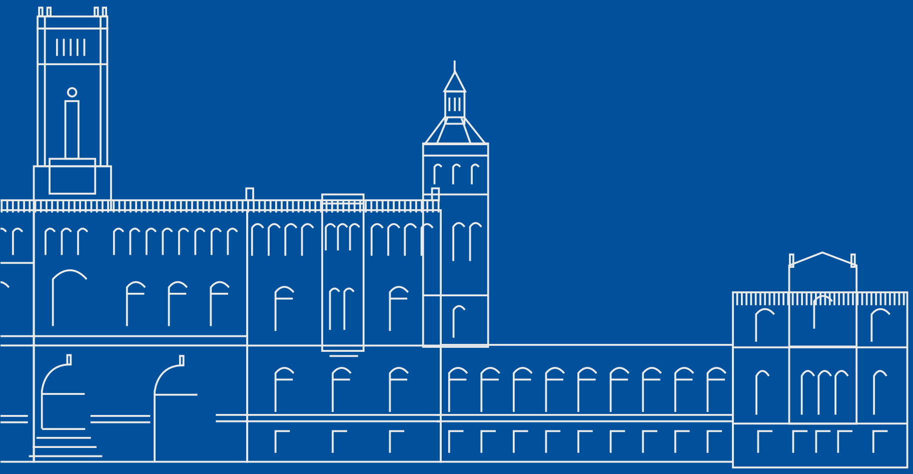
Leibniz Universität Hannover
Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur

Alexander Halbuer

Self-Contained Virtual-Memory Areas for Non-Volatile RAM in the Linux Kernel

Masterarbeit im Fach Technische Informatik

16. November 2022



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

www.sra.uni-hannover.de

Self-Contained Virtual-Memory Areas for Non-Volatile RAM in the Linux Kernel

Masterarbeit im Fach Technische Informatik

vorgelegt von

Alexander Halbuer

angefertigt am

Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur

Fakultät für Elektrotechnik und Informatik
Leibniz Universität Hannover

Erstprüfer:	**Prof. Dr.-Ing. habil. Daniel Lohmann**
Zweitprüfer:	**Prof. Dr.-Ing. Christian Dietrich**
Betreuer:	**Florian Rommel, M.Sc**
	Lars Wrenger, M.Sc

Beginn der Arbeit:	**16. Mai 2022**
Abgabe der Arbeit:	**16. November 2022**

Erklärung

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

Declaration

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

(Alexander Halbuer)
Hannover, 16. November 2022

ABSTRACT

Memory might be the most important resource managed by an operating system, and the amount of memory per machine increases steadily. Current memory management is based on the past assumptions of scarcity and hardware independence, and it is not designed to deal with high bandwidth devices, the high parallelism of multi-core CPUs, and new persistent main memory. Modern operating systems virtualize their memory using *paging*, where all management is done in the granularity of small *pages*, typically 4 KiB wide. Therefore, the costly handling of many individual pages is required for large amounts of data.

This thesis proposes *morsels*, a new memory primitive, as the solution to those problems. It sacrifices hardware independence in favor of performance by utilizing existing paging data structures to define a self-contained memory object that the hardware can directly interpret. This approach moves the management from individual pages to larger granularities while still relying on small pages to bypass the problems of huge pages. Atomic instructions should provide thread safety and crash consistency as well as good performance avoiding the need for *locks* and *logs*.

The prototypic implementation as a Linux kernel module for the x86-64 architecture shows better multi-core scalability than a shared mapping in terms of population speed. Furthermore, a morsel can be mapped in constant time, making it suitable for transferring large amounts of data between address spaces. In conclusion, a morsel is an efficient, general-purpose memory primitive with inherent support for persistent main memory.

KURZFASSUNG

Speicher ist eine der wichtigsten Ressourcen, die von einem Betriebssystem verwaltet werden, und die Menge an Speicher pro System steigt stetig an. Die aktuelle Speicherverwaltung basiert auf den alten Annahmen von Knappheit und Hardwareunabhängigkeit, und ist nicht für Geräte mit hoher Bandbreite, die hohe Parallelität von Mehrkernprozessoren, sowie neuen persistenten Hauptspeicher ausgelegt. Moderne Betriebssysteme virtualisieren ihren Speicher mit *Paging*, wobei alles in meist 4 KiB großen *Seiten* verwaltet wird. Dementsprechend ist für große Datenmengen der teure Umgang mit vielen einzelner Seiten notwendig.

Diese Arbeit schlägt *Morsels*, eine neue Speicherprimitive, als Lösung für diese Probleme vor. Sie gibt Hardwareunabhängigkeit zu Gunsten besserer Performance auf, indem die existierenden Paging-Datenstrukturen verwendet werden, um ein eigenständiges Speicherobjekt zu definieren, das direkt von der Hardware interpretiert werden kann. Mit diesem Ansatz wird Speicher nicht mehr als einzelne Seiten sondern als größere, unteilbare Objekte verwaltet. Auf unterster Ebene werden die kleinen Seiten jedoch beibehalten, um die Nachteile von *Huge Pages* zu umgehen. Atomare Operationen sollen durch die Vermeidung von *Locks* und *Logs* für Threadsicherheit und Absturzkonsistenz, sowie gute Performance sorgen.

Die prototypische Implementierung als Linux Kernel-Modul für die x86-64 Architektur zeigt eine bessere Skalierung auf Mehrkernprozessoren im Vergleich zu einem geteilten *Mapping* in Bezug auf die *Population*-Geschwindigkeit. Darüber hinaus kann ein Morsel in konstanter Zeit gemappt werden, was sie für den Transfer von großen Datenmengen zwischen Adressräumen eignet. Zusammengefasst ist ein Morsel eine effiziente, allgemein anwendbare Speicherprimitive mit inhärenter Unterstützung für persistenten Hauptspeicher.

CONTENTS

1 INTRODUCTION

Operating systems (OSs) have two major tasks: managing hardware resources and providing appropriate abstractions of the machine for user applications. Memory might be the most important hardware resource the OS manages.[1] Current abstractions virtualize memory using paging, which divides physical memory into equally sized chunks managed by the OS in page tables, a tree-like data structure. All memory management, e.g., allocation and swapping, is done in the granularity of those chunks, called pages. The whole memory subsystem is based on the past assumptions that physical and virtual memory are scarce resources and that the abstraction should be hardware-independent. These assumptions fit well in the past, and the resulting functions still work but often lead to non-ideal efficiency. With the continuous progress of hardware and software, we should question those principles.

New non-volatile RAM (NVRAM) functions like conventional main memory and adds persistency to a so far volatile domain. With this change in data lifetime, memory management must become long-term oriented and keep its state across reboots. Persistent memory modules offer large capacities, boosting the trend of the steadily increasing amount of main memory. This evolution contradicts the assumption of scarcity and makes techniques like swapping and the costly, fine-grained management of the virtual and physical address space pointless.

To overcome the limitations of general-purpose processors, many users add specialized processing elements to a machine, e.g., graphics processing units (GPUs) and devices connected via remote direct memory access (RDMA). RDMA-capable network interfaces like *Ethernet* or *InfiniBand* can directly access main memory and provide connection speeds of 100 Gbit/s or more [Int22c; Inf22]. To utilize such a connection and to keep those amounts of data under control, we must prevent copying and reduce memory management overhead wherever possible. Hardware-independent solutions add complexity resulting in computation and memory overheads. Architecture-tailored implementations come without these overheads.

The ongoing growth in core counts increases parallelism on the level of independent threads. This parallelism, in combination with current implementations, leads to high contention on memory management data structures [KDI20; Cor13; ZGF21].

There is a demand for new memory abstractions that get rid of the old assumptions. Huge amounts of physical and virtual memory do not need to be managed as scarce resources, and hardware-specific implementations can provide the required performance for modern applications. Making memory objects self-contained can decouple the runtime state from the process lifetime and, therefore, prepare them for NVRAM. Special algorithms must ensure consistency at every time to survive crashes. These algorithms must be lock- and log-free because some devices cannot

[1] Listing A.1 shows an analysis of the Linux kernel 5.17 source code. Memory management source code is at least as big as 40% of the core kernel subsystems.

participate in complex locking protocols, and logging wears out write-limited NVRAM. Additionally, both affect performance negatively.

This thesis presents *morsels*, a new memory abstraction that sacrifices hardware independence in favor of performance, using self-contained data structures managed by crash-consistent algorithms to utilize persistent memory and provide direct compatibility with peripheral devices. The main idea is to define subtrees of the page tables as independent units not managed in the granularity of individual pages. The hardware dependence limits an implementation to a specific hardware platform, whereas the general concept can be applied to any architecture with paging support. Due to its practical relevance, the first implementation focuses on the x86-64 architecture.

Morsels are much more flexible than huge pages, which also aim to reduce management overhead by merging a whole page table into a single unit. Morsels can be sparsely populated, do not require large chunks of contiguous physical memory, and support larger sizes on most machines. By combining all those features into a single memory primitive, morsels might be the required improvement to prepare an OS's memory subsystem for current and future challenges.

This work is part of the *Parallel Persistency OS* (ParPerOS) project, which examines new abstractions for low-level memory management in heterogeneous memory systems. The morsel memory primitive is an early piece of work in the context of a modern memory subsystem.

This thesis is structured into six chapters. The current chapter introduced the topic and outlined the research question. Chapter 2 discusses the required fundamentals and relevant pieces of related work. It is followed by the requirement analysis and concept definition in Chapter 3 and the implementation in Chapter 4. Afterwards, Chapter 5 provides an analysis of the prototypic morsel implementation and an evaluation with respect to the requirements. Finally, Chapter 6 summarizes all contents and provides an outlook on future work.

FUNDAMENTALS 2

This chapter introduces fundamental topics required to understand the problem and the proposed solution of this thesis. Furthermore, it presents relevant pieces of related work regarding modern memory management.

Current memory abstractions define a virtual address space whose addresses are transparently translated into physical addresses by the hardware. The commonly used technique is paging, which maps chunks of the virtual address space to the physical address space based on page tables, a tree-like data structure managed by the OS. Intel's hardware implementation uses page tables with up to five levels, supported by the Linux kernel on the software side.

Heterogeneous memory systems combine different types of memory within a single machine. The abstraction provided by memory virtualization acts as a common interface between different kinds of processing elements and memory providers. One memory type can, for example, be NVRAM that functions like conventional main memory and adds persistency to a so far volatile domain. It enables applications and OSs to keep data across reboots and, in combination with crash-consistent programming models, even across system crashes. The specific algorithms depend on the guarantees the hardware platform makes regarding the different levels in the memory hierarchy, e.g., registers and caches.

The discussion of related work presents an overview of the current state of the art in memory management research. Examples show that existing implementations come with noticeable overheads when dealing with huge amounts of memory and fast storage devices. Furthermore, NVRAM adds new requirements to memory management not consequently addressed by current memory subsystems.

2.1 Virtual Memory Management

The most fundamental approach for memory management is to use physical memory addresses directly without any additional abstraction layer. As stated by Tanenbaum and Bos, this primitive approach has several drawbacks. The running application can access the whole memory, even the OS memory, without restrictions, and it is difficult to run multiple applications concurrently on the same machine because they must be aware of their memory locations to prevent collisions. [TB15].

Code that can be executed from any memory location is called position-independent code (PIC). The opposite is absolute code because it uses absolute memory addresses [Int84]. The use of PIC comes with a measurable overhead [Kli16]. The default case for code generation with GCC[2] is absolute code [GNU22b].

[2] GCC, the GNU Compiler Collection, is a collection of compilers, e.g., for C and C++, provided as free software [GNU22a].

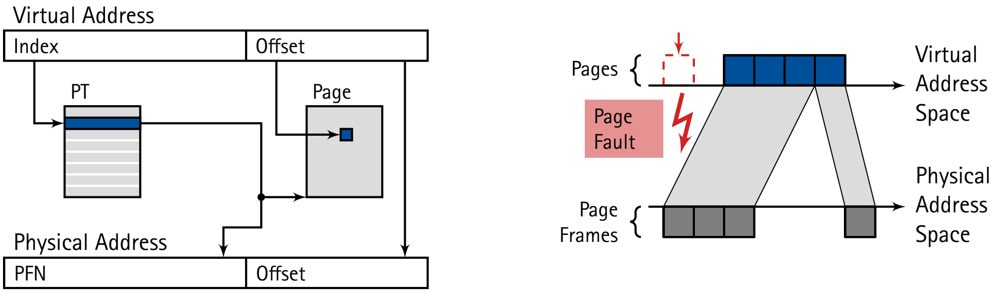
If two applications that use absolute code assume the same memory locations, they will collide. Memory virtualization provides the missing protection and relocation ability. One implementation is segmentation, which describes a virtual address space using multiple logical segments, one for each purpose (e.g., code, stack) [SGG13]. Segments are defined in a segment table through a base address and usually a limit and access rights. A specific memory location is then identified as a tuple of segment number and offset within the segment. Despite the simplicity, segmentation is not the preferred way of memory management in modern OSs.

Instead, they use the more flexible paging, which divides the virtual and physical address space into equally sized chunks [BL18]. The virtual address space is then defined as a mapping of virtual to physical chunks. Every application has its own virtual address space to achieve isolation. It is sparsely populated, containing only the mappings required for execution, and often larger than the available or even the supported amount of memory.

2.1.1 Paging

This section briefly introduces the paging concept based on the explanations of Tanenbaum and Bos [TB15]. The following sections discuss further improvements (see Section 2.1.2) and specific realizations (see Section 2.1.3 and Section 2.1.4).

Paging is the most common implementation of virtual memory. The virtual and the physical address space are divided into equally sized chunks called *pages* and *page frames*. The memory management unit (MMU) transparently translates virtual to physical addresses based on a so-called page table. The virtual address is split into an upper and a lower part. The upper part is the virtual page number (VPN) used as an index to select an entry from the page table containing the corresponding page frame number (PFN). The lower part of the virtual address is called offset, the relative address within a page. The physical address is assembled from the PFN as the upper part and the offset as the lower part. The translation process is depicted in Figure 2.1a.

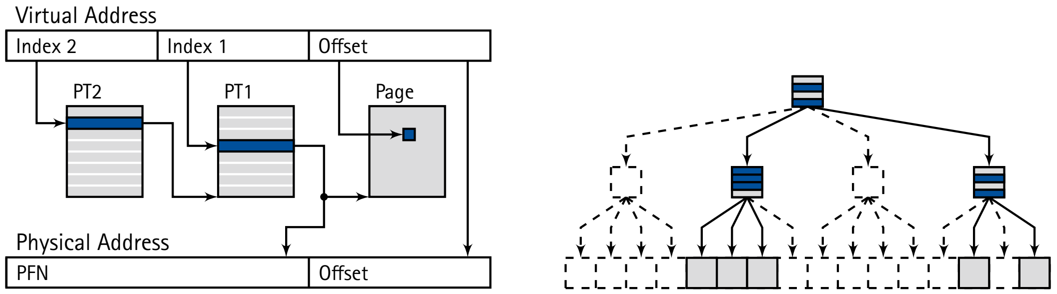


(a) Single-level paging: The input virtual address is split into an index and an offset. The index references an entry in the page table (PT). The physical address is a combination of the resolved page frame number (PFN) and the offset.

(b) The mapping enables great flexibility. Accesses to unmapped virtual addresses raise page faults that must be handled by the OS.

Figure 2.1 – Single-level paging and fault handling

Not every page table entry necessarily references a page frame. Entries can also be empty. If a program tries to access an unmapped virtual address, the MMU raises a page fault that must be handled by the OS (see Figure 2.1b). In practice, this simple approach based on a single page table is not viable for large virtual address spaces because it would require a giant page table. Therefore, the translation process is split into multiple steps using several small page tables, forming a tree-like data structure. The virtual address is not split into a single index plus an offset but multiple indices plus an offset. For example, a 32-bit virtual address can be divided into two 10-bit indices and a 12-bit offset, as shown in Figure 2.2a.



(a) Two level paging: The input virtual address is split into two indices plus an offset. The indices reference entries in the related page tables (PT1 and PT2). The physical address is a combination of the resolved PFN and the offset.

(b) Sparsely populated page table hierarchy: The multi-level paging approach can save much memory due to omitted lower-level page tables.

Figure 2.2 – Two level paging

The upper index (Index 2) selects an entry from the uppermost page table (PT2). The entry does not directly reference a physical page frame but a lower-level page table. The lower index (Index 1) is used for the lower-level page table (PT1) to get the referenced physical page frame. This two-level paging saves memory because the lower-level page table can be omitted if it does not contain any mapping. Figure 2.2b shows an example of possible savings. Due to the sparsely populated surface, two page tables can be omitted. Depending on the size of the virtual address space, resulting from the address width, more levels of page tables may be used. However, the effort required for address translation increases with every additional layer. Most current x86-64 central processing units (CPUs) support up to five paging levels (see Section 2.1.3). The following section addresses the problem of the time-consuming virtual to physical address translation requiring multiple expensive memory indirections.

2.1.2 Translation Lookaside Buffer

The paging-based implementation of virtual memory induces a considerable address translation overhead. This overhead significantly decreases performance if not accelerated because every memory-related instruction needs additional memory lookups for address translation.

Modern hardware uses a so-called translation lookaside buffer (TLB), which is part of the MMU, in order to speed up translation [ADAD18]. The TLB stores the VPN to PFN relation for a subset of recently used addresses. Because the TLB can only hold a few entries, it relies on spatial and temporal locality for achieving high hit rates. The fast path, the TLB contains the required entry, is called hit. The other case, the slow path, is called miss. TLB misses require a complete page table walk to obtain the PFN. Misses can be handled either by hardware or software. The latter is typical for reduced instruction set computer (RISC) architectures. As this thesis focuses on the x86 architecture, which is based on the complex instruction set computer (CISC) paradigm, we only consider hardware-managed TLBs.

Another important aspect is explicit synchronization with the page tables [Dre07]. TLB entries can become invalid due to page table modifications, e.g., due to a call to *munmap*[3], and, therefore, they must be explicitly removed. Because TLBs are a core-local resource and there is no automatic coherence protocol between cores, related entries in the TLBs of other cores must also be invalidated. This inter-core invalidation is called a *TLB shootdown* and relies on inter-processor interrupts [ATW20].

The analysis of different page sizes using simulated machine configurations shows a significant effect on overall application performance [WW09]. Smaller pages reduce internal fragmentation and, thereby, memory usage. Internal fragmentation names the wasted memory due to rounding allocation sizes up to an integer multiple of the page size. Larger pages increase TLB coverage, the amount of memory that can be accessed based on cached TLB entries. Higher TLB coverage means fewer TLB misses, resulting in an improved performance. Most modern architectures support multiple page sizes.

2.1.3 Intel 5-Level Paging

In November 2016, Intel initially released the white paper *5-Level Paging and 5-Level EPT* [Int17] introducing the planned extension of the effective address width used with the x86-64 architecture. Originally in 64-bit mode, four paging levels only utilized 48 address bits. The effective address width increases to 57 bits with an additional fifth paging level. The first CPU generation supporting the newly introduced 5-level paging was *Ice Lake* featuring the *Sunny Cove* microarchitecture [Cut18]. Corresponding server processors were released in April 2021 [Int21c].

The developer manual describes the actual implementation of 5-level paging [Int22b]. Regardless of its level, a page table neatly fits into a single 4 KiB page. Every table entry is 64-bit wide, so a single table contains $2^9 = 512$ entries. With a 9-bit index for every table in the hierarchy and a 12-bit offset (4 KiB pages), this is a total of 57-bit effective virtual address width. Figure 2.3 shows a simplified version of the address resolution routine with 5-level paging.

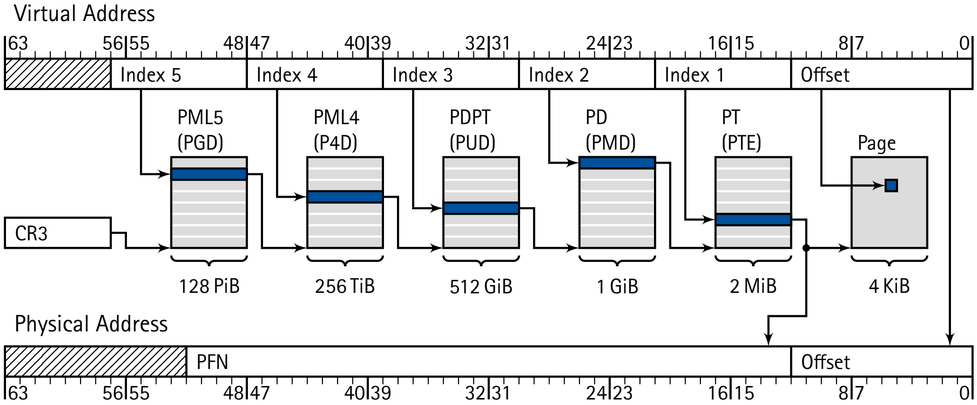


Figure 2.3 – Intel 5-level paging: The *CR3* register holds the address of the topmost paging data structure. Going through five levels of page tables the PFN is looked up. It is combined with the offset to form the physical address. The labels above the page tables show the Intel names and the Linux names in brackets. The labels below the page tables show the amount of virtual memory managed by a single page table. Shaded address parts indicate unused bits.

[3] *munmap* is a POSIX/UNIX system call that deletes a memory mapping (e.g., file mapping) from a virtual address space and lets further references to the specified virtual memory region generate access violations [Lin22a].

The *CR3* register holds the address of the topmost paging structure, the page map level 5 (PML5). The most significant index selects an entry in the PML5. The entry references the lower level paging structure, the page map level 4 (PML4). The second index selects an entry in the PML4. The entry references the page directory pointer table (PDPT). The procedure continues with the page directory (PD) and the page table (PT) to finally get the PFN of the mapped page frame. The least significant part of the virtual input address is the offset within the selected page. Together PFN and offset form the physical address. The labels below the page tables show the amount of virtual memory managed by a single page table. The complete virtual address space comprises 128 PiB of memory. Figure 2.4 shows the layout of a single page table entry.

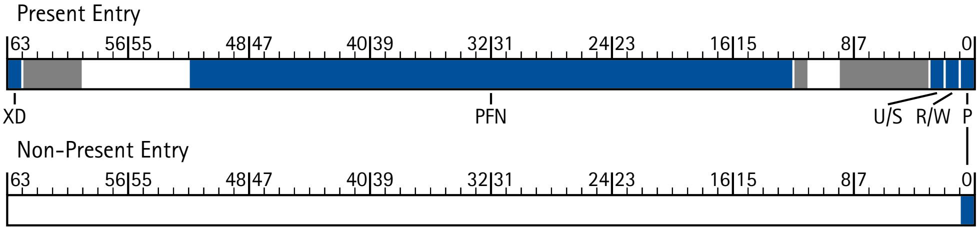


Figure 2.4 – Simplified x86-64 page table entry: The general layout is the same for all page table levels. Two different interpretations of the page table entry need to be distinguished (pages larger than the base page size are not considered here): one for present entries and one for non-present entries depending on the *present* bit *P*. Other important fields are the *execute-disable* flag *XD*, the PFN, the *user/supervisor* flag *U/S* determining access rights and the *read/write* flag *r/w* controlling write access. Not explicitly labeled segments (gray) are reserved or used for functions not further discussed here. Unfilled areas are free to use.

The basic page table entry layout is the same for all paging levels. In general, two cases need to be distinguished. An entry can refer to a physical page frame containing either a lower-level page table or user data, depending on the level. Also, an entry can be empty if the address range managed by this entry is not mapped at all. The *present* bit *P* indicates which case is active. If an entry is empty, the remaining bits are not interpreted and can be used for other purposes. For present entries, most bits have a specific meaning, but some are free to use. Predefined flags are, for example, the *XD* flag to disable code execution, the *U/S* flag to define the required privilege level for access (user/supervisor), and the *R/W* flag to allow write access. If flags are specified differently in different levels of the paging hierarchy, the most restrictive value applies.

Modern x86-64 CPUs support multiple page sizes. Alongside classic 4 KiB pages, they support 2 MiB pages, and some processors even support 1 GiB pages. For 2 MiB pages, a complete PT (PTE) table with its 512 entries is merged into a single page. The PD (PMD) entry refers directly to the page instead of a lower-level page table. The resulting page size is $512 \cdot 4\,\text{KiB} = 2\,\text{MiB}$. If supported, the same applies to the PD (PMD) table. The PDPT (PUD) entry then refers to a page with a size of $512 \cdot 2\,\text{MiB} = 1\,\text{GiB}$.

2.1.4 Linux Kernel

Before the introduction of Intel's 5-level paging, the Linux kernel only implemented four paging levels. The tables of the different levels were called page global directory (PGD), page upper directory (PUD), page middle directory (PMD), and page table entries (PTEs) from the uppermost to the lowest hierarchy level [BC05]. To support the fifth paging level, the developers introduced an additional layer called page level 4 directory (P4D) located between PGD and PUD [Shu16]. The kernel uses the page table data structures predefined by the hardware but with different names (see Figure 2.3). The 5-level paging support must be enabled in the static kernel configuration before building the kernel [Ker22a]. Builds with 5-level paging enabled can still be used on hardware supporting only four paging levels. The additional paging level P4D is then folded at runtime.

Modern architectures provide multiple page sizes. In Linux terminology, pages larger than the base page size are called *huge pages*. The first implementations only used huge pages for kernel address space mappings and made them available to user applications via a specialized interface called *hugetlbfs*. Bringing the benefits to all applications, the *transparent huge pages* feature tries to use huge pages in appropriate situations automatically [Cor11].

In the Linux kernel, every process is associated with an address space [BC05]. It denotes all valid addresses within the process context. The address space is a collection of memory regions called virtual memory areas (VMAs). A VMA is a linear range of addresses specified by a start and an end value. It is associated with several flags, e.g., access rights, and a set of functions implementing VMA-specific behavior, e.g., fault handling. A VMA can be imagined as a blueprint of virtual memory, not necessarily backed with physical memory. The kernel uses *demand paging* to better utilize the available system memory at the expense of a small runtime overhead. That means that the creation of a VMA does not allocate memory. Instead, accesses to the memory region specified by a VMA raise a page fault. The fault handler figures out the associated VMA based on the faulty virtual address and lazily allocates the requested page. The page is inserted into the page table, and the application can retry the failed instruction, which will now succeed. Figure 2.5 visualizes the VMA concept.

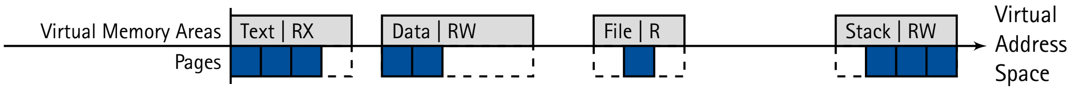


Figure 2.5 – A process address space defined through multiple virtual memory areas (VMAs): Each VMA has an individual set of flags and functions, e.g., access right (read, write, execute). Actual pages are typically mapped on demand.

A list of different VMAs for different purposes define the virtual address space of a process. Each VMA is associated with a set of flags and functions. For example, the VMA mapping the text segment allows execution but no write access. It spans a range of virtual memory but is typically populated on demand.

The kernel uses VMAs for all kinds of mappings [Lov10]. For example, there is one VMA for an executable's text section, one for the data section, and another for the stack. Also, mappings can be dynamically created or destroyed during runtime using the `mmap` and `munmap` POSIX/UNIX system calls [Lin22a], for example, to map files or shared memory or to allocate anonymous heap memory.

2.2 Heterogeneous Memory Systems

With the progress on the memory side as well as on the processing side, more and more different types of hardware elements need to be integrated into a computing system. Memory virtualization has been established as a common interface between processing elements and memory providers. Figure 2.6 illustrates this structure.

For example, memory providers can be conventional random-access memory (RAM), high-bandwidth memory (HBM) or NVRAM. Additional processing elements can be accelerators such as GPUs or RDMA-connected devices. The interface between both sides is a virtualization layer that

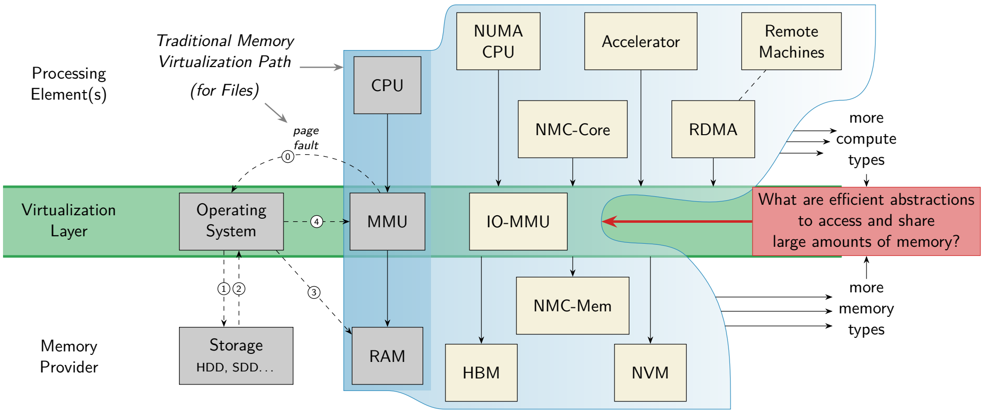


Figure 2.6 – Heterogeneous memory and different types of processing elements (figure from ParPerOS project description): Processing elements (upper side) can access different kinds of memory (lower side) through the (IO)MMU as common interface.

abstracts physical memory by providing a virtual address space. On the left, the figure shows the traditional procedure for file access requiring OS interaction. Section 2.2.1 explains the file access path in detail, before Section 2.2.2 describes the input-output MMU (IOMMU), a specialized MMU allowing memory virtualization for direct memory access (DMA) devices.

Another aspect when dealing with multiple processing elements is the underlying memory paradigm. For large multi-socket server systems, this is typically non-uniform memory access (NUMA), where all processors can access the whole memory, but performance characteristics differ depending on the memory location [Lam13]. A processor can access its local memory faster than other processors' memory.

2.2.1 Traditional File Access Path

Traditionally, files are stored on persistent storage devices, like hard disk drives (HDDs) or solid-state drives (SSDs). To speed up access, the OS transparently caches data in the so-called *page cache* located in RAM [BC05]. On read requests, the OS first refers to the page cache. On success, the read request can be satisfied directly from the main memory. On a miss, the OS loads the whole respective page from the underlying storage device and adds it to the page cache before answering the request. On write requests, the data in the page cache is manipulated, and the page is marked dirty. The OS defers the write-back of dirty pages because multiple consecutive writes to adjacent locations are a usual access pattern. The batching of multiple writes reduces the number of actual write-backs to disk. The page cache is used for read/write system calls as well as memory-mapped files.

Figure 2.6 shows the traditional file access path for memory-mapped files on the left. On first access to a file-backed mapping, the virtual memory location is not mapped to a physical page. Therefore, the MMU raises a page fault to be handled by the OS (step 0). The fault handler uses the faulty virtual address to determine the related VMA, which holds a reference to the file. Then it reads the respective data page from disk (steps 1 and 2) and inserts it into the page cache (step 3). The routine also modifies the page table (step 4) before it hands back the control to the user application. This example points out that file handling requires a lot of OS interaction and main memory for caching.

2.2.2 Input-Output Memory Management Unit

Nowadays, most input/output (I/O) devices use DMA for better performance. With DMA, devices can access memory without CPU interaction. Traditionally, those devices directly use physical addresses, but this has several drawbacks. One problem is the unrestricted access of devices to the whole memory without protection. Also, virtualized guests running on the host machine cannot use DMA because the guest-physical addresses used by the virtualized guests differ from the host-physical addresses used by DMA devices. Thus, guest OSs cannot localize the DMA buffers. Another problem is legacy devices that only support 32-bit addresses because they cannot utilize the whole address space of 64-bit host systems. The IOMMU addresses these problems at the expense of a small runtime overhead [ABYY10]. It acts as a translation layer between the devices' virtual address space and the host system's physical address space. Hierarchical page tables define the mapping between both address spaces. The approach is similar to the MMU, which virtualizes the memory for the CPU. The additional translation layer adds runtime overhead caused by page table walks. To accelerate translation, the IOMMU has an input-output translation lookaside buffer (IOTLB) similar to the TLB of the MMU. The management of IOMMU page tables induces additional overhead on the CPU side.

Implementations for the x86-64 architecture are Intel VT-d from Intel, and AMD-Vi from AMD [Int22a; AMD21]. Both implementations cannot be used interchangeably but follow similar principles. Intel reuses the existing paging data structures from the MMU. AMD calls its version a generalization of the existing MMU paging, where MMU page tables are compatible with the IOMMU. Both implementations can share page tables between MMU and IOMMU, at least for the lower levels. In contrast to the MMU, the IOMMU has no similar mechanism to handle page faults with a custom handler routine. Generally, faults are communication errors that possibly require a device reset. Accordingly, invalid accesses should be avoided.

2.3 Non-Volatile Memory

There is a large gap in the classic memory hierarchy between dynamic random-access memory (DRAM) and storage devices like SSDs and HDDs. On one side, volatile DRAM loses data on power loss. It provides low latencies and is directly accessible in byte-granularity, but technical restrictions and cost-effectiveness limit capacity. On the other side, storage drives are persistent and keep data across power cycles. They have large capacities, but accesses are slow and can be performed only in the granularity of blocks. New memory technologies for NVRAM are designed to fill this gap by combining the advantages of both sides. They are meant to deliver large capacities with fast, byte-granular access while maintaining persistency.

2.3.1 Intel Optane Persistent Memory

With its *Intel Optane Persistent Memory* modules, Intel introduced the first commercially available NVRAM in 2019 [Int19]. The modules are a drop-in replacement for conventional RAM, offer similar speeds, allow bytewise access hiding their internal block structure, and provide persistency on main memory level. The dual in-line memory modules (DIMMs) are designed to be used with second generation *Intel Xeon Scalable* processors. Module sizes of 128 GiB, 256 GiB, and 512 GiB, and up to six DIMMs per socket result in a maximum of 3 TiB per socket. In combination with up to 1.5 TiB of conventional DRAM, a single CPU can use 4.5 TiB of main memory. Applications scaling with the pure amount of memory can benefit as more data is directly available. In contrast to traditional DRAM, the Optane modules have limited write endurance, similar to SSDs. However, the limit specified in the datasheet is much higher in the magnitude of 10^6 write cycles.

In 2020, Intel presented the second generation of its Optane technology [Int21b]. While module sizes remained unchanged, bandwidth and endurance increased. In combination with the third generation *Intel Xeon Scalable* processors, up to eight DIMMs per socket can be used, lifting the limit to 4 TiB of NVRAM per CPU.

To get an idea of the practical performance, Izraelevitz et al. performed some benchmarks using 256 GiB modules of the first Optane generation [Izr+19]. It turned out that random read latency is about three times higher than the access latency of traditional DRAM (305 ns to 81 ns). Write latency is nearly the same (94 ns to 86 ns) if we assume that a write is finished when the data is effectively persistent (additional information about persistency properties follow). The measured bandwidth of a single module (read: 6.6 GB/s, write: 2.3 GB/s) roughly matches the values specified in the datasheet (read: 6.8 GB/s, write: 2.3 GB/s). Benchmarks with different block sizes reveal the internal block size of 256 B. Smaller accesses can reduce overall performance because the memory modules only work with complete blocks internally. An internal address indirection table maps incoming physical addresses to internal addresses, implementing a wear leveling mechanism. Even if performance is not on the level of traditional DRAM, especially latencies are much better compared to SSDs.

Not all levels in the memory hierarchy are persistent by default, even with NVRAM. Caches and the write pending queue (WPQ) are still volatile. To solve this issue, Intel introduced a feature called asynchronous DRAM refresh (ADR) [Int21a]. It ensures that pending writes in the WPQ get finished in case of a power loss. With ADR, both memory and the WPQ are part of the so-called power failure domain. Programs have to flush caches explicitly to make data persistent. Explicit flushing increases application complexity and can reduce performance. An improved version of ADR is extended asynchronous DRAM refresh (eADR). It extends the power failure domain to include the caches as well. With eADR explicit flushing is not required anymore. Persistency equals global visibility, simplifying the programming model significantly. Nevertheless, special crash-consistent algorithms must be used to ensure recoverability.

2.3.2 Crash-Consistent Algorithms

Software using persistent memory must implement failure atomicity to allow the recovery to a consistent state after a crash [Sca20]. Different mechanisms can achieve failure atomicity depending on the guarantees a hardware platform provides. This thesis focuses only on systems with eADR because it simplifies programming significantly. With eADR, the CPU caches seem to also be persistent from the software point of view. As a result, written data that is globally visible for other cores is also persistent. This fact allows the use of conventional techniques for lock-free, thread-safe programming to achieve crash consistency.

Typically, logging is used to ensure consistency for storage data structures. Changes are written to a log first before they are applied. In the case of a crash, the log provides enough information to restore a consistent state.

Locking is the primarily used tool to ensure the consistency of memory data structures in multi-threading environments. Whereas it synchronizes accesses in concurrent environments and, thus, makes actions atomic from the viewpoint of a thread, it does not provide failure atomicity. In concurrent programming, a taken lock indicates that one thread has exclusive access to a shared resource. If another thread also needs access, it has to wait until the first thread finishes. The problem with system crashes is that a previously taken lock is still taken after a crash if placed in persistent memory, but the runtime state of the threads is lost, and as a result, no thread will ever release this lock. Manually clearing the lock as part of a recovery procedure is also no solution because it would expose an inconsistent data state.

In the area of databases, the mechanism of transactions [Lan22] has been developed to guarantee persistency and failure atomicity. A transaction combines multiple operations into a single atomic unit. If it fulfills the *ACID* properties, the system is reliable. ACID stands for atomicity, consistency, isolation, and durability. *Atomicity* means that a transaction either completely finishes or does not happen at all. It moves the system from one valid state to another valid state, implying *consistency* at any time. *Isolation* denotes that multiple concurrent transactions do not corrupt the system state. The resulting state should be the same as for sequentially executed transactions. The last property, *durability*, means that external influences, e.g., power failures, must not affect the system state. Transactions can be implemented based on logging or atomic pointer updates. An atomic pointer update activates prepared changes by atomically overriding a single pointer. Alternatives to transactions are copy-on-write (COW) and versioning, where the basic concept is equal for both variants. The original object is copied, the copy gets modified, and then the copy replaces the original. When implementing atomic updates, we have to pay special attention to memory reordering due to optimizing compilers or out-of-order execution of superscalar processors [McK07]. Memory reordering can break failure atomicity when changes get activated before they are entirely applied. Memory barriers can solve this problem.

2.4 Related Work

Memory management and persistent memory are broad fields of research. In this thesis, we distinguish five subsections that directly address the same question of finding appropriate abstractions for persistent memory or are related to the morsel implementation. The first viewed topic is sharing of page tables between multiple address spaces to save memory and improve performance (see Section 2.4.1). The second subsection discusses the drawbacks of general-purpose OS interfaces and suggests application-specific tailoring instead (see Section 2.4.2). This is followed by an evaluation of classic file systems on top of NVRAM (see Section 2.4.3). Another approach to persistent memory support is to build an entirely new OS specially designed for NVRAM to eliminate all legacy functions that hold back the full potential (see Section 2.4.4). The last subsection is about a persistent memory allocator that allocates page frames in a persistent and crash-consistent manner, allowing the construction of persistent data objects (see Section 2.4.5).

2.4.1 Page Table Sharing

The Linux memory subsystem puts much effort into page sharing to reduce memory usage and improve performance (e.g., lazy copy when forking). McCracken discussed whether the sharing should be extended to page tables [McC06]. It turns out that using the same concept for page tables as for pages, including COW, adds a noticeable runtime overhead to applications with a small memory footprint. COW in the context of page sharing means that pages are mapped read-only to different locations. On write access, the MMU raises a page fault, and the fault handler duplicates the page. Then the application actually writes to the created copy. Because most applications have a small memory footprint, the author discards the first approach. An improved version only shares truly shareable mappings, e.g., shared, file-backed mappings, avoiding the need for COW.

For 64-bit systems, the author suggests sharing page tables on PTE and PMD levels. By default, the kernel deletes page tables when the corresponding memory regions get unmapped. Shared page tables must not be deleted if they are also mapped elsewhere. To realize that, the kernel has to count the number of references. Another difference is that an address-space-wide page table lock cannot protect shared page tables because they can be part of multiple address spaces. A solution is to extend the existing *split page table lock* to all shared page tables.

Originally an address-space-wide lock was used to protect the page tables. It turned out that this global locking scales poorly in multi-threading environments. Fine-grained locking for the lower page table levels has been introduced to improve scalability [Ker22c]. First, only PTE tables got separate locks. Later this approach was extended to the PMD level to improve scalability when using huge pages [Cor13]. This multi-lock approach is called *split page table lock*. The upper page tables still rely on the global lock. Figure 2.7 visualizes the sharing concept.

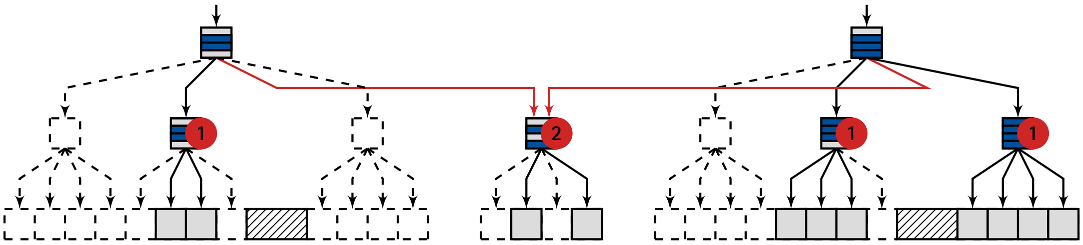


Figure 2.7 – Page table sharing: Shared page tables do not uniquely belong to a single address space. Therefore, the common mechanisms of page table management and locking using an address-space-wide lock do not work. Reference counting allows distinction between shared and private page tables.

In this example, two address spaces share a PTE table. All other page tables are private. Every page table has an associated reference count to distinguish between shared and private page tables.

The improved implementation does not induce a performance penalty to applications with little memory usage. Applications using large amounts of shared memory, e.g., database applications, show a significant performance gain. Page table sharing has advantages over huge pages, which conceptually address similar problems. Huge pages require large contiguous slices of physical memory, whereas shared page tables still use the default page size. Also, the page table sharing approach can be combined with huge pages to extend the size of a shared unit, e.g., use sharing on the PMD level while replacing PTE tables with huge pages. One drawback of the current implementation is that it only shares mappings to the same virtual address. The only hard requirement is that the mappings have a common alignment. Without modifying the page table, the mapping location can be adopted in the granularity of the uppermost shared page table.

About 14 years later, Zhao, Gong, and Fonseca revisit the page table sharing approach but from a different point of view [ZGF21]. The latency of the fork system call increases linearly with the amount of memory the calling application uses. The implementation spends most time copying page tables to set up the virtual memory of the forked child. The most expensive part is the reference counting in page granularity because, for every shared page, the fork routine has to increment the counter in the corresponding management data structure. The use of huge pages can drastically reduce the number of pages. However, on the downside, it will increase COW latency because the amount of memory that must be copied at once increases drastically. The increased COW latency can also affect the overall application performance negatively. Another way to decrease the number of shared elements is to implement COW on the PTE level instead of the page level. The new fork implementation clears the write bit of the PMD entry for every shared PTE table to achieve the desired behavior. On write access, the fault handler duplicates the PTE table, and the COW falls back to the default implementation in page granularity for this specific PTE table. Sharing page tables between processes makes it necessary to adapt kernel routines because shared tables can exceed process lifetime.

Lifting the COW implementation from page to PTE level improves the performance of fork drastically, e.g., by factor 65 for a 1 GiB memory area. On the other side, it can increase COW latency when a shared PTE table must be resolved. However, we can see notable performance gains and a decrease in worst-case latency in real-world applications that make heavy use of fork, e.g., for snapshotting. In contrast to the first approach by McCracken the sharing only applies to read-only sections that get resolved on write access. The performance gains in most situations justify the negligible increase of COW latency that can occur in some situations due to PTE table duplication.

2.4.2 Application-Specific OS Interfaces

Leis et al. discuss the approach of tailoring an OS's memory subsystem interface to the needs of a specific application [Lei+23]. In this case, it focuses on database management systems (DBMSs). DBMSs cache pages from a storage device in main memory for better performance. With the `mmap` system call, POSIX-compliant OSs support this feature out of the box, but it is too slow to utilize modern SSDs. To circumvent this issue, most DBMSs implement their own page caching mechanism. The idea of the paper is to implement a virtual-memory-based caching mechanism in user space supported by a custom memory subsystem interface that gives explicit control to the user application.

One problem with the current implementation in Linux is that user space applications have no control over page faulting and page eviction. The OS decides when it allocates pages and reads the respective data from the storage device. The same applies to write-back and freeing. As a result, accesses to a file-backed virtual memory mapping are entirely unpredictable. In the best case, the accessed virtual page is mapped to a physical page frame already in the CPU caches. In the worst case, the OS must allocate a new physical page frame and read the corresponding data from the storage device. The access times differ in the size of multiple magnitudes. In the paper, this problem is solved by performing explicit reads and writes in combination with an anonymous mapping. Removing the storage bottleneck reveals even more issues that cannot be solved in user space:

1. The frequent mapping and unmapping of pages lead to a high number of TLB shootdowns.
2. The Linux page allocator does not scale well with an increasing number of threads as it uses centralized data structures.
3. Also, it fills every page with zeros for security reasons.
4. Mapping/unmapping pages involves page table manipulation, guarded through a lock.

To solve those kernel space issues, the author introduces a new interface to the memory subsystem, called *exmap*, giving explicit control of page allocation and eviction to the user space application. Batching multiple operations reduces the number of TLB shootdowns as multiple invalidations can be performed with a single interprocessor interrupt (issue 1). Exmap preallocates memory during its initialization phase to remove the page allocator from the hot path (issue 2). This memory is exclusive to a single process. Therefore, it does not need to be zeroed every time before mapping it into the virtual address space (issue 3). For page table manipulation without locking, exmap suggests the use of atomic compare-and-swap (CAS) instructions. Because page table entries have machine-word size on most machines, they can be manipulated with a single atomic instruction (issue 4).

The combination of virtual-memory-based caching in the user space and the optimized memory subsystem interface outperforms all other tested competitors in terms of I/O performance. The implementation is fast enough to move the bottleneck from the virtual memory subsystem to the storage device. In conclusion, we can say that replacing a general-purpose implementation with an application-specific, tailored implementation can improve performance significantly. However, it comes at the cost of an additional interface and increased application complexity because the allocation and eviction management moved from kernel to user space.

2.4.3 File Systems on Non-Volatile Memory

There are different approaches to the integration of NVRAM into current systems on the software side. One is to build a classic file system on top, as described by Corbet [Cor14]. The problem is that these file systems are designed to work with block devices, whose data can only be accessed through the page cache (see Section 2.2.1). This caching is necessary for block devices because they provide data only in the granularity of larger blocks. NVRAM is directly mappable into the virtual address space, and thus, the page cache adds unnecessary overhead. The direct access (DAX) feature addresses this issue by enabling data on compatible devices to skip the page cache and be directly mapped. DAX is compatible with existing file systems, like ext4. However, even if DAX fixes the biggest issue, these file systems are not designed to work with directly accessible, byte-addressable memory. Tailored file systems would be required to utilize the full potential of NVRAM. It seems obvious to look at current RAM disk implementations that build file systems directly in the main memory. At a second glance, they are not prepared to deal with persistency. They start with a fresh initial state on every system startup and, therefore, do not have to worry about locating files or using a hardware- and kernel-independent implementation.

Different works target the design of specialized file systems for NVRAM, for example, wrJFS [Che+18], DurableFS [KBS19], and iNVMFS [WKC22]. They mainly take advantage of the byte-addressability to achieve lower write amplification of journaling file systems to reduce overhead and power consumption. Those file systems are not further discussed here because building a file system is beyond the scope of this thesis.

2.4.4 Twizzler

Twizzler by Bittman et al. is an OS designed to fully utilize the potential of NVRAM [Bit+17; Bit+20]. The main idea is to build everything around persistent data objects. Therefore, they call it a data-centric model. An object is a virtually contiguous region of memory identified through a 128-bit ID, currently limited to 1 GiB of size in the prototypic implementation. The size is no architectural limit but a compromise that will be explained later. The preferred object access method is memory mapping to achieve the best performance because it removes the OS from the access path. Twizzler heavily relies on nested paging, an enhanced virtualization feature of the MMU, to realize its two levels of mapping. The MMU first maps the object virtual address space to the object physical address space, which is mapped to the actual physical memory in a second step. The MMU also enforces the specific access rights of a powerful rights management system. Twizzler distinguishes between protection, which applications define themselves, and permissions, which are defined by security contexts. The rights management allows fine-grained control on the level of objects.

One major challenge is the management of cross-object pointers, which outlive the processes dealing with it. Because the object IDs are too large to directly use them for addressing, Twizzler uses indirect references. Every object contains a foreign object table (FOT) which maps an index to an object ID for every external reference. A pointer is then a tuple of an index and an offset combined into a single 64-bit value. The sizes of both components are a trade-off between the number of external references and the maximum object size. The advantage is that the pointer size does not increase compared to native address pointers. Before accessing memory, the application must resolve such a reference to an actual address via the FOT. The indirect referencing decouples addressing from mapping locations and, therefore, allows the mapping of objects to arbitrary locations without invalidating incoming references. On the downside, the pointer resolution increases access latency. Hardware support for pointer resolution could decrease this latency. The concepts employed in Twizzler seem promising but are incompatible with existing operating systems, e.g., the proposed cross-object pointers are incompatible with plain pointers.

2.4.5 Page Allocator for Non-Volatile Memory

The implementation of persistent data objects needs a persistent page allocator. Persistent in the context of a page allocator means that the allocator must provide the same guarantees as the data objects based on it. The allocator must keep its state, the allocated and free pages, across power cycles, and it must be able to tolerate crashes. Besides these NVRAM-specific properties, general requirements, such as good performance and high multi-core scalability, apply. Wrenger proposes a log- and lock-free allocator specifically designed for NVRAM [Wre22]. Locks are unsuitable as they do not scale well with high core counts and increase recovery complexity. Logging is not optimal because current NVRAM has limited write endurance. The implementation avoids locks and logs by using atomic CAS operations for critical updates. Another problem of allocators is fragmentation which is even worse for non-volatile memory (NVM) because allocations outlive reboots. Allocators for volatile memory can start with a fresh initial state at every boot, but allocators for NVRAM cannot. Therefore, the allocator must use a good heuristic for its allocations to avoid fragmentation. The final implementation even outperforms the Linux kernel allocator in some disciplines.

The allocator cannot avoid losing pages completely if a crash happens right before changes can be persisted to memory. To prevent page losses, the allocator requires the support of the persistent data structures that use the allocated pages. If the used pages can be determined from the data structures, the page allocator can use this information to fix its internal state after a crash. It would not even be necessary for the page allocator to persistently store its state as it can be completely recovered from this metadata.

2.5 Summary

Current OSs virtualize memory using paging. The virtual and the physical address space are divided into equally sized chunks called pages and page frames. The page tables, a tree-like, hierarchical data structure, define the mapping of pages to page frames. The MMU transparently resolves virtual to physical addresses on every memory access. The TLB, which is part of the MMU, caches recently used relations to speed up translation. Modern x86-64 CPUs use up to five paging levels to reduce the required memory for sparsely populated virtual address spaces. The hardware-defined paging data structures provide a few free bits in the table entries, which the OS can use for its purposes. In modern operating systems, such as Linux, there is an additional abstraction layer on top of paging. The Linux kernel associates every process with an address space, a collection of VMAs defining the virtual memory layout. The VMAs can be seen as a blueprint of the actual page tables, which implement the address space on the hardware level.

Modern computing systems combine different processing elements and memory technologies within a single machine, connected by the (IO)MMU as a virtualization layer. For traditional file access, the OS manages copies of the actual data blocks in the page cache. This procedure adds computing and memory overheads which are intolerable for byte-addressable NVRAM. NVRAM is nearly as fast as DRAM, directly accessible by the processor and other processing elements, but provides persistency as classic storage devices do. A commercially available NVRAM technology is Intel Optane. The second Optane generation even guarantees persistency for data in the CPU caches, simplifying programming significantly. Nevertheless, specific algorithms must provide failure atomicity to allow recovery to a consistent state after a crash.

The idea of utilizing hardware data structures and features to optimize OS efficiency is not new. Sharing (lower-level) page tables between address spaces saves memory and reduces the latency of the fork system call. Challenges are the required reference counting of page tables and the synchronization of page table modifications because an address-space-wide lock cannot be used. The current memory subsystem is already a bottleneck in practical applications because it has not been designed to deal with low-latency storage devices such as SSDs. A specialized interface tailored to the application's needs can improve performance at the expense of increased complexity.

One idea for the use of NVRAM is to build a classic file system on top. The problem is that those file systems are designed to work with block devices and cannot utilize the full potential of directly byte-addressable NVRAM. Another approach is to build an entirely new OS for persistent memory, e.g., Twizzler. It manages everything as a data object with a unique ID. An indirect referencing allows cross-object pointers independent of the actual mapping locations. Persistent data objects require a persistent page allocator providing allocations that survive reboots. To fully avoid page losses due to crashes, persistent data structures must be aware of their used pages, so the allocator can use this information to fix its internal state.

ARCHITECTURE 3

In this chapter, we conceptually design the morsel, a new memory primitive, and all related operations. Sacrificing hardware independence limits a morsel implementation to a specific hardware platform. Nevertheless, the general concept can be applied to every architecture with paging support. This thesis focuses on the x86-64 architecture due to its practical relevance and well-defined structure. The requirements are based on the research question and the knowledge gained from the previous chapter. Besides the requirement definition, we also discuss the scope of this work.

A morsel is a subtree of the paging data structures identified through its entry in the parent page table. The root level in the tree data structure defines the virtual size. The basic operations for morsel handling are creation, (un)mapping, and destruction. The creation routine allocates only the morsel root element to reduce latency and save memory. Utilizing the fault handler, the actual population of morsels with pages happens lazily on demand. Atomic instructions achieve thread safety and crash consistency, as well as good performance, without locks and logs.

Mapping and unmapping require only the manipulation of a single page table entry of the related address space, making it very efficient. The destruction operation frees morsel pages and page tables in a way that limits the damage to persistent morsels in the case of a system crash during execution.

The morsel data structures do not provide space for additional metadata, e.g., a fixed mapping address. The nesting of two morsels solves this problem by using a dedicated morsel to store metadata.

3.1 Design Goals and Scope

This thesis covers the development of morsels as independent, self-contained memory objects using existing hardware-specific data structures to overcome the limitations of current memory primitives. Morsels must be sharable between multiple processes and address spaces (see Section 2.4.1). Therefore, the data structures need to be thread-safe to allow concurrent access of multiple processors and other processing elements. Furthermore, they should scale well across many simultaneously running threads because the parallelism of new processors rises steadily due to an increasing number of cores. An additional requirement is inherent crash consistency to be ready for persistent memory technologies. Reasonable NVRAM support includes that the implementation needs to limit the number of lost pages in the case of a crash. Combined with a compatible allocator, it should be possible to prevent page losses completely (see Section 2.4.5). Also, morsels must not wear out memory with limited write endurance. The previously mentioned requirements imply the prevention of locking and logging for synchronization and failure atomicity (see Section 2.3.2 and Section 2.3.1).

The underlying morsel data structure is a page table subtree. Page tables form a hierarchical data structure with up to five levels (see Section 2.1.3). For a variety of different use cases, morsels have to support multiple sizes according to the levels of the paging hierarchy. This concept depends on the abstraction layer provided by the MMU and IOMMU to allow access from all devices (see Section 2.2.2).

The traditional file access path heavily relies on OS interaction (see Section 2.2.1). To overcome bottlenecks (see Section 2.4.2) and avoid unnecessary copies (see Section 2.4.3), the OS should not be on the access path. Another interesting aspect of in-memory files is byte-addressability, suggesting the use of pointer-based data structures without the need for serialization and deserialization (see Section 2.4.4). For security reasons, protection in object granularity allows the enforcement of enhanced access rights policies (see Section 2.4.4).

As this is the first proof of concept design, the implementation includes only the basic functionality: creation, (un)mapping, destruction, and lazy population. We can safely ignore swapping because physical memory is no scarce resource nowadays.

Even with the hardware's support of multiple page sizes, the prototypic morsels only use the basic page size of 4 KiB to limit complexity (see Section 2.1.3). The same applies to the cross-object pointers provided by Twizzler. They are a promising concept but are beyond the scope of this work (see Section 2.4.4). A similar aspect is upper-level management of known morsels and access rights. The first implementation expects trusted applications that do management themselves. For the sake of simplicity, crash consistency will only be provided for systems with an extended persistency domain, including the CPU caches (see Section 2.3.2). Allocation of physical morsel page frames would be a standalone topic, so this thesis assumes the existence of a suitable page allocator (see Section 2.4.5). The morsel design should respect the existence of the IOMMU, but the first implementation does not specially target DMA devices (see Section 2.2.2). All mentioned restrictions may be addressed in future work. The primary requirements, which are also the main contributions of this thesis, can be summarized as follows:

Requirment 1 *Morsels are self-contained, sharable memory objects using existing paging data structures.*

Requirment 2 *They are inherently crash-consistent, thread-safe, lock- and log-free.*

Requirment 3 *Sparse population in combination with lazy allocation achieves low creation latency and memory efficiency.*

Requirment 4 *Direct access without OS interaction achieves reasonable runtime efficiency.*

These requirements are the basis of the morsel concept and will be used later to evaluate the prototypic implementation (see Section 5.6). The following section starts with the definition of basic morsel properties.

3.2 Basic Concept

The general idea behind the morsel concept is to build memory objects based on the existing hardware-specific paging data structures. It has the advantage that the MMU can directly interpret those data structures without any additional conversion required and, therefore, without OS

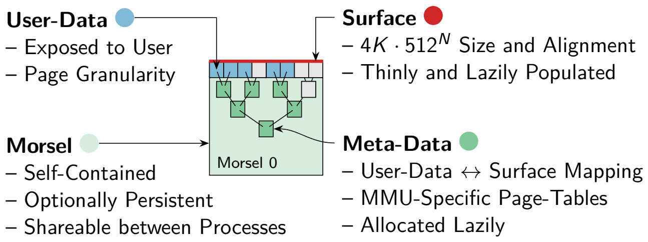

User-Data
– Exposed to User
– Page Granularity
Surface
– $4K \cdot 512^N$ Size and Alignment
– Thinly and Lazily Populated
Morsel
– Self-Contained
– Optionally Persistent
– Shareable between Processes
Morsel 0
Meta-Data
– User-Data ↔ Surface Mapping
– MMU-Specific Page-Tables
– Allocated Lazily

Figure 3.1 – Basic morsel structure as defined in the ParPerOS project proposals (figure from ParPerOS project description)

interaction. No conversion means no computational overhead or possible inconsistency between different representations. Figure 3.1 visualizes the described concept.

A morsel consists of pages and page tables. The pages contain the user data, and the page tables define the arrangement of those pages. Thus, the page tables add a small memory overhead to the total physical size. Morsels can be optionally persistent if pages and page tables are placed in NVRAM.

A consequence of the hierarchical paging structure is the possibility of identifying a complete page table subtree by its entry in the parent page table. We call this entry *morsel ID*. It is effectively a combination of a PFN and some flags. The morsel ID refers to a specific morsel but is not part of the morsel itself. The *self-contained* property means that no additional information is required to interpret a morsel memory object, simplifying morsel placement in persistent memory. A morsel is transferable between processes by sharing the morsel ID, granting access to the same memory object. It does not matter how the morsel ID is shared. One possibility is to use UNIX pipes[4].

Mapping a morsel into a process's address space exposes the user data to the application. A morsel can also be mapped into multiple address spaces at the same time to share data between processes. Mapping, in this case, means inserting (a slightly modified version of) the morsel ID at a vacant position into an existing page table of the corresponding level. Section 3.3.2 discusses the mapping process in more detail. The concept, on top of the existing hardware data structures, limits the virtual size of a morsel to specific values. Because a morsel is defined through a page table entry, it covers the complete subtree, also restricting the alignment of mappings. Possible mapping positions are the discrete entries in a page table. As a result, the mapping address must be naturally aligned, meaning the address is an integer multiple of the virtual morsel size. Table 3.1 provides an overview of the supported virtual sizes.

The smallest possible morsel is a single page. It is defined to be a morsel of order 0, and it is fully identified through its entry in the PTE table. The next larger morsel of order 1 covers a whole PTE table referencing 512 pages. Its entry in the PMD table identifies it. On systems with 5-level paging support, this can be continued up to a morsel of order 4, covering a whole P4D table. The virtual morsel size grows exponentially with the order N. It can be calculated as $4\,\text{KiB} \cdot 512^N$. Every additional layer, from page level upwards, takes 512 elements of the previous level and combines them into a single indivisible object. In addition to the user data, a morsel needs some memory for the page tables. The *Max. Overhead* column shows the overhead for a fully populated morsel. Relatively to the virtual size, it is about 0.2% independent of the morsel order.

Virtual size does not denote the actual physical size. It rather defines a blueprint that can be populated with pages in page granularity. Especially for higher-order morsels, we can expect that they will be sparsely populated. The page fault handler can allocate and insert additional page

[4] In Linux, a pipe is a unidirectional data channel between two processes [Lin22b].

Order	Element	ID	Virt. Size	Max. Overhead	Notes
0	Page	PTE Entry	4 KiB	0 B	
1	PTE Table	PMD Entry	2 MiB	4 KiB	
2	PMD Table	PUD Entry	1 GiB	~2 MiB	
3	PUD Table	P4D Entry	512 GiB	~1 GiB	With 5-lvl paging
3	PUD Table	PGD Entry	512 GiB	~1 GiB	With 4-lvl paging (no P4D)
4	P4D Table	PGD Entry	256 TiB	~513 GiB	Requires 5-lvl paging

Table 3.1 – Supported virtual morsel sizes: The virtual size grows exponentially with the morsel order N following the formula $4\,\text{KiB} \cdot 512^N$. Morsels of order 4 can only be used in combination with 5-lvl paging. With 4-lvl paging a morsel of order 3 is identified through its entry in the PGD table because the P4D level does not exist. The *Max. Overhead* column shows the memory needs for the page tables of a fully populated morsel.

frames on demand without the need for explicit management by the user. Explicit population is a separate feature that can speed up the population by batching multiple page allocations into a single system call. As described in Section 3.1 this is not covered in this thesis.

3.3 Primary Operations

A morsel must be created before it can be used, so creation is the first required operation. Using a morsel means reading and writing data. For efficient access, morsels employ the memory mapping paradigm. Therefore, (un-)mapping is the second required operation. The used memory should be reclaimed at the end of a morsel's life span. Thus, destruction is the third required operation. The following subsections discuss these three operations: creation, (un-)mapping, and destruction. It focuses on the general implementation as well as possible problems, especially crash consistency.

3.3.1 Creation

Morsels can have different virtual sizes depending on the depth of the paging subtree. Therefore, the creation requires the specification of an order that determines this depth. For execution speed and memory efficiency, only the uppermost element, the morsel root, is allocated. Figure 3.2 shows the minimal morsels for different orders.

For example, a morsel of order 0 consists of a single page. A morsel of the next higher order 1 gets an empty PTE Table. For a morsel of order 4, an empty P4D table is allocated. Every created morsel is identified through its morsel ID, which is its entry in the parent page table. For example, a newly created morsel of order 2 consists of an empty PMD table and is identified through a PUD entry. If a morsel ID of a prior unknown morsel is given, there must be a way to get the morsel order. Otherwise, it would not be possible to interpret the morsel correctly because it is unknown whether it should be interpreted as a PTE entry, a PMD entry, or something else. This problem becomes even more critical when dealing with mapping in the following subsection. Suppose a morsel gets inserted into the paging hierarchy at the wrong level. In that case, it either exposes page table data to the user application or, even worse, the MMU tries to interpret user data as page table entries.

As stated in Section 2.1.3, the page table entries of different levels in the paging hierarchy are very similar. As a result, it is impossible to determine the order directly from the morsel ID. Additionally, the page table entries have a few free bits that are not interpreted by the hardware and, therefore, can be used for other purposes by the OS. Encoding five different orders (0-4)

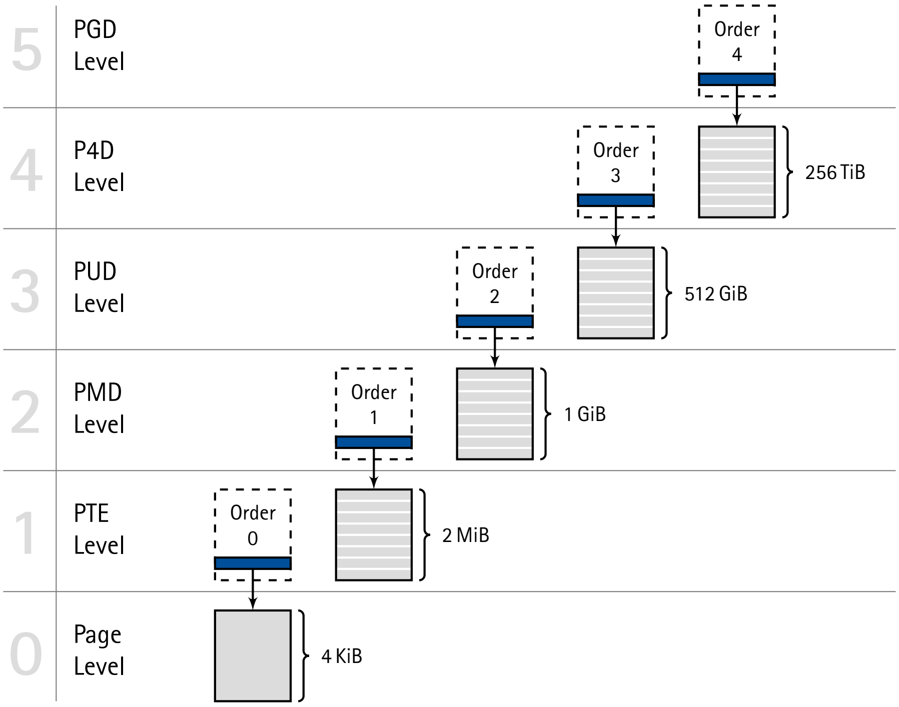


Figure 3.2 – Minimal morsels as they are directly after creation for different orders: The creation process allocates a single element only. For order 0 morsels, this is the user data page. For higher-order morsels, this is an empty page table. The morsel ID, which is the page table entry in the parent page table, points to the allocated page. The values next to the morsel page tables indicate the virtual sizes.

requires 3 bits. The encoding starts at 1 (binary 001_2) for order 0 to separate default page table entries from morsel IDs. The three-bit-wide order tag is placed into an empty part of the morsel ID.

Access rights are not considered at this point because they should be applied per mapping and not per morsel. Another aspect is that morsels are not globally registered. Such higher-level management would increase the complexity because many aspects, such as performance, coherence in multi-threading environments, and crash consistency, must be considered. Higher-level morsel management functions are a possible improvement for future versions.

3.3.2 Mapping

Mapping memory into a process's address space allows direct access by the user application without the need for OS interaction. In the case of morsels, mapping means inserting the morsel ID into an address space's page table of the corresponding level. In the first step, the mapping routine must determine the morsel order, which specifies the page table level on which the morsel ID should be inserted. According to the `mmap` POSIX/UNIX system call [Lin22a], it should be possible to specify a (naturally aligned) mapping address manually or to let the system select a suitable address automatically. After successfully validating the mapping address, the mapping routine can insert the morsel ID into an existing page table at the corresponding position. Possibly the required parent page table does not exist. In this case, the regular routines for page table allocation down to the required level can be used because this is not morsel specific, and the involved page tables are not shared. Figure 3.3 illustrates the mapping routine.

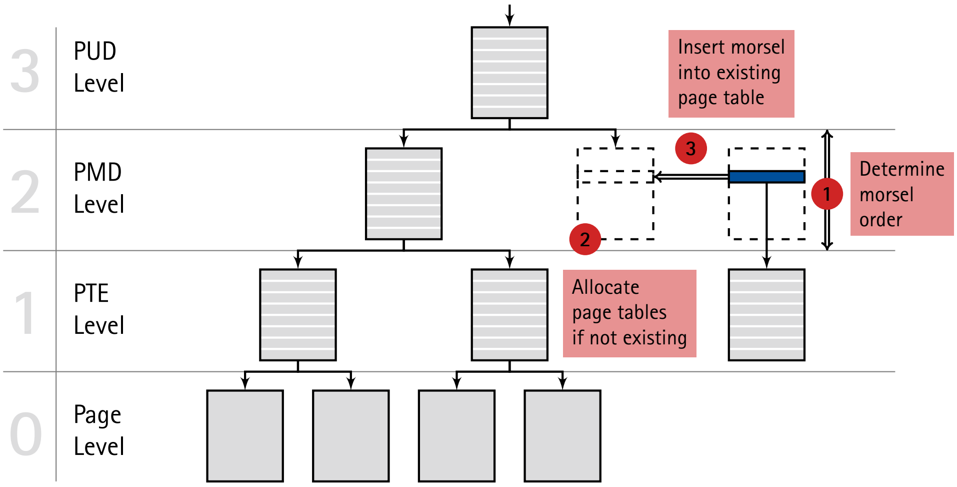

3
PUD Level
2
PMD Level
1
PTE Level
0
Page Level
Insert morsel into existing page table
Determine morsel order
Allocate page tables if not existing
1
2
3

Figure 3.3 – Mapping a morsel into a process's address space: 1. Determine the morsel order, 2. Allocate parent page tables if necessary, 3. Insert the morsel ID into an existing page table.

To realize specific access rights (read, write, execute), the mapping routine must adapt the flag bits of the morsel ID before writing the value into the parent page table. As stated in Section 2.1.3, the most restrictive value applies if rights are specified differently on different levels in the paging hierarchy. To realize access rights per mapping, only the uppermost page table entry, the morsel ID, restricts access. The lower entries are permissive in order to achieve the desired behavior. This approach has the advantage that the morsel data structures are unaffected. Therefore, the mapping is inherently thread-safe. It also allows the mapping of the same morsel into multiple address spaces at the same time with different access rights.

The inverse operation, unmapping, is straightforward. The morsel ID has to be removed from the address space's page tables. Overriding the entry with a non-present entry blanks out the complete subtree. The morsel data structures themselves are unaffected. Possibly cached translations must be removed from the TLB to apply the previously made address space changes.

For mapping as well as unmapping, there are no concerns regarding crash consistency. Because the morsel data structures themselves are not involved, they cannot break.

3.3.3 Destruction

The destruction operation deletes a morsel that is no longer needed to give back the used page frames to the page allocator. Assuming that the system can crash at any time during the destruction process, the number of lost pages should be minimized, and the morsel must never be in an invalid state. These requirements are essential when dealing with persistent memory and a compatible page allocator. A page can get lost if the destruction routine has dropped it from the morsel but has not yet given it back to the page allocator. In order to minimize the number of possibly lost pages, only a single page should be in this transient state. A page table must not be deleted if it still holds valid references because otherwise, those references would be lost. On the other side, freed physical page frames that are referenced somewhere would lead to invalid accesses. A tree traversal strategy that fulfills these requirements is the post-order traversal [ARLB03]. A recursive function can implement it, iterating through the tree structure until it has freed the complete morsel. Every element's reference is removed from the parent before the physical page frame is freed. Figure 3.4 visualizes the destruction routine.

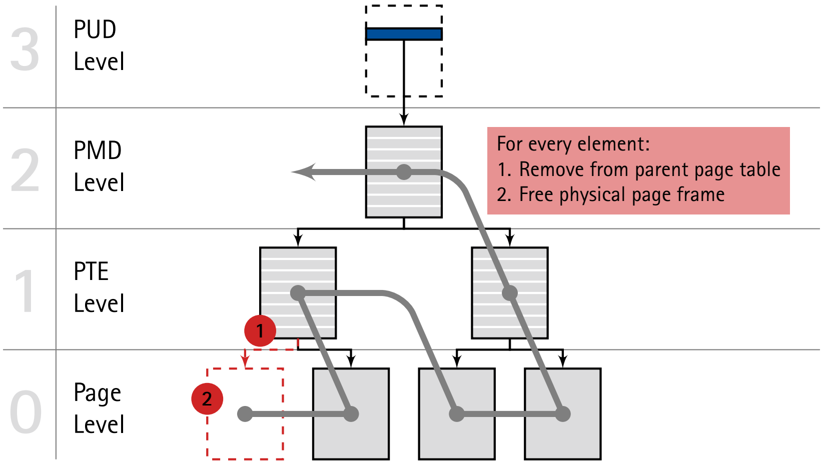


Figure 3.4 – Destroying a morsel to free the used pages: Iterate through all pages and page tables in a post-order pass. For each element, remove the reference from its parent and give back the physical page frame to the page allocator.

Because there is no global morsel registration, the user must ensure that the morsel is not mapped somewhere before destroying it. Destroying a mapped morsel can have unintended consequences when a process tries to access an already freed page or the MMU tries to interpret a non existing page table.

3.4 Lock-Free Lazy Population

The creation routine creates morsels as minimal roots to maximize memory efficiency and minimize latency. These roots do not contain user data pages. Exceptions are morsels of order 0, which only consist of a single user data page that can be directly accessed. For higher orders, the idea is to populate morsels on demand. Demand is expressed through a page fault, indicating a user space application tries to read or write from/to an unpopulated part of the morsel surface. The fault handler only populates the specific page at the requested virtual address, aiming at memory efficiency. An alternative would be an explicit population routine that a user space application must trigger. This approach would have the advantage that batching can improve performance by populating larger parts of the morsel surface at once. It eliminates the need for expensive page faults for every single page. The improvement in terms of speed would come along with the downside that the developers of user space applications need to put much effort into explicit population management (see Section 2.4.2). For this reason, this thesis focuses on the page-fault-based implicit population. The additional implementation of an explicit population function could be a future improvement for applications where the population speed is critical.

On a page fault, the page fault handler walks down the page table tree to the requested page. It allocates missing nodes on this path, including page tables and user data pages, and inserts them into the respective page tables. Because morsels can be used in multiple address spaces simultaneously, the population routine must be thread-safe. The implementation must also provide failure atomicity because morsels can be located in persistent memory. In Section 3.1, we already identified that logging and locking are no viable techniques to achieve the desired behavior. Section 2.3.2 alternatively suggests atomic instructions to prevent invalid transition states. The population routine for a single page table level can then be implemented as follows:

1. Get the corresponding page table entry and check whether the allocation of a new page is necessary. Possibly another thread already inserted a child element in the meantime. This can happen because it takes some time until the execution reaches the custom page fault handler after the page fault has occurred. If the check observes that the child element already exists, nothing has to be done. In the other case, the fault handler continues with step 2.
2. Allocate a new page.
3. Prepare a page table entry that references the allocated page.
4. Do an atomic CAS operation to override the old entry with the prepared value. If the old entry is unchanged, the CAS operation succeeds, and the population step finishes. The operation fails if another thread has been faster. Continue with step 5 in this case.
5. The CAS operation failed, but this does not mean that the population failed. Another thread has just been faster. Undo page allocation by giving the no longer needed page back to the page allocator.

The described process handles only a single level in the page table hierarchy. Multiple nodes can be missing on the path down to the requested page. In such a case, the fault handler must execute the routine for every missing node, even if a CAS operation fails. Possibly the concurrent thread tries to access a different memory location sharing only some upper-level page tables. Figure 3.5 summarizes the population process.

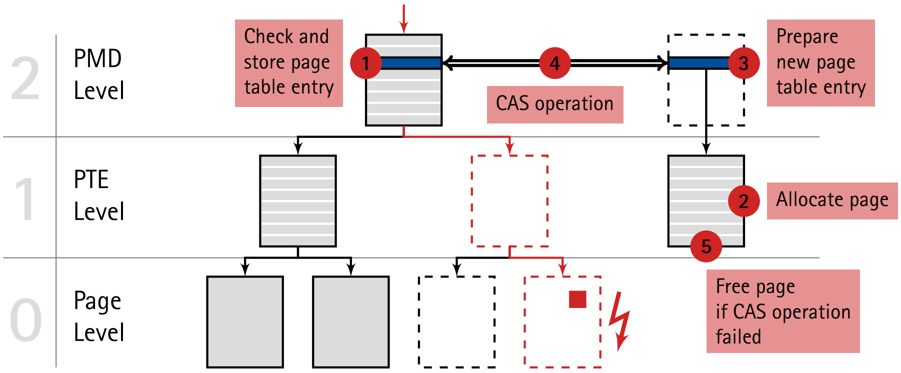

2
PMD Level
1
PTE Level
0
Page Level
Check and store page table entry
1
4
CAS operation
3
Prepare new page table entry
2
Allocate page
5
Free page if CAS operation failed

Figure 3.5 – Morsel lazy population: On a page fault, the page fault handler allocates related page tables and pages for the requested memory location. The figure shows the allocation and insertion of a single element.

3.5 Additional Meta Data

One benefit of NVRAM is the ability of random access in byte-granularity. This feature allows the usage of pointers directly in persistent data objects without the need for serialization. The problem is that pointers become invalid when the memory gets mapped to a different location in the virtual address space. In order to support pointer-based data structures within morsels, the mapping address must always be the same. The mapping routine allows the specification of a target address so that this functionality can be implemented manually in the user space application. On the other hand, this is likely to be a commonly used feature, so it would be better to integrate it into the morsel memory primitive.

The implementation requires some memory to store metadata. For the fixed mapping location, this is only the start address of the mapping, which is 64-bits-wide on an x86-64 machine. Adding additional features in the future may require more space for metadata, so the solution should be scalable. For a consistent feature set, a morsel should support all features independently of its order. As a result, even the smallest morsels must provide sufficient space for metadata. A morsel of order 0 consists of a single data page and its entry in the parent page table, the morsel ID. The data page is unsuitable for storing metadata because it is reserved for user data and exposed to the user application when mapped. The morsel ID has a few unused bits but does not have enough space to store a complete virtual address. Also, this approach would not be scalable. A more flexible way than using the morsel to reference the metadata is to use the metadata to reference the morsel. Figure 3.6 visualizes the idea.

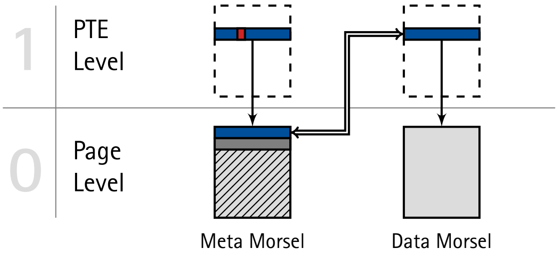


Figure 3.6 – Meta morsel concept: The morsel ID given to the user does not directly reference the morsel containing user data. Instead, it references a separate morsel with metadata, called *meta morsel*. The meta morsel contains the actual morsel ID referencing the so-called *data morsel*. The meta morsel flag, part of the morsel ID, indicates whether an ID refers to a meta morsel or a data morsel. In this example, both meta and data morsel are of order 0. In general, the concept works with arbitrary, even different orders for meta and data morsel.

The morsel ID does not directly reference the morsel containing the user data. Instead, it references a separate morsel storing the metadata. The metadata contains a second morsel ID referencing a second morsel with the actual user data. To precisely refer to one of them, we introduce the terms *meta morsel* and *data morsel*. The meta morsel contains the metadata for exactly one data morsel. The data morsel contains the actual user data. This arrangement makes it impossible to get the metadata if only the data morsel is known. For this reason, it is important that the morsel ID returned by the creation routine points to the meta morsel.

Besides the advantage of flexibility, the additional indirection has some drawbacks. It introduces a memory overhead of at least 4 KiB, the minimum size of an order 0 meta morsel. This overhead seems negligible for larger data morsels, but it is a relative overhead of 100% for an order 0 data morsel. Another aspect is the mapping performance. As discussed in Section 3.3.2, the mapping process is very efficient because it simply inserts the morsel ID into a process's page table. When dealing with meta morsels, the routine should not map the meta morsel itself but the referenced data morsel to the user space. As a result, it requires an additional step to get the data morsel ID from the meta morsel. To prevent those memory and computing overheads when enhanced features are not required, meta morsels should be optional. The morsel ID must provide a possibility to distinguish meta and data morsels because this is the only information that is known a priori. As already done for the morsel order, another free bit, called meta morsel flag, marks meta morsels.

Another aspect not mentioned before is crash consistency. The metadata should not have invalid states corrupting the data morsel. Because it is not necessary to dynamically change the metadata to implement the fixed mapping address feature, the meta morsel can be read-only after creation. If this is insufficient for other enhanced features in the future, their implementation must provide different consistency guarantees.

The previously defined operations (see Section 3.3) and the population routine (see Section 3.4) have been designed for plain morsels. The meta morsel support may require some changes. Generally, meta and data morsels are independent objects. For example, the creation operation can construct two morsels and assemble the requested meta morsel configuration. If a crash occurs before the driver can hand out the ID to the user application, both morsels are lost because no reference exists. The mapping operation is straightforward. It should only map the data morsel, so it uses the meta morsel to figure out the data morsel ID and then continues as usual. For the unmapping operation, there is no difference at all. The destruction operation is more complicated because it should prevent invalid references and reduce the number of possibly lost pages. Extending the existing destruction routine, which uses a post-order traversal, from node to morsel level solves that problem. The adapted routine destroys the data morsel as usual but keeps the uppermost element. Then it removes the reference from the meta morsel and deletes the remaining data morsel element. Finally, it destroys the meta morsel the usual way. The lazy population routine does not need to be changed because there is no difference to plain morsels in the mapped state.

3.6 Summary

In this chapter, we defined morsels, a new memory primitive, as a minimal-necessary abstraction for memory virtualization. A morsel is a page table subtree used as a self-contained memory object that is optionally persistent if placed in NVRAM. It is memory-mappable, sharable between processes, and can be directly interpreted by the MMU without overhead because it is based on hardware-specific data structures.

For the usage of morsels, we discussed three primary operations: creation, (un-)mapping, and destruction. The creation routine only allocates the morsel root to achieve memory and runtime efficiency. A morsel is identified through its entry in the parent page table, called morsel ID. Inserting the morsel ID into an address space's page table tree exposes the user data to the application, making the mapping very efficient. Unmapping works the opposite way but requires additional TLB invalidations. Access rights are applied per mapping by modifying the flags of the morsel ID before inserting the value into the page table.

The destruction operation deletes morsels that are no longer needed. It iterates through all page tables and pages in a post-order pass, removing the elements from their parents and freeing the physical page frames. This procedure limits the number of lost pages in the case of a crash to a single page.

Morsels are populated lazily on demand, expressed through a page fault. The page fault handler allocates page tables and pages and atomically inserts them into the morsel page table subtree. The atomicity prevents race conditions without relying on locks.

The implementation of enhanced features needs additional metadata. Because morsels do not provide enough space for metadata, the nesting of a data morsel into an additional meta morsel solves that problem.

IMPLEMENTATION 4

This chapter provides an overview and discusses the challenges of the morsel implementation as a loadable Linux kernel module. The module provides a character device as an application interface. Applications can interact with this device file using classical file I/O in order to access morsels.

The kernel automatically allocates some upper-level page tables on a page fault using the default routines, which break thread safety for shared morsels. A workaround, only affecting large morsels, prevents extensive kernel code changes. Another problem introduced by page table sharing is different process and morsel lifetimes. A callback mechanism allows the unmapping of morsels before the kernel deletes related page tables.

Typically, the kernel manages memory access rights on the level of individual pages and does not tolerate related flags in upper page table entries. Slight modifications of those checks inside the kernel code solve that problem, allowing rights specification per morsel mapping.

4.1 General Structure

The prototypic morsel implementation relies on the Linux kernel (version 5.17). It is realized as a loadable kernel module to prevent extensive changes to the kernel source code and increase portability. The module comprises about 2400 effective lines of code (see Listing A.3). Required kernel code changes include about ten lines of code affecting only four small check functions. User applications do not require any special library to use morsels because all required functionality is mapped into the file system, so applications only need common file operations. Nevertheless, a small user space library provides some helper functions simplifying morsel handling, e.g., a function to determine the virtual morsel size required for the `mmap` system call.

The module registers a character device on load to expose its functions to user applications. A character device is a virtual file located in the `/dev` directory [Kut18]. Applications can interact with this device using classical file I/O. The driver provides implementations for those I/O functions by specifying the function pointers in an instance of `struct file_operations`. The complete struct is shown in Listing A.2, while Listing 4.1 contains only the most relevant functions in terms of the morsel implementation.

```
1  struct file_operations {
4          ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
5          ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
13         long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
15         int (*mmap) (struct file *, struct vm_area_struct *);
17         int (*open) (struct inode *, struct file *);
19         int (*release) (struct inode *, struct file *);
24         unsigned long (*get_unmapped_area)(struct file *, unsigned long, ↘
               unsigned long, unsigned long, unsigned long);
42 } __randomize_layout;
```

Listing 4.1 – Shortened `struct file_operations` from `/include/linux/fs.h` (kernel 5.17)

The morsel driver registers a character device at `/dev/morsel`. Applications can open this device file to retrieve an uninitialized file descriptor (FD) that must be bound to a morsel ID before further usage. The `open` system call internally calls the specified `open` function (line 17) to perform some basic setup, e.g., allocating memory for data structures. The counterpart is the `release` function (line 19), which is called when an application terminates or explicitly closes a FD.

The `ioctl` system call and the related `unlocked_ioctl` function (line 13) allow the implementation of arbitrary functionality based on a single system call, e.g., initialization. The FD can either be initialized by creating a new morsel or by selecting an existing one to get access to the morsel data.

Typically, character devices implement the `read` and `write` operations (lines 4 and 5) to interact with a data stream serially. However, morsels are designed for random access and, thus, for direct mapping into an address space. Also, read/write operations require additional copying between kernel and user space which is unsuitable for an efficient memory primitive.

The `mmap` system call allows user space applications to map files, devices, or memory into its address space. The generic kernel part of the execution path creates a new VMA object before handing over control to the specialized part, e.g., the `mmap` function (line 15) specified by the device driver (see Section 2.1.4). The driver part can use the VMA to specify additional attributes or to register related callback operations, e.g., a fault handler. In the case of the morsel implementation, the driver also performs the actual mapping by inserting the morsel ID at the corresponding position into the related page table (see Section 3.3.2).

Implementing the `get_unmapped_area` function (line 24) is also necessary because morsels have special alignment requirements. The function is intended to find a viable mapping location, which, in this case, is a naturally aligned, vacant part of the virtual address space.

The resulting usage from the application point of view in order to access morsel data is as follows:

1. Open the driver device file: `int fd = open("/dev/morsel", ...)`
2. Initialize the retrieved FD: `ioctl(fd, ...)`
3. Map the initialized FD into the address space: `mmap(..., fd, ...)`

Nothing special is required for morsel unmapping and releasing the FD.

4.2 Page Fault Handling

The implementation of lazy population requires a custom page fault handler. As described in the previous section, the mapping routine can associate a mapping with a set of related functions via its VMA. When an application triggers a page fault, the kernel interrupt handler is invoked. The execution passes a certain amount of kernel code before it reaches the specified custom fault handler, which implements the population routine described in Section 3.4.

It turns out that the kernel part of the fault handling already allocates missing page tables down to the PMD level before calling the specified fault handler. This behavior breaks thread safety because the default locking mechanism does not work for shared page tables. Preventing this unwanted page table allocation would require modifications of essential kernel code. A workaround is the allocation of those upper page tables on morsel creation. This enlarges the created morsel roots, shown in Figure 4.1, in contrast to the initial concept, shown in Figure 3.2.

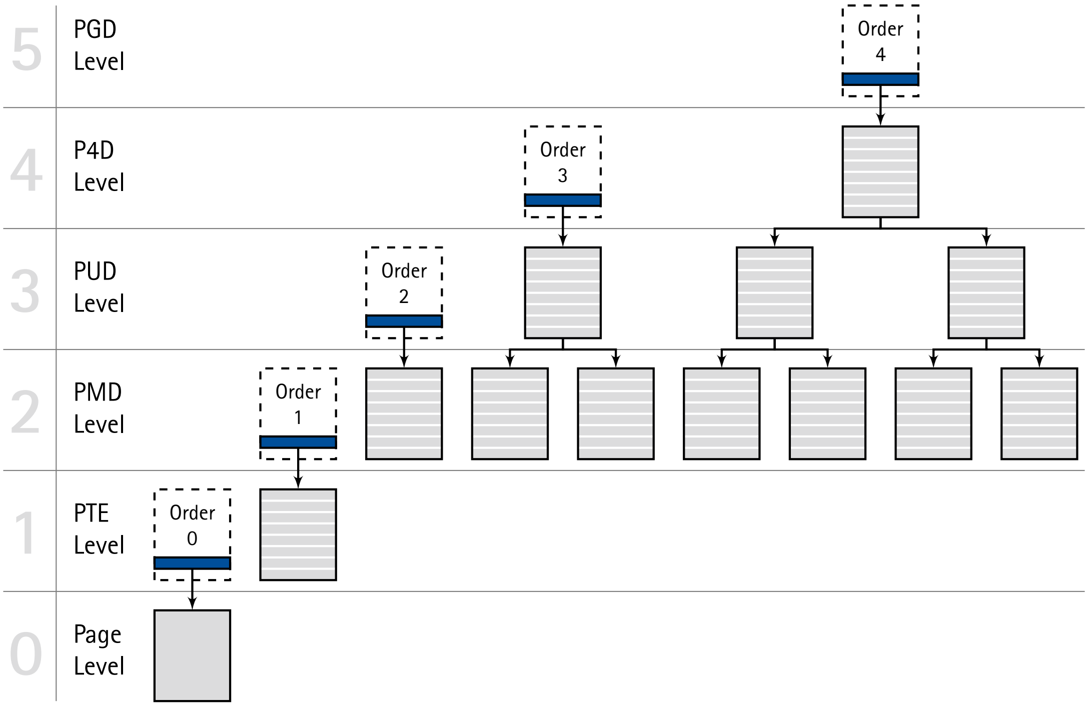


Figure 4.1 – Morsel creation workaround to bypass automatic page table allocation: On a page fault, the kernel part of the page fault handling routine automatically allocates page tables down to the PMD level. The creation routine already allocates the involved page tables to prevent this behavior.

This change does not affect morsels of orders 0, 1, and 2. For morsels of order 3, additional 512 PMD tables are allocated during creation. For morsels of order 4, it is even worse. 512 PUD tables are required, which themselves hold 512 PMD tables each. The preallocation of page tables comes with an increased size of the minimal morsel roots. Table 4.1 lists the actual numbers.

The additional overhead might be arguable for order 3 morsels, but it is entirely unacceptable for order 4 morsels, particularly if only sparsely populated. This workaround needs to be replaced by a real solution to the automatic population problem in order to bring the implementation to a productive level. For now, it enables further analysis of the morsel concept without requiring massive kernel changes.

	Number of Pages / Page Tables						
Order	PGD	P4D	PUD	PMD	PTE	Pages	Total Size
0	-	-	-	-	-	1	4 KiB
1	-	-	-	-	1	0	4 KiB
2	-	-	-	1	0	0	4 KiB
3	-	-	1	512	0	0	~2 MiB
4	-	1	512	262144	0	0	~1 GiB

Table 4.1 – Structure of extended morsel roots

4.3 Decoupled Morsel Lifetime

One characteristic of morsels is their decoupled lifetime, meaning that they can outlive processes. Typically, all page tables uniquely belong to an address space. They can be safely deleted on address space termination or when unmapping a virtual address range. The deletion must be prevented for morsels because they are conceptually independent of the address space they are mapped to. Morsel page tables are managed by the morsel driver. Therefore, it is enough to drop the reference from the address space instead of destroying all subsequent page tables.

The most primitive solution would be a dedicated system call (via `ioctl`) to remove a morsel from an address space before further page table manipulations happen. The implementation is very straightforward, but it has several drawbacks. The application developer must actively manage those system calls to prevent damage to morsels. For ungracefully terminating applications this is impossible at all. Consequently, this cannot be the solution.

Another approach is to utilize an existing callback. A VMA provides a `close` callback, similar to the fault handling mechanism. It is executed when a VMA is removed from an address space, but unluckily this is after the page table manipulations happened, so this cannot be the utilized either.

Integrating the special handling of morsel page tables directly into the kernel code is also possible. On the downside, this intrusive procedure contradicts the position of preventing kernel changes wherever feasible. *MMU notifiers* are a tool to register callbacks and get notified when memory management events occur [Cor08]. They have been initially designed to allow swapping of memory used by virtual machines running in KVM, a hypervisor directly integrated into the Linux kernel. The morsel implementation uses the `release` callback to get informed when the whole address space terminates and the `invalidate_range_start` callback to catch modifications limited to a specific range. With MMU notifiers, it is possible to safely remove a morsel, as described in Section 3.3.2, before further page table manipulations happen. This procedure prevents the morsel page tables from being damaged, even in the case of an ungraceful application stop.

4.4 Access Rights Management

For rights definition of memory mappings, the kernel uses a different mechanism than intended for morsels. Traditionally, page table entries are permissive for referenced lower-level page tables, and specific rights are only applied to individual pages. This approach allows fine-grained permission control in the granularity of pages. As stated in Section 3.3.2, the opposite way is required to specify rights per mapping instead of per morsel.

The kernel performs internal checks to verify page table entry integrity at some points. These checks do not allow flags different from the intended layout and classify corresponding entries as invalid. The kernel cannot work with invalid entries, so it clears them. It requires the modification of those checks to allow rights management on page table level. It is impossible to do this outside the kernel because it is an inherent part of the kernel source code. Consequently, kernel code changes are unavoidable in this case. Listing 4.2 shows the check for PUD entries. Checks for PGD, P4D, and PMD levels are similar. No check exists for PTE entries because they refer to pages where all flags are allowed.

```
static inline int pud_bad(pud_t pud)
{
        return (pud_flags(pud) & ~(_KERNPG_TABLE | _PAGE_USER)) != 0;
}
```

Listing 4.2 – Integrity check for PUD entries from `/arch/x86/include/asm/pgtable.h` (kernel 5.17)

The check compares the flags of a specific page table entry with a mask of allowed flags. This mask can be extended to include the *R/W* and the *NX* flag (*XD* flag in Intel terms) as well (see the Intel page table layout described in Section 2.1.3). Listing 4.3 shows the modified version of the PUD check. The adjustments for the other levels are similar.

```
static inline int pud_bad(pud_t pud)
{
        unsigned long ignore_flags = _KERNPG_TABLE | _PAGE_USER | _PAGE_NX
                | _PAGE_RW;

        return (pud_flags(pud) & ~ignore_flags) != 0;
}
```

Listing 4.3 – Page table integrity check adapted to allow upper level rights specification

The mask definition has been moved out to a separate variable for better readability. All allowed flags are combined into a single mask that is compared with the flags of the given entry. So far, we have only discussed the additional flags for rights management. However, some more bits are used differently compared to the default kernel implementation: the encoded morsel ID and the meta morsel flag. Further adaptions would be required to allow such bits in active page table entries. Because the information provided by those bits is not necessary for mapped morsels, the mapping routine can clear them from the morsel ID before inserting it into an address space. Thus, the checks do not need to tolerate those bits in page table entries.

4.5 Summary

This chapter discussed the most critical aspects and challenges of implementing the previously defined concept. The prototypic morsel is implemented as a loadable driver module for the Linux kernel (version 5.17). The driver provides a character device, located at `/dev/morsel`, as an application interface. Applications can open the device file to retrieve an uninitialized FD, which must be initialized by selecting a known morsel or creating a new one. The initialized FD enables access to the user data via memory mapping.

The page fault mechanism is used to implement lazy population. The automatic page table allocation down to the PMD level in the kernel part of the page fault handler bypasses the special lock-free insertion. Using the conventional kernel routines for page table allocation breaks thread safety for shared morsels. As a workaround, the creation routine already allocates affected page tables for large morsels. This workaround does not affect morsels of orders 0, 1, and 2, but it adds a massive memory overhead for morsels of orders 4 and 5 up to about 1 GiB. The prepopulation is no real solution but allows further evaluation of the morsel concept without extensive kernel code changes.

Typically, page tables uniquely belong to a single address space. The kernel deletes unneeded page tables on unmapping operations or process termination, which must be prevented for morsel page tables. The MMU notifier mechanism allows the registration of callbacks to memory management events. Such a callback is used to remove a morsel from an address space before the kernel cleans up page tables.

The kernel traditionally uses rights management on the level of pages, but morsels require rights management on page table level. At some points, the kernel performs integrity checks on page table entries, not tolerating unexpected flags. Therefore, the checks must be adapted to ignore additional rights flags in upper page table entries. The encoded morsel order and the meta morsel flag as part of the morsel ID are no problems because the mapping routine clears the respective bits before inserting an entry into a page table.

ANALYSIS 5

In this chapter, we analyze the prototypic morsel implementation. All tests are performed using conventional DRAM because a morsel is primarily a general-purpose memory primitive. Additional evaluation on NVRAM would exceed the scope of this work. Moreover, no system with NVRAM implementing the required eADR feature is currently available for use.

The evaluation uses two differently powerful systems to cover efficiency as well as multi-core scalability. The lazy population of morsels is compared with an anonymous, shared mapping for different thread counts. Profiling reveals the bottlenecks of both competitors. Due to the conceptual design of morsels and their management in the granularity of complete page table subtrees, mapping should execute in constant time. A benchmark compares the latency for differently sized memory objects with the established POSIX shared memory (SHM). Another test setup utilizes this efficient mapping to transfer large amounts of data between address spaces. Two connected sender and receiver applications exchange data packets. Arrival timestamps are converted to a data rate.

The in-memory caching software Memcached is a suitable real-world example for the use of morsels because persistent main memory could significantly improve startup performance. Replacing the backing storage with a morsel shows the adaption of existing software. Finally, the prototypic morsel implementation is evaluated based on the previously defined requirements (see Section 3.1).

5.1 Evaluation Hardware

There is no single system that fits all requirements. As a result, the evaluation is performed on two different hardware systems. Table 5.1 shows the specifications of both platforms.

	Desktop PC	Server System
Type	Dell OptiPlex 3040	Dell PowerEdge R740
CPU	Intel Core i5-6500T	2x Intel Xeon Gold 6252
Base Frequency (Max.)	2.5 GHz (3.1 GHz)	2.1 GHz (3.7 GHz)
Cores (Threads)	4 (4)	2x 24 (2x 48)
DRAM	1x 8 GiB DDR3 @1600 MHz	12x 32 GiB DDR4 @2666 MHz

Table 5.1 – Hardware specifications of the used evaluation systems

A small desktop PC is the basis for latency and performance measurements. The simple platform works out of the box with the default Linux kernel configuration because it does not require any extraordinary drivers. On the downside, it only provides a limited amount of cores and memory.

Benchmarks regarding the scalability of morsels on large systems are performed on a dual-socket server system. Its two NUMA nodes also allow investigating possible NUMA effects.

Both systems are configured to produce deterministic and reproducible results. The server provides two logical cores for every physical core, sharing hardware resources. This feature is called simultaneous multithreading [TEL95]. It improves the utilization of superscalar processors, but logical cores belonging to the same physical core affect each other, competing for hardware resources. Therefore, simultaneous multithreading is inappropriate for deterministic measurements and is disabled. Another aspect is dynamic frequency scaling [Bao+16]. It dynamically adapts the processor frequency based on different factors, e.g., power consumption or processor utilization. Fluctuations in the CPU frequency influence the benchmark results and, thus, should be prevented. Fixing the frequency to the specified base value solves the problem for both evaluation systems. To further reduce external factors, the benchmarks execute with the highest priority reducing the effects of concurrently running applications or system services. Pinning benchmark threads to specific processor cores reduces scheduling effects. On the small system, this configuration is even extended to a lower level by setting kernel command line parameters that optimize the system behavior for benchmarking purposes [Ker22b]:

```
GRUB_CMDLINE_LINUX="isolcpus=nohz,domain,managed_irq,2-3 irqaffinity=0-1"
```

The specified values isolate two CPU cores from regular scheduling and optimize interrupt handling. Benchmarks running on the isolated cores achieve a maximum of deterministic behavior.

5.2 Population Speed

The morsel implementation relies on atomic CAS operations for page table manipulation instead of locking as used by the kernel. Thus, it should be faster and provide better scalability along multiple threads. The first benchmark compares the population speed of morsels with anonymous, shared mappings. It allocates and maps a memory object of the respective type, which is then divided into equally sized, virtually contiguous slices, one for each thread. Every thread populates its memory slice by triggering page faults with write accesses. Only a single byte per page is written to reduce the overall data volume and shift the memory bandwidth bottleneck to higher thread counts. The benchmark result is the duration required for populating the whole memory region. The procedure repeats for different thread counts and two different NUMA policies.

Both policies start with core 0 and assign additional cores in ascending order. The first policy sequentially assigns the cores of different NUMA nodes. It takes all cores of one node before it starts using the next node. The second policy assigns cores of all NUMA nodes alternatingly. As a result, two points in the policies are identical: The first, only a single core, and the last, all cores of the machine.

Figure 5.1 visualizes the benchmark results for 32 GiB of memory and 32 iterations per configuration executed on the server system. The points mark the median values of all iterations. Error bars indicate the deviation as range from the 10th to the 90th percentile. This method has the advantage of being more resistant to outliers in contrast to mean values paired with standard deviations.

The left column shows the results for the sequential, and the right column the results for the alternating NUMA assignment policy. The upper row displays the total time needed for the population, and the lower row the population speed per thread calculated from the measured times. Starting with the left column, we can see that about eight threads are enough to saturate the population of an anonymous, shared mapping. In contrast, about twelve threads are required to saturate the population of morsels. For 25 threads and above, the second NUMA node is also

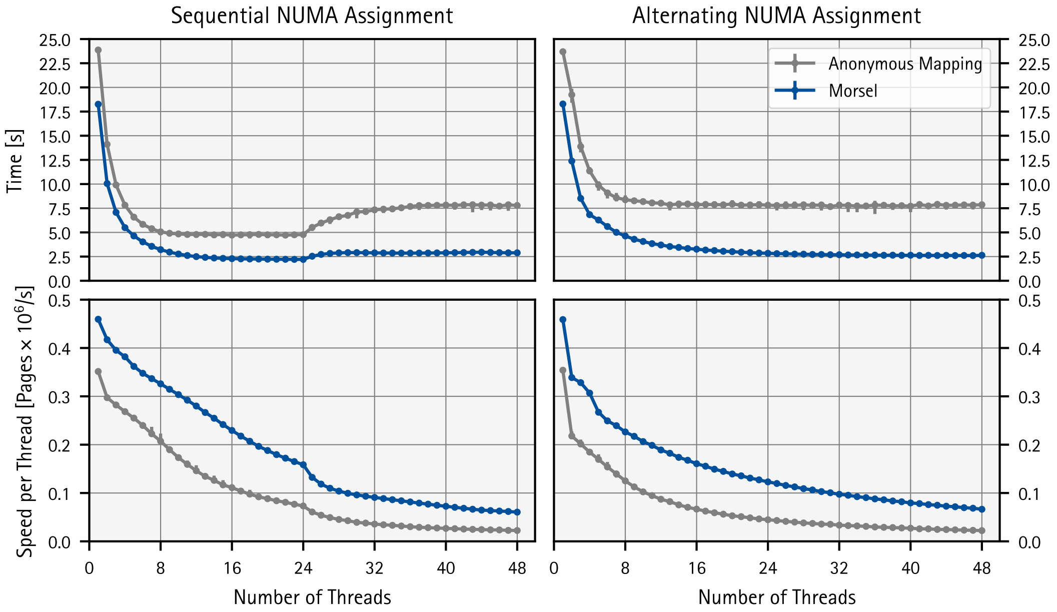


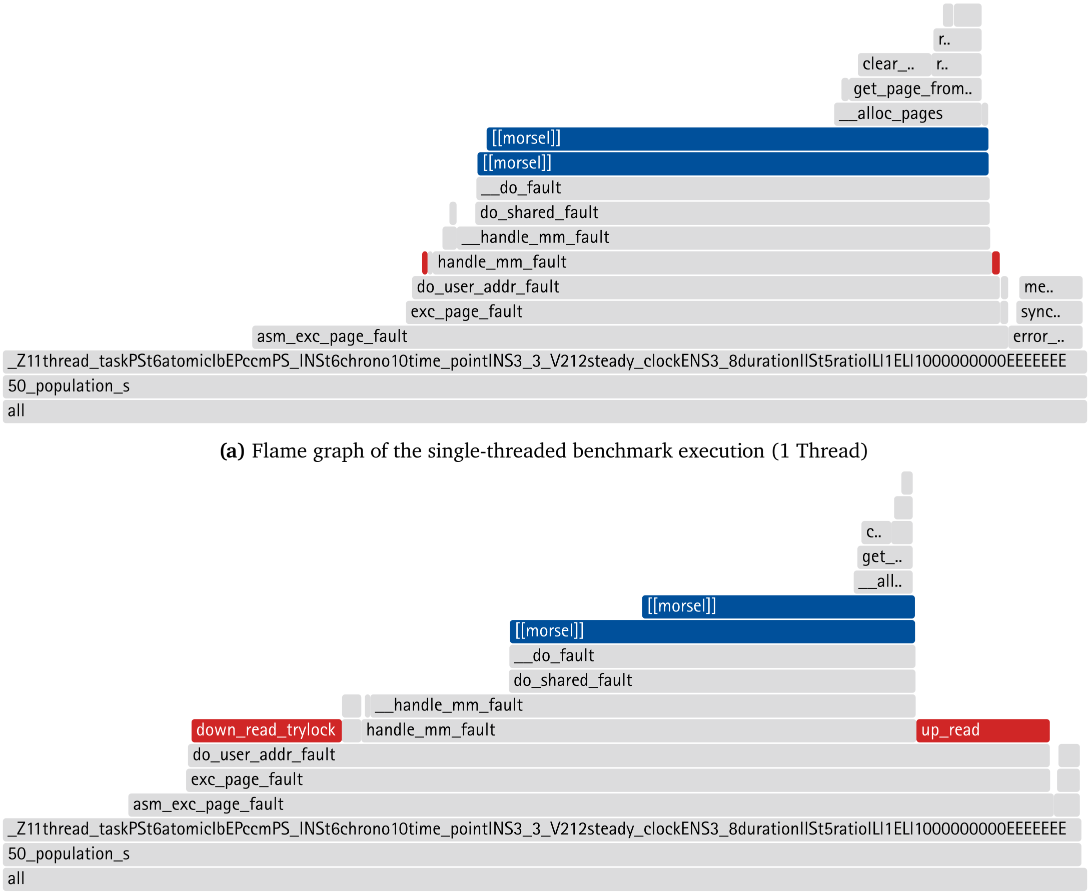
Figure 5.1 – Measuring results of population performance: A 32-GiB-wide virtual memory region is populated by triggering page faults. The work is uniformly distributed to the specified number of threads. The two columns represent two strategies for thread-to-core assignments based on the NUMA affiliation of the cores.

involved into population. The graph shows a significant performance penalty when cores of both NUMA nodes are working on the same memory object. This effect is especially high for anonymous mappings.

The second NUMA policy shifts the saturation points to higher thread counts. They are at about twelve for the anonymous mapping and 24 for the morsel. Using cores of both NUMA nodes generally reduces the performance compared to the same core count on a single node. The overall result is that the population of morsels is approximately 30% faster when using only a single thread. The advance increases up to 100% for high parallelism on a single NUMA node. With two NUMA nodes, the morsel performance is nearly three times as fast as the population speed of its competitor for high thread counts. Nevertheless, the performance per thread significantly descends with every additional thread in all cases, even for low thread counts. One possible reason could be the system page allocator that must provide pages with an appropriate speed or internal synchronization mechanisms. Determining the actual bottleneck requires further investigation.

Profiling is a standard tool to analyze CPU time consumption. Data gathered from a profiler is typically not very handy. Flame graphs are a visualization technique that presents hierarchical data in an understandable way. In the case of CPU profiling, the data is numerous stack traces. Boxes symbolize function calls, and the y-axis represents the depth in the call graph from bottom to top. The width of a box scales with the number of samples containing this function and, therefore, gives a hint of the actual share of execution time.

We compare the flame graphs of morsel population for a single thread and 24 threads to determine the bottleneck in multi-core scaling. The multi-threaded benchmark runs only on a single processor to bypass additional NUMA effects. Data has been collected using the Linux profiler `perf` with a sample rate of 999 Hz for a period of 10 s. Figure 5.2 shows the resulting flame graphs.



(a) Flame graph of the single-threaded benchmark execution (1 Thread)

(b) Flame graph of the multi-threaded benchmark execution (24 Threads)

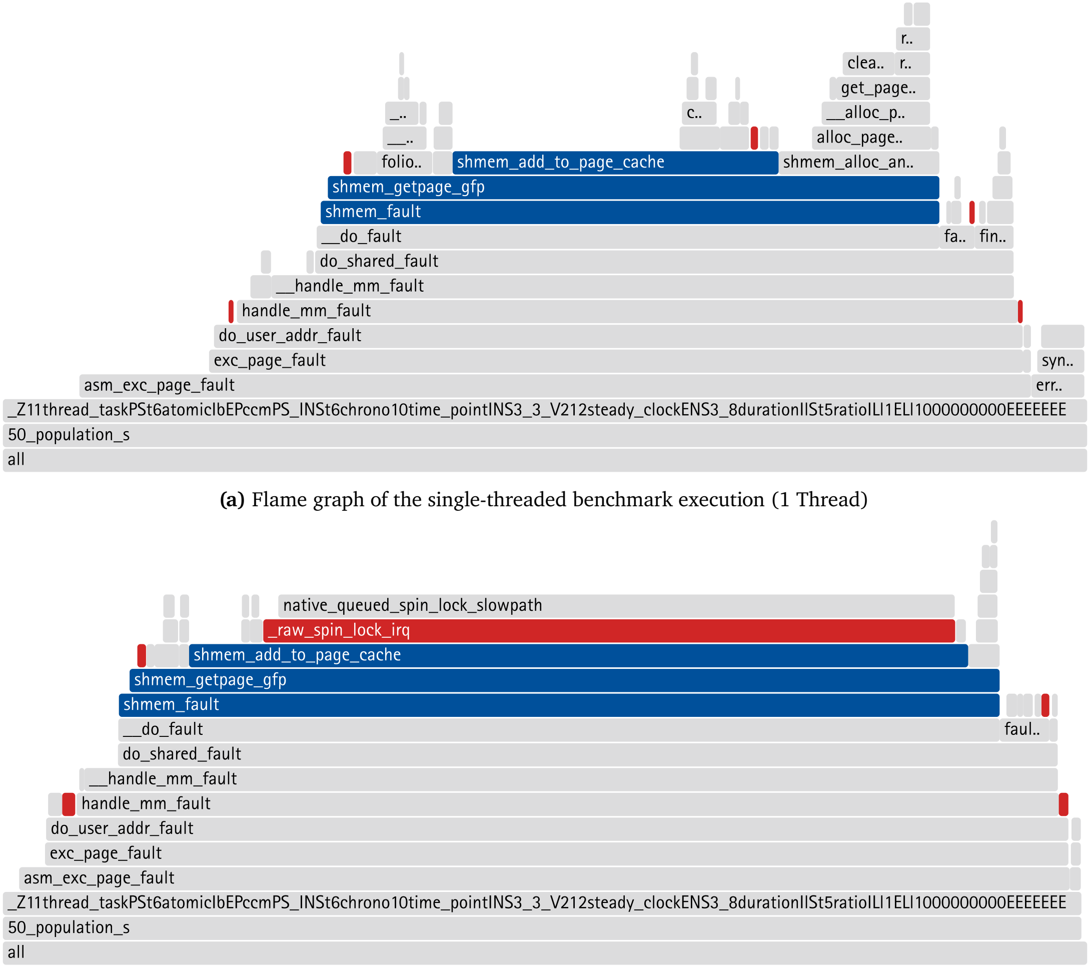
Figure 5.2 – Flame graphs showing profiling results for morsel population

The general layout for both flame graphs is identical because both graphs show profiling results for the same benchmark. For the sake of clarity, very small boxes are suppressed. The bottom rectangle is the basis, including the whole measurement. The rectangle above is the benchmark execution with the worker threads on top. The major part of the worker threads is page fault handling as indicated by the `asm_exc_page_fault` function. After going through six more layers of kernel functions, the execution reaches the morsel page fault handling routine. The additional layer on top is an internal helper function. We can see that the actual morsel page fault handler takes less than half of the total execution time. The only external functionality is the page allocator, indicated by the `__alloc_pages` function. In the single-threaded benchmark, the page allocations take between a quarter and a third of the total time the morsel page fault handler needs.

With multiple threads, the distribution of execution time changes, e.g., the portion of the page allocator decreases relative to the morsel fault handler. Therefore, page allocation is no bottleneck in this test case. One reason might be the high utilization of the memory bus highlighting other expensive memory operations. One significant difference between the single-thread and the multi-thread benchmark is the `down_read_trylock` and `up_read` functions on the seventh layer next to the `handle_mm_fault` function. Both functions are nearly invisible in the upper flame graph but appear in the lower graph with a considerable percentage.

The kernel source code reveals that those functions acquire and release the reader/writer semaphore `mm_struct→mmap_lock`, which allows either multiple readers or a single writer to synchronize address space changes. Even if it is only taken for read access, the acquire and release operations lead to frequent cache invalidations. This bottleneck is a known issue for applications with frequent parallel page faults [Cor22]. It is impressive that the morsel implementation performs much better than the anonymous, shared mapping, even if both competitors have to deal with this limitation.

Doing the same analysis for the anonymous, shared mapping reveals the reason of its bad multi-core scalability. Figure 5.3 shows the corresponding flame graphs using the same settings as for the morsels.



(a) Flame graph of the single-threaded benchmark execution (1 Thread)

(b) Flame graph of the multi-threaded benchmark execution (24 Threads)

Figure 5.3 – Flame graphs showing profiling results for the population of a shared, anonymous mapping

The ten lower layers are equal to the morsel implementation. Starting with the single-threaded execution, we can see that page cache management, in this case, adding a page to the page cache as indicated by the `shmem_add_to_page_cache` function, is the most expensive part of the shared-mapping-specific page fault handler `shmem_fault`. The already known acquire and release operations for the `mmap_lock` semaphore can be found at the expected locations on the level of the `handle_mm_fault` function. An additional release path is on the level of the `shmem_add_to_page_cache` function.

In the multi-threaded execution, the share of the page cache management increases significantly. The `_raw_spin_lock_irq` function, an unimportant fraction in the upper graph, changes to an expensive portion in the lower graph. The name suggests that it implements a spinlock acquirement. Another spinlock that does not seem to be a bottleneck is on the level of the `shmem_add_to_page_cache` function.

A look into the source code discloses the reason for the spinlock bottleneck. The kernel manages shared mappings in the page cache. On every page allocation, the new page must be added to the cache. Different data objects are separate structures in the page cache, and every structure has a lock that protects it. Multiple simultaneous claims to the same lock due to parallel page faults create high contention, resulting in poor multi-core scalability. Morsels have the advantage of implementing memory sharing without relying on the page cache, so they do not have to deal with this problem.

5.3 Mapping Speed

A second benchmark addresses the mapping speed of shared memory, including the population with existing pages. Morsels are compared with POSIX shared memory (SHM) [Yas22]. The benchmark prepares memory objects in advance to measure the raw mapping and population performance without page allocation overheads. The preparation includes the creation, the mapping to the address space, and the initialization using pseudo-random data.

For the actual measurement, the benchmark maps the prepared memory object a second time to a different location and accesses every page once to ensure its presence and possibly trigger lazy population. Finally, it removes both mappings and deletes the memory object. This procedure is executed for different object sizes on the desktop PC. Figure 5.4 shows the combined results for mapping and read access.

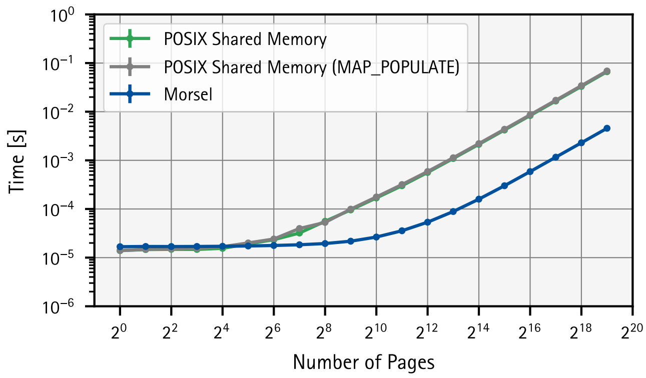


Figure 5.4 – Measurement results of mapping and accessing differently-sized memory objects: Median values combined with error bars showing the 10th to 90th percentile range

For SHM, two different configurations are considered. The first one maps the memory object the default way, and the second one with the MAP_POPULATE attribute specified. This attribute ensures that all required page table manipulations are performed during the mapping system call and not lazy on first access. This behavior is essential when analyzing the runtimes of mapping and accessing all pages separately in the next step. The values shown in the graph are median values of 256 iterations per test case. Error bars represent the range from the 10th to the 90th percentile. The fluctuations in the measured values are so small that the error bars are not visible in the graph. We can see that the runtimes for SHM with and without the MAP_POPULATE flag are almost equal. From 2^0 to 2^4 pages the runtime for SHM and morsels is nearly constant, and SHM is slightly faster. Starting at around 2^5 pages, the runtime of the SHM test case exceeds the morsel implementation. For 2^{12} pages and more, the lines run in parallel again with a difference of more than a whole magnitude. Figure 5.5 shows the runtime values split into its mapping and accessing components to analyze those measurement results further.

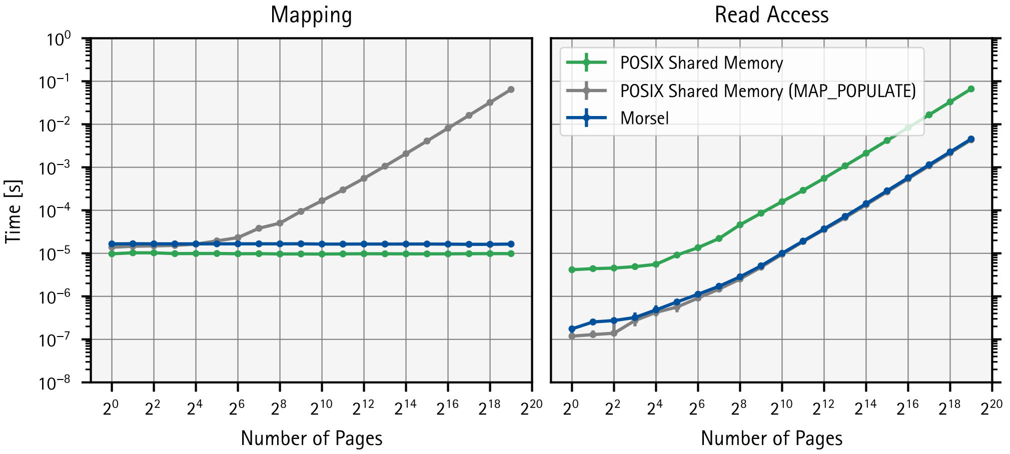


Figure 5.5 – Split results of mapping and accessing differently-sized memory objects

The left graph shows the mapping time, and the right graph shows the time required to read the first four bytes of every single page. Measurement values and graph parameters (median values combined with 10th to 90th percentile error bar) equal the previous graph (see Figure 5.5). As expected, we can see that the mapping of SHM without additional attributes defers page table manipulations to the first access. The runtime of the mapping system call is constant and faster than the other two competitors. On the other hand, the access is about one magnitude slower.

The mapping of SHM with the MAP_POPULATE attribute allows a direct comparison with the morsel implementation because, in both cases, all required page table manipulations are performed during the mapping system call. We can see that the mapping time of SHM with MAP_POPULATE specified linearly increases with the number of pages. For small page numbers, the logarithmic scaling distorts the line because there is a base runtime as general overhead induced by the mapping system call. The mapping of morsels executes in constant time independent of the page count. For counts below 2^4, the morsel mapping is slightly slower, for 2^4 pages, the runtime is equal, and for higher page counts, the morsel mapping is faster. The time for accessing all pages is nearly equal for both memory objects, with a small advantage of SHM with MAP_POPULATE for low page counts. The time increases linearly with the number of pages, reflecting the runtime complexity. The equal runtimes imply that, in both cases, the accesses do not trigger page faults. In the case of SHM, this is prevented due to the specified MAP_POPULATE attribute, whereas for morsels, this is a conceptual design feature. The small difference for low page counts could be an effect of caching. Due to the preceding population of SHM in page granularity, some related page tables may already be in the CPU caches. Hence, the first few page table walks execute faster. For higher page counts, the increased total runtime hides this effect. Caching might also be the reason for the deviation from perfectly straight lines. Another explanation could be measurement inaccuracies for short durations in the magnitude of a few µs and below.

In total, this benchmark shows a great advantage for morsels. Their conceptual design allows mapping by manipulating only a single page table entry because lower-level page tables are directly included. On the other side, for SHM, page tables must be allocated and populated, either during the mmap system call or lazily on demand. Managing the memory object on the level of individual pages instead of a single indivisible unit adds significant computational overhead.

5.4 Sharing Data Between Address Spaces

The third benchmark further investigates the handling of morsels as indivisible units. With fast mapping, the data transfer between address spaces should also be fast.

The test setup comprises two applications on two different cores, one sender and one receiver. A pipe linking the standard output stream of the first and the standard input stream of the second connects both applications [Mod18]. The setup is depicted in Figure 5.6.

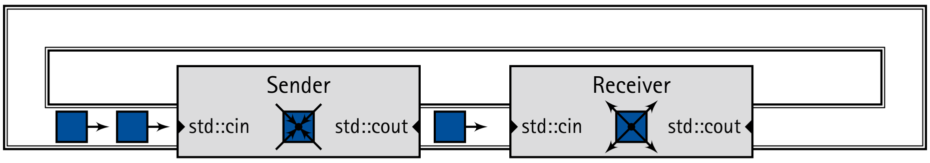


Figure 5.6 – Benchmark setup for measuring data transfer speed between address spaces: Data packets are transferred from a sender application to a receiver application by passing a reference through a pipe. The receiver sends back packets for reuse to prevent high setup costs.

Both applications exchange memory objects as data packets, comparing the morsel memory primitive against SHM. The sender takes a memory object, maps it into its address space, and fills it with data. Then it unmaps it again and sends a reference via the standard output stream. The reference is a name, in the case of SHM, or a morsel ID, in the case of morsels. The receiver maps the received memory object into its address space, accesses all pages, and unmaps it again. The receiver measures the timestamps of received packets that can be converted into a data rate if the packet size is known.

To avoid the overhead of setting up new packets, including the population discussed in Section 5.2, the sender prepares a fixed number of packets during the initialization phase. Additionally, the receiver gives back processed packets to the sender for reuse via a second pipe. The benchmark uses a packet size of 32 MiB to reduce the communication overhead induced by the input and output streams. Nevertheless, it is still small enough to argue that it is a realistic size used in practical applications. Because morsels are only available in a few discrete sizes, the next larger size is used, which is a morsel of order 2 and a virtual size of 1 GiB. Although, only the first 32 MiB are actually populated with pages. Figure 5.7 shows the results of the described benchmark executed on the desktop PC for 1000 iterations and five circulating packets.

In addition to both compared shared memory mechanisms, the graph shows measurements of the `memset` function as the upper limit. The sender cannot provide packets faster than filling

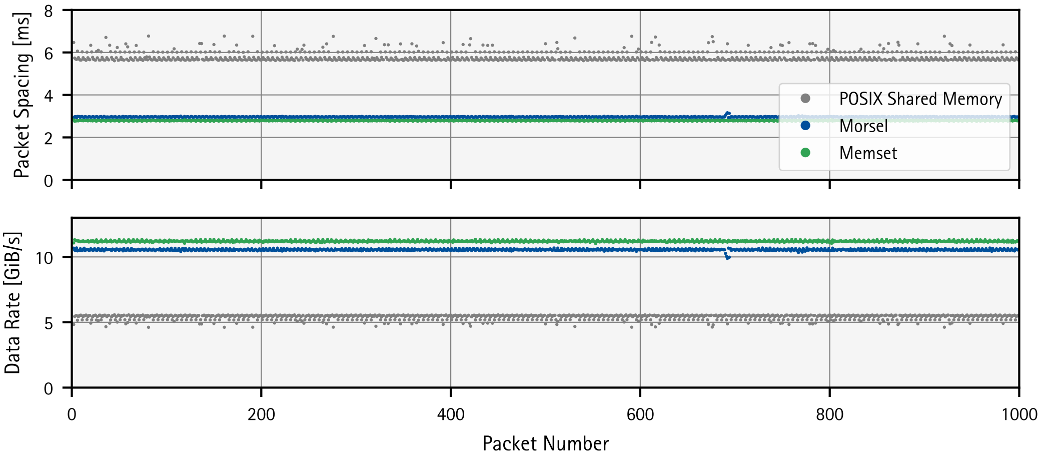


Figure 5.7 – Benchmark results for measuring data transfer speed between address spaces: 32 MiB packets are transferred between a sender and a receiver application (see Figure 5.6) using different mechanisms, POSIX shared memory (SHM) and morsels. Additionally, the raw `memset` speed is given as the theoretical upper limit based on the achievable memory bandwidth of a single core.

them with data. The measurement of the `memset` function uses a modified version of the benchmark, where packets are prepared but not sent through a pipe. Therefore, mapping and unmapping system calls are not required, so it measures almost the raw memory bandwidth of single core.

The upper graph shows the time passed between the reception of two consecutive packets. The lower graph shows those values converted to a data rate based on the packet size of 32 MiB. The data rate of SHM fluctuates around 5.5 GiB/s with some drops due to latency spikes in packet reception. The morsel implementation is relatively constant at about 10.5 GiB/s, which is near the upper limit provided by the `memset` function at 11.2 GiB/s. The benchmark shows a great benefit of morsels in this application. This is probably the result of the efficient mapping and unmapping of whole page table subtrees avoiding the costly handling of many individual pages.

5.5 Real World Example Memcached

The integration of morsels into a real-world application should demonstrate their practical usability. A good target is Memcached, an open-source, in-memory key-value database [Dor22]. It acts as a cache to speed up primarily web applications. It can benefit from persistent main memory because it provides a larger quantity of memory per system. It also removes the need for cache warming because data is still available from the last run. The nature of NVRAM removes the need for serialization to disk in order to keep the cache contents over reboots. Driven by the significant advantages of NVRAM for caching applications, there are already attempts at Intel Optane PMem support of Memcached [Dor19]. The implementation can benefit from a memory primitive optimized for NVRAM. Integrating morsels is the first step toward complete persistent memory support, which has multiple levels. The first level targets graceful shutdowns/reboots required due to software updates or hardware maintenance. The second level also includes ungraceful shutdowns, such as quickly bringing a machine back online after a failure. The latter requires much more work because the whole implementation must use crash-tolerant algorithms.

A possible entry point is the replacement of the backing storage with a morsel. This simple patch is primarily a piece of advice on what can be done with morsels in practice and only a tiny step toward full NVRAM support of Memcached. Nevertheless, it shows the easy adaption of existing software. For better comparison, the reference implementation is also configured to allocate the whole memory for object storage at once during the startup phase. The evaluation uses the *memtier_benchmark* executed on the server system to demonstrate that the modified version of Memcached still works as intended. The benchmark is a tool for measuring key-value database performance [Red13]. Figure 5.8 shows the measured results for the reference and the adapted version of Memcached.

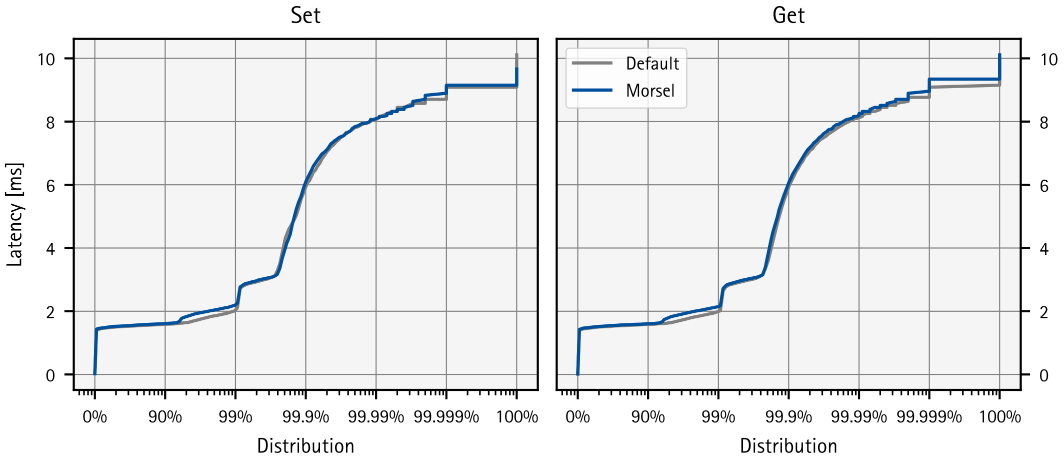


Figure 5.8 – Evaluation of a morsel-backed Memcached variant using the *memtier_benchmark*

In the used configuration, Memcached allocates 64 GiB of memory for the object cache. The benchmark setup utilizes 16 threads, 50 clients per thread, and 10 000 requests per client to stress the database. The benchmark results are the latencies of set and get operations. The graph visualizes them as a distribution on an inverse logarithmic scale to highlight rare latency spikes. Comparing both lines reveals that using morsels has no significant effect on performance. The average set latency of the morsel variant is 1.55 ms, and 1.52 ms for the reference. Average get latencies are 1.54 ms and 1.51 ms. In both cases, the deviation is approximately 2%, which is within the measurement tolerance. The benchmark was executed using DRAM, so the values do not include possible effects induced by NVRAM. However, those effects would result from the underlying memory technology, not the used memory primitive. The observations from this section show that Memcached is a promising target for morsel integration that should be further addressed in future work.

5.6 Functional Evaluation

In this section, we evaluate the work based on the requirements defined in Section 3.1.

Requirment 1 *Morsels are self-contained, sharable memory objects using existing paging data structures.*

Self-containment is an inherent design feature that is part of the implemented solution. The sender-receiver-benchmark illustrates this characteristic (see Section 5.4). The morsel ID is the only attribute passed through the pipes, and it is enough to perform page table manipulation without additional information. The mentioned benchmark is also a suitable example for the sharing property because it transfers data between two independent applications with their own address spaces.

Requirment 2 *They are inherently crash-consistent, thread-safe, lock- and log-free.*

The implementation is based on atomic instructions resulting in thread safety and crash consistency if used with compatible NVRAM that provides the eADR feature. The crash consistency has not been proved in practice as part of this thesis due to the lack of compatible hardware. Therefore, it is currently a theoretical feature that must be evaluated in future work. A good example demonstrating thread safety is the population benchmark (see Section 5.2).

Requirment 3 *Sparse population in combination with lazy allocation achieves low creation latency and memory efficiency.*

The concept considers sparse population by only allocating the uppermost element at morsel creation. The current implementation hurts this requirement for morsels of orders 3 and 4 due to complications with the kernel part of the page fault handling routine (see Section 4.2). This shortcoming can be removed by adapting the kernel source code. This improvement is another point of future work.

Requirment 4 *Direct access without OS interaction achieves reasonable runtime efficiency.*

Morsels provide access using memory mapping to avoid OS interaction and additional data copying. Furthermore, they do not rely on the page cache for sharing, which prevents the bottleneck of shared, anonymous mappings (see Section 5.2). Overall, the prototypic morsel fulfills the requirements. A little polishing is required to remove some minor imperfections.

5.7 Summary

The evaluation has been performed on two test systems: a small desktop PC, using isolated cores for benchmark execution to remove scheduling effects from the results, and a dual-socket server, providing insights into the scalability with many cores and much memory.

The first benchmark analyzes morsel population speed by triggering page faults with write accesses. An anonymous, shared mapping acts as the baseline. The morsel implementation is generally faster and scales better with an increasing number of threads. A reader/writer semaphore that synchronizes address space changes prevents even better scalability due to frequent cache invalidations, although only the read lock is actually taken. Populating the anonymous, shared mapping involves the page cache, which is the bottleneck in this case. A lock synchronizes changes to the related page cache object degrading the overall performance.

The second benchmark addresses the mapping latency of morsels. Due to their handling in the granularity of whole page table subtrees, it executes in constant time. In contrast, the mapping time of the competitor SHM increases linearly with the number of pages. A third benchmark utilizes this morsel property to exchange large amounts of data between address spaces, using a sender and a receiver application. The morsel variant nearly reaches the theoretical limit drawn by the maximum memory bandwidth. The reference using SHM is slightly below half of that limit.

A possible real-world use case of morsels is Memcached, an in-memory caching software. It would greatly benefit from persistent memory due to increased capacity and the ability to restore cache contents after reboots without serializing/deserializing to/from disk. Replacing the backing storage of the cached objects with a morsel shows the easy adaption of existing software. Even if it demonstrates the applicability of morsels in practice, it is only a small step toward full NVRAM support of Memcached. The final evaluation concludes that the prototypic implementation fulfills all requirements.

6 CONCLUSION

In the final chapter, we revisit the most important aspects of this work and discuss some perspectives of future work. The morsel memory primitive is an early piece of a larger research project drawing a bigger vision of modern memory management.

6.1 Summary

Current memory management in OSs is based on the past assumptions of scarcity and hardware independence. Nowadays, memory is not a scarce resource anymore, and hardware independence adds complexity resulting in runtime overhead. Moreover, current memory subsystems are not designed to deal with persistency introduced by emerging NVRAM technologies.

Much research targets the improvement of memory management. Page tables, especially if sparsely populated, induce a notable memory overhead. One idea to reduce this overhead is to share page tables between address spaces that map the same memory. This sharing introduces new problems because page tables typically belong to a single address space, so they can be safely deleted if not required. With sharing, a mechanism, e.g., reference counting, is required to prevent the kernel from deleting shared page tables.

Another research topic is memory management OS interfaces. Some general-purpose interfaces are already a bottleneck for modern applications. Giving explicit control to the user application via specialized interfaces can bypass that bottleneck at the expense of increased complexity.

One more subject is the utilization of NVRAM in software. Twizzler is an OS explicitly designed for NVRAM, providing an indirect referencing mechanism that allows pointers across persistent memory objects. On the downside, that concept is incompatible with existing OSs.

The contribution of this work is *morsels*, a new memory primitive based on the existing hardware data structures to overcome the limitations of the existing memory management. A morsel is a page table subtree that acts as a self-contained, sharable memory object, optionally persistent if placed in NVRAM. Atomic instructions achieve crash consistency and thread safety without locks and logs, as well as good performance and multi-core scalability. Sparse population and lazy allocation provide low creation latency and memory efficiency. Memory mapping allows access without OS interaction and, therefore, without copy or system call overheads. The existing paging data structures predefine the supported virtual morsel sizes: 4 KiB, 2 MiB, 1 GiB, 512 GiB, and 256 TiB.

The tree-like data structure allows the identification of a morsel due to its entry in the parent page table, called morsel ID. For memory efficiency, the creation routine only allocates the root element. The page fault handler implements lazy population to insert lower-level page tables and pages on demand.

Due to the conceptual design, the modification of a single page table entry is enough to insert a morsel into an address space. This page table entry also includes the corresponding access rights allowing access control per mapping without modifying the morsel data structures. A downside of the morsel concept is that it does not provide space for additional metadata, e.g., a fixed mapping address. The nesting of two morsels fixes this issue. One morsel contains the metadata and a reference to the second morsel, which contains the actual user data.

The implementation relies on a loadable kernel module to prevent excessive changes to the kernel source code and increase portability. One issue with that approach is that morsels of the two largest virtual sizes cannot be created as minimal roots due to problems with the kernel part of the fault handling mechanism. A workaround introduces a memory overhead which is 1 GiB for the largest available morsel with a virtual size of 256 TiB. A callback mechanism called *MMU notifiers* allows decoupling the page table lifetime from the address space lifetime, bypassing the previously mentioned page table sharing problems. The access rights implementation requires minor kernel modifications to allow the corresponding flags in the upper-level page table entries.

Analyzing the prototypic implementation shows that the lazy population of morsels is faster than the population of an anonymous, shared mapping and scales better with multiple threads. Scalability is limited by a reader/writer semaphore protecting the address space. Although only the read lock is actually taken, the frequent cache invalidations decrease performance. The page cache limits the anonymous, shared mapping, which uses a spinlock to synchronize accesses.

Another benchmark addresses the mapping performance. Due to its conceptual design, morsels can be mapped in constant time, while the mapping time of the competitor, SHM, scales linearly with the number of pages. The fast mapping makes morsels suitable for transferring large amounts of data between address spaces. A sender-receiver example shows significant advantages over SHM, and the results are close to the upper limit drawn by the memory bandwidth.

In contrast to the synthetic benchmarks, the in-memory caching software *Memcached* is a suitable real-world application for morsels. It can benefit from persistency because it removes the need for cache reconstruction from disk by keeping the data directly in the main memory. The example demonstrates the easy adaption of existing software, but full NVRAM support would require further changes. The final, overall evaluation points out that the prototypic morsel implementation fulfills the requirements of a modern memory primitive.

6.2 Outlook

The current implementation provides basic functionality, but many open points are still on the way to a fully functional memory primitive. First, the creation of larger morsels as minimal roots should be fixed by modifying the kernel part of the fault handling routine. Another point is the evaluation that only provides results for conventional DRAM. Further tests using NVRAM are required to confirm the compatibility of morsels with persistent memory. Completing the Memcached integration can show the advantages in practical applications.

Besides the raw morsel memory primitive, an upper management layer is required to improve usability. One goal is the elimination of morsel IDs in user applications because it opens a security vulnerability enabling applications to generate morsel IDs referring to arbitrary data. The new control mechanism can be extended with an actual rights management. The current implementation only provides the raw mechanism to restrict access on the hardware level but needs a component that enforces specific rights, e.g., by integrating morsels into the file system using the existing permission control.

The management layer can also improve the basic morsel operations, e.g., verifying a morsel's existence before executing a mapping request. If mapping locations are also tracked in the form of a usage counter, it can prevent the destruction of mapped morsels. This tracking would further allow the implementation of page eviction. Currently, only lazy population is supported, but pages cannot be removed from the morsel surface. The removal requires knowledge about mapping locations because those destructive page table manipulations must be explicitly synchronized with the TLBs.

Another useful feature of persistent morsels would be the ability to specify persistent pointers, even allowing cross-morsel references. In the best case, the developed concept should be compatible with raw pointers to simplify application integration.

The raw morsel memory primitive itself also has room for improvement. Morsels can benefit from huge page support, which reduces the number of page faults, increases TLB coverage, and saves memory due to fewer page table levels. Another feature to bypass known OS bottlenecks is the implementation of explicit population, as shown by Leis et al. [Lei+23]. Batch population can bypass the bottleneck of the reader/writer semaphore faced in Section 5.2.

Furthermore, a complete file system around morsels as an alternative to conventional files on disks can be established. A hardware-supported COW implementation would allow efficient snapshotting to keep multiple file versions. All kinds of processing elements could directly access those in-memory files through the IOMMU. For this purpose, the theoretically discussed IOMMU support must be evaluated in practice.

All considerations within this thesis focused on the x86-64 architecture, but the general concept should be compatible with any architecture that supports paging. Due to its rising relevance in high-performance mobile, desktop, and server systems *ARM* would be another interesting target. The specific implementation will probably be more challenging because the ARM architecture has much more variability than x86-64.

LIST OF ACRONYMS

ADR	asynchronous DRAM refresh
CAS	compare-and-swap
CISC	complex instruction set computer
COW	copy-on-write
CPU	central processing unit
DAX	direct access
DBMS	database management system
DIMM	dual in-line memory module
DMA	direct memory access
DRAM	dynamic random-access memory
FD	file descriptor
FOT	foreign object table
eADR	extended asynchronous DRAM refresh
GPU	graphics processing unit
HBM	high-bandwidth memory
HDD	hard disk drive
I/O	input/output
IOMMU	input-output MMU
IOTLB	input-output translation lookaside buffer
KVM	Kernel-based Virtual Machine
MMU	memory management unit
NUMA	non-uniform memory access
NVM	non-volatile memory
NVRAM	non-volatile RAM
OS	operating system
PFN	page frame number
PML5	page map level 5
PML4	page map level 4

PDPT	page directory pointer table
PD	page directory
PGD	page global directory
PIC	position-independent code
PMD	page middle directory
PT	page table
PTE	page table entry
PUD	page upper directory
P4D	page level 4 directory
RAM	random-access memory
RDMA	remote direct memory access
RISC	reduced instruction set computer
SHM	POSIX shared memory
SSD	solid-state drive
TLB	translation lookaside buffer
VMA	virtual memory area
VPN	virtual page number
WPQ	write pending queue

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

REFERENCES

[ABYY10] Nadav Amit, Muli Ben-Yehuda, and Ben-Ami Yassour. “IOMMU: Strategies for mitigating the IOTLB bottleneck.” In: *International Symposium on Computer Architecture*. Springer, 2010, pp. 256–274. ISBN: 978-3-642-24322-6. DOI: 10.1007/978-3-642-24322-6_22.

[ADAD18] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Version 1.00. Arpaci-Dusseau Books, 2018. URL: http://www.ostep.org (visited on 07/04/2022).

[AMD21] Advanced Micro Devices Inc. *AMD I/O Virtualization Technology (IOMMU) Specification*. Version 3.06. 2021. URL: https://www.amd.com/en/support/tech-docs/amd-io-virtualization-technology-iommu-specification (visited on 09/26/2022).

[ARLB03] A. Al-Rawi, A. Lansari, and F. Bouslama. “A new non-recursive algorithm for binary search tree traversal.” In: *10th IEEE International Conference on Electronics, Circuits and Systems, 2003. ICECS 2003. Proceedings of the 2003*. Vol. 2. 2003, 770–773 Vol.2. DOI: 10.1109/ICECS.2003.1301900.

[ATW20] Nadav Amit, Amy Tai, and Michael Wei. “Don’t Shoot down TLB Shootdowns!” In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys ’20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 8-1-4503-6882-7/2. DOI: 10.1145/3342195.3387518.

[Bao+16] Wenlei Bao et al. “Static and Dynamic Frequency Scaling on Multicore CPUs.” In: *ACM Trans. Archit. Code Optim.* 13.4 (2016). DOI: 10.1145/3011017.

[BC05] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel. From I/O Ports to Process Management*. O’Reilly Media, 2005. ISBN: 978-0-596-00565-8.

[Bit+17] Daniel Bittman et al. *Twizzler: An Operating System for Next-Generation Memory Hierarchies*. Tech. rep. Technical Report UCSC-SSRC-17-01, University of California, Santa Cruz, 2017.

[Bit+20] Daniel Bittman et al. “Twizzler: a Data-Centric OS for Non-Volatile Memory.” In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 2020, pp. 65–80. ISBN: 978-1-939133-14-4. URL: https://www.usenix.org/conference/atc20/presentation/bittman.

[Bit+21] Daniel Bittman et al. “Twizzler: A Data-Centric OS for Non-Volatile Memory.” In: *ACM Trans. Storage* 17.2 (2021). ISSN: 1553-3077. DOI: 10.1145/3454129.

[BL18] Abhishek Bhattacharjee and Daniel Lustig. *Architectural and Operating System Support for Virtual Memory*. Springer Cham, 2018. ISBN: 978-3-031-01757-5. DOI: 10.1007/978-3-031-01757-5.

[Che+18] Tseng-Yi Chen et al. “wrJFS: A Write-Reduction Journaling File System for Byte-addressable NVRAM.” In: *IEEE Transactions on Computers* 67.7 (2018), pp. 1023–1038. DOI: 10.1109/TC.2018.2794440.

[Cor08] Jonathan Corbet. *Memory management notifiers*. 2008. URL: https://lwn.net/Articles/266320/ (visited on 10/19/2022).

[Cor11] Jonathan Corbet. *Transparent huge pages in 2.6.38*. 2011. URL: https://lwn.net/Articles/423584/ (visited on 09/23/2022).

[Cor13] Jonathan Corbet. *Split PMD locks*. 2013. URL: https://lwn.net/Articles/568076/ (visited on 09/23/2022).

[Cor14] Jonathan Corbet. *Supporting filesystems in persistent memory*. 2014. URL: https://lwn.net/Articles/610174/ (visited on 09/24/2022).

[Cor22] Jonathan Corbet. *The ongoing search for mmap_lock scalability*. 2022. URL: https://lwn.net/Articles/893906/ (visited on 10/29/2022).

[Cut18] Ian Cutress. *Intel’s Architecture Day 2018: The Future of Core, Intel GPUs, 10nm, and Hybrid x86. Sunny Cove Microarchitecture: A Peek At the Back End*. 2018. URL: https://www.anandtech.com/show/13699/intel-architecture-day-2018-core-future-hybrid-x86/2 (visited on 07/07/2022).

[Dan22] Albert Danial. *cloc - Count Lines of Code*. 2022. URL: https://github.com/AlDanial/cloc (visited on 11/02/2022).

[Dor19] Dormando. *The Volatile Benefit of Persistent Memory*. 2019. URL: https://memcached.org/blog/persistent-memory/ (visited on 10/30/2022).

[Dor22] Dormando. *Memcached Website*. 2022. URL: https://memcached.org/ (visited on 10/30/2022).

[Dre07] Ulrich Drepper. *What Every Programmer Should Know About Memory*. Version 1.0. Red Hat, Inc., 2007. URL: https://www.akkadia.org/drepper/cpumemory.pdf (visited on 07/07/2022).

[GNU22a] Free Software Foundation, Inc. *GCC, the GNU Compiler Collection*. 2022. URL: https://gcc.gnu.org/ (visited on 10/04/2022).

[GNU22b] Free Software Foundation, Inc. *Using the GNU Compiler Collection (GCC). 3.17 Options for Code Generation Conventions*. 2022. URL: https://gcc.gnu.org/onlinedocs/gcc/Code-Gen-Options.html (visited on 10/04/2022).

[Inf22] InfiniBand Trade Association. *InfiniBand™ Architecture Specification*. 2022. URL: https://www.infinibandta.org/ibta-specification/ (visited on 10/16/2022).

[Int17] Intel Corporation. *5-Level Paging and 5-Level EPT*. Revision 1.1. 2017. URL: https://www.intel.com (visited on 07/07/2022).

[Int19] Intel Corporation. *Brief: Intel® Optane™ Persistent Memory. The Challenge of Keeping Up with Data*. 2019. URL: https://www.intel.de/content/www/de/de/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html (visited on 09/25/2022).

[Int21a] Intel Corporation. *eADR: New Opportunities for Persistent Memory Applications*. 2021. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html (visited on 09/24/2022).

[Int21b] Intel Corporation. *Intel® Optane™ Persistent Memory 200 Series Brief. Achieve Greater Insight From Your Data with Intel® Optane™ Persistent Memory*. 2021. URL: `https://www.intel.de/content/www/de/de/products/docs/memory-storage/optane-persistent-memory/optane-persistent-memory-200-series-brief.html` (visited on 09/25/2022).

[Int21c] Intel Corporation. *Media Alert: Intel to Launch 3rd Gen Intel Xeon Scalable Portfolio*. 2021. URL: `https://newsroom.intel.com/news-releases/media-alert-launch-3rd-gen-intel-xeon-scalable-portfolio/` (visited on 07/08/2022).

[Int22a] Intel Corporation. *Intel Virtualization Technology for Directed I/O. Architecture Specification*. Version 4.0. 2022. URL: `https://software.intel.com/sites/default/files/managed/c5/15/vt-directed-io-spec.pdf` (visited on 09/27/2022).

[Int22b] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Apr. 2022. URL: `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html` (visited on 07/07/2022).

[Int22c] Intel Corporation. *Product Guide - Intel® Ethernet 800 Series*. 2022. URL: `https://cdrdv2.intel.com/v1/dl/getContent/709766` (visited on 10/16/2022).

[Int84] Intel. *iRMX 86 Application Loader Reference Manual*. Version Rev 6. 1984. URL: `https://archive.org/details/bitsavers_inteliRMXi196BurstiRMX86ApplicationLoaderReference_2401611` (visited on 10/04/2022).

[Izr+19] Joseph Izraelevitz et al. "Basic performance measurements of the intel optane DC persistent memory module." In: *arXiv preprint arXiv:1903.05714* (2019).

[KBS19] Chandan Kalita, Gautam Barua, and Priya Sehgal. "DurableFS: a file system for NVRAM." In: *CSI Transactions on ICT* 7.4 (2019), pp. 277–286. DOI: `10.1007/s40012-019-00215-0`.

[KDI20] Alex Kogan, Dave Dice, and Shady Issa. "Scalable Range Locks for Scalable Address Spaces and Beyond." In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 978-1-4503-6882-7. DOI: `10.1145/3342195.3387533`.

[Ker22a] Kernel Development Community. The Linux Kernel Documentation: 5-level paging. 2022. URL: `https://www.kernel.org/doc/html/latest/x86/x86_64/5level-paging.html` (visited on 07/09/2022).

[Ker22b] Kernel Development Community. The kernel's command-line parameters. 2022. URL: `https://www.kernel.org/doc/html/v5.17/admin-guide/kernel-parameters.html#` (visited on 10/26/2022).

[Ker22c] Kernel Development Community. *Split page table lock*. 2022. URL: `https://www.kernel.org/doc/html/v5.7/vm/split_page_table_lock.html` (visited on 09/23/2022).

[Kli16] Evan Klitzke. *The Curious Case of Position Independent Executables*. 2016. URL: `https://eklitzke.org/position-independent-executables` (visited on 10/04/2022).

[Kut18] Oleg Kutkov. *Simple Linux character device driver*. 2018. URL: `https://olegkutkov.me/2018/03/14/simple-linux-character-device-driver/` (visited on 10/19/2022).

[Lam13] Christoph Lameter. "An Overview of Non-Uniform Memory Access." In: *Communications of the ACM* 56.9 (2013). DOI: `10.1145/2500468.2500477`.

[Lan22] Niklas Lang. *Database Basics: ACID Transactions. Understanding the ACID properties of Databases*. 2022. URL: `https://towardsdatascience.com/database-basics-acid-transactions-bf4d38bd8e26` (visited on 09/26/2022).

[Lei+23] Viktor Leis et al. “Virtual-Memory Assisted Buffer Management.” In: *Proceedings of the ACM SIGMOD/PODS International Conference on Management of Data (SIGMOD’23)*. Accepted at SIGMOD’23, to appear. Seattle, WA, USA: ACM, June 2023.

[Lin22a] Linux. *mmap(2) — Linux manual page*. 2022. URL: `https://man7.org/linux/man-pages/man2/mmap.2.html` (visited on 07/07/2022).

[Lin22b] Linux. *pipe(2) — Linux manual page*. 2022. URL: `https://man7.org/linux/man-pages/man2/pipe.2.html` (visited on 10/18/2022).

[Lov10] Robert Love. *Linux Kernel Development*. 3rd ed. Addison-Weseley, 2010. ISBN: 978-0-672-32946-3.

[McC06] Dave McCracken. “Shared page tables redux.” In: *Linux Symposium*. 2006, p. 117.

[McK07] Paul E. McKenney. “Memory Ordering in Modern Microprocessors.” In: (2007). URL: `http://www.rdrop.com/users/paulmck/scalability/paper/ordering.2007.09.19a.pdf` (visited on 11/06/2022).

[Mod18] Archit Modi. *An introduction to pipes and named pipes in Linux*. 2018. URL: `https://opensource.com/article/18/8/introduction-pipes-linux` (visited on 10/27/2022).

[Red13] Redis. *memtier_benchmark: A High-Throughput Benchmarking Tool for Redis & Memcached*. 2013. URL: `https://redis.com/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached/` (visited on 10/30/2022).

[Sca20] Steve Scargall. *Programming Persistent Memory. A Comprehensive Guide for Developers*. 1st ed. Apress Berkeley, CA, 2020. ISBN: 978-1-4842-4932-1. DOI: `10.1007/978-1-4842-4932-1`.

[SGG13] Abraham Silberschatz, Peter Bear Galvin, and Greg Gagne. *Operating System Concepts*. 9th ed. Wiley, 2013. ISBN: 978-1-118-06333-0.

[Shu16] Kirill A. Shutemov. *[RFC, PATCHv1 00/28] 5-level paging*. Linux kernel mailing list. 2016. URL: `https://lore.kernel.org/lkml/20161208162150.148763-1-kirill.shutemov@linux.intel.com/` (visited on 07/09/2022).

[TB15] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. 4th ed. Pearson, 2015. ISBN: 978-0-13-359162-0.

[TEL95] D.M. Tullsen, S.J. Eggers, and H.M. Levy. “Simultaneous multithreading: Maximizing on-chip parallelism.” In: *Proceedings 22nd Annual International Symposium on Computer Architecture*. 1995, pp. 392–403. ISBN: 0-89791-698-0.

[WKC22] Ying-Jan Wu, Ching-Yu Kuo, and Li-Pin Chang. “iNVMFS: An Efficient File System for NVRAM-Based Intermittent Computing Devices.” In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2022). DOI: `10.1109/TCAD.2022.3197485`.

[Wre22] Lars Wrenger. “Lo(ck|g)-free Page Allocator for Non-Volatile Memory in the Linux Kernel.” MA thesis. Leibniz Universität Hannover, 2022.

[WW09] P. Weisberg and Y. Wiseman. “Using 4KB page size for Virtual Memory is obsolete.” In: *2009 IEEE International Conference on Information Reuse & Integration*. 2009, pp. 262–265. DOI: `10.1109/IRI.2009.5211562`.

[Yas22] Aqsa Yasin. *POSIX Shared Memory with C Programming*. 2022. URL: `https://linuxhint.com/posix-shared-memory-c-programming/` (visited on 10/27/2022).

[ZGF21] Kaiyang Zhao, Sishuai Gong, and Pedro Fonseca. “On-Demand-Fork: A Microsecond Fork for Memory-Intensive and Latency-Sensitive Applications.” In: *Proceedings of the Sixteenth European Conference on Computer Systems*. EuroSys ’21. Online Event, United Kingdom: Association for Computing Machinery, 2021, pp. 540–555. ISBN: 978-1-4503-8334-9. DOI: 10.1145/3447786.3456258.

APPENDIX

```
$ du -k -d 1 --apparent-size . | sort -n
34      ./certs
47      ./usr
182     ./init
231     ./LICENSES
232     ./virt
245     ./ipc
1201    ./samples
1769    ./block
2783    ./scripts
2802    ./security
3197    ./crypto
4442    ./mm
6281    ./lib
10904   ./kernel
31382   ./net
36378   ./tools
36688   ./include
40212   ./Documentation
40346   ./sound
41622   ./fs
102090  ./arch
730163  ./drivers
1094059 .
```

Listing A.1 – Linux kernel 5.17 source code structure: Every line states the accumulated size of a subdirectory in KiB. We can see that the hardware-independent part of the memory subsystem located in `./mm` is as big as 40% of the core functionalities located in `./kernel`. Additionally, many other parts of the kernel outside the `./mm` directory also implement functions closely related to memory management, e.g. the implementation of fork (`/kernel/fork.c`).

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
        ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
        int (*iopoll)(struct kiocb *kiocb, struct io_comp_batch *,
                        unsigned int flags);
        int (*iterate) (struct file *, struct dir_context *);
        int (*iterate_shared) (struct file *, struct dir_context *);
        __poll_t (*poll) (struct file *, struct poll_table_struct *);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        unsigned long mmap_supported_flags;
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, loff_t, loff_t, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
        int (*check_flags)(int);
        int (*flock) (struct file *, int, struct file_lock *);
        ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
        int (*setlease)(struct file *, long, struct file_lock **, void **);
        long (*fallocate)(struct file *file, int mode, loff_t offset,
                                loff_t len);
        void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
        unsigned (*mmap_capabilities)(struct file *);
#endif
        ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
                        loff_t, size_t, unsigned int);
        loff_t (*remap_file_range)(struct file *file_in, loff_t pos_in,
                                        struct file *file_out, loff_t pos_out,
                                        loff_t len, unsigned int remap_flags);
        int (*fadvise)(struct file *, loff_t, loff_t, int);
} __randomize_layout;
```

Listing A.2 – struct file_operations from /include/linux/fs.h (kernel 5.17)

```
$ cloc . --exclude-dir=.cache,.clang-format
       9 text files.
       9 unique files.
       1 file ignored.

github.com/AlDanial/cloc v 1.90  T=0.01 s (731.7 files/s, 368454.9 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C                                4            725           1219           2183
C/C++ Header                     4             86             66            212
make                             1             10              5             26
-------------------------------------------------------------------------------
SUM:                             9            821           1290           2421
-------------------------------------------------------------------------------
```

Listing A.3 – Analysis of Kernel Module Implementation: Lines of Code using the `cloc` tool [Dan22]