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

Leibniz
Universität
Hannover

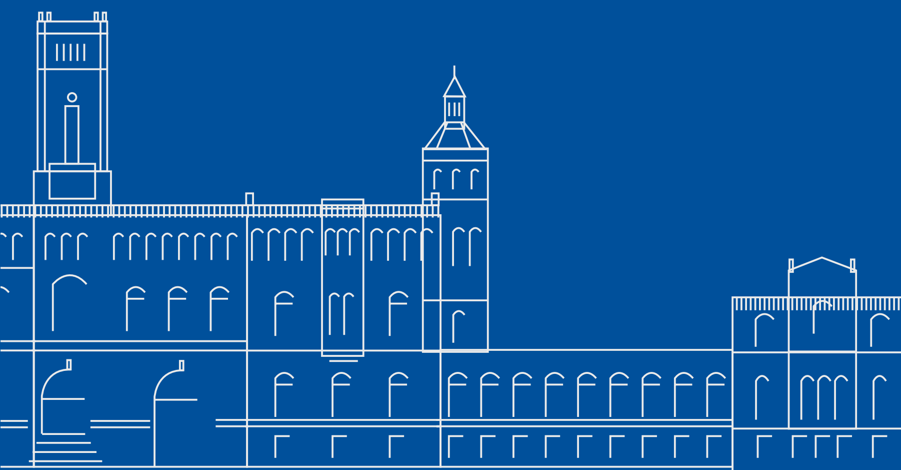
Tim Hollmann

Page Eviction for Self-Contained Virtual-Memory Objects in the Linux Kernel

Masterarbeit im Fach Informatik

03. Juli 2024

Please cite as:
Tim Hollmann, "Page Eviction for Self-Contained Virtual-Memory Objects in the Linux Kernel"
Master's Thesis, Leibniz Universität Hannover, Institut für Systems Engineering, January 1980.



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

www.sra.uni-hannover.de

Page Eviction for Self-Contained Virtual-Memory Objects in the Linux Kernel

Masterarbeit im Fach Informatik

vorgelegt von

Tim Hollmann

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. Jan Simon Rellermeyer**
Betreuer:	**Alexander Halbuer, M.Sc.**

Beginn der Arbeit:	**3. Januar 2024**
Abgabe der Arbeit:	**3. Juli 2024**

Erklärung

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

Declaration

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

(Tim Hollmann)
Hannover, 03. Juli 2024

ABSTRACT

With increasing memory heterogeneity, modern systems challenge established concepts of low-level memory management in the kernel, driving the need for new abstractions. Morsels, self-contained indivisible virtual memory objects, render such a new memory management primitive. When mounted into a virtual address space, morsels offer a surface which is initially not backed by physical memory. The underlying physical memory for data pages and paging structures gets allocated either transparently on the access path (lazy population) or via an explicit population operation.

While it is possible to populate the morsel surface, the reverse is currently not possible. At this time, the only way to reclaim memory from a morsel is to destroy it.

Traditional methods of page frame reclamation do not apply to morsels, the kernel has no way to actively deallocate memory from morsels, so they currently represent a sink for what should be a reusable system resource. Just by accessing more pages on the morsel surface than the system has physical page frames available gets the system out-of-memory. The implementation of eviction requires invalidation of translation lookaside buffer entries, which in turn requires reverse mapping morsel pages into all virtual addresses resolving to them, which is currently not possible due to lack of knowledge about the mapping locations.

This thesis makes two contributions. The first addresses the lack of eviction, and is the design and implementation of a novel morsel operation that enables the eviction of underlying physical memory on a given range on a morsel's surface. This is supported by the second contribution, the design and implementation of a mapping tracking mechanism, which implements reverse mapping on morsel-granularity, on top of which precise translation lookaside buffer flushing is possible.

The prototypical implementation proves being functional while causing no degradation of functionality or performance, but is also competitive performance-wise, significantly outperforming eviction on POSIX shared memory.

KURZFASSUNG

Mit zunehmender Speicherheterogenität stellen moderne Systeme die etablierten Konzepte der Speicherverwaltung auf niedriger Ebene im Kernel vor Herausforderungen, so dass neue Abstraktionen gefragt sind. Morsel, in sich geschlossene, unteilbare virtuelle Speicherobjekte, stellen eine solche neue Speicherverwaltungs-Primitive dar. Wenn sie in einen virtuellen Adressraum eingeblendet werden, bieten Morsel eine virtuelle Oberfläche, die zunächst nicht mit physischem Speicher hinterlegt ist. Der zugrunde liegende physische Speicher für Daten-Seiten und Paging-Strukturen wird entweder transparent auf dem Zugriffspfad alloziert (Lazy Population) oder durch eine explizite Populations-Operation.

Während es so möglich ist, die Morsel-Oberfläche zu besetzen, ist der umgekehrte Weg derzeit nicht möglich. Gegenwärtig ist die einzige Möglichkeit, Speicher von einem Morsel zurückzugewinnen, es zu zerstören. Da herkömmliche Methoden der Kachelrückgewinnung nicht für Morsel gelten, hat der Kernel keine Möglichkeit, aktiv Speicher von einem Morsel zu deallozieren, somit stellen sie aktuell eine Senke für ein eigentlich auf Wiederverwendbarkeit ausgelegtes Betriebsmittel dar. Allein der Zugriff auf mehr Seiten auf der Morsel-Oberfläche als das System physische Seitenrahmen zur Verfügung hat, reicht aus, damit dem System der Speicher ausgeht. Die Implementierung von Eviction erfordert Invalidierung von Einträgen im Translation-Lookaside-Puffer, was wiederum eine umgekehrte Abbildung von Morsel-Seiten auf alle virtuellen Adressen, die auf sie auflösen, erfordert. Jedoch ist dies aktuell nicht möglich, da Wissen über die Einblende-Positionen fehlt.

Die vorliegende Arbeit leistet zwei Beiträge. Der erste Beitrag behandelt das Problem der fehlenden Rückgewinnung und ist der Entwurf und die Implementierung einer neuartigen Morsel-Operation, die die Entfernung des zugrunde liegenden physischen Speichers in einem bestimmten Bereich auf der Oberfläche eines Morsels ermöglicht. Dies wird durch den zweiten Beitrag unterstützt, dem Entwurf und der Implementierung eines Mapping-Tracking-Mechanismus, der eine umgekehrte Abbildung auf Morsel-Granularität implementiert, auf dessen Grundlage ein präzises Entfernen von Einträgen aus den Translation-Lookaside-Puffern möglich ist.

Die prototypische Implementierung erwies sich als funktionsfähig und verursachte keine Beeinträchtigung der Funktionalität oder Leistung. Sie ist auch in Bezug auf die Leistung konkurrenzfähig und übertrifft die Eviction auf POSIX Shared Memory deutlich.

CONTENTS

Abstract v

Kurzfassung vii

1 Introduction 1

2 Fundamentals 3
- 2.1 Virtual Memory Management 3
  - 2.1.1 Paging 3
  - 2.1.2 Reverse Mapping Anonymous Memory in Linux 4
- 2.2 Translation Lookaside Buffers 5
  - 2.2.1 TLB invalidation 6
  - 2.2.2 Linux TLB Flushing Interface 8
- 2.3 Read-Copy-Update 9
- 2.4 Morsels 9
  - 2.4.1 Concept 10
  - 2.4.2 Morsel Lifecycle 10
  - 2.4.3 Morsel Mapping 11
  - 2.4.4 Morsel Population 11
  - 2.4.5 Huge Pages 11
  - 2.4.6 Copy-on-Write 12
- 2.5 Related Work 13
  - 2.5.1 Page Table Sharing 13
  - 2.5.2 Approaches to the TLB Consistency Problem 13
  - 2.5.3 Morsels 14

3 Architecture 15
- 3.1 Depopulation Operation 15
  - 3.1.1 Basic Operation Definition 15
  - 3.1.2 Discussion: About Flushes, Batches and dynamic Memory 16
  - 3.1.3 Chosen Architecture in this Thesis 19
  - 3.1.4 Discussion: Determining the Pages to Evict 20
- 3.2 Eviction 20
  - 3.2.1 Punching Holes into the Morsel Surface 20
  - 3.2.2 Eviction via Lock-Free Subtree Pruning 21
  - 3.2.3 Worst-case Amount of Pruned Subtrees 24

3.2.4 COW aware Subtree Pruning . . . 25
3.3 Precise TLB Flushing . . . 27
3.3.1 Need for Reverse Mapping . . . 28
3.3.2 Reverse Mapping via Tracking of Morsel Mappings . . . 28
3.4 Morsel Mapping Tracking Subsystem . . . 29
3.5 Clean-Up . . . 30
3.6 Summary . . . 31

4 Implementation **33**
4.1 General Structure . . . 33
4.2 A new System Call for Depopulation . . . 34
4.2.1 Specifying the Eviction Range . . . 34
4.2.2 Permissions . . . 35
4.3 Eviction . . . 35
4.4 Morsel Gather . . . 35
4.5 Morsel Mapping Tracking Subsystem . . . 36
4.6 Securing the Destroy Operation . . . 37
4.7 Safe Deallocation . . . 37
4.8 Summary . . . 38

5 Analysis **41**
5.1 Evaluation Platform . . . 41
5.2 Functional Evaluation . . . 41
5.2.1 Conservative Criterion . . . 41
5.2.2 Progressive Criterion . . . 42
5.3 Performance Evaluation . . . 43
5.3.1 Influence on Mapping Speed . . . 43
5.3.2 Microbenchmarks of Depopulation Subroutines . . . 44
5.3.3 Depopulation Speed compared to POSIX Shared Memory . . . 46
5.4 Summary . . . 48

6 Conclusion **51**
6.1 Summary . . . 51
6.2 Future Work . . . 51

Lists **53**
List of Acronyms . . . 53
List of Figures . . . 55
List of Tables . . . 57
List of Algorithms . . . 59
Bibliography . . . 61

1 INTRODUCTION

Memory virtualization is one of the core tasks of modern operating systems. Nowadays, both sides of this traditional memory virtualization layer become more and more diverse, forming heterogeneous memory systems (HMSs). This thesis resides in the context of the Parallel Persistency OS (ParPerOS) project, which, in the presence of heterogeneous memory systems, aims to find new abstractions for low-level memory management in the kernel [DL]. In this context, Halbuer designed *morsels*, which are novel fully self-contained, indivisible virtual memory objects that build on top of the page table sharing paradigm [Hal22]. They purposely sacrifice hardware independence for the sake of performance gains through memory management unit (MMU)-interpreted data structures, may optionally reside on non-volatile memory, can be mapped in constant time and arbitrarily often. They shift the focus of memory management from individual pages to larger objects, while still managing to avoid internal fragmentation through sparse population.

Morsels currently suffer from two limitations that are connected with each other through the presence of the translation lookaside buffer (TLB): Lack of knowledge regarding mapping locations, and as a result the disability to destructively alter the morsel page tables while the morsel is mapped, including the eviction of allocated memory. Let us take a closer look. A morsel, by design, might be mapped into multiple address spaces simultaneously. If the morsel page tables happened to be altered in a way that invalidates the cached information in the TLB, such as the page frame number (PFN), presence or more restrictive access controls, this requires an immediate invalidation of the respective cache entries. TLB invalidation can be, depending on the concrete flush type and the necessity of remote shootdowns, an expensive operation. Because we don't want to use the most collateral flush type, we need information about the mapping locations of the morsel in order to provide the virtual addresses required by the hardware. This information is currently not available. Therefore, certain changes to morsels are currently not possible without full TLB flush, and are therefore not implemented yet. This includes the relocation of physical memory, swapping, huge-page merging, restrictive changing of access controls and eviction. The latter is interesting, because on the one side there is no way to deallocate memory off a morsel (unless destroying it), and on the other side it is well possible to populate it, either by lazy or explicit population. This leads to the situation that a user can get the system out of memory by just accessing memory on the surface of a morsel. The disability to change access controls to be more restrictive hinders the copy-on-write (COW) implementation from Fistanto to re-map the COW-originals read-only for morsels with more than one mapping location [Fis24].

As the first contribution, this thesis designs and implements a novel morsel operation that enables the user to explicitly evict a given range on a morsel surface and therefore is able

to deallocate memory off a morsel without destroying it. This inverts the existing explicit population operation. Special challenges lie in the fact that this is performed in a lock-free manner, in parallel to other morsel operations and potential lazy population. To support the eviction, a second contribution of this thesis is a mapping tracking mechanism that addresses the lack of knowledge regarding mapping locations and therefore enables precise TLB flushes, which in turn not only enables eviction, but also the other mentioned modifications, like general COW or swapping.

2 FUNDAMENTALS

This chapter forms the foundation of this thesis and is intended to provide the reader with the necessary knowledge to be able to follow the problem statement, the challenges faced and the solutions proposed in this thesis.

2.1 Virtual Memory Management

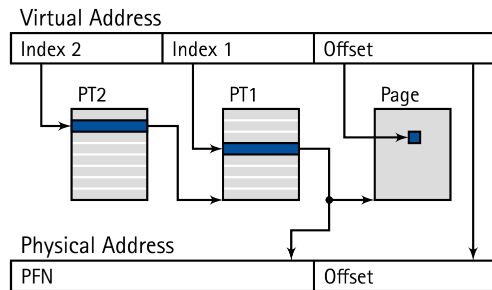
Virtual memory is a fundamental abstraction provided by the operating system (OS). For every process, the OS configures the MMU hardware to create the illusion of an exclusive, contiguous and very large address space, the *virtual address space*. The *mapping* of virtual to physical addresses then limits what is accessible by a process, which is used to isolate processes from each other and the OS. The management of virtual memory is a privileged operation reserved to the OS. Virtual address spaces can be sparsely populated and the allocation of corresponding physical memory deferred until access. It furthermore allows the transparent relocation of contents on physical memory, or even *swapping* it out. For a virtual address, the underlying memory provider can be arbitrary, from non-volatile memory (NVM), high-bandwith memory (HBM) over virtual objects like files to memory-mapped I/O devices. On the other side of the virtualization layer, the consumer of the virtual memory also does not need to be CPU, but can also well be a remote direct memory access (RDMA)-enabled network card, for example. Virtual memory allows heterogeneous memory systems to be homogeneous towards the user.

Two implementations of virtual memory are *segmentation* and *paging*, with paging being the most prevalent on modern systems [TB24] and which is discussed in the following section.

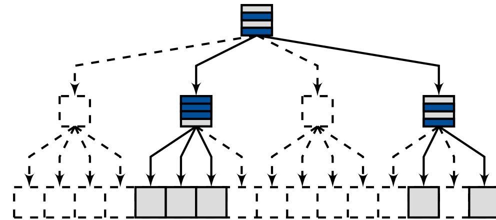
2.1.1 Paging

Paging divides the physical and virtual memory into equally sized chunks, the *page frames* in physical memory and *pages* in virtual memory, which can be mapped onto each other [TB24]. A virtual address consists of an upper part, its virtual page number (VPN), and a lower part, the offset into the translated page frame. A page frame is uniquely identified by its PFN.

The OS manages the structures that control the translation of pages to page frames, the *page tables (PTs)*. Depending on the levels of PT indirection (*paging level*) n, the VPN is split up into n parts used to index into the n different page tables. Figure 2.1a shows an example of this for $n = 2$. The most common paging level on modern AMD64 systems is $n = 4$ [AMD23]. With 4 KiB page size, 512 entries per PT, this allows to address $4\ \text{KiB} \cdot 512^4 = 256$ TiB memory. Modern hardware may support $n = 5$, which is called *5-level paging* and illustrated in figure 2.2. This



(a) Example of two level paging. The VPN of the virtual address is split into two parts, which each index into a page table. The last page table points to the page frame, which is then indexed by the offset. Image by [Hal22].



(b) Example of how a sparsely populated paging hierarchy can save physical memory. Each dashed square references to a page frame that does not need to be allocated. Image by [Hal22].

allows to address 4 KiB $\cdot$ $512^5 = 128$ PiB. The page tables[1] form a tree, the page table hierarchy, with pages as leafs. The address translation is performed, transparently to the process, by a hardware called the memory management unit (MMU). As the process of address translation involves following a chain of tables, this is called *table walk*. Every MMU table walk requires to read up to n page table entries. If every memory access necessitated a table walk, paging would become unusably slow [ADAD23]. The common solution to this is to use a cache for address translations, which is discussed further in section 2.2.

The translation might fail at any level of table walk, if a non-present PT entry is met. In this case, a *hard miss*, the MMU raises a *page fault*. Because it is possible to have non-present entries on any level of the paging hierarchy, the virtual address space can be sparsely populated (see figure 2.1b), which allows to minimize the amount of allocated page frames used to hold data pages or PTs. The OS can use the page fault mechanism to transparently allocate and map a page frame for a page in the moment that it is being accessed. In the page fault handler, it allocates a page frame and maps it such way that a the accessed address translates to the allocated page. This is called *demand paging*.

Paging allows further to share memory between address spaces, by mapping pages from different virtual address spaces onto the same page frame. This allows to hold only one copy of, for example, a library in physical memory and transparently sharing it read-only to multiple address spaces. Explicit memory sharing can be used as an inter-process communication (IPC) channel. A special kind of transparent sharing is copy-on-write (COW), a scheme which creates virtual copies a memory region by creating additional references on the original and re-mapping the original read-only. This defers the actual copying to the time of a write access, reducing both latency during the copy and usage of physical memory [ADAD23].

2.1.2 Reverse Mapping Anonymous Memory in Linux

In order to evict a shared anonymous page, the Linux memory management subsystem must find all mapping locations of that respective page. This process is called *reverse mapping*.

The first step is to resolve the PFN via `pfn_to_page()` to its `struct page`. The next step is to decide which type of memory it is, since this determines how the `.mapping` pointer in the `struct page` is

[1] Depending on their level in the page table hierarchy, page tables may be named differently by different nomenclatures. This thesis avoids sticking to any of them and just refer to them as page tables and use integers to reference a certain level, with 0 being leaf pages.

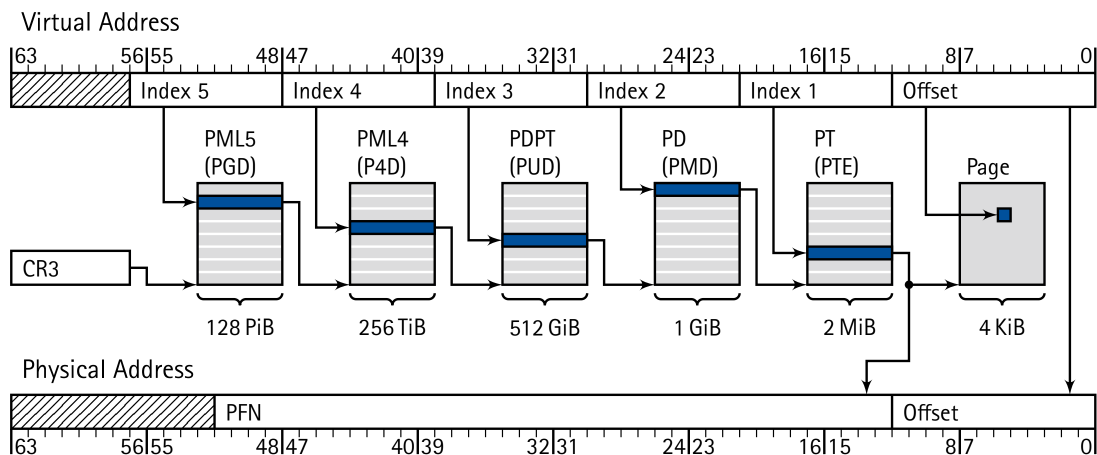


Figure 2.2 – Scheme of address translation on modern systems with 5-Level-Paging. Image by Halbuer [Hal22].

to be interpreted. While for *file-backed* memory, this is a `address_space*`, in case of *anonymous* memory this has to be cased to a `anon_vma*`. The pointed to `anon_vma`, which itself is part of a tree, then has a member `.rb_root`, which is the root of a red black tree containing `anon_vma_chains` that belong to it. These are intermediate chain-structures to connect a single `anon_vma` with a single `vma`, therefore the desired virtual memory area (VMA) can be reached via `.vma`. From there on, the only thing left is to determine the virtual address of the page frame in the VMA, by using `page_address_in_vma()`. Figure 2.3 visualizes this process.

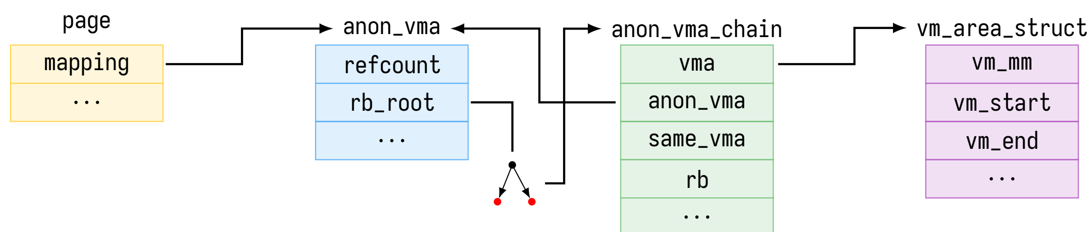


Figure 2.3 – Visualization reverse mapping anonymous memory in Linux. A page frame, via its `struct page`, is translated a VMA.

We can see that this kind of reverse mapping requires both a `struct page` as well as properly maintained linkages into the various complex kernel data structures, which also requires appropriate locking. Both is not the case for morsels. Therefore, we have to design a new way for reverse mapping of morsel page frames in section 3.3.

2.2 Translation Lookaside Buffers

The implementation of virtual memory based on pages requires address translation on every memory related instruction. Without further optimizations, this process involves multiple memory accesses and therefore renders a significant overhead [ADAD23]. With more and more

levels being added to the paging hierarchy, the address translation becomes even more expensive. Since spatial and temporal locality of instruction and data fetches can often be observed, using a cache here is predestined to solve this problem.

For this reason, paging-enabled architectures usually allow to cache the results of address-translations on a processor, which is the case for IA-32, Intel64 and AMD64 [AMD23, p. 157][Int23, Vol. 3A 4-41]. The cached information may include VPN, PFN and a logical AND of the access bits [Int23, Vol. 3A 4-41]. The idea is to speed up future address translations for the same VPN through a cache hit (*fast path*), thereby avoiding the time-consuming MMU table walk (*slow path*).

A cache miss can be handled, depending on the architecture, in hardware or in software, therefore the caches are called *hardware-managed* (common on CISC, like x86) or *software-managed* (common on RISC, like MIPS R10k or SPARC v9) [ADAD23]. Although it is technically left open, the RISC-V Privileged Architecture specification advises against implementing software-managed TLBs, because "Software TLB refills are a performance bottleneck on high-performance systems [...]" [WAH21, p. 79]. In case of hardware-managed caches, the MMU itself handles the cache miss by performing the address translation and in case of a translation success, insert the result into the cache and repeat the instruction which now results in a cache hit. This requires hardware-interpretable page tables. An advantage of hardware managed caches is that it can perform the table walk faster than this could be done software. Furthermore, the OS is only activated if the table walk fails or detects invalid access due to permissions. This follows the concept of a sleeping OS, which sets things up and is not active during normal operation. In case of software-managed caches, the MMU raises a *Miss Exception* to the OS, which is handled by a special trap handler. The advantage of software-managed caches is that they are more flexible in the regard that the paging structures are not defined by hardware, the OS is free to implement the address translation process in any kind it wants, and also might dynamically adapt the process any time, without requiring change in hardware. Furthermore the hardware is much simpler, it just raises an exception. In the following, we will only consider hardware-managed caches, as this is the case of x86-64.

The caching is realized by hardware called the translation lookaside buffers (TLBs), which are part of the processor's memory management units (MMUs). Usually, the TLB entries are rotated in a least recently used (LRU) fashion, therefore TLBs hold the respective last accessed VPNs of that processor, however also other algorithms for TLB entry management exist, as this is a detail up to the implementation. The TLB may be implemented to be accessed by the stored VPN, therefore it is a content-addressable memory (CAM). Just like other caches, it is possible to hierarchically order multiple layers of TLBs with different characteristics regarding size and speed into a cache pyramid. Also, there usually are separate TLBs depending on the fetch target, like instruction translation lookaside buffers (iTLBs) and data translation lookaside buffers (dTLBs), which is called a *split* TLB. Usually there are also separate TLBs for different page sizes, or TLBs shared among certain sizes (like a 4 KiB dTLB and a shared 2 MiB/1 GiB dTLB).

It is up to the processor manufacturer if and how the caching is implemented and this is an active area of performance optimization. TLBs therefore can be found in a wide variety.

2.2.1 TLB invalidation

In the event that the cached information gets outdated by a change to the original paging structures, for example if a page is logically removed from an address space or a page is changed regarding its access controls, the corresponding TLB entry *should* be invalidated. We call outdated

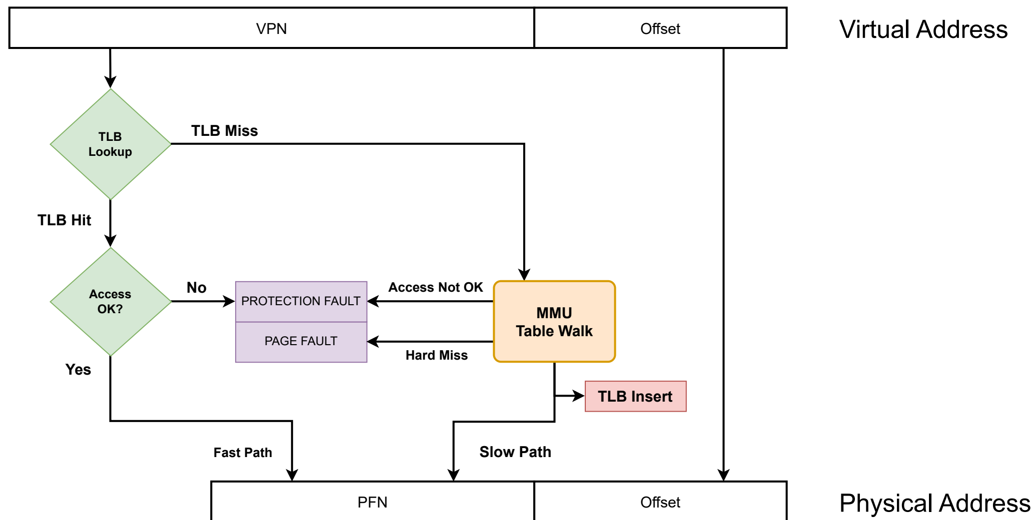


Figure 2.4 – Scheme of address translation involving a translation lookaside buffer. The address is first decomposed into the VPN and the offset. The MMU then performs a TLB lookup, which results in either a hit or a miss. In case of a miss, the PFN can be read from the TLB entry. It further checks the attempted access type against the stored resolved permissions, potentially raising a protection fault. In case of a miss, a table walk is performed, which is depictured in more detail in figure 2.2. The table walk either exits successfully with a PFN or may raise faults like protection fault or page fault. The PFN is inserted into the TLB for possible reuse, typically displacing the least recently used entry.

TLB entries *stale*. There is, in contrast to other types of caches, usually no coherence provided by hardware [ATW20], so it is up to the software to ensure the absence of stale TLB entries every time it updates paging structures. Because TLBs are core-local, this includes invalidation of TLB entries on remote cores, e.g. via inter-processor interrupts (IPIs), which is called TLB *shootdown* [ATW20], while the invalidation of local TLB entries is called a TLB *flush*. Because there is usually no way to check for entry presence, TLBs invalidation is performed by suspicion. A usual optimization is to reduce the set of invalidated TLBs by those that can not possibly hold the entries to invalidate. Another optimization is to *batch* independent flushes on remote shootdowns [LF23].

Local invalidation on x86-64

In the following, we will give a brief overview how the TLB can be invalidated for the local processor (*flushed*) by software on the x86-64 architecture [Int23, Vol. 3A 4-48].

- **`MOV` to `CR3`.** This is the traditional address space switch and it makes sense to flush the local TLB in this case, because otherwise the switched-to address space may be able to access page frames via TLB hits that it should not be able to, breaking isolation. However, writing to `CR3` does not necessarily flush the whole TLB, depending on the usage of process-context identifiers (PCIDs), indicated by the `CR4.PCIDE` (`PCID`-enable) flag. If enabled, the CPU identifies the current context with a 12 bit value and the TLB can *tag* entries with it TLB *tagging*, which allows preserving TLB entries across context switches [Maa+20].

- **`INVLPG` instruction.** Invalidates all TLB entries with a VPN that equals that of a given linear address and the *current* PCID. It also invalidates all global TLB entries for the VPN, regardless of PCID. This realizes TLB flushing on page-granularity for the current PCID and is thereby quite precise.
- **`INVPCID` instruction.** This is an all-rounder for *non-current* PCIDs. It accepts a flush *type*, and an optional PCID and linear address in a *descriptor* operand. It can either flush individual pages for the given VPN and PCID, everything for the given PCID or unconditionally everything (with or without global entries).
- Full flushes in the cases of disabling paging at all (`CR0.PG=0`), disabling of global pages (`CR4.PGE=0`) or disabling PCIDs (`CR4.PCIDE=0`), task switches or VMX transitions.

2.2.2 Linux TLB Flushing Interface

Because it will be used in the prototypical implementation in section 4, the TLB Flushing interface of Linux is briefly presented in the following.

In Linux terminology, the term *flush* refers to both flushes and shootdowns. The Linux kernel provides an internal interface for TLB invalidation [Mil]. It should be noted that this interface is architecture agnostic. It targets a cross-architecture concept of a TLB and is implemented by the architecture specific code in order to fulfull the defined criteria. On SMP systems, this concept is extended to all cores and is up to the interface implementation to take care of coherence using remote invalidations (*shootdowns*).

The following list is based on Linux 6.1 source code and the interface documentation [Mil].

- `void flush_tlb_all(void)`

  After invocation of this interface, all modifications in the paging structures are guaranteed to be visible to the CPU, which implies the absence of TLB entries. This is the most collateral flush type.
- `void flush_tlb_mm(struct mm_struct *mm)`

  After invocation of this interface, no TLB entries exist for the provided address space.
- `void flush_tlb_page(struct vm_area_struct *vma, unsigned long addr)`

  After invocation of this interface, no TLB entries for the VPN of the provided address exist.
- `void flush_tlb_range(struct vm_area_struct *vma, unsigned long start, unsigned long end)`

  After invocation of this interface, no TLB entries exist for the provided virtual address range.

The respective address space can be retrieved from the provided VMA as `.vm_mm`. The VMA is provided because in case of architectures with software-managed TLBs, it is relevant if the region is executable (`vma->vm_flags & VM_EXEC`) and thus either the iTLB or dTLB is concerned.

The `flush_tlb_range()` interface allows to take advantage if the architecture may have more efficient methods available for range flushing than invalidating each affected single page individually.

If the flush type is limited, e.g. to a certain address spaces, the implementation can conduct further optimizations like reducing the IPI targets to those CPUs that actually once had the address space active. This is information available in the `mm_struct`'s `.cpu_bitmap` bitmap, interpreted by `mm_cpumask()`.

Since Linux version 4.14, Linux employs address space identifiers (ASIDs) [Maa+20], which translate to use the underlying PCIDs feature on x86-64, resulting in tagged TLB entries. The x86-64 reserves only 12 bits per PCID with 0 being a special value, allowing for 4095 contexts per-processor. Because this is not enough for global process identification, Linux assigns recycled ASIDs to processes on a CPU basis. With enabled kernel page-table isolation (KPTI), processes have two versions of kernel memory, which can be cheaply switched in between with the help of ASIDs tagged TLBs [Cor17], which reduces the amount of usable ASID pairs, and therefore processes, to 2047.

Another technique employed in Linux is *Lazy TLB*, which avoids TLB flushes when switching the context to processes that *only* access kernel memory (like kernel threads), using that kernel memory is visible in all address spaces at the same location [Gor07]. An address space therefore can be *borrowed* by anonymous users who switch to it without TLB flushes. This is why the `mm_struct` has different counters for actual *users* (`mm_users`) *usages* (`mm_count`), the latter including anonymous users.

Another mechanism is the use of *global* TLB entries for the kernel portion of every address space. These are TLB entries that survive the normal address space switch by `MOV` to `CR3`. This way, CPUs can transparently share TLB entries when accessing kernel memory in different address spaces.

2.3 Read-Copy-Update

The implementation will make use of the read-copy-update (RCU) mechanism in order to synchronize access to a complex data structure (see section 4.5), therefore it is briefly explained here. The RCU mechanism is a synchronization mechanism that was added to the Linux kernel in 2002 [McK07] and since has received large adoption within the kernel.

Traditional read-writer locks might allow shared reading *or* exclusive writing at the same time, designated readers block while a write is happening and designated writers blocks until there is a point in time without reader. This approach slices time into global phases, either reading-phases or writing-phases. It therefore requires global agreement, which scales poorly.

RCU, on the other hand, allows concurrent readers *and* a single writer. Furthermore, readers always perceive intact data even if a concurrent writer partly updates the shared data. This works by performing the actual update on a copy of the RCU protected object. After the update is finished, references to the original are atomically replaced with references to the modified copy (*Removal Phase*), thereby the readers only perceive the old version or the new version, but never a mid-update state. As soon as the last pre-existing read-side critical section is left (the writer can either await this blocking via `synchronize_rcu()` or provide a callback), the original can be safely freed (*Reclamation Phase*) [McK07].

2.4 Morsels

The following section presents the concept of morsels, as introduced by Halbuer et al. [Hal22; Hal+23]. It will, however, be limited to the aspects of morsels that are relevant for this thesis.

2.4.1 Concept

Morsels are a novel memory-management paradigm that allows efficient sharing of data between address spaces [Alb23]. Morsels render a shared subtree of the page-table hierarchy, so they include both user data-pages, as well as the related management page tables and therefore are considered both an implementation of the *page table sharing* paradigm and fully *self-contained* [Hal+23]. The page table portion of morsels are based on the hardware specific MMU interpreted data structures, they purposely sacrifice hardware independence in favor of performance [Hal+23]. Morsels are single, indivisible units of memory [Bia23] and represent a shift in paradigm, from the management of individual pages to that of memory objects with larger granularity [Fis24].

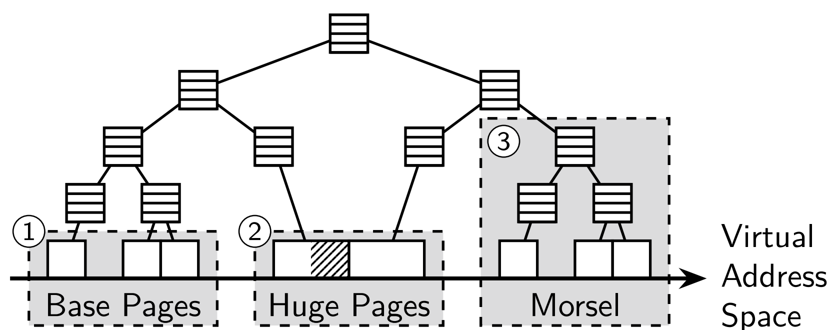


Figure 2.5 – A morsel as indivisible, self-contained subtree of the page table hierarchy. Compared to huge pages, morsels are able to avoid internal fragmentation by sparse population. Figure from [Hal+23]

The size of the virtual morsel *surface* depends on the level of the morsel root node in the paging hierarchy. This height, ranging from 0 to 4 on systems with 5-level paging or 0 to 3 with 4-level paging, is named the morsel *order* and is specified during creation [Hal22; Alb23]. The morsel order can not be changed afterwards. Table 2.1 depicts the virtual surface size depending on the morsel order, it can be calculated by the formula $4\text{ KiB} \cdot 512^{\text{Order}}$.

Morsel Order	0	1	2	3	4
Surface Size	4 KiB	2 MiB	1 GiB	512 GiB	256 TiB

Table 2.1 – Morsel surface size by order.

Morsels may reside on NVM and, in tandem with a suitable page frame allocator, can survive crashes of the surrounding volatile system [Alb23].

2.4.2 Morsel Lifecycle

The morsel is explicitly created via the *Create* operation and from there on identified by its *morsel ID*, which is the root nodes PFN and some flags. [Hal22]. Its lifetime is decoupled from the entity that created it. A morsel may be mapped and selected arbitrarily often, until it is explicitly destroyed by any permitted process via the *Destroy* operation. Destruction of a mapped morsel is semantically illegal, but currently not enforced technically.

2.4.3 Morsel Mapping

A key strength of morsels is their ability to be rapidly mapped into an address space, by setting a single page table entry pointing to the *morsel root*. The mapping time is constant and independent of its order. Morsels must be mapped as a whole and are limited to certain address alignments determined by their order. A morsel is designed to be simultaneously mapped into an arbitrary amount of address spaces [Bia23], and even multiple times into the same address space. Morsels therefore are suitable for fast mapping and unmapping of large shared memory, for example serving as a data package that is passed around from one process to another.

Unmapping or morsels is possible in the same fashion as mapping, having the same advantages in speed, however it requires an additional TLB range flush for the previous mapping location.

2.4.4 Morsel Population

At the time of their creation, morsels consist only of the root node. Therefore the paging hierarchy, and with it the virtual morsel surface, is sparsely populated. Order 0 morsels are an exception, as they solely consist of the root node and are therefore fully populated from the momemt of their creation. It is possible to create morsels, whose surface size exceeds the available physical memory.

Lazy Population

In the event of an access to an unpopulated page on the morsel surface, the MMU raises a page fault due to a hard miss. This page fault is handled by the morsel driver, which allocates a new page frame and inserts it into the morsel's paging hierarchy in such a way that the attempted accessed address translates to the newly allocated page frame, thereby populating the previously absent page on the morsel surface. A following access will therefore not cause a page fault. This process is generally referred to as *lazy population* or *demand paging*. It shifts the allocation overhead (once) into the access path. Figure 2.6 provides a visual example of this. As soon as a page is actually present on the morsel surface, accesses to it have no software on the access path (*fast path*) anymore, because the morsel paging structures are directly interpreted by the MMU hardware. This population is carried out in a lock-free manner, using atomic compare-and-swap (CAS) operations [Hal22].

Explicit Population

The morsel Population operation can be used to explicitly populate a given range on a morsel's surface. It is suitable for situations in which the runtime overhead in the access path introduced by lazy population is not desired, or in situations when it is foreseeable that a large area is being accessed, where page-wise lazy population introduces an avoidable overhead in the form of frequent context switches. It is semantically similar to `madvise(MADV_WILLNEED)` or, more explicit, `madvise(MADV_POPULATE_WRITE)`.

2.4.5 Huge Pages

Bolowski implemented support for huge pages in morsels [Bol24]. Morsels now have a base page size embedded in their ID that allows to increase the previous 4 KiB to 2 MiB or 1 GiB, as available huge page sizes on x86-64. Using huge pages as base pages in morsels provides several advantages, for example the lazy population overhead is reduced for contiguous memory areas.

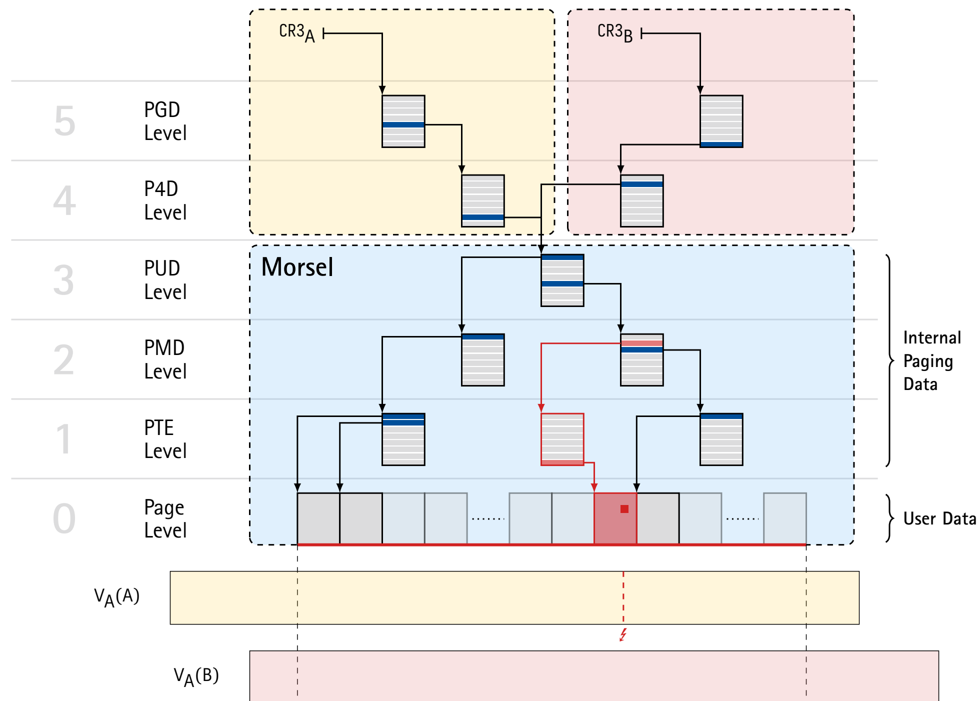


Figure 2.6 – An order 3 morsel on a system with 5-level-paging support, simultaneously mapped into two address spaces *A* and *B*, therefore rendering a shared subtree of the paging hierarchy. The morsel consists of leaf pages, called *User Data*, and internal MMU-specific *Paging Data*. The morsel is initially sparsely populated. In address space *A* occurs a memory access to an address which resides on an absent page, causing a page fault. This page fault is handled by the morsel driver, which allocates a new page frame and inserts it into the morsel's paging hierarchy in such a way that the accessed address translates to the newly allocated page frame, thereby *lazy-populating* the absent page. If required, intermediate page tables are allocated and appropriately inserted as well. A following access will not cause a page fault.

In addition, the higher TLB coverage increases the probability of TLB hits, leading to speedup, especially of non-local accesses across page boundaries. However, the usual disadvantages of huge pages also apply, such as internal fragmentation and increased costs of resolving copy-on-write (COW).

2.4.6 Copy-on-Write

Recently, support for COW for morsels was implemented by Fistanto [Fis24]. Fistanto applied the COW concept for morsels, by associating a reference count (RC) with every node in the paging hierarchy except the root node. The RC represents the amount of direct parent page tables, which are inside of morsels. Per definition, the morsel root node has RC = 1. Elements with a RC ≥ 2 represent the root of a COW-shared subtree. Copying a morsel, for example during a `fork()` now only requires to copy the root node and is therefore constant, independent of morsel order or population.

2.5 Related Work

This thesis relates to a broad field of research, including page table sharing, the lock-free manipulation of paging structures and the challenge of dealing with the resulting TLB inconsistencies, and last but not least the concept of morsel itself.

2.5.1 Page Table Sharing

The concept of sharing page tables, which is the underlying paradigm of morsels, is not a particularly new idea. While the sharing of pages is widely used in Linux, especially to enable COW during a `fork()`, sharing of upper level paging structures is not, until today. Each process holds its exclusive set of page tables down to paging level 1 and the sharing happens on page-level. This implies that if a process `fork()`s, copying its page tables is necessary, even if on the leaf level pages are COW-shared with the originating process. The amount of page tables, and therefore the copying overhead, scales linear to the amount of used memory [ZGF21], which causes an avoidable latency for frequently forking processes with much mapped memory. McCracken investigated if the sharing of pages could be extended one step upwards, to level 1 page tables [McC03; McC06]. A first iteration applied the sharing paradigm to all pages, including those that are COW shared and in the result suffered in runtime overhead for applications with small memory footprint, which are most. In a second approach, McCracken applied the sharing only to those that are actually suited to be shared. In a later approach to speed up the `fork()` system call, Zhao, Gong, and Fonseca also suggested to extend the COW sharing from level 0 to level 1, which they call *Last-level Page Table Sharing* [ZGF21]. With their approach, they managed to speed up the performance of COW significantly, with the downside of a more expensive COW resolve, since this requires to additionally copying of page tables, and they furthermore introduce reference counting overhead.

2.5.2 Approaches to the TLB Consistency Problem

Efficient invalidation of stale TLB entries in multi-processor systems is a challenging task, which is commonly called the *TLB consistency problem*. This topic has been the subject of much scientific work for a long time, and still is. At a time when multicore systems were just about to rise, Teller provided a number of solutions to the TLB consistency problem [Tel90]. Some of them relied on special hardware to provide the consistency, such as a *bus monitor*. Others, purely software solutions, include *TLB shootdown*, originally proposed by Black et al. [Bla+89], which are based on sending IPIs with flush instructions and include busy-waiting of the sender. The *lazy devaluation* uses the TLB tagging feature by assigning a new ASID to a process as soon as it releases a region of virtual memory, making the outdated TLB entries inaccessible to itself and others. The *read-locked TLB* solution makes a processor hold a read lock on a page table as long as its TLB holds an entry that translated using that PT, and which prevents any modification on the PT for the time. The last two approaches do not allow parallel execution in the same address space, or modifying the paging structures of a process that is currently executing. Another approach is *validation*, rather than invalidation. In the event of a TLB hit, a CPU augments the virtual address with the *generation count* stored in the TLB entry, which is then externally compared to the current generation count of the page frame in a *validation table*. In case the generation in the access is outdated, the accessor is notified to invalidate its entry. Another approach by Moon-Seek Chang and Kern Koh, is *lazy TLB* (not to confuse with that of Linux),

a technique that postpones the TLB synchronization until the time of an acquire access to the shared data [MK97]. A processor can send invalidation requests to queues of other processors, which do not need to interrupt their current operation and the sender does not need to wait. Villavieja et al. also propose *pending invalidation buffer* for asynchronous TLB flushes, and also design a *shared second-level TLB directory*, an additional hardware that tracks which TLB holds a translation for a given address, in order to allow targeted flushes [Vil+11].

These are just a fraction of the proposed solutions and underline the active research on this performance-critical topic. Still, carrying out remote *shootdowns* via IPIs is the most widespread solution for the TLB consistency problem. A common strategy is to avoid shootdowns whenever possible. Therefore, it will be a primary goal of this thesis to reduce the total amount of issued TLB flushes to a minimum when implementing the eviction.

2.5.3 Morsels

This section will briefly summarize the related work regarding the context of Morsels.

The morsel concept was designed by Halbuer and implemented in a prototype as a Linux kernel module in tandem with minimal modifications on the Linux 6.1 kernel.

Based on this work, morsels were published via a paper by Halbuer et al. in October 2023 [Hal+23], which demonstrated the strength of morsels in various use-cases.

At the same time, Biastoch designed and prototypically implemented a management layer for morsels, prominently including the morsel *daemon*. The management layer provides various convenience and security functionalities. It maintains a database of existing morsels with associated data like a name and the credentials of the creator. Based on this, user-friendly name resolution can be implemented instead of referring to morsels by their morsel ID. Furthermore, this allows to introduce the concept of a morsel owner, and enforce access control on incoming mapping requests. Another feature of morsels is the sharing of pointer-based data structures across address spaces, which makes it necessary to map the morsel onto the same address, which therefore must be available in every address space. The management layer assists in finding and internally allocating address ranges suitable for this.

Albes implemented support for morsels and specific IOMMUs in 12/2023 [Alb23]. As it turns out, the paging structures of the x86-64 MMU and, at least for the AMD IOMMUs implementation, are sufficiently compatible such that simultaneous interpretation of the same structures by these two hardware components is possible, allowing to extend the page table sharing paradigm even across hardware boundaries. Using the strengths of morsels regarding constant mapping time of almost arbitrarily large memory areas, Albes was able to outperform traditional methods, like *Shared Virtual Memory* and *bounce buffers*, by multiple orders of magnitude. This renders morsels very attractive to speed up I/O operations. Future work regarding this is in progress.

Bolowski implemented support for huge pages in morsels in 03/2024 [Bol24]. Morsels now have a base page size embedded in their ID that allows to increase the previous 4 KiB to 2 MiB or 1 GiB, as available huge page sizes on x86-64.

Recently, support for COW for morsels was implemented by Fistanto [Fis24]. Fistanto applied the COW concept for morsels, by associating a reference count (RC) with every node in the paging hierarchy except the root node. Copying a morsel, now only requires to copy the root node and is therefore constant, independent of morsel order or population.

Swapping of morsel contents is currently worked on, which is useful if morsels are used on systems without NVM, which morsels were originally designed for, but this technology has not received much adoption.

ARCHITECTURE 3

In this chapter, we will define the depopulation operation and design a concept for its implementation on a theoretical level. In doing so, we remain on an abstract level that is limited to the design of suitable algorithms and data structures. Some architecture specifics and details of certain operating systems (like Linux) will be mentioned. However, these are only examples and the generality of the concept remains unaffected.

3.1 Depopulation Operation

The main contribution of this thesis is the design and implementation of a Morsel operation that enables the eviction of underlying physical memory on a given range on a Morsel's surface. In reference to the existing semantically inverse explicit morsel *Population* operation, we call it *Depopulation*. While the Population operation is intended to pre-fault given ranges of the morsel with memory, avoiding demand paging overhead during runtime, the Depopulation inverts the population process by removing and deallocating data pages on the affected surface range, as well as corresponding page tables.

3.1.1 Basic Operation Definition

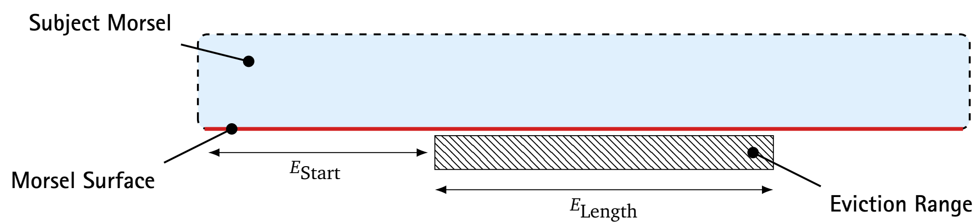


Figure 3.1 – Figure visualizing the basic notions of morsel Depopulation.

Subject

The Depopulation operation affects the surface of a morsel, the latter we call the *Subject* in the following. We define Depopulation as an operation *on* a morsel, therefore the subject can be derived from the context.

Arguments

The input to the Depopulation operation is a specification of the contiguous part of the subject's surface, which is supposed to be depopulated. We name this the *eviction range* in the following. The eviction range is uniquely identified by a start offset E_{Start} on the morsel surface and a length E_{Length} in bytes.

Result

After successful completion, the specified eviction range will be indistinguishable from the state prior to its population, which manifests itself in two separate dimensions:

1. **Dimension: Removal of Contents**. A following read operation will return uninitialized memory, which is by definition zeroed for morsels. Every previously stored content in the eviction range is considered lost, except it was zeroed in the first place. This dimension is directly visible to accessors of the surface.

2. **Dimension: Deallocation of backing memory**. Every page, that resides completely within the provided region, has been logically removed from the morsel and the containing page frame is deallocated, if it lost the last reference onto it. A following memory access will trigger lazy population. This dimension is transparent to accessors of the surface, but can perceived in non-functional properties like access time.

Therefore, next to the modification of the morsel itself, an important side effect of this operation is the deallocation of page frames. This renders the depopulation, next to the collateral *Destroy* operation, useful in low-memory scenarios. The Depopulation can be applied to the entire morsel surface, but in contrast to the Destroy operation, it will never deallocate the morsel root node, because this would destroy the morsel. Furthermore, a Destroy operation is semantically only permitted when the morsel is unmapped, whereas a Depopulation operation is conducted on morsels which are potentially mapped into many address spaces simultaniously and subject to concurrent accesses. This results in specific requirements, especially with regard to the reduction of inconsistencies caused the TLBs of the accessors.

The Depopulation operation is semantically similar to `madvise()` with `MADV_REMOVE` advice value.

3.1.2 Discussion: About Flushes, Batches and dynamic Memory

On paging-enabled systems, the deallocation of a page frame, which was the target of an address translation, requires previous TLB invalidation for all the VPNs that translated to that page frame. Therefore the usual three-step of consists of[2]

1. Remove the last reference to the page frame in question,

2. TLB invalidation

3. Free page frame

We assume for now that there are lots of page frames to be freed, for example because we were in the situation to depopulate the surface of a morsel.

Generally speaking, TLB flushes should be a rare event, because, firstly, the delivery of the flush IPIs themselves interrupt the receiving core. Secondly, the address translations of unrelated

[2] In reference to Linux's description of the `mmu_gather` API in `/include/asm-generic/tlb.h`

workloads may suffer from collateral damage caused, depending on the degree of precision of the flush. Usual mitigations include changing semantics or batching of independent shootdowns. Because we can not weaken guarantees when returning page frames to the operating system's page frame allocator, we resort to batching in the following. For lots of pages, it is desirable to *gather* unhooked pages in order to issue *batched* flushes. In the most extreme, one gathers all pages and flushes only once (for example, a *range flush*), therefore minimizing the overall amount of issued flushes.

However, gathering requires allocation of new memory to store references to the pages that lost their last reference elsewhere. Considering that we are designing an operation that is supposed to deallocate physical memory from a morsel, allocating new memory in order to free memory may sound paradox, and it may even render a problem if the depopulation is motivated by a lack of available memory in the first place.

Furthermore, time spent with gathering before flushing increases the *inconsistency time window* in which accessors can have different views of the morsel's surface, depending on their individual TLB state:

- **TLB hit**: The core can still hit the page frame of an already logically removed page with accesses that conform with the cached access controls. Data written this way would be lost.

- **TLB miss**: An access triggers lazy population, read unitialized/zeroed contents and writes are persistent. This does not further interfere with the running depopulation operation.

Since each accessing core might hold its individual set of TLB entries, it may therefore perceive an individual temporary view of the morsel, regarding contents and persistencies of *each* individual page on the eviction range.

As we can see, there are multiple optimization dimensions that we can optimize for:

- Minimize the **amount of issued TLB flushes**. Positively influenced by larger batches, which negatively influences gather storage.

- Minimize the **amount of memory required for gathering**. Positively influenced by smaller batches which then requires more frequent flushes.

- Minimize the **size of the inconsistency time window**. Positively influenced by fast eviction and small batch size, which then implies more frequent flushes.

Example: Linux MMU Gather

The MMU Gather in Linux is an interesting example of a compromise. If one happens to wish to free a lot of page frames that still might have TLB entries for them, one may start a new gather session using `tlb_gather_mmu()` , and then submit all the frames via the `tlb_remove_page*` functions. The MMU Gather is responsible to flush the TLB before it deallocates the pages. It is however allowed to batch the received page frames for performance reasons and it does so[3] by managing a dynamically allocated and singly linked list of page-sized `mmu_gather_batch` structures. The MMU Gather might decide by itself, that it is time to issue a flush, for example when the amount of gathered items reaches a certain threshold. Figure 3.2 visualizes this scenario. Due to the internal threshold, the space consumption over time of the MMU Gather for large amounts

[3] If not opted-out via `CONFIG_MMU_GATHER_NO_GATHER`.

of pages has the form of a sawtooth wave. The user might also explicitly signal the end of the gather session via `tlb_finish_mmu()`, which also issues a (final) flush.

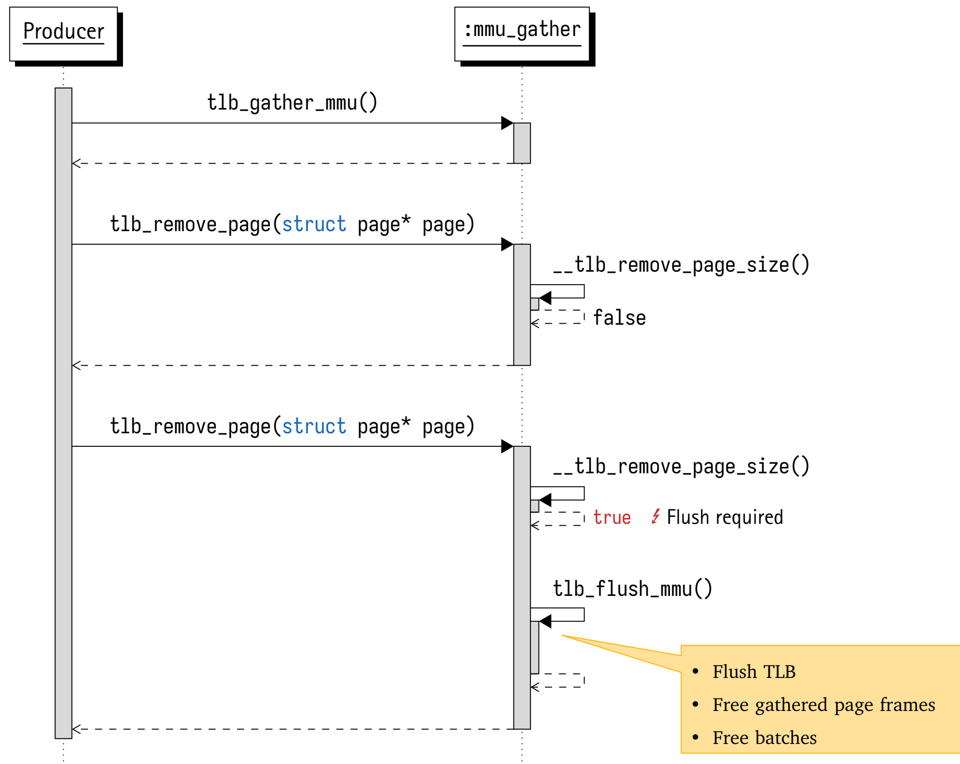


Figure 3.2 – Example of a `mmu_gather` instance that itself decides to flush and free the gathered page frame batches upon insertion of a page frame, based on an internal threshold.

Regarding the optimization dimensions, the MMU Gather renders a trade-off between flush count on the one side and gather storage on the other, which can be adjusted by the threshold value.

Approach in this Thesis

Section 3.2 proposes an algorithm with a runtime complexity far below linear to the eviction range size. As we will see, the key lies in not iterating the leafs at all, but rather avoid descending wherever possible and to evict based on higher tree nodes. We therefore expect to significantly outperform leaf-iterating algorithms like Linux's `zap_{pte,pmd,pud,p4d}_range`. Therefore, the time in between the first evicted page and the last finished flush, the time window of inconsistency, is minimized.

At the same time, because of evicting subtrees and not leaf pages, the amount of references to store is, compared to leaf-storing approaches like MMU Gather, significantly reduced. As section 3.2.3 will show, in the worst case, evicting circa 256 TiB, it is required to store 3576 references. With the same amount of references, traditional MMU Gather covers circa 39 MiB of eviction range (assuming 4 KiB pages in both cases). Therefore it is feasible storage-wise to collect all gathered items into a single batch and then issue only one range flush per mapping

location of the morsel, which renders both the least amount of, and at the same time, a precise type of TLB flush.

To summarize, the approach of this thesis is

- Minimizing the window of inconsistency by focusing on eviction speed and delaying time-consuming clean-ups to after the TLB flush.
- Minimizing the amount of required gather storage and thereby it is feasible to batch into only one batch, reducing the amount of TLB flushes to a minimum.

3.1.3 Chosen Architecture in this Thesis

As a consequence of the previous section, we design the Depopulation operation to be divided into three successively executed steps which we call the *Depopulation Subroutines* in the following.

1. **Eviction**, logically removes all affected data pages and no longer required page tables from the morsel. Covered in detail in section 3.2. Optimized for performance.
2. **TLB Flushing**, ensures the absence of stale TLB entries that cache outdated information and thus would indirectly prevent the safe deallocation of page frames in the following. Optimized to be fast and precise. Covered in detail in section 3.3.
3. **Clean-up**, iterates the evicted structures and deallocates page frames for leafs and intermediate paging structs. Covered in detail in section 3.5.

Figure 3.3 provides a high-level overview on the Depopulation operation, with an emphasis on the three subroutines and the flow of the gathered data from the Eviction to the Clean-Up.

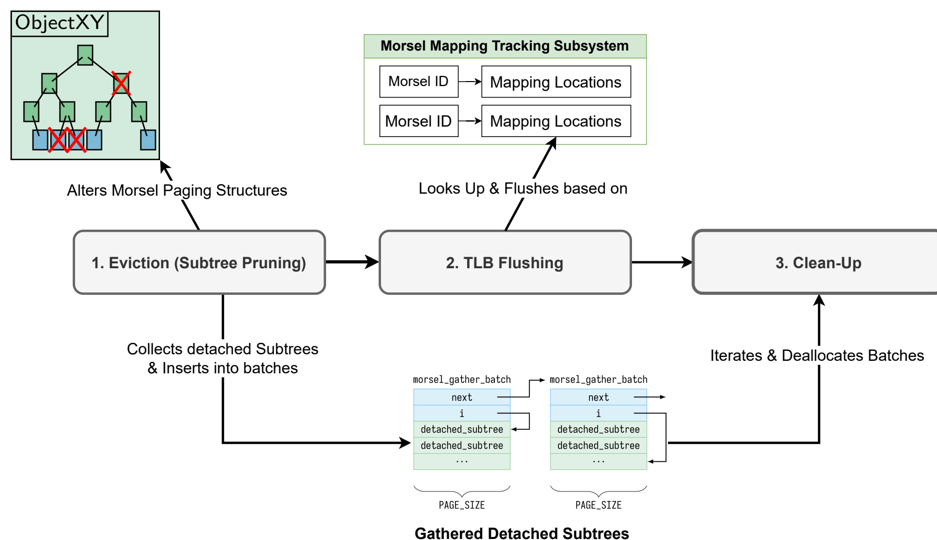


Figure 3.3 – Overview on the Depopulation subroutines. We can see that the Eviction alters the morsel paging structures and submits references to the evicted items into the Morsel Gather. The TLB Flush performs range flushing based on the mapping information stored in the morsel mapping tracking subsystem (MMTS). The Clean-Up finally consumes the gathered references and performs deallocation.

The first two routines (Eviction and TLB Flushing) are optimized for speed and delay all time-consuming tasks to the Clean-up routine.

3.1.4 Discussion: Determining the Pages to Evict

The pages to evict must be derived from the specification of the eviction range. However, the eviction range might not be aligned to the borders of the actual pages. Rounding the eviction range to the next inner-laying page boundary removes too less and is a violation of result dimension 1. Rounding the eviction range to the next outer-laying page boundary removes too much and is certainly not an option, as this may cause unintentional loss of data. Another approach would be to insist on aligned addresses. However, this comes with the burden for the user to memorize the exact page layout of the morsel in mixed size scenarios. Bolowski implemented support for huge pages in morsels, which previously consisted solely of base-pages [Bol24]. Although currently only one granularity can be used at a time, this thesis assumes there will be support for mixed page size granularities within morsels in the future and therefore all theoretical concepts and the implementation are compatible to that.

We decided for a user-friendly approach and to not enforce any alignment of the eviction range. This allows the user to byte-precisely specify the depopulation range.

- All pages that reside completely in the eviction range will be removed.
- Partly covered pages on the range ends are evicted via null-writes and remain present.

Sub-base-page-size depopulations are thereby possible, although this would just be a `memset` equivalent. It removes the burden from the user to memorize the page/huge page layout of the morsel and avoids the risk of unintentional data loss when rounding interval borders to actual page borders. This also has the effect that for the user, depopulation almost never fails, as long as the user is permitted to and the provided interval definition is valid.

We accepted the downside that strictly speaking, result dimension 2 – removal of backing storage, see section 3.1.1 – is loosened for the interval ends. However, dimension 1 – removal of contents – is always guaranteed.

3.2 Eviction

In the following, the term *Eviction* may describe the destructive process of altering the page table hierarchy with the goal of logically removing virtual memory, usually by clearing present bits. The *range eviction* aims to remove a certain virtual address interval, the *eviction range*.

Deallocation of the acquired memory is out-of-scope for eviction and can be delayed until a later clean-up phase, see section 3.5. Eviction also does not include flushing the TLB for the evicted virtual address range, for this see section 3.3.

3.2.1 Punching Holes into the Morsel Surface

The first observation to make is, that, regarding the eviction of an individual page, it is irrelevant at which point the MMU terminates its table walk due to non-presence of entries. For a 4 KiB page it doesn't matter if its PT-entry is cleared or its parent PT's PMD-entry, as displayed in figure 3.4.

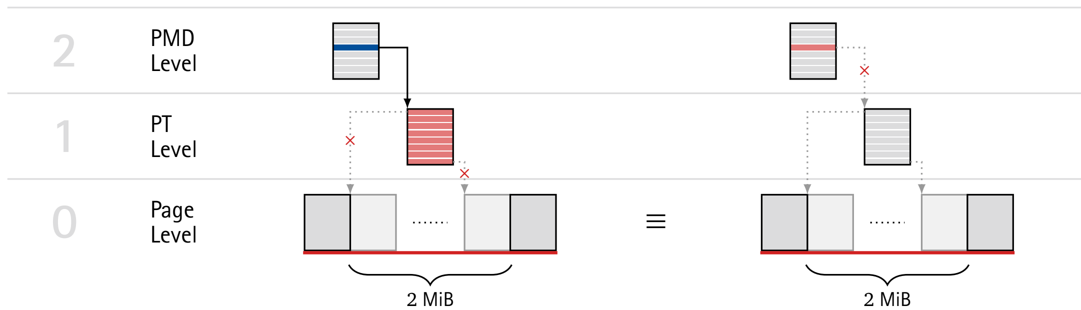


Figure 3.4 – Example how 2 MiB of aligned contiguous virtual memory can be evicted in functionally equivalent ways that differ in runtime of the eviction. The left approach cuts on leaf-level and requires 512 write operation while the right approach cuts on PT-level and requires one write operation.

By clearing present bits on the higher levels of the paging hierarchy, we are able to punch holes in the sizes of 4 KiB, 2 MiB, 1 GiB or 512 GiB into the morsel surface – by a *single atomic write* operation. Regarding the eviction algorithm itself, the same result can be achieved with 511 write operations less, resulting in a significantly reduced runtime. The speedup of leaf-cutting versus pruning of subtrees even exponentially increases by powers of 512 with increasing height.

Higher cutting also speeds up future lookup *hard misses*, because the MMU is able to terminate the table walk earlier. Furthermore, given that the TLBs are no negative caches – they don't cache the absence of pages – and therefore a table walk on every singe access to a page on the evicted range is necessary, a speedup of the latter is even more valuable in the context of eviction.

However, this approach is subject to two limitations. First, the granularity is constrained to the described sizes. Second, it is limited to certain alignments.

3.2.2 Eviction via Lock-Free Subtree Pruning

We implement range eviction based on the hole-punching idea, by using a top-down approach that avoids descending wherever possible. The algorithm is depicted in algorithm 3.1 and will be discussed shortly.

```
 1: procedure prune(node, start, end)
 2:     if node completely covers [start, end] and node is not morsel root then     ▷ prune
 3:         detach node from its parent via atomic CAS operation
 4:         enqueue a reference to node for delayed clean-up
 5:     else if node is leaf then                                                    ▷ not-prunable leaf
 6:         enqueue a delayed zero write for interval [start, end]
 7:     else                                                      ▷ not-prunable non-leaf ⇒ descend
 8:         for all children affected by [start, end] do
 9:             prune(child, max(child.start, start), min(child.end, end))
10: prune(morsel root node, eviction start, eviction end)
```

Algorithm 3.1 – The Subtree Pruning algorithm

As we can see, it is formulated recursively in the form of a depth-first traverse and always treats one node at a time. At the beginning, this is the morsel root node. Furthermore, two

addresses, *start* and *end*, define an interval within the part of the morsel surface that the respective node translates to, that is to be evicted. We call this *eviction interval*. Initially, this is equal to the eviction range.

The algorithm distinguishes four cases.

I. **The node is completely covered by the eviction interval.** The eviction interval [start, end] is identical to the part of the morsel surface translated to by the node. In other words, the node does not translate to any address outside the eviction range, therefore is safe to be detached via a single atomic CAS operation to its parent. Because this *prunes* the subtree that the current node is the root of, we name this algorithm *Subtree Pruning*.

   A reference to the detached node is enqeued for delayed clean-up (see section 3.5).

   Note that we are not allowed to prune the morsel root node, as this would destroy the morsel. In that case, a depopulation of the whole morsel, we simply omit pruning. We will automatically fall into case III, descend once into every child and prune it immediately. For fully populated morsels, this always results in (just) 512 gathered references, independent of its order.

II. **The node is a leaf and partly covered by the eviction interval.** A leaf (page) that is partly covered must not be pruned, because this would falsely affect the uncovered parts. This case is possible because we intentionally do not force alignment in the eviction range specification (see section 3.1.4). Furthermore, this can happen only twice, namely at the range ends. In this case, we must perform a destructive null-write into the affected page's contents (*zero write* in the following). However, because the page itself is not altered regarding its presence, the zero write is not relevant for the TLB flushing and therefore is delayed into the Clean-Up subroutine.

III. **The node is not a leaf and partly covered, at least one affected child exists.** The node is a page table and translates both into and outside of the eviction interval. We descend into every child that does translate to the eviction interval. The eviction interval is reduced to the intersection with the respective child's translated memory, so the child will see only the parts of the eviction interval that are relevant for it.

IV. **The node is not a leaf and partly covered, no affected child exists.** This case is implicit, if the loop over the affected children never does an actual iteration. In this case, nothing further is (to be) done.

The maximum recursion depth is equal to the morsel order, which is 4 or 3, depending on the hardware support for 5-level paging, and thereby very limited. Figure 3.5 gives a visual example of an eviction.

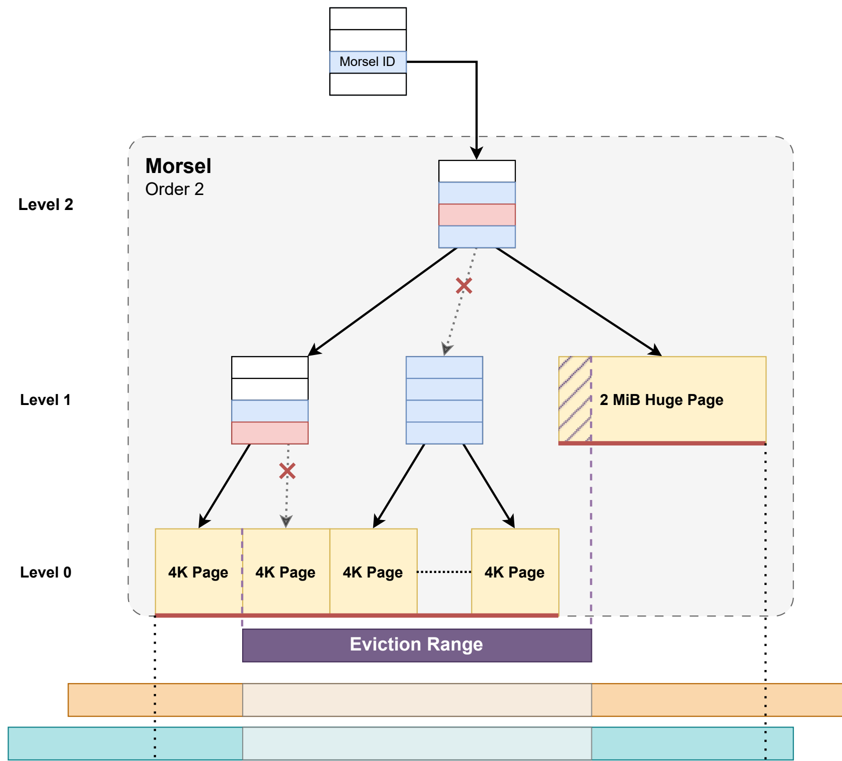


Figure 3.5 – The surface (red) of an order 2 morsel with mixed page granularities, that is mapped into two locations, is evicted via Subtree Pruning. Two subtrees (one order 0, one order 1) are detached (marked via red crosses), a zero-write for the partly covered huge-page is delayed (hatched area). In this case, the algorithm required 2 write operations to evict 513 whole 4 KiB pages (not including the delayed zero write).

Considering Concurrent Modifications

Morsels' page tables ultimately form a non-synchronized data structure, subject to potentially true-parallel write access. Write access to the page tables can happen at any time, by fault handlers performing lazy population, because accesses to the morsel surface can not be prevented unless unmapping the morsel from every mapping location (including a TLB flush), which is the approach that Fistanto chose for the expensive COW operations [Fis24]. Next to fault handlers, we must consider potential other concurrent eviction operations.

If two concurrent eviction operations happen to attempt pruning the same subtree, only one of them should *get* it and be in charge of cleaning it up, in order to avoid double freeing. Using atomic CAS operations, a subtree pruning control flow can ensure that *it* atomically changed a present page table entry to a non-present entry and therefore is in exclusive possession of the removed entry. If the CAS did not succeed, someone else apparently removed the entry in the meantime and is therefore responsible for clean-up, so there is nothing left to do.

After atomically pruning a subtree, immediately no other eviction or fault handler can *enter* it. Even if someone was already in there, this is not a problem, because neither the fault handler nor the eviction require that the subtree they operate on is still mounted. The only remaining

challenge is to ensure that everyone has left the subtree before cleaning it up, which will be discussed later in section 4.7 and is not relevant for the eviction.

3.2.3 Worst-case Amount of Pruned Subtrees

The basic characteristics of the paging hierarchy of x86-64 with physical address extension (PAE) are given as maximum tree height 5, maximum node degree 512. Furthermore, because the maximum morsel order is 4, the highest detached subtree has its root on level 3. Based on this knowledge, we can construct the cases in which the highest amount of subtrees has to be pruned in order to evict a given range, depending on the morsel order. In all cases, the morsel is fully populated with 4 KiB pages. For order 0 morsels, trivially no subtrees can be pruned. For order 1 morsels, all the children can be pruned off the morsel root, resulting in 512. Not evicting the whole morsel results in fewer subtrees. For order 1 however, the most amount of subtrees is pruned if the first and last 4 KiB of the morsel surface are left out in the eviction range. Therefore the first and last order 1 subtree can not be pruned, instead descending and individually removing the inner-lying 511 order 0 subtrees (pages) is necessary. Figure 3.6 visualizes this for the orders 0, 1 and 2.

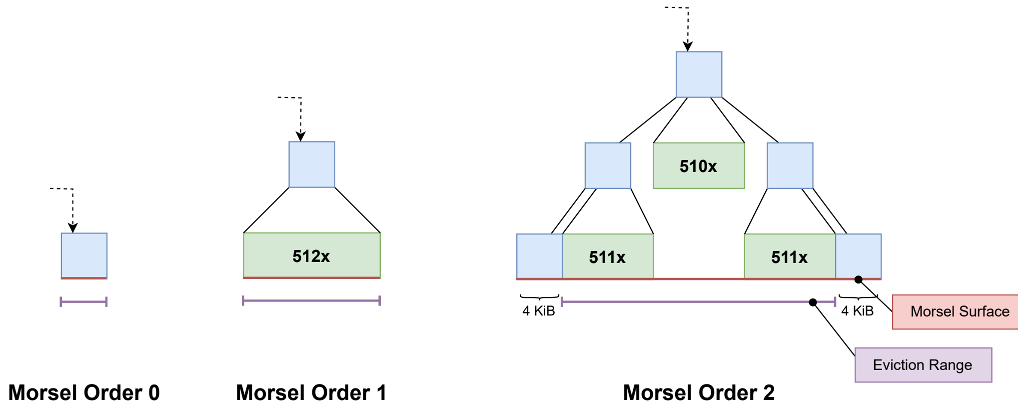


Figure 3.6 – Example of the situations with the most amount of pruned subtrees for the lowest 3 morsel orders. In all cases the morsel is fully populated. For orders ≥ 2, the eviction range omits the first and last 4 KiB page.

This slightly smaller eviction range can be applied to the morsel orders 3 and 4 as well, which result in a total of 2554 and 3576 pruned trees, respectively. See table 3.1.

Given table 3.1, the worst-case amount of detached subtrees can be described as a function f of the morsel order n as

$$f(n) = \begin{cases} 0, & n = 0 \\ 512, & n = 1 \\ 510 + (n-1) \cdot 2 \cdot 511, & n \in \{2,3,4\} \end{cases} \tag{3.1}$$

The overall worst case is therefore in the event of evicting 256 TiB $-$ 2 $\cdot$ 4 KiB off an order 4 morsel fully populated with 4 KiB pages. In this case, 3576 tree references need to be stored. Given that each reference to a tree node should not exceed a few bytes, the memory required to store those is quite low compared to the gigantic eviction range.

Morsel Order	# detached subtrees (by order)				
	3	2	1	0	Σ
0	0	0	0	0	0
1	0	0	0	512	512
2	0	0	510	$2 \cdot 511$	1532
3	0	510	$2 \cdot 511$	$2 \cdot 511$	2554
4	510	$2 \cdot 511$	$2 \cdot 511$	$2 \cdot 511$	3576

Table 3.1 – Worst-Case amount of pruned subtrees by morsel order

3.2.4 COW aware Subtree Pruning

In parallel to this thesis, Fistanto implemented copy-on-write support for morsels [Fis24]. Although it could not be considered in the prototypical implementation due to time constraints, the COW concept is addressed in the following theoretical extension of the subtree pruning algorithm.

With COW enabled, all elements in the morsel paging hierarchy have an associated reference count (RC). Per definition, the morsel root node has $RC = 1$. Elements with a $RC \geq 2$ represent the root of a COW-shared subtree, which is, per definition, read-only. In other words, it is illegal to write to an element that itself or one of its ancestors has a $RC \geq 2$, because this would unintentionally tamper with multiple COW copies simultaneously. A write operation requires the accessor to claim exclusive ownership or perform COW resolve on the target first.

Eviction is a destructive, write-alike operation that should only affect one copy of the morsel. A naive approach would be to perform a COW resolve over the eviction range to make it exclusive and writable, and then run the unmodified pruning algorithm. However, this would involve allocating fresh memory and copying the content that is to be removed in the next step anyway. This is clearly unnecessary. The available COW resolve is unsuitable in cases where the resolved area is to be evicted.

A more efficient approach is to make the Subtree Pruning algorithm COW-aware, avoiding copying wherever possible while still profiting from the pruning concept. An advantage of subtree pruning is that it is a top-down approach, so it necessarily has encountered all ancestors of the nodes it processes and thereby, without additional overhead, can ensure the absence of ancestors with $RC \geq 2$.

In the following, the algorithm 3.1 shall be extended such that for nodes with $RC = 1$ the old behaviour is kept. If the current node has a $RC \geq 2$, there are four scenarios:

I. **The node is completely covered by the range to evict.** In this case it does not matter if the node is a leaf or not, the subtree can be pruned with an atomic write access to its parent and a modification of the node itself is not required. Unlike in the unmodified case, it is not necessary to check that the current node is not the morsel root node, as those by definition cannot have an $RC \geq 2$ in the first place.

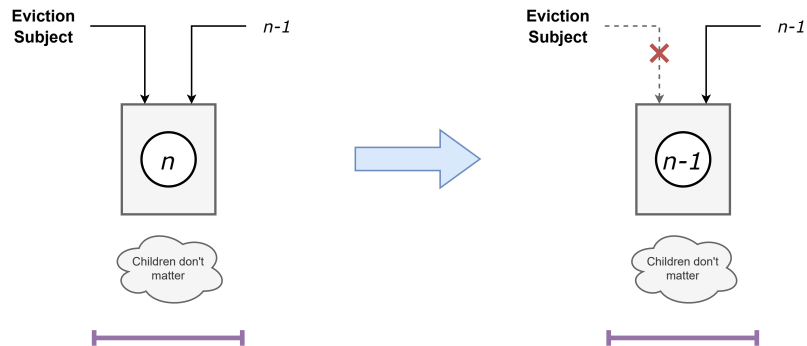


Figure 3.7 – Example of COW scenario I.

II. **The node is a leaf and partly covered.** Resolving the COW on the node is required, because a partial write access to the node is necessary. This involves copying the node onto a newly allocated page frame of the same size and then switch the parent over to the copy. The RC of the original is decremented by one. Assuming that the newly allocated page frame is zeroed initially, only the unaffected part needs to be copied. No delayed zero write is required because the affected part is zeroed in the first place.

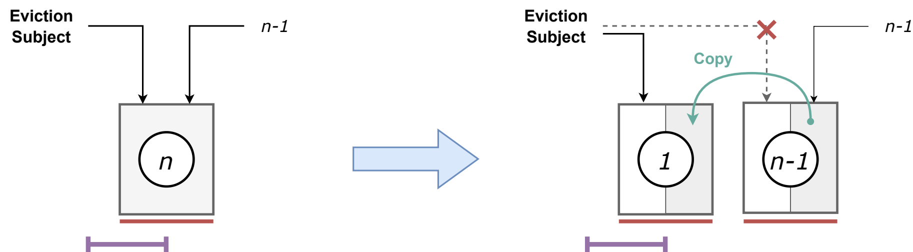


Figure 3.8 – Example of COW scenario II. Red indicates morsel surface.

III. **The node is not a leaf and partly covered, at least one affected child exists.** Although descending into the affected children does not require modifications of the node, its RC ≥ 2 prevents all potential write accesses in the descendants. Therefore a COW resolve of the node is necessary, including copying the node and switching it over by an atomic write to the parent. The RC of the original node decrements and the RCs of all children increment by 1. This way, unaffected children will stay unmodified and COW-shared among the other owners.

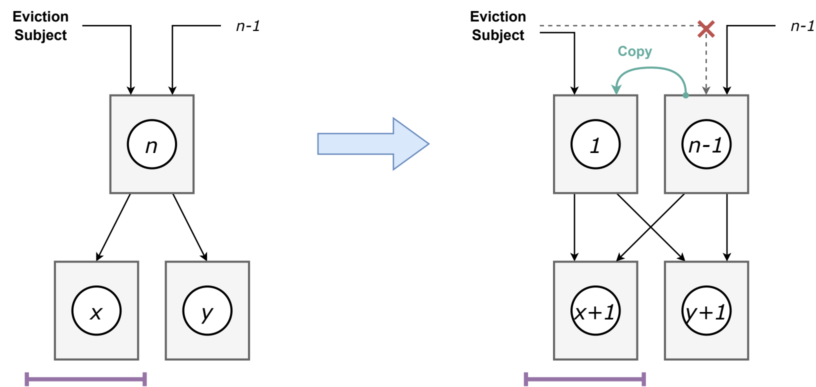


Figure 3.9 – Example of COW scenario III.

IV. **The node is not a leaf and partly covered, but no affected children exist.** In this case, no action is required. Because there is no affected children that can be descended into, the parent also doesn't need to become writable. This is also the only case without write access to the parent.

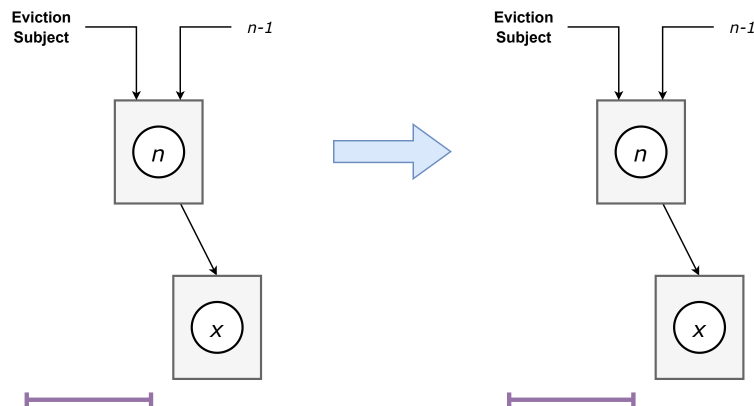


Figure 3.10 – Example of COW scenario IV. No modification is required.

In all the COW-shared cases, no subtrees are queued for clean-up, because due to RC ≥ 2, even the RCs decremented by 1 are still above 0 and therefore still in use.

3.3 Precise TLB Flushing

Once the last reference to a morsel page frame is removed by eviction, it shall be released. However, prior to returning the page frame to the page frame allocator, it is necessary to ensure that no translation lookaside buffer is caching a translation to the page frame's physical address (PFN). This is a general necessity when deallocating page frames and not specific to morsels.

The absence of stale translations is ensured via TLB flushes. Linux has established the in-kernel TLB flushing interface, which abstracts away the concept of a TLB from the underlying,

architecture specific details. This also takes care of optimizations, like batching and reducing the IPI recipients to those CPUs actually affected, and performance considerations like at which threshold a full flush is carried out instead of single page flushes (`tlb_single_page_flush_ceiling`), which is also customizable. The TLB flushing interface offers flushing on various granularities, ranging from complete flush (`flush_tlb_all`) over address spaces (`flush_tlb_mm`), ranges within an address space (`flush_tlb_range`) down to a single page (`flush_tlb_page`).

For morsel depopulation, the best matching granularity is flushing ranges, in order to flush the eviction range for each address space that the morsel is mapped to. The operating system is of course free to translate this into larger granularity flushes if it wants to, but this is not a decision that the morsel driver needs to make or should make. This way, morsel pages behave the same regarding flushing the TLB as every other page, and they are subject to the same global flushing strategy and performance optimizations.

3.3.1 Need for Reverse Mapping

It is, by hardware limitations, not possible to flush translations out of the TLBs based on the target of the translation, the PFN. The Linux's TLB flushing interface reflects this and requires virtual addresses, at least for range and single-page flushes.

Therefore, when the goal is to flush all stale translations for a specific page frame, one needs to know all virtual addresses translating to it. Determining the virtual addresses of a page frame inverts the traditional address translation and is typically referred to as *reverse mapping*.

The traditional way of reverse-mapping anonymous memory in Linux is a complex process, further explained in section 2.1.2. However, although a statically allocated `struct page` exists for every of the morsel's page frames, they are intentionally not used. Updating the metadata of every single morsel page frame on every `map` and `unmap` would significantly slow down those operations. In fact, work has been done to even avoid the allocation of `struct page` for morsel page frames in the first place [Aum24].

Since the usage of `struct pages` for morsels and thereby the traditional Linux way of reverse mapping is not an option, we design another solution to reverse-map morsel page frames, which is presented in the next section.

3.3.2 Reverse Mapping via Tracking of Morsel Mappings

Each position on the morsel's surface has a unique offset to the beginning of the morsel, we call it *surface offset* in the following. Because morsels can only be mapped as a whole, the virtual addresses of the same position on the morsel surface have the same offset to the respective mapping beginning. Provided we knew all mapping locations of a morsel, we can therefore determine the virtual addresses of a position on the morsel surface by adding the respective surface offset to each known mapping location. Figure 3.11 visualizes this. This realizes reverse mapping for morsels, because we are able to determine for every page frame the virtual addresses that translate to it. By reverse mapping the eviction range we can determine all its virtual address-ranges and then issue range-flushes against the operating system's TLB flushing interface.

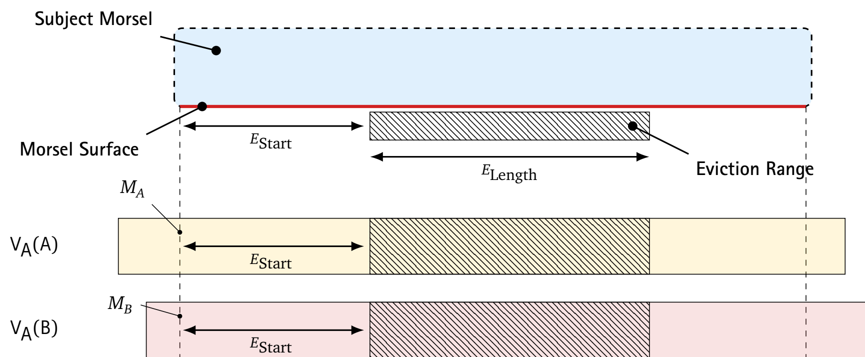


Figure 3.11 – Example of how the start of the eviction range can be reverse-mapped to all virtual address ranges translating to it, utilizing the known mapping locations $\{M_A, M_B\}$ and the constant-offset property.

The key here is that morsels can only be mapped as a whole and therefore reverse-mapping on page-granularity, like the traditional Linux way via `struct pages`, would be redundant – it can equally be implemented on morsel-granularity using the described way with significantly less bookkeeping overhead in time and space.

Because the mappings of a morsel were not tracked yet, the design and implementation of a tracking mechanism is part of this thesis, see sections 3.4 and 4.5.

3.4 Morsel Mapping Tracking Subsystem

The morsel mapping tracking subsystem (MMTS) is a new part of the morsel driver that is supposed to keep track of morsel mappings. The MMTS is motivated to enable precise TLB flushes, for which one use-case is Depopulation. Another use-case for flushing is to remap COW-originals as read-only, because the TLBs also cache access control information. The MMTS is not responsible for monitoring the *existence* of morsels, as this is the domain of the daemon.[4] Mappings, as the connection between the persistent morsels and the volatile surrounding system, are volatile themselves and the tracking information therefore does not require any extended persistence. It can therefore be held in volatile kernel memory, which also allows fast access compared to e.g., storing it in the daemon. A key requirement is that the MMTS must be able to store an arbitrary amount of mappings and therefore inevitably must allocate dynamic memory at runtime. The second key requirement is the lookup speed to retrieve the mappings of a morsel, as this is critical for TLB flushes. A data structure, that is both dynamic and optimized for fast in lookup, is a hashtable. Another important point to consider was that the MMTS is subject to potentially true-parallel access by multiple cores. Consequently, the integrity of the internal data structures must be protected through the implementation of appropriate synchronization measures. Depending on the concrete protection mechanism, blocking effects might occur. A potential block on the path of `mmap` and `munmap` conflicts, to an extent, with the lock and log free concept of morsels. Although map, unmap and destroy might not be the most frequent and time-critical operations, the implementation should be very careful about this and reduce the possibility and costs to a minimum.

The MMTS exposes the following functionalities to the morsel driver

[4]Although the MMTS de-facto tracks the existence of those morsels that have at least one mapping.

- Insert a new mapping. This is used then a new mapping is created on `mmap()` or an existing mapping is duplicated on `fork()`.

- Remove a stored mapping. Used when unmapping a morsel, either actively via `munmap()` or when a process/address space terminates.

- Check whether at least one mapping exists for a given morsel. Used to enhance the security of the Morsel *Destroy* operation, which before had to rely on the user to unmap the morsel prior to destroying it.

- Invoke a provided callback function for every known mapping of a given morsel. Used to implement TLB range flushes and acquiring the `mm-lock` writable on every address space the morsel is mapped into (see secion 4.7).

For the implementation, see section 4.5.

3.5 Clean-Up

The objective of the cleanup subroutine is to traverse the gathered pruned subtrees and deallocate all page frames that have lost their last reference. It consumes the root node references from the Morsel Gather structure and descends into the trees. Only those trees whose root lost their last reference were scheduled for cleanup in the eviction, therefore at least the root node can always be deallocated. It is reasonable to iterate the trees in a post-order fashion, deallocating a node after handling its children. This was the approach chosen in algorithm 3.2. It is important to keep in mind that one may encounter nodes with RC $\neq 0$, which mark the root of a subtree that is COW-shared between the current detached tree and one or more other owners. These nodes must not be altered and can simply be omitted.

```
1: procedure cleanup(node)
2:     if RC(node) ≠ 0 then
3:         return
4:     if node is not leaf then
5:         for all children do
6:             RC(child) ←RC(child) −1
7:             cleanup(child)
8:     free(node)
```

Algorithm 3.2 – The Clean-Up algorithm. Lines 2,3 and 6 are extensions for COW.

Figure 3.12 illustrates the procedure on an example detached order 2 tree with a COW-shared subtree.

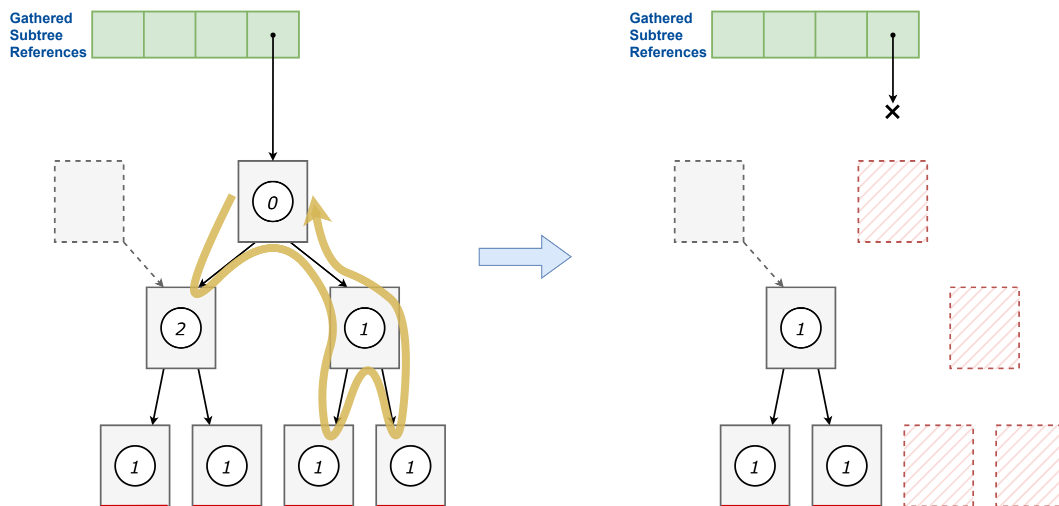


Figure 3.12 – Example cleanup of a detached order 2 tree that contains a COW-shared order 1 subtree. The shared subtree is not altered (except the RC) and four page frames were deallocated in this case.

3.6 Summary

In this section, we presented the design of a novel morsel operation intended to evict a specified subset of a morsel's surface, the Depopulation. We designed the Subtree Pruning algorithm, which implements range eviction and which we expect to significantly outperform traditional leaf pruning because of its below-linear runtime complexity and storage footprint. We also extended Subtree Pruning for COW. Based on the properties of the subtree pruning approach, it is feasible to gather all subtree references into a single batch and issue only the minimal amount of high-precision TLB flushes. We demonstrated the design of a new tracking mechanism for morsel mappings, on top of which we can implement reverse mapping on morsel-granularity. Compared to traditional page-level reverse mapping, this achieves massive reduction in bookkeeping overhead, both in space and time. Finally, we designed a Clean-Up procedure to deallocate the evicted subtrees in a COW-compatible way.

4 IMPLEMENTATION

This chapter presents the practical implementation of the concepts developed previously, through an extension to the prototypical morsel implementation by Halbuer, which includes a modified Linux kernel in version 6.1, the morsel driver kernel module and userland libraries [Hal22].

4.1 General Structure

The Depopulation, like the other morsel operations, is implemented inside the morsel driver kernel module, which therefore is subject to the most modifications. Most notably, this includes the system call handling in `driver.c` and the implementation of the eviction and cleanup subroutines in `morsel_mem.c`. Wherever possible, new functionality is added as encapsulated modules that have a well-defined interface to the rest of the driver, namely the morsel mapping tracking subsystem (`morsel_track.{c,h}`, see section 4.5) and the Morsel Gather mechanism (`morsel_gather.{c,h}`, see section 4.4). In order to ease the use of the new system call from the end-users perspective, the userspace libraries are also enhanced accordingly. Notably, this work does not require modification of the Linux kernel.

Figure 4.1 provides an architectural overview of the extended morsel prototype. It also outlines the steps involved in a Depopulation operation, whose implementations will be presented in following sections in more detail. An application invokes `depopulate()` (1) on a `MorselMem::Morsel` object provided by the morsel C++ Library, which delegates the request to the underlying morsel C library which issues (2) a `MORSEL_IOCTL_DEPOPULATE` system call (see section 4.2). The `ioctl` handler of the morsel driver receives (3) the system call, copies-in the arguments and performs validation of the user's permissions (see section 4.2.2), the subject morsel and the range definitions (see section 4.2.1). It then delegates the depopulation to the Eviction implementation (see section 4.3) in `morsel_mem.c` successively for every of the provided ranges, which performs the subtree pruning (4) and inserts the gathered subtree roots into an instance of the Morsel Gather (5) (see section 4.4). Subsequently, it flushes the TLB for every known mapping location of the eviction range (6) by invoking the MMTS (see section 4.5) with a certain callback function pointer. In order to ensure a safe deallocation (see section 4.7), it acquires (7) and immediately releases the `mmap_lock` writable once for every address space that the morsel is mapped into. Then the Clean-Up (8) is conducted, successively consuming (9) and deallocating items off the Morsel Gather and descending into the referenced subtrees.

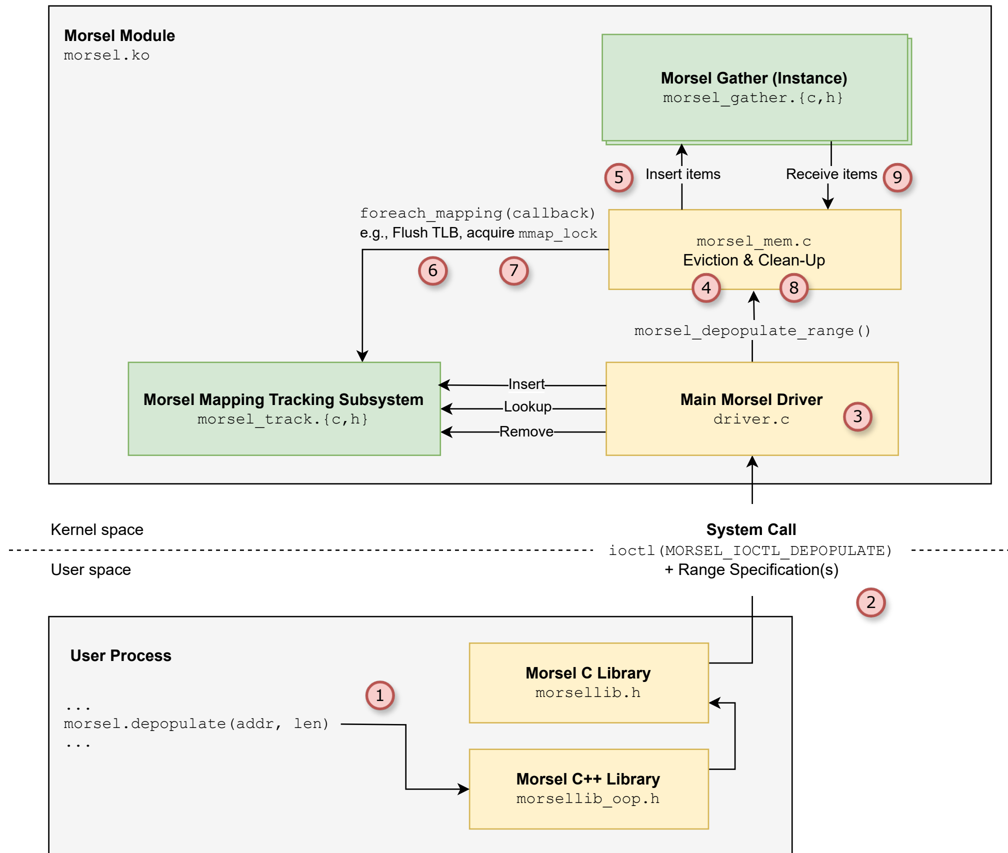


Figure 4.1 – Overview of the extended morsel prototype. Entirely new parts are indicated in green, modified parts are indicated in yellow. Furthermore, this figure focuses on a Depopulation process, for which section 4.1 provides a detailed description.

4.2 A new System Call for Depopulation

In order to expose the new Depopulation functionality to userspace, a new `ioctl` request for morsel file descriptors is added – `MORSEL_IOCTL_DEPOPULATE`. This enables the user to explicitly depopulate a given set of ranges to the morsel surface. The system call accepts multiple range definitions in a single invocation and is therefore considered a *vector operation*. This is an advantage over similar syscalls like `madvise`, which accepts only one range per invocation.

We therefore extend `morsel_calls.c` by the definition of the new system call and the `ioctl()` handler in `driver.c` was extended to receive and decode it.

4.2.1 Specifying the Eviction Range

The invoker provides a set of eviction ranges as arguments to the system call. The definition in section 3.1.1 defines the eviction range on the morsel surface. However, in order to be consistent with other morsel system calls, like `MORSEL_IOCTL_POPULATE`, as well as traditional similar system

calls like `madvise(MADV_REMOVE)` on anonymous shared memory, we choose to let the invoker specify the start position of the eviction range *on a mapping of the morsel* into the invoker's address space. If the invoker wants to evict by surface offset, he can still achieve this by adding the offset to the mapping start address, which is known to him. We accept the weak downside that this requires the invoker to have the morsel mapped. However, because mapping morsels is possible in constant time, this is no serious limitation. From the implementation perspective, it is very convenient to have a mapping available, as it makes iterating over the morsel surface, and potential writes to it, in the kernel much easier.

The mapping, on which a range is defined, should be writable. This is because if the invoker specifies an eviction range that is not aligned with the underlying page boundaries, zero writes are necessary from the ends until the next inner-lying page boundary, which are carried out as a `memset` on the provided mapping. Traditional similar system calls like `madvise(MADV_REMOVE)` on anonymous memory in fact require writable mappings. However, because the mapping is merely a vehicle for the user to specify the range and a helper for the kernel to iterate it, a `force_ro` flag is implemented that even allows depopulation on a read-only mapping if the user is sure to provide page-aligned addresses and thus no writing to the mapping will be necessary.

4.2.2 Permissions

Depopulation can be classified as a destructive write-alike operation. A morsel therefore may only be depopulated by processes that are in possession of a writable file descriptor for it. The driver is in charge to check the `f_mode` against `FMODE_WRITE` when receiving the depopulation system call, and to return `-EPERM` in case of unauthorized attempts.

4.3 Eviction

Eviction is implemented, in `morsel_mem.c`, very close to algorithm 3.1, split into one function per level, `morsel_evict_range_*`, which is conceptally near to Linux's `zap_{pte,pmd,pud,p4d}_range` functions. The implementation makes extensive use of the abstractions provided by Linux, such as the table entry types and the various macros to check entry properties like presence or being a leaf. Technically, the implementation does not iterate the tree but rather the eviction range, in the address space of the invoker, in descending granularities and looks up the respective page table via the `p{g,4,u,m}d_offset` and `pte_offset_map` macros. Those take care of transparent folding during runtime on systems without 5-level paging support. Theoretically, using solely abstractions could even make the implementation architecture independent, but this has not been tested.

In order to delay the cleanup of the detached subtrees, the algorithm submits their references to an instance of the *Morsel Gather*, which is presented in the next section.

4.4 Morsel Gather

Delaying the treatment of the detached trees from the eviction until their consumption during the cleanup makes it necessariy to temporarily store them. The Morsel Gather is designed to manage a dynamically allocated backing storage for references submitted to it, similar to Linux's MMU Gather, but for subtrees instead of individual pages frames. The primary motivation for the gather is to avoid pressuring the slab allocator with bursts of requests for the small

detached_subtree structs, which store the references. Instead, whole pages are requested that serve as *batches* of those structs, and singly chained together. Figure 4.2 illustrates this. This approach was inspired by the Linux's mmu_gather_batch. At the end of the day, the Morsel Gather is a technique to optimize the speed of reference insertion, as part of the eviction.

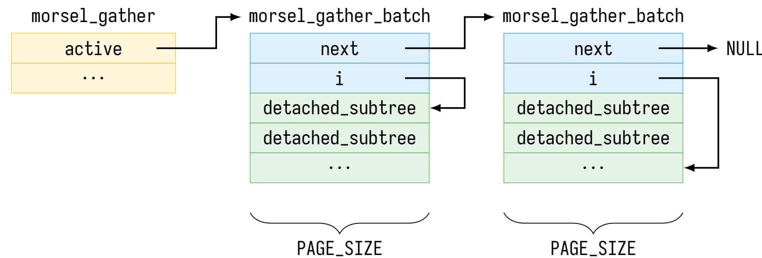


Figure 4.2 – Example of two page-sized batches of detached_subtrees. The batch header contains a pointer to the respective next batch, or nullptr in case of the last batch. The header also contains an index of the next free slot in the respective batch. If a batch is full, a new page/batch is allocated and placed in front, becoming the new active batch. Therefore, all batches, which are not the first one, are full, and the access to the non-full batch is constant.

By design, each Gather instance is only accessed by a single depopulation control flow, eliminating the need for synchronization. In the current implementation, a batch can hold up to 102 detached_subtree structs. In the worst-case event of evicting a bit less than 256 TiB of memory, the amount of pages required to store the 3576 subtree references (see 3.2.3) plus two zero writes is therefore limited to 36 pages. The user must provide a nomem-callback function, which is invoked in case that the Gather fails to allocate a new batch and which is supposed to early-cleanup the already gathered items after issuing a flush. After that, the Gather re-tries the previously failed allocation. This way, the Gather stays working even in very low memory scenarios.

4.5 Morsel Mapping Tracking Subsystem

The MMTS implementation, located in morsel_track.{c,h}, is at its core a *hash list table* data structure, a special form of a hash table that is able to store multiple elements for the same key. A suitable hash list implementation is Linux's rhltable, a variation of the in-tree generic hash-table implementation rhashtable. The elements inserted into the hash table have the morsel ID as their key and essentially hold a pointer to the vm_area_structs, a kernel structure which represents a mapping of that morsel. From the VMA, certain other attributes like the surrounding address space (.vm_mm) or the start address (.vm_start) are accessible. Hash list tables are optimized for lookup by key, so this allows to look up the mappings of a morsel by its ID (key) very fast. The runtime complexity of hash table look-ups usually approximates $\mathcal{O}(1)$. Since the inserted items are just pointers, the storage footprint of each entry is quite low.

The driver.c is adapted such that it

- Initializes a global instance of the MMTS during kernel module initialization (mod_init).
- Registers a new mapping to the MMTS after successful mmap.

- Unregisters an existing mapping from the MMTS in the event of an `vm_unmap`, which may be caused by an explicit `munmap` or due to address space termination.
- Deinitializes the MMTS instance during kernel module deinitialization (`mod_exit`).

Therefore, unloading the module leads to losing the stored tracking information.

The MMTS requires protection of the underlying hashtable, because there exists only one global instance which is subject to potential true-parallel access. Protection of the hash table is realized using the RCU synchronization mechanism, that is well-established in the Linux kernel and has first-class support in the `rhltable` implementation. With RCU, reads are never blocking, even then writing, because the writing happens on a copy and replaces the original after every potential reader left the protected area. See section 2.3. Writing, and with it morsel `mmap`, therefore is potentially blocking, however the protected area is quite small, as hash table insertion, or especially lookup, are usually quite fast operations. The tracking of morsel mappings works completely independently from the existence tracking `TRACK_MORSELS` mechanism.

4.6 Securing the Destroy Operation

Semantically, the morsel *Destroy* operation is illegal on mapped morsels. However, due to limited information, the driver had to rely on the user having unmapped the morsel previously. Now with the MMTS, this information is available and the absence of mappings can be enforced technically. We extended the driver to look up if at least one mapping is known to the MMTS for the subject morsel of a Destroy operation, and returns `-EBUSY` if this is the case.

We decided to make this new behavior the default and thereby introduce a change breaking backwards compatibility, which is acceptable because it was a semantically incorrect use case in the first place. However, we preserved the old behavior behind a `force` flag that disables the new safety check and therefore can be used to force the destruction of a still mapped morsel.

4.7 Safe Deallocation

Page tables can only be considered safe for deallocation if it can be guaranteed that there is no one who still accesses it. This can be the case if, at the same time as the deallocation, there is still a fault-handler operating within the evicted subtree, or another eviction routine. While we can be sure that no *new* control flows can enter, we must wait for all *existing* to leave the collected structures.

In traditional Linux address spaces, the reading and writing to the paging structures is synchronized through the `mmap_lock`, which allows multiple simultaneous readers or a single writer [Cor22]. Therefore, claiming the lock in reading mode ensures the absence of writers, and claiming it in write mode ensures the absence of any readers and other writers. A page fault handler acquires the lock reading, in order to prevent any unexpected simultaneous modifications to the paging structures while it handles the page fault.

However, in the case of morsels, the assumption that an address space has exclusive control over its paging structures does not hold anymore. Because, as an implementation of page table sharing, the page tables of a morsel being part of multiple address spaces simultaneously allows concurrent modifications by fault handlers in different address spaces, which are not synchronized by the same `mmap_lock`. Therefore, we add the following

1. Hold an mmap_lock (reading) during eviction. This is necessary because we want to be able to wait for evictions, like we can for page fault handlers, in the next step.

2. Acquire the mmap_lock (writable) once on *every* address space that a morsel is mapped into. The lock can be released immediately after acquiring it. This ensures that neither a fault handler nor a eviction is still active in the detached subtree.

Speaking in RCU terms, we can block until finish of the Removal phase (= Eviction) and begin the Reclamation phase (= Clean-Up). Figure 4.3 illustrates the procedure.

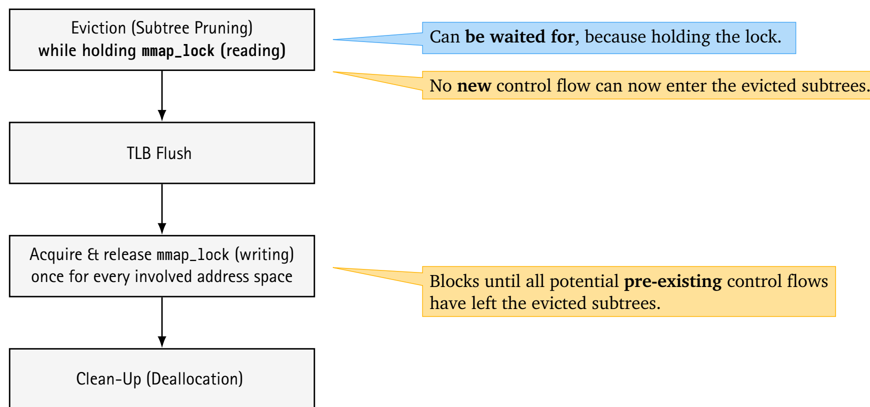


Figure 4.3 – Illustration of the process that ensures that no control flow still operates on the evicted subtrees and therefore deallocation is safe. First, new access to the evicted subtrees is prevented by eviction. Then the exclusive writable mmap_lock is acquired on each address space into which the morsel is mapped, which implies the absence of fault handlers and eviction routines in that address space. If this is the case for every address space *after* eviction, there can be no control flow still be within the collected structures, because no new ones can enter..

It should be noted that this is also only possible with the help of the MMTS. Fistanto, who also used the mmap_lock approach to synchronize the morsel COW modifications with the fault handlers, did not have this information available and therefore was limited to morsels which are only mapped once [Fis24, p. 30].

4.8 Summary

In this chapter, we have seen the implementation of the Depopulation operation as an extension to the morsel prototype, especially to the morsel kernel driver module. On the back of subtree pruning, we are able to evict rapidly and to flush the TLB only once per mapping location, while also having to store very few references. We have seen the morsel Gather, which is designed to efficiently handle bursts of incoming tree references by allocating page-sized batches. The morsel mapping tracking subsystem (MMTS), a second contribution of this thesis, is implemented using Linux's in-tree generic hash list table. On the back of the MMTS, it is possible to reverse map positions on the morsel surface, which allowed us to implement precise TLB range flushing. The MMTS furthermore enables to shortly acquire the mmap_lock on every mapping location, which is used to await that every potential reference holder has left the detached subtrees and they are

therefore safe to deallocate. Finally, the MMTS enables to actually make the COW concept of Fistanto usable for all morsels, not just those mapped only once.

5 ANALYSIS

In the following, we will evaluate the implementation of morsel Depopulation regarding functionality and performance. In the following, the *unmodified prototype* references commit `7d197a0` and the *modified prototype* commit `9ad42b7` of the `research/morsel/eviction/morsel-linux` git repository. Because the modification does not require kernel changes, the kernel is the same in all cases, commit `68873e8` of `research/morsel/morsel-linux-kernel`.

5.1 Evaluation Platform

The evaluation was performed on two systems, with the following hardware specifications.

	(*A*) Server System	(*B*) Workstation
Model	Dell PowerEdge R740	Dell OptiPlex 5000
CPU	2x Intel Xeon Gold 6252	1x Intel Core i7-12700
Architecture	x86-64	x86-64
Base Frequency (Max.)	2.1 GHz (3.7 GHz)	2.1 GHz (4.9 GHz)
Cores (Threads)	2x 24 (2x 48)	1x 12 (1x 20)
DRAM	12x 32 GiB DDR4 @ 2666 MHz	2x 16 GiB DDR4 @ 3200 MT/s
Operating System	Linux `6.1.0-morsel-32094-g68873e89ab77`	

Table 5.1 – Specifications of the evaluation platform

5.2 Functional Evaluation

5.2.1 Conservative Criterion

Fortunately the existing behavior was covered by a set of tests, which can serve as regression tests. I demonstrate that the existing morsel operations, to the extent that is covered by the tests, did not degrade by successfully executing the tests with the unmodified and the modified version. I tested on virtual and bare metal systems, with and without 5-level-paging support.

A regression is not found. Because the existing operations stay untouched, except registering (*Map*), unregistering (*Unmap*) and lookup (*Destroy*) of mapping locations to the hash list table of the MMTS, this was also not likely to happen.

5.2.2 Progressive Criterion

A new set of functional tests is implemented, which tests the modified prototype against the specified behavior. These are intended to demonstrate the functionality on the one hand and to serve as future regression tests on the other. However, it is limited to those effects that are visible on the morsel surface. While it can verify the removal of contents (and therefore result dimension 1), it can not verify removal of backing memory (result dimension 2) via the surface.

For the sake of demonstrating the removal of backing memory, the following example is conducted, where a program successively populates sizes of 1 GiB up to 5 GiB on different positions of a morsels surface and depopulates the area thereafter. During this, the amount of available page frames, as reported by `sconf(_SC_AVPHYS_PAGES)`, is measured. The morsel is not destroyed in between. The results are visualized in figure 5.1. As we can see, the populations lower the amount of available page frames, followed by the Depopulation which causes the complementing rising edges. This shows the following things:

1. The Depopulation in fact deallocates memory off a morsel.

2. The amount of deallocated memory equals, from what we can see, to the respective previously allocated memory, since the graph always comes back to the same baseline. Therefore no (obvious) memory leaks are present. After, in sum, 15 GiB, the graph still returns to the baseline.

3. The Depopulation is faster than the Population for each of the sizes.

4. The position on the morsel surface seems to be irrelevant for the functionality.

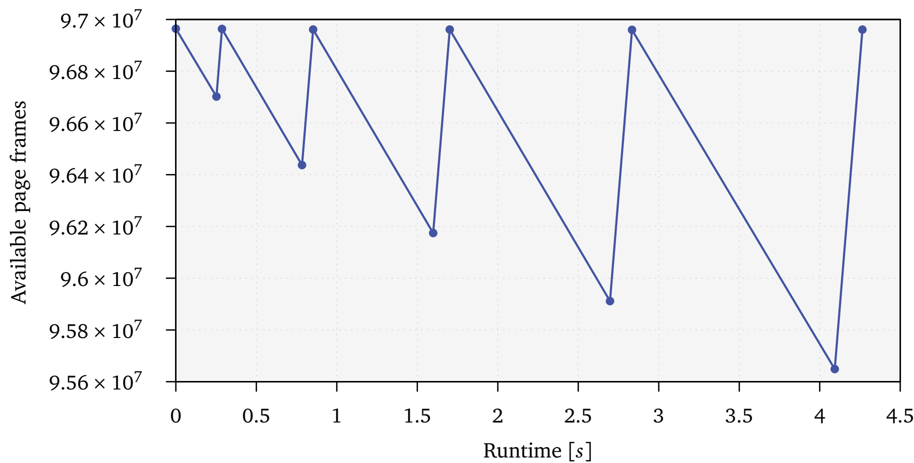


Figure 5.1 – Number of available page frames as returned by `sconf(_SC_AVPHYS_PAGES)` during the runtime of a program that populates and depopulates different ranges on the morsel surface. Measured on evaluation system *A*. Rising edges are caused by Depopulation.

5.3 Performance Evaluation

5.3.1 Influence on Mapping Speed

Because the *Map* operation in the modified prototype registers a mapping to the MMTS, which includes allocation of a mapping entry and inserting it into the hashtable, it raises the question if this might have a measurable impact on the mapping time. Constant mapping time is an important property of morsels, which should not be degraded.

For this purpose, the mapping speed benchmark of Halbuer [Hal22, p. 40] (`bench54`) was repeated for the unmodified and modified version. The results are visualized in figure 5.2. In both cases, only the actual mapping time is displayed. We can see that the mapping time remains constant for all range sizes, which was expected. Furthermore we can see that the runtimes for the unmodified and modified prototypes are identical. Therefore, even though the modified version includes additional allocation and hash table insertion overhead, this not have a negative impact on mapping speed at all, according to the benchmark. We can see that the variance, beginning at the 2^{12} pages mark, is increasing. Because this is the case also for the unmodified code, this is some effect that is outside of my modifications.

It should be noted that *Map* and *Unmap* now have a potentially blocking case, if it had to wait for RCU readers to leave, after modification of the hash table. Given that even the hash table insertion overhead is small to an extent that could not be measured here, I assume that waiting for a lookup (which should be in general faster, and is also never blocked by RCU), should not be a concern.

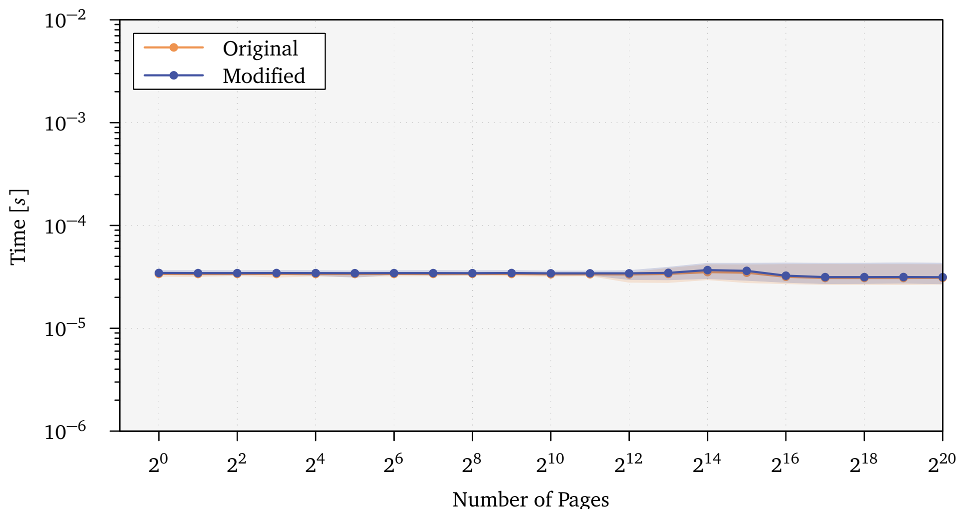


Figure 5.2 – Comparing modified and unmodified versions by results of mapping speed benchmark `bench54` by Halbuer [Hal22]. Mapping time by exponentially increasing number of pages. Connected medians with 80 % point-wise confidence bands. 1024 samples per data point. Measured on evaluation system *B*.

5.3.2 Microbenchmarks of Depopulation Subroutines

The Linux's kernel `ktime` interface allows to retrieve reliable, highly-precise monotonic timestamps that allow measuring short time intervals accurately. It was therefore possible to zoom into the Depopulation and benchmark the runtime of the three individual subroutines, performing the measurements in the morsel driver itself, removing the system call overheads. The in-kernel Depopulation benchmarking was implemented using `ktime_get_ns()` and is enabled via a `BENCH_DEPOPULATION` flag. The morsel driver then prints the results using `ERR_OUT`, this way the outputs can be collected via `dmesg` and without the need to enable debugging in the morsel driver, which would negatively impact its performance.

All presented benchmark results in the following subsections were retrieved in a single run, evicting exponentially increasing sizes from 4 KiB up to 4 GiB. Each individual range size was measured 1024 times on evaluation system *B*. The combined benchmark result is visualized in figure 5.3 and will be discussed in the following.

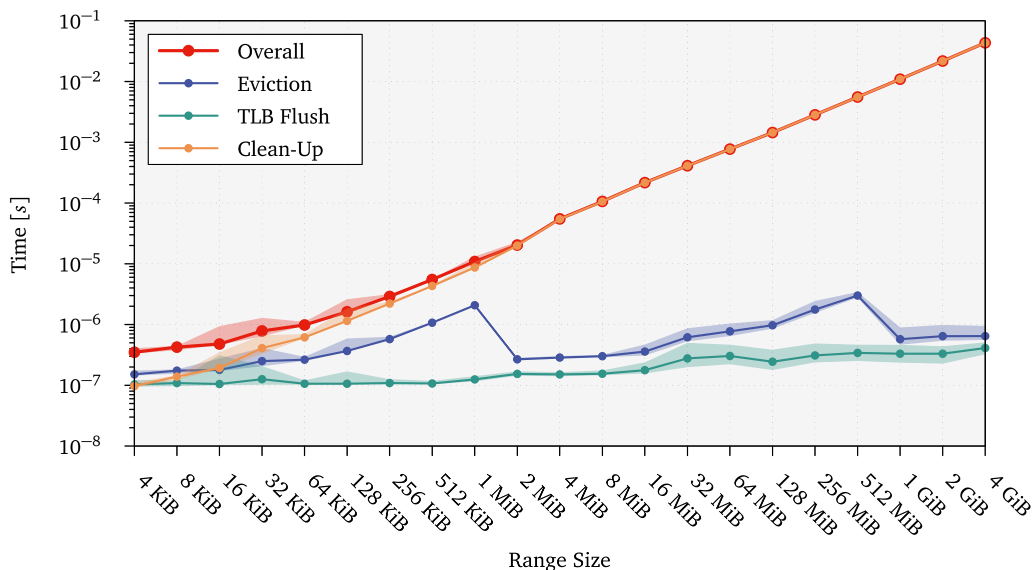


Figure 5.3 – Proportion of subroutines to overall runtime. Connected medians with 80 % point-wise confidence bands. 1024 samples per data point. Double-logarithmic scale.

Eviction Subroutine

The first thing to note is, that no observed datapoint for the eviction exceeds 4 μs, although the range size is exponentially increasing. The clearly visible sudden drop of runtime at the 2 MiB and 1 GiB marks can be explained by the subtree pruning algorithm. For these sizes, the whole eviction range is completely covered by a single subtree, and the algorithm is able to prune it with a single operation. No further descending is required. This correlation is confirmed by figure 5.4, which shows the amount of pruned subtrees by eviction range size. From there on, more and more subtrees need to be descended into and pruned, so the runtime is increasing as well, until the next whole-subtree prune is possible.

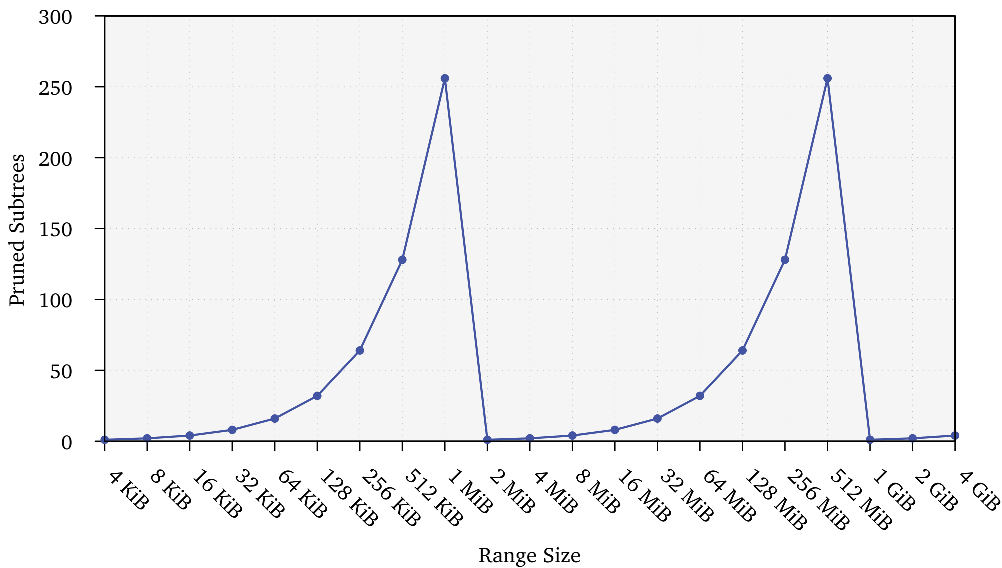


Figure 5.4 – Amount of pruned subtrees by range size, which is equal among all samples of the same range size. Is almost identical to the blue line in figure 5.3 (mind that that the ordinate in that figure is scaled logarithmically). It clearly shows that the eviction runtime depends on the amount of pruned trees.

TLB Flushing Subroutine

The measurement results are included in figure 5.3. We can see that, as far es the benchmark is concerned, the TLB flushing is constantly below 1 μs, and thereby even below the eviction. Note that the severe effect of the flush, the impact on future address translations, is not measured here, this is purely the time to issue the flush locally and potentially carrying out remote shootdowns. From this data, it is hard to tell whether this included remote shootdowns.

The influence of the morsel module to the runtime of this phase is very limited, as it is solely the hashtable lookup. Everything else is up to the architecture specific code that implements TLB range flushing on x86.

If two processes that have the same morsel mounted once shared the same CPU, the IPIs to this CPU should be batched together. If the flushing code not already does it, this could be a way for future optimization.

The `tlb_single_page_flush_ceiling` in the x86 TLB code is on the evaluation system 33 , which is the default[5]. Therefore, I expected to see an increase of runtime in $33 * 4\,\text{KiB} = 132\,\text{KiB}$, because at this point the implementation will choose to perform full flushes.

Furthermore, if a flush altered page tables and this is indicated via `freed_tables`, this causes all CPUs in Lazy TLB mode to flush.

Clean-Up Subroutine

As we can see in figure 5.3, while the eviction and TLB flush always stay below 10 μs, the clean-up scales linear to the exponential amount of memory and thus also the runtime increases. In the case of 4 GiB ranges, the clean-up takes circa 50 000 μs, while the other two combined

[5] see `arch/x86/mm/tlb.c` of Linux 6.1 sources, line 937.

around 3 μs. This is, because the clean-up has to iterate the detached tree and visit every leaf, and therefore is linear to the eviction range. That is confirmed by the flame graph in figure 5.7a, which depicts the internals of a morsel Depopulation. We can see that the `morsel_gather_free` takes up `morsel_depopulate_range` completely, the other methods for eviction or TLB flushing are not even recorded. I intentionally removed the `inline` statements for these functions while recording the flame graph, but this did not change the outcome.

The flame graph also shows, that the actual majority of the runtime is spent outside the module, in kernel functions related to freeing page frames. This is time that we could hardly further optimize. A possible optimization would be to collect batches and return multiple page frames at once back to the page frame allocator, if it would support that. The page frames are not necessarily physically aligned, though. Another optimization approach would be to parellelize the clean-up step. A coordinator could delegate the cleanup of individual subtrees to worker threads. Trees are quite suited for this, because the subtrees are independent of each other.

5.3.3 Depopulation Speed compared to POSIX Shared Memory

Conventional POSIX shared memory (SHM) could be considered a competitor to morsels functional wise. They render a piece of virtual memory that can be referenced by name, shared between processes and mounted into them simultaneously, serving as a data package or IPC channel. The equivalent of morsel Depopulation on SHM is `madvise(MADV_REMOVE)`. A comparison of these operations, regarding performance, was conducted via the `depopulation-speed` benchmark. In the benchmark, a morsel is subsequently created, mapped, populated via morsel Population, depopulated via morsel Depopulation, unmapped and destroyed. For comparison, a SHM is requested, mapped, populated via `madvise(MADV_POPULATE_WRITE)`, depopulated via `madvise(MADV_REMOVE)` and unmapped. In both cases, the eviction range size is identical and the depopulation time is measured using a precise monotonic clock suitable for measuring short intervals. For both variants, the measurement is repeated 1024 times for exponentially increasing range sizes, starting at 4 KiB and ending at 4 GiB. The results of the measurement are visualized figure 5.5. We can see that morsel Depopulation is constantly below its competitor, except for the smallest size, 4 GiB, where it is equal. While, for short ranges, the difference is small to non-existent, with rising range sizes the speedup of morsel Depopulation approaches an almost constant speedup factor towards the SHM. The speedup factor between the medians is displayed in figure 5.6.

It should be mentioned that both measurements are done in userspace and wrap a system call, so they both include context switching overhead. I assume the context switching overhead of `ioctl` and `madvise` to be similar, so it does not influence the outcome of a relative comparison if both are affected by the same offset. From the perspective of an end user, only the total runtime of the eviction would be relevant anyway.

In an attempt to explain why morsel Depopulation is considerably faster than eviction on SHM, I conducted a qualitative evaluation of the two methods and recorded flamegraphs showing the internals, which can be seen in figure 5.7. In both cases the same range size was evicted (32 GiB) and the runtime of the Depopulation took 1.16 s and 6.21 s, which is a factor of 5.1, confirmed by figure 5.6 and which indicates that the trend extends further to 32 GiB. For the morsel, we can see that the runtime of `morsel_depopulate_range` is dominated by `morsel_gather_free`, which is an effect further discussed in section 5.3.2, and that it spends almost all of its runtime outside the morsel module in kernel methods related to freeing pages.

For the SHM, we can see that the `madvise_remove` in the case of SHM translates to `shmem_fallocate` (which was to expect because it is implemented on top of `fallocate` with `FALLOC_FL_PUNCH_HOLE`

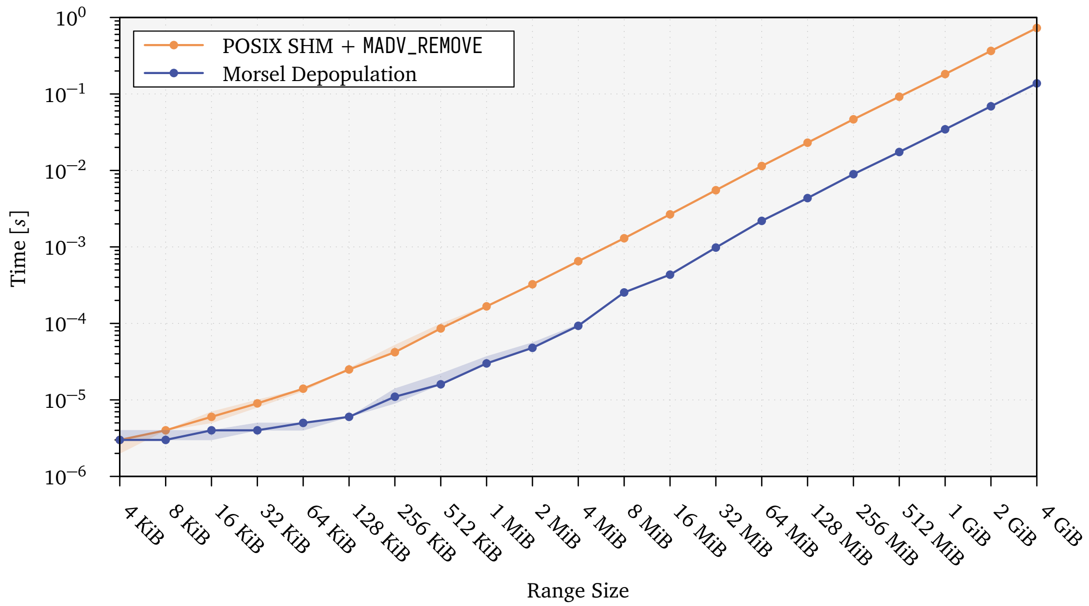


Figure 5.5 – Comparing the `depopulation-speed` benchmark results of morsel Depopulation and `madvise(MADV_REMOVE)` on SHM. Connected medians with 80 % point-wise confidence bands. 1024 samples per data point. Measured on evaluation system *A*.

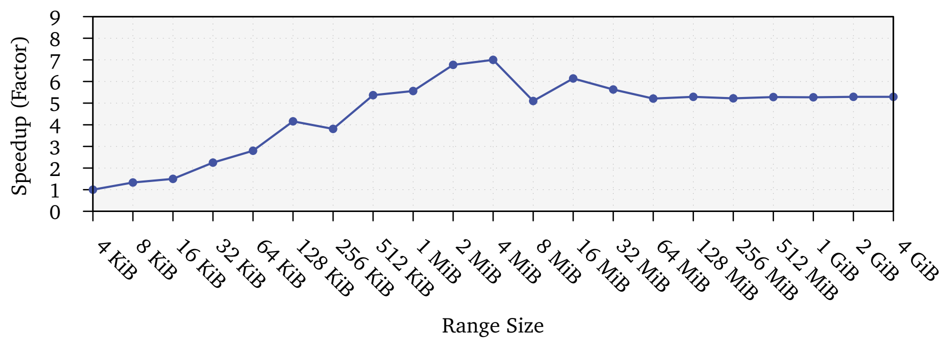
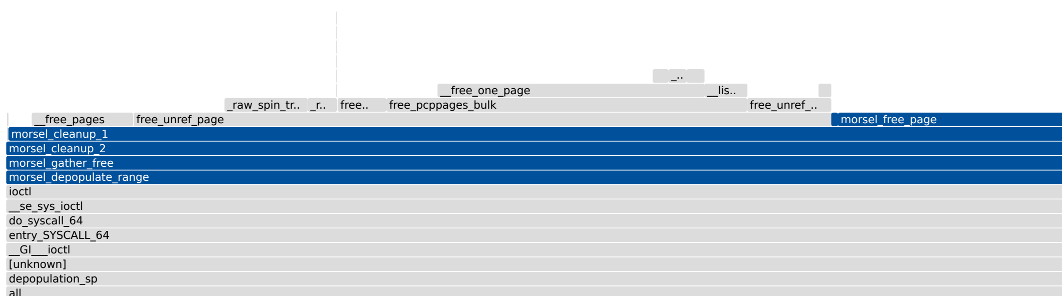
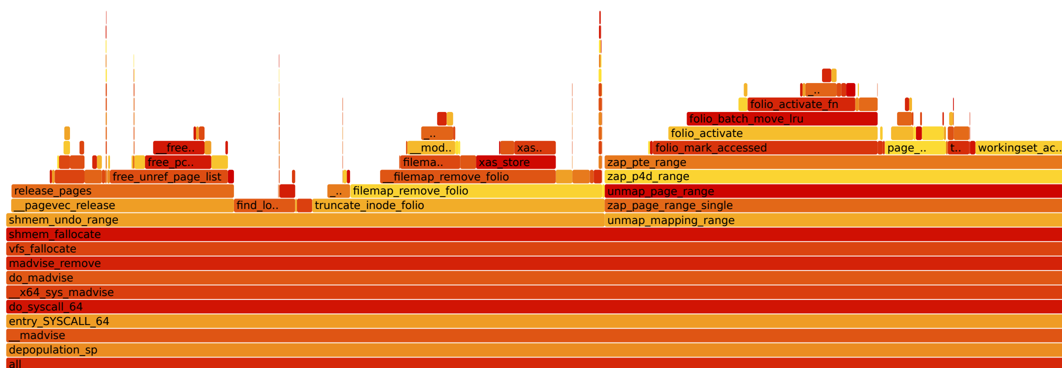


Figure 5.6 – Speedup of morsel Depopulation compared to SHM depopulation. Factor between the medians in figure 5.5.

mode), which splits circa 60 % into `shmem_undo_range` and 40 % into `unmap_mapping_range`. While the latter can be considered to implement the actual eviction (it iterates and destructively alters the page tables of a given range), the former is related to “remove range of pages and swap entries from page cache, and free them”.[6] This is, because SHM in Linux seems to be implemented based on file-backed memory. Morsels, in comparison, are implemented purely on what Linux would consider anonymous memory. Therefore, morsels inherently do not require managing of page cache. At the same time, morsels do not maintain their page frames’ metadata in the `struct page`, like reverse mappings via `anon_vmas`. The larger the size, the more pays off not managing each page frames metadata. Those are the two reasons that I suspect to be responsible for the considerable speedup.



(a) Morsel Depopulation, zoomed into the `morsel_depopulate_range` method. Methods that belong to the morsel driver are colored blue. Runtime circa 1.16 s.



(b) Eviction of SHM using `madvise(MADV_REMOVE)`, zoomed into the `madvise_remove` method. Runtime circa 6.21 s.

Figure 5.7 – Flame graphs depicting the internals during depopulation of 32 GiB of Morsels and POSIX SHM.

5.4 Summary

In this chapter, we evaluated the prototypical implementation of the morsel Depopulation operation, on top of the morsel prototype by Halbuer. We could see that the modifications did not degrade the pre-existing morsel operations functional or performance wise. We could see

[6] See `mm/shmem.c` of Linux 6.1 sources, line 902.

that the prototype is functionally working as expected, evicting specified regions of pages on the morsel surface off a morsel and returning them properly to the operating system. Performance wise, we could see that Eviction is scaling very well to the range size, remaining constantly in the range of a few microseconds. We saw that Clean-Up clearly dominates the Depopulation runtime, because opposed to the other subroutines, it scales linear to the eviction range size. However, since a clean-up necessarily must visit every page in order to deallocate it, below-linear runtime complexity is hardly possible. We further saw that morsel Depopulation significantly outperforms `madvise(MADV_REMOVE)` on POSIX shared memory by up to 7 times. This can be explained by morsels being purely anonymous memory, further without any overhead in metadata (except for, now, tracking of mapping locations).

6 CONCLUSION

6.1 Summary

This thesis contributed the design and implementation of a novel morsel operation, the *Depopulation*. It enables to explicitly evict a range on the surface of a morsel, rendering the first way to deallocate memory off a morsel without destroying it. Based on the design of a lock-free range eviction algorithm that simultaneously has a low runtime complexity and a low storage footprint, it is possible to issue only the minimal amount of precise TLB range flushes, while also reducing the inconsistency time window and the required temporary gather storage. The amount of pruned subtrees is the upper bound for both the eviction runtime and the gather storage. In the worst case on x86-64 with 5-level-paging – depopulating circa 256 TiB off a fully 4 KiB-populated order 4 morsel – this upper bound is just 3576, which can be pruned off in microseconds and requires just a few bytes of storage each. The implementation issues the TLB flushes, using the Linux's TLB flushing interface, in the most precise way, one range flush for each mapping location. The gathering of the bursts of temporary tree root references avoids using the slab allocator and manages an own list of page-sized batches, inspired by Linux's MMU Gather.

A second contribution of this thesis is the design and implementation of a tracking mechanism, the morsel mapping tracking subsystem. Based on this, it was possible to implement reverse mapping in morsel-granularity, which enables precise TLB flushes. Compared to page-granular reverse mapping, this requires significantly less bookkeeping overhead, both in space and time. This was not only the foundation for Depopulation, but also enables general COW for morsels, as well as other operations like swapping or transparent physical memory relocation in mapped morsels.

The evaluation demonstrated that the implementation did not degrade existing functionality and performance, is functional in both result dimensions (removal of contents and deallocation of backing storage), and significantly outperforms range eviction on POSIX shared memory, for range sizes above 512 KiB the speedup is even constantly above factor 5. The clean-up, being the only subroutine scaling linear to the eviction range size, dominates the runtime.

6.2 Future Work

The first starting point for future work would certainly be to implement the theoretical COW extension for subtree pruning and clean-up into the prototype.

A future improvement, regarding eviction of morsel memory, could be to make morsels support `madvise`, especially `MADV_POPULATE_WRITE` and `MADV_REMOVE` are candidates that just need to invoke the corresponding morsel operation (Populate and Depopulate). This would make morsels a less special kind of memory, from the users perspective.

Another feature could be to implement a mechanism that allows the kernel to actively steal null-pages from morsels in the case of memory pressure. Because uninitialized morsel pages are per convention null, it is functionally equivalent if a page is present and contains null, or is absent.

LIST OF ACRONYMS

API application programming interface
ASID address space identifier
CAM content-addressable memory
CAS compare-and-swap
CISC complex instruction set computer
COW copy-on-write
CPU central processing unit
dTLB data translation lookaside buffer
HBM high-bandwith memory
HMS heterogeneous memory system
IOMMU input–output memory management unit
IPC inter-process communication
IPI inter-processor interrupt
iTLB instruction translation lookaside buffer
KPTI kernel page-table isolation
LRU least recently used
MMTS morsel mapping tracking subsystem
MMU memory management unit
NVM non-volatile memory
OS operating system
PAE physical address extension
ParPerOS Parallel Persistency OS
PCID process-context identifier
PFN page frame number
POSIX portable operating system interface
PT page table
RC reference count (in the context of Copy-on-Write)
RCU read-copy-update
RDMA remote direct memory access
RISC reduced instruction set computer
SHM POSIX shared memory
SMP symmetric multiprocessing
TLB translation lookaside buffer
VMA virtual memory area
VPN virtual page number

LIST OF FIGURES

2.2 Scheme of address translation on modern systems with 5-Level-Paging 5
2.3 Visualization reverse mapping anonymous memory in Linux 5
2.4 Scheme of address translation involving a translation lookaside buffer 7
2.5 A morsel as indivisible, self-contained subtree of the page table hierarchy 10
2.6 Example of a shared order 3 morsel being populated. 12

3.1 Figure visualizing the basic notions of morsel Depopulation. 15
3.2 Example of a mmu_gather instance that itself decides to flush and free the gathered page frame batches upon insertion of a page frame, based on an internal threshold. 18
3.3 Overview on the Depopulation subroutines 19
3.4 Example how 2 MiB of aligned contiguous virtual memory can be evicted in functionally equivalent ways that differ in runtime of the eviction 21
3.5 Subtree Pruning Example on an order 2 morsel with mixed page granularities . . 23
3.6 Example of the situations with the most amount of pruned subtrees for the lowest 3 morsel orders 24
3.7 Example of COW scenario I. 26
3.8 Example of COW scenario II. Red indicates morsel surface. 26
3.9 Example of COW scenario III. 27
3.10 Example of COW scenario IV. No modification is required. 27
3.11 Example of how the start of the eviction range can be reverse-mapped to all virtual address ranges translating to it 29
3.12 Example cleanup of a detached order 2 tree that contains a COW-shared order 1 subtree 31

4.1 Overview of the extended morsel prototype 34
4.2 Example of two page-sized batches of detached_subtrees 36
4.3 Illustration of the process that ensures that no control flow still operates on the evicted subtrees and therefore deallocation is safe 38

5.1 Number of available page frames as returned by sconf(_SC_AVPHYS_PAGES) during the runtime of a program that populates and depopulates different ranges on the morsel surface 42
5.2 Comparing modified and unmodified versions by results of mapping speed benchmark bench54 by Halbuer 43
5.3 Proportion of subroutines to overall runtime 44

5.4 Amount of pruned subtrees by range size, which is equal among all samples of the same range size . . . 45
5.5 Comparing the `depopulation-speed` benchmark results of morsel Depopulation and `madvise(MADV_REMOVE)` on SHM . . . 47
5.6 Speedup of morsel Depopulation compared to SHM depopulation . . . 47
5.7 Flame graphs depicting the internals during depopulation of 32 GiB of Morsels and POSIX SHM. . . . 48

LIST OF TABLES

2.1 Morsel surface size by order. 10

3.1 Worst-Case amount of pruned subtrees by morsel order 25

5.1 Specifications of the evaluation platform 41

LIST OF ALGORITHMS

3.1 The Subtree Pruning algorithm 21
3.2 The Clean-Up algorithm 30

REFERENCES

[ADAD23] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Version 1.10. Arpaci-Dusseau Books, Nov. 2023.

[Alb23] Kenny Albes. "IOMMU-unterstütztes Speichermanagement: Teilen virtueller Speicherobjekte mit PCIe-Geräten im Linux Kern." Master's Thesis. Institut für Systems Engineering: Leibniz Universität Hannover, Dec. 3, 2023.

[AMD23] AMD. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. Revision 3.41. June 2023.

[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: Fifteenth EuroSys Conference 2020. Heraklion Greece: ACM, Apr. 15, 2020, pp. 1–14. ISBN: 978-1-4503-6882-7. DOI: `10.1145/3342195.3387518`. (Visited on 06/25/2024).

[Aum24] Paul Aumann. "Optimizing Memory Metadata: Dynamic Allocation of Struct Pages in the Linux Kernel." Master's Thesis. Institut für Systems Engineering: Leibniz Universität Hannover, June 14, 2024.

[Bia23] Darian Biastoch. "Extending Linux for Managing Volatile and Non-Volatile Virtual Memory Objects as Files." PhD thesis. Institut für Systems Engineering: Leibniz Universität Hannover, Oct. 26, 2023.

[Bla+89] D. L. Black et al. "Translation lookaside buffer consistency: a software approach." In: *Proceedings of the third international conference on Architectural support for programming languages and operating systems*. ASPLOS89: Int'l Conference on Architecture Support for Programming Lang & Operating Systems. Boston Massachusetts USA: ACM, Apr. 1989, pp. 113–122. ISBN: 978-0-89791-300-3. DOI: `10.1145/70082.68193`. URL: `https://dl.acm.org/doi/10.1145/70082.68193` (visited on 07/02/2024).

[Bol24] Marko Bolowski. "Huge-Page-Unterstützung für eigenständige virtuelle Speicherobjekte im Linux Kernel." Bachelor's Thesis. Institut für Systems Engineering: Leibniz Universität Hannover, Mar. 18, 2024.

[Cor17] Jonathan Corbet. *The current state of kernel page-table isolation*. Dec. 20, 2017. URL: `https://lwn.net/Articles/741878/` (visited on 06/26/2024).

[Cor22] Jonathan Corbet. *Concurrent page-fault handling with per-VMA locks*. Sept. 5, 2022. URL: `https://lwn.net/Articles/906852/` (visited on 06/22/2024).

[DL] Christian Dietrich and Daniel Lohmann. *ParPerOS: Parallel Persistency OS*. URL: `https://gepris.dfg.de/gepris/projekt/501887536` (visited on 06/30/2024).

[Fis24] Pasha Alghifari Fistanto. “Efficient Copy-on-Write for Self-Contained Virtual-Memory Objects in the Linux Kernel.” Master’s Thesis. May 2, 2024.

[Gor07] Mel Gorman. *Understanding The Linux Virtual Memory Manager*. July 9, 2007.

[Hal22] Alexander Halbuer. “Self-Contained Virtual-Memory Areas for Non-Volatile RAM in the Linux Kernel.” Master’s Thesis. Institut für Systems Engineering: Leibniz Universität Hannover, Nov. 16, 2022.

[Hal+23] Alexander Halbuer et al. “Morsels: Explicit Virtual Memory Objects.” In: *Proceedings of the 1st Workshop on Disruptive Memory Systems*. DIMES ’23: 1st Workshop on Disruptive Memory Systems. Koblenz Germany: ACM, Oct. 23, 2023, pp. 52–59. ISBN: 9798400703003. DOI: 10.1145/3609308.3625267.

[Int23] Intel. *Intel® 64 and IA-32 Architectures Software Developer’s Manual. Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*. Sept. 2023.

[LF23] Daniel Lohmann and Björn Fiedler. *OS Challenges for Modern Memory Systems*. June 5, 2023.

[Maa+20] Steffen Maass et al. “ECO TLB: Eventually Consistent TLBs.” In: *ACM Transactions on Architecture and Code Optimization* 17.4 (Dec. 31, 2020), pp. 1–24. ISSN: 1544-3566, 1544-3973. DOI: 10.1145/3409454. (Visited on 06/26/2024).

[McC03] Dave McCracken. “Sharing Page Tables in the Linux Kernel.” In: *Proceedings of the Linux Symposium*. Linux Symposium. 2003, pp. 315–320.

[McC06] Dave McCracken. “Shared Page Tables Redux.” In: *Proceedings of the Linux Symposium*. Linux Symposium. 2006, pp. 117–122.

[McK07] Paul McKenney. *What is RCU, Fundamentally?* Dec. 17, 2007. URL: https://lwn.net/Articles/262464/ (visited on 06/26/2024).

[Mil] David S. Miller. *Cache and TLB Flushing Under Linux*. URL: https://www.kernel.org/doc/html/latest/core-api/cachetlb.html (visited on 06/26/2024).

[MK97] Moon-Seek Chang and Kern Koh. “Lazy TLB consistency for large-scale multiprocessors.” In: *Proceedings of IEEE International Symposium on Parallel Algorithms Architecture Synthesis*. IEEE International Symposium on Parallel Algorithms Architecture Synthesis. Aizu-Wakamatsu, Japan: IEEE Comput. Soc. Press, 1997, pp. 308–315. ISBN: 978-0-8186-7870-7. DOI: 10.1109/AISPAS.1997.581683. URL: http://ieeexplore.ieee.org/document/581683/ (visited on 07/02/2024).

[TB24] Andrew S. Tanenbaum and Herbert Bos. *Modern operating systems*. 5th edition. OCLC: 1422578969. Harlow, United Kingdom: Pearson Education Limited, 2024. ISBN: 978-1-292-72789-9.

[Tel90] P.J. Teller. “Translation-lookaside buffer consistency.” In: *Computer* 23.6 (June 1990), pp. 26–36. ISSN: 0018-9162. DOI: 10.1109/2.55498. URL: http://ieeexplore.ieee.org/document/55498/ (visited on 07/02/2024).

[Vil+11] Carlos Villavieja et al. “DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory.” In: *2011 International Conference on Parallel Architectures and Compilation Techniques*. 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT). Galveston, TX, USA: IEEE, Oct. 2011, pp. 340–349. ISBN: 978-1-4577-1794-9 978-0-7695-4566-0. DOI: 10.1109/PACT.2011.65. URL: http://ieeexplore.ieee.org/document/6113842/ (visited on 07/02/2024).

[WAH21] Andrew Waterman, Krste Asanovic, and John Hauser. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*. Version 20211203. Dec. 2021.

[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: Sixteenth European Conference on Computer Systems. Online Event United Kingdom: ACM, Apr. 21, 2021, pp. 540–555. ISBN: 978-1-4503-8334-9. DOI: 10.1145/3447786.3456258. URL: https://dl.acm.org/doi/10.1145/3447786.3456258 (visited on 07/01/2024).