Morsels: Explicit Virtual Memory Objects

Alexander Halbuer
Leibniz Universität Hannover
Germany
halbuer@sra.uni-hannover.de

Christian Dietrich
Hamburg University of Technology
Germany
christian.dietrich@tuhh.de

Florian Rommel
Leibniz Universität Hannover
Germany
rommel@sra.uni-hannover.de

Daniel Lohmann
Leibniz Universität Hannover
Germany
lohmann@sra.uni-hannover.de

Abstract

The tremendous growth of RAM capacity – now exceeding multiple terabytes – necessitates a reevaluation of traditional memory-management methods, which were developed when resources were scarce. Current virtual-memory subsystems handle address-space regions as sets of individual 4-KiB pages with demand paging and copy-on-write, resulting in significant management overhead. Although huge pages reduce the number of managed entities, they induce internal fragmentation and have a coarse copy granularity.

To address these problems, we introduce Morsels, a novel virtual-memory–management paradigm that is purely based on hardware data structures and enables the efficient sharing of virtual-memory objects between processes and devices while being well suited for non-volatile memory. Our benchmarks show that Morsels reduce the mapping time for a 6.82-GiB machine-learning model by up to 99.8 percent compared to conventional memory mapping in Linux.

1 Introduction

Memory management in operating systems has historically been shaped by the fundamental assumptions that both physical main memory (RAM) and – to a lesser degree – *virtual memory (VM)* [6] are scarce resources, which should be provided in an architecture-independent way [23].

Demand paging and *copy-on-write (COW)* [22] are fundamental OS techniques to cope with the limited physical memory: All physical memory is allocated, swapped, shared, or transferred [28] between address spaces on the granularity of individual page frames, typically sized at 4 KiB and mounted one-by-one to the respective page tables. This allows for lazy and memory-efficient implementations of `fork()` and `mmap()`, but comes at the price of significant bookkeeping overhead and contention [20, 32] inside the kernel and makes transferring and sharing large in-memory objects across devices and processes expensive. As mitigation, huge pages merge a complete page-table subtree into a large contiguous memory unit (on AMD64: 2 MiB/1 GiB), thereby reducing the overhead for large objects. However, although Linux and BSD support huge pages [17, 15], they have been poorly adopted as an explicit measure for data sharing and transfer, as their (now too coarse) granularity can lead to memory bloat caused by internal fragmentation and high latencies of COW faults [15, 19].

Given the ongoing trend towards in-memory databases [8, 24] and large AI models (e.g., up to 56 GiB for Meta's current LLaMA 2 models), applications require efficient means to provision and share large in-memory objects not necessarily backed by files. Also, non-CPU processing elements, like GPUs and other accelerators, are more and more becoming an integrated part of the system, which will even gain cache-coherent memory access (i.e., with CXL [4] and Gen-Z [9]) in the near future. Combined with system disaggregation [25], this will push the (local) CPU from its exclusive place at the top of the memory hierarchy. Therefore, the OS must also expose its interfaces [30] to these external devices. For memory, the page-transfer–oriented I/O model [20] is already being replaced by *shared virtual memory (SVM)* [31], where memory ranges are continuously shared with I/O devices through the IO-MMU. All this calls for more explicit management of in-memory objects, which should become first-class entities – similar to files but independent from the file abstraction.

About This Paper

We propose Morsels, a new memory abstraction based on the *memory-managing unit (MMU)* data structures in addition to the existing abstractions in OS kernels. The design follows an exokernel-like philosophy [7] by sacrificing hardware independence in favor of performance and reduced management overhead. Our prototypical implementation focuses on the AMD64 architecture, but the Morsel concept is generally applicable to all architectures with multi-level paging support. We claim the following contributions:

DIMES '23, October 23, 2023, Koblenz, Germany

ACM ISBN 979-8-4007-0300-3/23/10.
https://doi.org/10.1145/3609308.3625267

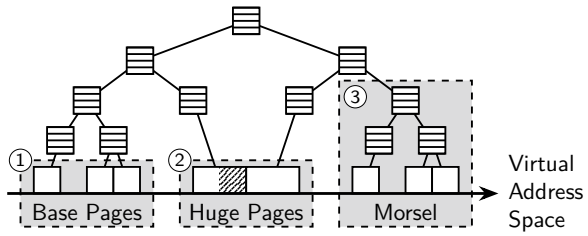

Figure 1. Conceptual design of a Morsel – take a subtree of the page table hierarchy as an indivisible, self-contained memory object.

- We explore the weakness of classic file-backed mappings for applications with large memory footprints.
- We propose Morsels as a new memory abstraction and provide a Linux implementation to make them available to a wide range of applications and use cases.
- We demonstrate the benefits of Morsels over the existing shared-memory primitives with two case studies.

2 Concept

With Morsels, we lift the management of memory from individual pages to larger, indivisible memory objects in order to overcome the shortcomings of page-by-page management. Our design aims to achieve the following objectives:

1. Scalable memory-object management for huge memory capacities and a high degree of parallelism.
2. Support different types of memory in a heterogeneous system (e.g., volatile, persistent, high bandwidth).
3. Store densely packed, serialized data as well as scattered, pointer-based data structures.
4. Efficiently share large memory objects between processes and devices.
5. Avoid memory bloat by fine-grained and adaptive memory provisioning.

In a nutshell, a Morsel is a subtree of the page-table hierarchy that includes all related management, page-table, and user data, making it fully self-contained. Fig. 1 visualizes our concept using a simplified page-table hierarchy (4 levels, 2 entries/level), showing the same 3-page VM layout with different primitives: (1) With a standard VM area composed of base pages, three individual pages need to be managed (e.g., individually accounted for and mapped by the pager). (2) With huge pages, we only have to keep track of two entities but lose one base page worth of memory due to internal fragmentation. (3) The Morsel is managed and shared as a single indivisible unit, while we keep fine-grained control over memory provisioning as we can populate the Morsel sparsely with base pages. In comparison to standard mappings, Morsels require no additional metadata, such as per page-frame reference counters or reverse-mapping information. For Morsels, the information stored in the page tables is sufficient.

2.1 Near-Hardware Design

We restrict Morsels to architecture-specific virtual-object sizes and enforce a natural alignment to achieve a low-overhead design. The smallest possible Morsel is a base page, while the largest is a subtree of the root page table. To refer to specific Morsel sizes, we define the order N as the number of page-table levels covered by a Morsel so that a base-page Morsel has order $N = 0$. With base-page size S_{base} and the number of entries per page table c, the VM extent of an order-N Morsel is $S_{base} \cdot c^N$.

For the AMD64 architecture with 4 KiB base-page size ($S_{base} = 4\,\text{KiB}$), 5-level paging ($N \in \{0, ..., 4\}$), and 512 entries per page table ($c = 512$), Morsels can be of VM sizes 4 KiB, 2 MiB, 1 GiB, 512 GiB, and 256 TiB. As Morsels can be sparsely populated, their virtual size must not be confused with the actual memory usage. For high-order Morsels, it will often even exceed the system's physical memory. The significant discrepancy between the supported virtual sizes likely results in internal fragmentation of the *virtual* address space. However, this is not a significant concern since the virtual address space is rarely scarce in most applications.

2.2 Usage and Interface

On the application side, we represent Morsels by *file descriptors (FDs)* to take advantage of the existing descriptor primitives (i.e., inter-process descriptor passing). However, Morsels are not connected to regular files. They live exclusively in the main memory. Transparent persistence, like in file-backed mappings, is only possible by placing the Morsel, its user *and* meta data, in *non-volatile memory (NVM)*.

We plan to provide a small library that hides interaction with the Morsel kernel module from the user and provides name resolution. The following code snippet shows the intended usage from a developer's point of view:

```
int fd = create_morsel("my_morsel", 2); // Create new morsel (N = 2)
int fd = select_morsel("my_morsel");    // or select an existing one.

void* addr = mmap(NULL, 1<<30, PROT_READ | PROT_WRITE,
                  MAP_SHARED | MAP_NORESERVE, fd, 0);
```

To get a Morsel FD, either a new Morsel can be created, or an existing one can be selected via its identifier in the Morsel namespace. The retrieved FD is then used to map the Morsel into the address space.

2.3 Efficient Memory Mapping

Due to being self-contained, mapping and unmapping a Morsel is very efficient: As Morsels bring their own page tables, the kernel only has to change a single page-table entry in the process' address space to map a Morsel. As a result, the page tables of a Morsel can be referenced from different address spaces. In comparison: Working with a memory-mapped file entails not only the allocation of page tables but also the need to populate them with each accessed page, even if the file is already loaded and mapped in another address

space. This typically happens lazily through the page-fault mechanism, leading to many expensive faults and contention within the VM subsystem and the page cache.

Please note, although we share the page tables for all mappings of a Morsel, we can nevertheless use different access protections per mapping, as we can specify the desired permissions in the *mounting* page-table entry that points to the Morsel.

2.4 Creation and Identification

As Morsels can be sparsely populated, the construction routine only creates a minimal page-table subtree covering only the uppermost element, reducing memory consumption and system-call latency. The lower-level page tables, as well as the data pages, are installed either on demand by the fault handler or by an explicit system call that will allow for batch allocation. A special case is order $N = 0$, which requires no page table; here, we allocate the user page upon creation.

To address a Morsel, we only have to reference its uppermost element, which is either a page table or a single user page. As both are 4-KiB page frames, a Morsel is identified by a page-aligned physical address. For security and convenience, our Morsel subsystem will not directly expose this address to the user-space applications.

2.5 Scalability

The primary design goals of Morsels are memory efficiency and scalable manipulations (design goal 1). The management of Morsels using MMU data structures comes with minimal overhead, which is required anyway for memory mapping. Unlike conventional mappings, where the enclosing address space owns the page tables, the Morsel owns its page tables. In the best case, a fully populated surface requires about one page table per 512 user pages ($\frac{1}{512} \approx 0.20\%$; assuming 4-KiB base pages and neglecting higher page-table layers). The page-table overhead is independent of the mapping count as the page tables are shared, and no additional costs arise.

For most architectures (including AMD64), page-table entries are word-sized, allowing for atomic manipulation. Combined with our metadata-less design, different threads can modify Morsels in parallel without requiring locks.

2.6 Fixed Mapping Addresses

Morsels are well suited for packed or serialized, file-like data, as well as for pointer-based, scattered data structures such as graphs (design goal 3). For the latter, the mapping address matters in order to maintain the validity of the contained pointers since they are absolute addresses. Therefore, we provide a mechanism to bind a Morsel to a specific virtual address on creation.

We are aware of the potential for address conflicts for bound Morsels caused by the limited usable virtual address spaces in current VM hardware (AMD64: 4/5-level paging $\widehat{=}$ 256 Tib / 128 PiB). We believe that such conflicts are effectively avoidable through careful system-wide management of virtual addresses. However, the discussion of such management strategies is beyond the scope of this paper.

2.7 Persistence

Morsels support all types of memory accessible via the memory bus and, accordingly, mappable by the MMU (design goal 2). With conventional DRAM, the lifetime of Morsels is naturally limited to the system's uptime. However, Morsels are predestined for usage with NVM: Since a Morsel is self-contained and includes all its necessary structural information, we do not have to rebuild it after a power loss.

A major challenge with persistent memory is maintaining a consistent state in the event of unexpected power losses and other outages. With atomic instructions for page-table modification and NVM with a *persist granularity* of at least 64 bit [21], Morsels themselves are already outage-tolerant and a suitable building block for persistent objects. In this regard, Linux lacks a persistent page allocator, which is another fundamental prerequisite for outage-tolerant persistency. Therefore, we plan to use an allocator implementation that provides the required guarantees (e.g., [32, 18]).

2.8 Sharing with Devices

The Morsel design also targets the direct sharing of memory objects with devices that are capable of *direct memory access (DMA)* via the IO-MMU, which provides virtualization and isolation features, similar to what the MMU does for the CPU (design goal 4). AMD's IO-MMU implementation uses MMU-compatible data structures, allowing for direct page-table sharing between the MMU and the IO-MMU. This compatibility makes it possible to efficiently share memory between the processor and devices, like network interfaces, NVMe SSDs, and accelerators. For example, large neural network models could be loaded from a storage device and passed to an accelerator without CPU interaction. RDMA-compatible network interfaces add even more possibilities by sharing Morsels with attached systems.

To also support IO-MMUs with a page-table layout that is incompatible with the MMU, we could use proxy page tables, which we keep synchronized with the original MMU tables. While this adds some overhead to the page-table manipulation routines and the overall memory usage, it enables the same fast mapping and unmapping as directly sharing the Morsel's page tables. For persistent Morsels, the proxy page tables are stored in volatile memory and are rebuilt from the persistent MMU page tables on startup.

2.9 Huge Pages

Regarding memory management overheads, Morsels and huge pages target similar shortcomings. However, huge pages increase the page size by merging a complete page table, which requires a huge page to be backed with physically

contiguous memory and leads to internal fragmentation. In contrast, Morsels internally can stick to smaller frame sizes to allow fine-grained control (design goal 5). Another difference is the sparse population of Morsels, whereas huge pages cannot contain holes. While huge pages are architecturally supported only by the lower levels of the page table hierarchy – typically 2 MiB, sometimes 1 GiB – Morsels support even larger virtual sizes.

Current huge-page implementations in Linux have been poorly adopted because they either need explicit support by applications (*hugetlbfs*) or provide unpredictable latency spikes (*transparent huge pages*) [19]. Adding huge-page support to Morsels will allow us to provide object-specific transparent huge pages, which enables the benefits of huge pages for specific objects (e.g., large machine-learning models) without the drawback of high CPU utilization caused by scanning and migrating the entire address space.

3 Case Studies

In the following, we will present two case studies that highlight the potential of Morsels to share and transmit large amounts of *volatile* memory between processes.

Benchmark System. We execute all benchmarks on a machine with two Intel Xeon Gold 6252 CPUs (2.10 Ghz, 2×24 cores, 96 HW threads) and 384 GiB of DRAM. We use a Linux 6.1.0 kernel with our Morsel extensions and Debian 11 (bullseye). Memory was always abundant.

3.1 Case Study: User-Space Read-Only File Cache

In our first scenario, we use Morsels to implement a read-only cache for large files in user space that allows reusing loaded data by different processes. We compare it to the operating system's page cache. As a use case, we consider `llama.cpp` [10, 29], a program that performs inference with a *large language model (LLM)* and requires large amounts of parameters to be accessible in the address space. We aim to speed up the loading process as it reduces the startup latency of the LLM process, benefiting its initial response time.

We introduce the *Morsel cache*, a daemon that provides processes with cached Morsels filled with file contents. A client connects to the daemon via a Unix domain socket and transmits the name of the requested file. If the file is not already loaded, the daemon creates a new Morsel, fills it with file contents, and returns the Morsel's FD. Otherwise, the daemon directly replies with the FD of the already loaded Morsel. The client can then mount the Morsel into its address space via `mmap()`, making the data accessible without additional page faults.

Our competitor is a conventional file mapping. Since it is populated from the page cache, the data pages are shared between all other mappings of the file. To also avoid the costs of lazy population, we pre-fault all pages of the mapping.

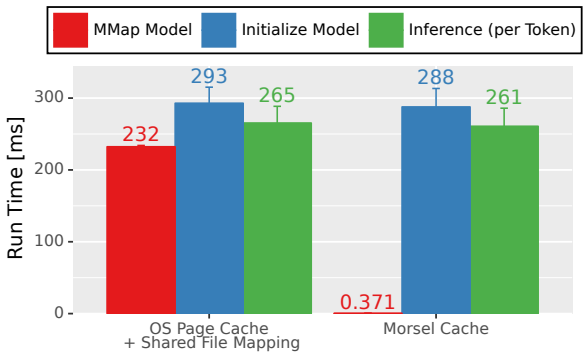

Figure 2. LLAMA.cpp: Model load and inference times with *warm* cache. Model: SelFee (13B parameters, q4_0, 6.82 GiB)

Within our LLM scenario, we either request the parameter file via the Morsel cache or via the conventional file mapping, the latter being the current loading strategy of `llama.cpp`. To focus on system overheads and to prevent domination by I/O wait times, we consider the case where the data is already present in main memory, either in a page cache or loaded in a Morsel.

We use the SelFee [33, 14] LLM with the prompt "Explain Virtual Memory!", perform CPU inference, and request 100 output tokens. Averaged over 50 runs, we report the time required for mapping the parameter file, the time for the rest of the model initialization, and the average time required for each output token.

As anticipated, the Morsel cache does not affect the rest of the model initialization or the actual inference (see Fig. 2). However, it does significantly decrease the time required for model mapping by *three* orders of magnitude, from 232 ms to 371 µs (-99.8%). This results in a 45 percent reduction in the overall initialization phase of `llama.cpp`.

A look at the required work for installing the shared memory mapping reveals the large difference between both caching methods: As Linux does not share page tables, it has to allocate and populate a new page-table subtree for each LLM process. This further results in a 0.2 percent memory overhead per inference process. Additionally, Linux must update the reference counters and the reverse mapping for every data page, as the Linux page cache is organized around 4-KiB frames.

In contrast, our Morsel cache only needs to transmit and install one *page-table entry (PTE)* into a freshly allocated *virtual-memory area (VMA)*. Moreover, for 6.82 GiB of model data, we require 15.65 MiB for the page tables, which are shared between all LLM processes and the Morsel cache.

3.2 Case Study: Processing Pipeline

Our second scenario is a four-step processing pipeline that prepares data packets to be sent via a network connection. The pipeline architecture increases throughput by utilizing multiple CPU cores, reduces complexity as every step has a

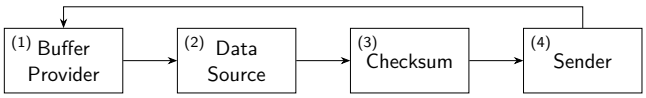

Figure 3. Second case study: 4-step processing pipeline. Multiple processes are connected through Unix domain sockets to transfer buffers as FDs between address spaces.

limited scope, and improves security due to isolation. The steps are connected without queues to prevent buffer bloat. Fig. 3 shows the architecture: (1) The first step provides memory buffers from a lazily-filled cache of shared-memory objects. (2) The second step inserts random data with a random length of up to 256 MiB. (3) The third step calculates a CRC-32 checksum of the data. (4) The fourth step is intended to send the packet via a network connection. In this evaluation, it just overwrites the packet and returns the buffer.

For the benchmark, we measure the round-trip time (1→2→3→4→1) and the accumulated user-space time of each packet. We compare the conventional shared-memory mechanism via anonymous files (*memfd*) with Morsels. The buffer provider yields buffers of the configured type as FDs, which are passed between the steps via Unix domain sockets.

Fig. 4 compares the round-trip times of 1 000 packets per type. The locally-weighted regression shows the trend of the total round-trip time and the actual processing time (user-space time). The difference comes from kernel processing and waiting for the next step. As we can see, the round-trip time is not a linear function of the packet size. There is a high variation, especially for small packets, due to larger preceding packets blocking the pipeline and causing processing delays. The user-space time is independent of previous packets, as it does not include the waiting time. Outliers may result from buffer initialization because newly created buffers are not initially backed with physical memory.

By comparing both subplots, we can see that Morsels are generally faster than *memfd*. The average round-trip time reduces from 174 ms to 107 ms (-38.7%). With *memfd*, every first access to a page triggers the page-fault handler that queries the page cache and populates the corresponding page-table entry, adding a noticeable delay. Morsels, on the other hand, are mapped as indivisible units, which avoids fault-driven population of individual pages.

We would expect the actual processing time (accumulated user-space time) to be the same for both competitors, which is not the case. We attribute the slightly longer time measured in the *memfd* variant to a potential inaccuracy in Linux's time measurement with many user/kernel transitions and to an increased occurrence of *translation lookaside buffer (TLB)* and cache evictions caused by the page-fault handler.

The accumulated user-space time marks the lower bound of the round-trip time. For *memfd*, a clear gap is evident, while with Morsels, the fastest round-trip times match the user-space time, indicating negligible overheads.

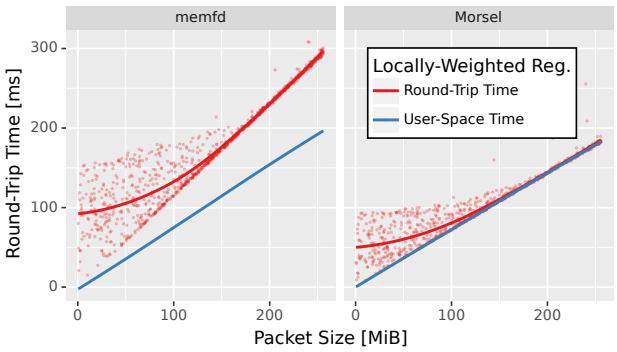

Figure 4. Round-trip times of 1 000 packets through the 4-step processing pipeline

Fig. 5 visualizes the runtime share of the pipeline for both shared-memory implementations as a customized flame graph [11]. The x-axis represents the relative runtime share in relation to the total runtime of the processing pipeline with *memfd* across all involved CPU cores. The data samples were collected using Linux's *perf* utility with a sampling frequency of 99 Hz for the whole system and a subsequent post-processing step to filter out unrelated data points.

We can clearly see all pipeline steps in the graph, except the buffer provider (step 1), which is negligible in terms of runtime. The *memfd* subplot reveals a high page-fault activity in each step. The pattern is similar for steps 2 and 4, but smaller for step 3, which might result from the differing access types (write: steps 2 & 4; read: step 3). Unmapping the buffers accounts for another notable portion of the runtime.

Morsels drastically reduce runtime costs. Since there is no need to handle individual pages, the page-fault components observed with *memfd* are absent in the Morsel subplot. However, the total reduction is even larger than the missing faults. As mentioned, we attribute this to the reduced cache load and the fewer user/kernel transitions.

4 Discussion

Our case studies show that the VM subsystem has a significant effect on the end-to-end performance of applications. As a new supplementary mapping type, Morsels effectively eliminate current bottlenecks by reevaluating mapping requirements and surpassing (in many cases) obsolete assumptions like memory scarcity. The current Morsel implementation only uses 4-KiB pages (like Linux's page cache), but we will extend Morsels with huge-page support in the future. This could further improve the file-cache use case (Sec. 3.1), as huge pages reduce the TLB pressure.

The fully self-contained design of Morsels does not rely on the per page-frame management data that Linux uses to store information, such as the mapping count or allocation information. This management data adds an overhead of 64 B per page frame (~ 1.56%), which is 6 GiB for our benchmark system with 384 GiB of main memory. Currently, we can

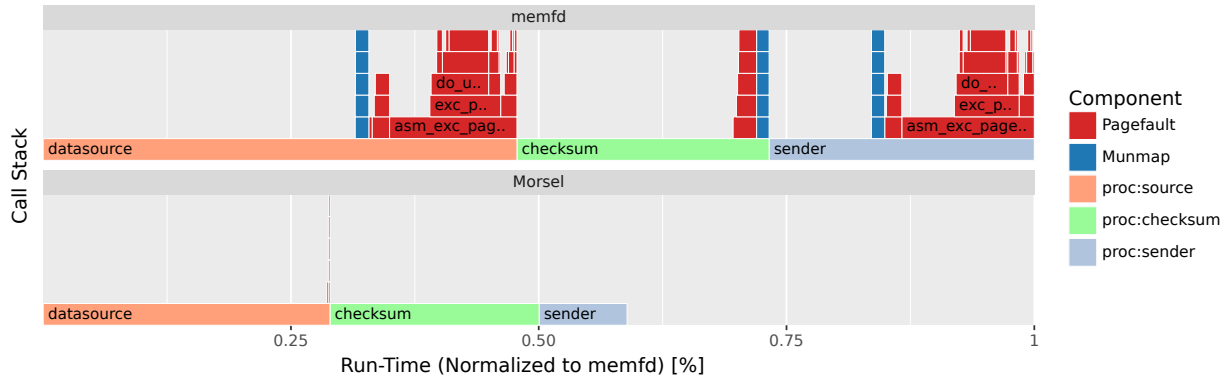

Figure 5. Normalized runtime share of the four pipeline steps, including calls to sub-functions and operating system code, both stacked on top, visualizing the call hierarchy.

not omit this overhead for Morsel pages because it is deeply entangled in the operating system, but with further measures, such as replacing the allocator with an implementation that does not rely on this data, we will be able to use the available memory more efficiently.

5 Related Work

The concept of mountable storage objects is reminiscent of single-level–store systems. Multics [1] represents memory objects as segments, which are effectively second-level page tables that can be directly mapped into multiple address spaces, similar to Morsels. Also, IBM's System/38 [13] and AS/400 [27, pp. 171–219], which pioneered the single–address-space concept, utilized hardware-supported segments, however, combined with inverted page tables. More recent single–address-space systems, using modern standard paging hardware, purely rely on software for managing memory objects [3, 5, 26, 12], whereas Morsels rely on hardware data structures to avoid additional metadata and bookkeeping overhead. The fundamental difference to all mentioned single-level–store systems is that they provide a general storage abstraction via demand paging and page swapping, whereas Morsels exclusively operate on main memory under the assumption of abundant memory resources.

Twizzler [2], an operating system explicitly tailored for NVM, is more similar to Morsels in this respect in that it only targets the directly accessible main memory. Although it proposes a fundamentally different data-centric single–address-space design, Twizzler utilizes conventional VM hardware behind the scenes and treats its memory objects as indivisible units, similar to Morsels. However, Twizzler allows (potentially nested) COW relationships with the associated bookkeeping and page-fault overhead. In addition, Twizzler is a clean-slate approach, while Morsels are integrated into the VM model of Linux.

There have also been proposed extensions for existing operating systems that have some similarities with Morsels: Zhao et al. modify the *fork* system call in Linux to share last-level page tables between parent and child process with the goal of reducing the fork latency. However, in contrast to Morsels, the approach does not touch the overall fault-driven VM model of Linux [34].

Similar to Morsels, Exmap [16] addresses the demand-paging bottleneck in Linux's memory subsystem but still keeps the fundamental hardware-agnostic memory management and does not consider NVM or IO-MMUs.

6 Conclusion

In this paper, we introduced Morsels, a novel memory-management approach that leverages VM objects as subtrees of hardware-defined paging data structures instead of individual 4-KiB pages, thereby minimizing management overhead. Morsels' fully self-contained design makes them suitable for NVM and enables efficient memory sharing between processes and devices.

In a scenario where a Morsel-based cache provides read-only instances of a file, Morsels outperform the Linux page cache by three orders of magnitude, reducing the mapping time of a 6.82-GiB machine-learning model from 232 ms to 371 µs (-99.8%). In another application, a four-step processing pipeline, the use of Morsels instead of conventional shared memory reduces the average round-trip time from 174 ms to 107 ms (-38.7%) for large packets with up to 256 MiB of data. An in-depth analysis shows that nearly all management overhead can be avoided.

In the future, we plan to investigate the use of huge pages as part of Morsels that can directly benefit the shown use cases due to increased TLB coverage, speeding up the average memory-access latency. We will also assess the advantages of direct sharing of Morsels with devices via the IO-MMU.

Acknowledgments

We thank our reviewers for their valuable feedback. This work was funded by the *Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)* – 468988364, 501887536.

References

[1] A. Bensoussan, C. T. Clingen, and R. C. Daley. "The multics virtual memory". In: *Proceedings of the second symposium on Operating systems principles*. SOSP '69. New York, NY, USA: Association for Computing Machinery, Oct. 1969, 30–42. ISBN: 978-1-4503-7456-9. DOI: 10.1145/961053.961069.

[2] Daniel Bittman, Peter Alvaro, Pankaj Mehra, Darrell D. E. Long, and Ethan L. Miller. "Twizzler: a Data-Centric OS for Non-Volatile Memory". In: *2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020, pp. 65–80. ISBN: 978-1-939133-14-4. URL: https://www.usenix.org/conference/atc20/presentation/bittman.

[3] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. "Sharing and Protection in a Single-Address-Space Operating System". In: *ACM Trans. Comput. Syst.* 12.4 (1994), 271–307. ISSN: 0734-2071. DOI: 10.1145/195792.195795.

[4] Compute Express Link Consortium, Inc. *CXL Specification, Revision 2.0*. Oct. 2020.

[5] Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, Anders Lindström, John Rosenberg, and Francis Vaughan. "Grasshopper: An Orthogonally Persistent Operating System". In: *Comput. Syst.* 7.3 (1994), 289–312. ISSN: 0895-6340.

[6] Izzat El Hajj et al. "SpaceJMP: Programming with Multiple Virtual Address Spaces". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '16. Atlanta, Georgia, USA: Association for Computing Machinery, 2016, 353–368. ISBN: 9781450340915. DOI: 10.1145/2872362.2872366.

[7] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. "Exokernel: An Operating System Architecture for Application-Level Resource Management". In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)* (Copper Mountain, CO, USA). New York, NY, USA: ACM Press, Dec. 1995, pp. 251–266. ISBN: 0-89791-715-4. DOI: 10.1145/224057.224076.

[8] Brad Fitzpatrick. "Distributed Caching with Memcached". In: *Linux Journal* 2004.124 (Aug. 2004), pp. 5–. ISSN: 1075-3583. URL: http://dl.acm.org/citation.cfm?id=1012889.1012894.

[9] Gen-Z Consortium. *Gen-Z Core Specification, Revision 1.1*. Oct. 2020.

[10] Georgi Gerganov. *llama.cpp: Port of Facebook's LLaMA model in C/C++*. June 2023. URL: https://github.com/ggerganov/llama.cpp.

[11] Brendan Gregg. "The flame graph". In: *Communications of the ACM* 59.6 (2016), pp. 48–57.

[12] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russel, and Jochen Liedtke. "The Mungi Single-Address-Space Operating System". In: *Software: Practice and Experience* 18.9 (July 1998).

[13] Merle E. Houdek, Frank G. Soltis, and Roy L. Hoffman. "IBM System/38 Support for Capability-Based Addressing". In: *Proceedings of the 8th Annual Symposium on Computer Architecture (ISCA)*. ISCA '81. Minneapolis, Minnesota, USA: IEEE Computer Society Press, 1981, 341–348.

[14] Tom Jobbins. *Selfee-13B-GGML-DOI (Revision 4dd57ef)*. 2023. DOI: 10.57967/hf/0822. URL: https://huggingface.co/TheBloke/Selfee-13B-GGML-DOI.

[15] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. "Coordinated and Efficient Huge Page Management with Ingens". In: *12th Symposium on Operating Systems Design and Implementation (OSDI '16)*. Savannah, GA, USA: USENIX Association, 2016, 705–721. ISBN: 9781931971331.

[16] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. "Virtual-Memory Assisted Buffer Management". In: *Proceedings of the ACM SIGMOD/PODS International Conference on Management of Data (SIGMOD'23)*. Seattle, WA, USA: ACM, June 2023. DOI: 10.1145/3588687.

[17] Juan Navarro, Sitaram Iyer, and Alan Cox. "Practical, Transparent Operating System Support for Superpages". In: *5th Symposium on Operating Systems Design and Implementation (OSDI '02)*. Boston, MA: USENIX Association, Dec. 2002.

[18] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. "Memory Management Techniques for Large-Scale Persistent-Main-Memory Systems". In: *Proc. VLDB Endow.* 10.11 (2017), pp. 1166–1177. DOI: 10.14778/3137628.3137629. URL: http://www.vldb.org/pvldb/vol10/p1166-oukid.pdf.

[19] Ashish Panwar, Aravinda Prasad, and K. Gopinath. "Making Huge Pages Actually Useful". In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '18. Williamsburg, VA, USA: Association for Computing Machinery, 2018, 679–692. ISBN: 9781450349116. DOI: 10.1145/3173162.3173203. URL: https://doi.org/10.1145/3173162.3173203.

[20] Omer Peleg, Adam Morrison, Benjamin Serebrin, and Dan Tsafrir. "Utilizing the IOMMU Scalably". In: *2015 USENIX Annual Technical Conference (USENIX ATC '15)*. Santa Clara, CA: USENIX Association, July 2015, pp. 549–562. ISBN: 978-1-931971-225. URL: https://www.usenix.org/conference/atc15/technical-session/presentation/peleg.

[21] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. "Memory Persistency". In: *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA '14)*. Minneapolis, Minnesota, USA: IEEE Press, 2014, 265–276. ISBN: 9781479943944.

[22] Richard F. Rashid and George G. Robertson. "Accent: A Communication Oriented Network Operating System Kernel". In: *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP '81)*. New York, NY, USA: ACM Press, 1981, pp. 64–75. ISBN: 0-89791-062-1. DOI: 10.1145/800216.806593.

[23] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures". In: *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems (ASPLOS '87)*. ASPLOS '87. Palo Alto, California, USA: IEEE Computer Society Press, 1987, 31–39. ISBN: 0818608056. DOI: 10.1145/36206.36181.

[24] Redislab. *Redis*. http://redis.io, visited 2019-07-21. 2019. (Visited on 07/21/2019).

[25] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. "LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 69–87. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/shan.

[26] Jonathan S. Shapiro and Jonathan Adams. "Design Evolution of the EROS Single-Level Store". In: *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*. ATEC '02. USA: USENIX Association, 2002, 59–72. ISBN: 1880446006.

[27] Frank G. Soltis. *Inside the AS/400*. Loveland, Colorado: 29th Street Press, 1996. ISBN: 1-882419-13-8.

[28] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa. "Pin-down cache: a virtual memory management technique for zero-copy communication". In: *Proceedings of the First Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS '98)*. 1998, pp. 308–314. DOI: 10.1109/IPPS.1998.669932.

[29] Hugo Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. 2023. arXiv: 2302.13971 [cs.CL].

[30] Ján Veselý, Arkaprava Basu, Abhishek Bhattacharjee, Gabriel H. Loh, Mark Oskin, and Steven K. Reinhardt. "Generic System Calls for GPUs". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 2018, pp. 843–856. DOI: 10.1109/ISCA.2018.00075.

[31] Pirmin Vogel. "Shared Virtual Memory for Heterogeneous Embedded Systems on Chip". en. PhD thesis. Zurich: ETH Zurich, 2018. ISBN: 978-3-86628-623-8. DOI: 10.3929/ethz-b-000292606.

[32] Lars Wrenger, Florian Rommel, Alexander Halbuer, Christian Dietrich, and Daniel Lohmann. "LLFree: Scalable and Optionally-Persistent Page-Frame Allocation". In: *2023 USENIX Annual Technical Conference (USENIX '23)*. Boston, MA: USENIX Association, July 2023, pp. 897–914. ISBN: 978-1-939133-35-9. URL: https://www.usenix.org/conference/atc23/presentation/wrenger.

[33] Seonghyeon Ye, Yongrae Jo, Doyoung Kim, Sungdong Kim, Hyeonbin Hwang, and Minjoon Seo. *SelFee: Iterative Self-Revising LLM Empowered by Self-Feedback Generation*. Blog post. 2023. URL: https://kaistai.github.io/SelFee/.

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