiprocessors." In: *SIGARCH Comput. Archit. News* 10.3 (Apr. 1982), pp. 299–308. ISSN: 0163-5964. DOI: 10.1145/1067649.801739.

[12] S. J. Eggers and R. H. Katz. "Evaluating the performance of four snooping cache coherency protocols." In: *Proceedings of the 16th Annual International Symposium on Computer Architecture*. ISCA '89. Jerusalem, Israel: Association for Computing Machinery, 1989, pp. 2–15. ISBN: 0897913191. DOI: 10.1145/74925.74927.

[13] *ftrace - Function Tracer 2014; The Linux Kernel documentation — kernel.org*. https://www.kernel.org/doc/html/v5.0/trace/ftrace.html. [Accessed 30-06-2024].

[14] J. Funston, M. Lorrillere, A. Fedorova, B. Lepers, D. Vengerov, J.-P. Lozi, and V. Quéma. "Placement of virtual containers on NUMA systems: a practical and comprehensive model." In: *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '18. Boston, MA, USA: USENIX Association, 2018, pp. 281–293. ISBN: 9781931971447.

[15] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quema. "Large Pages May Be Harmful on NUMA Systems." In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, June 2014, pp. 231–242. ISBN: 978-1-931971-10-2.

[16] *graphica.com.au*. https://www.graphica.com.au/files/numalink.pdf?ref=just.graphica.com.au. [Accessed 30-06-2024].

[17] *intel.cn*. https://www.intel.cn/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf. [Accessed 30-06-2024].

[18] *intel.cn*. https://www.intel.cn/content/dam/www/public/us/en/documents/technology-briefs/intel-labs-single-chip-cloud-overview-paper.pdf. [Accessed 30-06-2024].

[19] *Intel® Performance Counter Monitor - A Better Way to Measure CPU... — intel.com*. https://www.intel.com/content/www/us/en/developer/articles/tool/performance-counter-monitor.html. [Accessed 30-06-2024].

[20] T. Johnson and D. Shasha. "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm." In: *Proceedings of the 20th International Conference on Very Large Data Bases*. VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 439–450. ISBN: 1558601538.

[21] R. Lachaize, B. Lepers, and V. Quema. "MemProf: A Memory Profiler for NUMA Multicore Systems." In: *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, June 2012, pp. 53–64. ISBN: 978-931971-93-5.

[22] C. Lameter. "NUMA (Non-Uniform Memory Access): An Overview: NUMA becomes more common because memory controllers get close to execution units on microprocessors." In: *Queue* 11.7 (July 2013), pp. 40–51. ISSN: 1542-7730. DOI: 10.1145/2508834.2513149.

[23] J. Lawall, H. Chhaya-Shailesh, J.-P. Lozi, B. Lepers, W. Zwaenepoel, and G. Muller. "OS scheduling with nest: keeping tasks close together on warm cores." In: *Proceedings of the Seventeenth European Conference on Computer Systems*. EuroSys '22. Rennes, France: Association for Computing Machinery, 2022, pp. 368–383. ISBN: 9781450391627. DOI: 10.1145/3492321.3519585.

[24] B. Lepers, V. Quéma, and A. Fedorova. "Thread and memory placement on NUMA systems: asymmetry matters." In: *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '15. Santa Clara, CA: USENIX Association, 2015, pp. 277–289. ISBN: 9781931971225.

[25] *Linux kernel version history - Wikipedia — en.wikipedia.org*. https://en.wikipedia.org/wiki/Linux_kernel_version_history. [Accessed 30-01-2024].

[26] *LKML: Mel Gorman: Re: [PATCH 00/49] Automatic NUMA Balancing v10 — lkml.org*. https://lkml.org/lkml/2012/12/9/108. [Accessed 30-06-2024].

[27] *Multi-size THP for anonymous memory [LWN.net] — lwn.net*. https://lwn.net/Articles/954094/. [Accessed 30-06-2024].

[28] *NAS Parallel Benchmarks — nas.nasa.gov*. https://www.nas.nasa.gov/software/npb.html. [Accessed 30-06-2024].

[29] D. S. Nikolopoulos and T. S. Papatheodorou. "The architectural and operating system implications on the performance of synchronization on ccNUMA multiprocessors." In: *International Journal of Parallel Programming* 29 (2001), pp. 249–282.

[30] E. J. O'Neil, P. E. O'Neil, and G. Weikum. "The LRU-K page replacement algorithm for database disk buffering." In: *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*. SIGMOD '93. Washington, D.C., USA: Association for Computing Machinery, 1993, pp. 297–306. ISBN: 0897915925. DOI: 10.1145/170035.170081.

[31] *OMP_PLACES — openmp.org*. https://www.openmp.org/spec-html/5.0/openmpse53.html. [Accessed 30-06-2024].

[32] *Page Table Management — kernel.org*. https://www.kernel.org/doc/gorman/html/understand/understand006.html. [Accessed 30-06-2024].

[33] J. Park, M. Han, and W. Baek. "Quantifying the performance impact of large pages on in-memory big-data workloads." In: *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 2016, pp. 1–10. DOI: 10.1109/IISWC.2016.7581281.

[34] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim. "libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK)." In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 241–254. ISBN: 978-1-939133-03-8.

[35] *Perf Wiki — perf.wiki.kernel.org*. https://perf.wiki.kernel.org/index.php. [Accessed 30-06-2024].

[36] *perf-stat(1) - Linux manual page — man7.org*. https://man7.org/linux/man-pages/man1/perf-stat.1.html. [Accessed 30-06-2024].

[37] *PerfMon Events — perfmon-events.intel.com*. https://perfmon-events.intel.com/skylake_server.html. [Accessed 30-06-2024].

[38] S. Przybylski, M. Horowitz, and J. Hennessy. "Characteristics of performance-optimal multi-level cache hierarchies." In: *Proceedings of the 16th Annual International Symposium on Computer Architecture*. ISCA '89. Jerusalem, Israel: Association for Computing Machinery, 1989, pp. 114–121. ISBN: 0897913191. DOI: 10.1145/74925.74939.

[39] A. J. Smith. "Cache memories." In: *ACM Computing Surveys (CSUR)* 14.3 (1982), pp. 473–530.

[40] *Toward better NUMA scheduling [LWN.net] — lwn.net*. https://lwn.net/Articles/486858/. [Accessed 30-06-2024].

[41] *trace-cmd: A front-end for Ftrace [LWN.net] — lwn.net*. https://lwn.net/Articles/410200/. [Accessed 30-06-2024].

[42] *Transparent Hugepage Support 2014; The Linux Kernel documentation — kernel.org*. https://www.kernel.org/doc/html/next/admin-guide/mm/transhuge.html. [Accessed 30-06-2024].