ExtMem: Enabling Application-Aware Virtual Memory Management for Data-Intensive Applications

Sepehr Jalalian, Shaurya Patel, Milad Rezaei Hajidehi, Margo Seltzer, and Alexandra Fedorova, *University of British Columbia*

https://www.usenix.org/conference/atc24/presentation/jalalian

This paper is included in the Proceedings of the 2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

Open access to the Proceedings of the 2024 USENIX Annual Technical Conference is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

EXTMEM: Enabling Application-Aware Virtual Memory Management for Data-Intensive Applications

Sepehr Jalalian
University of British Columbia

Shaurya Patel
University of British Columbia

Milad Rezaei Hajidehi
University of British Columbia

Margo Seltzer
University of British Columbia

Alexandra Fedorova
University of British Columbia

Abstract

For over forty years, researchers have demonstrated that operating system memory managers often fall short in supporting memory-hungry applications. The problem is even more critical today, with disaggregated memory and new memory technologies and in the presence of tera-scale machine learning models, large-scale graph processing, and other memory-intensive applications. Past attempts to provide application-specific memory management either required significant in-kernel changes or suffered from high overhead. We present EXTMEM, a flexible framework for providing application-specific memory management. It differs from prior solutions in three ways: (1) It is compatible with today's Linux deployments, (2) it is a general-purpose substrate for addressing various memory and storage backends, and (3) it is performant in multithreaded environments. EXTMEM allows for easy and rapid prototyping of new memory management algorithms, easy collection of memory patterns and statistics, and immediate deployment of isolated custom memory management.

1 Introduction

Memory management plays a significant role in application performance and operational cost [18, 86, 91]. Due to the cost and scaling trends of DRAM, memory has emerged as a costly and scarce resource in modern datacenters [57, 63, 87]. This challenge has become even more critical today given workloads such as tera-scale machine learning [11, 64, 73], large-scale graph processing [38, 79], in-memory databases [34, 83], and other memory-intensive applications [17, 46, 53].

Operators have turned to innovative memory architectures such as disaggregated memory [15, 40, 63], tiered memory [61, 76, 93], and memory-centric computing [68, 69] to address the challenges presented by memory-hungry applications. These approaches have produced a dynamic and diverse memory hardware landscape, with myriad ongoing developments [12, 52, 54, 71]. Consequently, system designers must not only incorporate core kernel support for these novel memory devices but also revamp memory management policies. This transformation is essential to fully leverage hardware benefits and optimize memory utilization [16, 31].

In this evolving landscape, the concept of a one-size-fits-all memory management policy is no longer tenable [31, 33, 37]. While researchers have illustrated instances where general-purpose operating system memory managers fell short in catering to the needs of data-intensive applications [2, 21, 55, 92, 94], this inadequacy has been exacerbated with the ongoing innovations in memory architecture and the increasingly diverse range of hardware configurations [22, 33, 48]. For example, researchers showed that a major system bottleneck can be memory management and page faults rather than device bandwidth [13]. Consequently, the selection of memory management policies has become a case-specific endeavor. For instance, in the context of datacenters, TMO [91] achieved 20% memory saving without significant performance penalties by tuning the kernel policies that offload memory to SSD devices. Canvas [90] introduced a prefetcher for the JVM; Semeru [89] modified the in-kernel swap system to support JVM runtime disaggregation; and Dilos [95] demonstrated a prefetcher for Redis [25]. Unfortunately, these approaches do not work for the general case. Prior research has also underscored the inefficacy of traditional paging policies in tiered memory systems [33, 76], leading to the development of specialized policies.

Developing, testing, and deploying new memory managers in the kernel is challenging [39]. The Linux memory management system is a core kernel component that must function effectively for a wide range of responsibilities in the system, and in recent years, it has become increasingly complex, with some calling for emulators or simulators to help [4, 7]. Developing new memory policies is a research-intensive task that is further slowed by the kernel development process. For example, implementing the Leap prefetching algorithm [14] required modification of 20 kernel files, even though the algorithm itself can be expressed in 20 lines of code. Researchers that built Canvas [90] report spending 17 months on kernel development for paging policy optimizations. Furthermore, deploying new systems requires rigorous testing to maintain

system stability and avoid affecting other applications [9]. As a result, many developers are hesitant to add special-purpose memory management enhancements to the kernel [6].

Previous efforts addressing memory management challenges require either substantial kernel modifications [13, 22, 93] or entirely new kernel architectures [58, 80]. Dilos [95] implemented prefetching in its library OS to circumvent kernel paging stack limitations. FBMM [85] proposes memory management inside a VFS kernel module. While these conceptual advances hold value, they often prove hard to implement and deploy in real-world use cases. Prior attempts at user-space memory management have significant shortcomings, as they either require runtime support [41, 77, 89], application modifications [30, 88], are for specific applications [56, 81] or specific memory back-ends [76], or cause high overheads [10, 23, 67, 72].

We introduce ExtMem, a framework designed for application-based memory management in user space that avoids the shortcomings of prior work. ExtMem elevates the task of memory management policies and paging mechanisms to user space. It cleanly separates responsibilities, allowing the kernel to maintain security and isolation while delegating memory management policies to user space. Our framework provides ease of development and rapid testing in user space. With its observability layer, ExtMem offers data collection modules for metrics such as access bits and hardware counter samples.

ExtMem builds on well-known ideas and mechanisms, such as user-level page fault handling, kernel signals and upcalls, but it **brings them together in a novel way** to enable bespoke memory management policies in modern Linux deployments with overhead similar to that of in-kernel paging. ExtMem operates seamlessly without necessitating any modifications to application code, while also empowering application developers to implement application-specific memory management policies. Developers can use ExtMem for three main tasks: Developing and testing new memory managers, deploying highly customized memory managers, and gaining control and observability over the working memory of software such as serverless frameworks, garbage collectors, databases, and data-intensive applications.

We considered the design implications of the widely-used Linux subsystem for user space paging, userfaultfd [10], which relies on file-based IPC and server threads to handle page faults, and thus scales poorly for multithreaded workloads. In contrast, the ExtMem architecture is based on upcalls, allowing for scalable self-paging [43] even in highly multithreaded environments. We implemented our approach using existing userfaultfd and signaling interfaces in Linux. Supporting ExtMem requires modifications of only 200 lines of Linux code localized to the userfaultfd and signaling subsystems.

Our work makes the following contributions: First, we design and implement ExtMem, a framework for the rapid development of customized memory managers in user space. Second, we propose an innovative page fault handling mechanism in user space, designed to scale and perform effectively in multi-threaded environments. We demonstrate the applicability of our framework through two case studies. One involves the implementation of a Linux-like memory reclaim policy, written in only 300 lines of code, with performance that is similar to Linux in-kernel implementation for memory-intensive workloads. In the second case study, we demonstrate a substantial 50% performance improvement over native swap-based execution in the GAP benchmark suite's [20] PageRank algorithm. This boost was achieved through the development of a custom memory management policy with minimal development effort. The major contribution of this work is the framework and the associated techniques to enable efficient memory management at the user level. The policies provided and evaluated in this work are proof-of-concepts that demonstrate the practicality and utility of our framework.

ExtMem is publicly available as open-source software.[1]

2 The 2024 Case for External Paging

For more than four decades, researchers have advocated for granting applications control over memory management [24, 82]. Even today, the majority of large-scale commercial database servers implement their own memory management [2, 47, 78]. Despite substantial advances in the Linux memory management subsystem, situations where the kernel's memory manager is inadequate still arise [28, 55, 95]. While micro-kernels offer an elegant solution to the external paging problem, their adoption for production use has been hindered by complexities [26, 44]. In contrast, the monolithic paging model has gained widespread acceptance due to its transparency, ease of use, and strong performance in a wide range of common use cases.

Yet, memory management is even more important today than in the past, due to new disaggregated memory architectures [13, 42] and new memory technology, such as CXL [84] and processing-in-memory (PIM) [36]. We first discuss how the need for sophisticated memory management is even more pressing today than it was in times past (section 2.1). We then discuss how microkernels addressed this problem and how newer semi-kernels are a modern-day, deployable technique to leverage the benefits of the microkernel.

2.1 New Memory Architectures, New Needs

The evolving memory landscape and the increasing heterogeneity of execution environments [37, 60, 61] underscores the increasing importance of memory management. Each environment and architecture presents unique challenges and advantages, necessitating specialized memory management

[1] https://github.com/SepehrDV2/ExtMem

approaches. For instance, memory disaggregation [13, 40, 42] extends a system's memory capacity beyond physical DRAM by retrieving data from far memory and mapping it into the local address space. Effectively managing data placement and movement between local and disaggregated memory falls within the domain of memory management [15, 91]. This complexity highlights the need for tailored memory management solutions to optimize memory utilization and performance in diverse execution environments. Furthermore, although emerging byte-addressable memory architectures such as CXL-attached memory [1, 61] can eliminate major page faults, they require techniques such as page promotion for optimal performance [60, 67]. ExtMem offers a common framework for developing paging and page promotion algorithms, simplifying development and code reuse across these memory systems.

2.2 Microkernels and Semi-microkernels

Microkernels [75] offer improved code management [62, 74], enhanced security [49, 50], and isolation [44] by placing the operating system functionality in user-space servers. Microkernels address the external paging problem by implementing memory pagers in user space and using Inter-Process Communication (IPC) [50, 62].

Although for decades microkernels were considered impractical for wide deployment [26, 44], we are witnessing a renewed interest in this architecture. Notably, microkernels inspired a line of work called "semi-microkernels". The semi-microkernels, exemplified by projects such as Snap [66], uFS [65], Shenango [70] and ghOSt [45] are user-level processes that work alongside the conventional monolithic kernel while realizing either partial or entire OS subsystems, such as the networking stack. To facilitate the streamlined development of bespoke management policies for memory-intensive applications, our system adopts a semi-microkernel structure.

3 ExtMem: Design

Our design goals for ExtMem are to: a) enable experimentation with new memory management policies, b) provide observability over the memory of real executions, and c) allow quick deployment of highly customized memory managers without kernel modification across a wide range of scenarios, excluding only the most latency-sensitive situations.

To realize these goals, ExtMem must be *non-intrusive* – ensuring any kernel modifications are small and easily portable as Linux evolves, *extensible* – allowing rapid development of new memory managers, *transparent* – requiring no changes to applications, *safe* – preventing application-specific memory managers from compromising the kernel or unaffected applications, and *efficient*. These criteria advocate a library-OS-like design deployable in Linux.

3.1 The User View

ExtMem is designed as a dynamically linked library that can be transparently loaded into the application address space via LD_PRELOAD. User code interacts with ExtMem in two ways: explicitly through a library API and implicitly through native memory-related system calls such as `mmap` and `madvise`, which we intercept using Intel libsyscall_intercept [8] and handle them in ExtMem. When an ExtMem core function is bound to an intercepted system call such as `mmap`, this function is executed upon each invocation of that system call. The bound function can itself invoke a kernel system call and/or other ExtMem functions. Explicit interaction is for developers creating custom memory managers; implicit interaction is for unmodified applications running on top of ExtMem.

ExtMem is structured in three layers. The **core layer** is responsible for interacting with the kernel: it handles system calls and receives page faults. The **observability layer** provides the functionality commonly used by memory managers such as access to page table access bits and to hardware counter sampling. The **policy layer** contains implementations of policies, e.g., deciding which pages to evict, prefetch, etc.

3.2 The Core Layer

We use the existing userfaultfd interface in Linux to securely register areas for user-level page faults. Userfaultfd is a kernel subsystem that forwards page faults for these registered areas to a user process. Memory areas are registered and specific pages are mapped into the application address space via `ioctl` system calls and are unmapped using `madvise` with the MADV_DONTNEED flag. ExtMem relies on this API to transparently register/deregister the memory of applications it manages upon intercepting the `mmap`, `munmap`, and `madvise` system calls; these system calls are used by libc to grow and shrink the heap area.

ExtMem maintains direct control over storage space for swap and file backings; we only require that the backend support synchronous and asynchronous reads/writes. In the current prototype, we've implemented an interface to NVMe SSD using Linux io_uring. RDMA interfaces provide similar semantics. ExtMem manages the local or remote memory dedicated to it. In scenarios where multiple applications or nodes share a remote memory area, a daemon or global manager would do coarse grain management and allocate slabs to each ExtMem instance. This case is well studied in the previous work [15, 40].

Second-tier memory devices, such as CXL-attached memory, are typically presented as additional NUMA nodes without CPUs. We perform basic page migration using the Linux `move_pages` system call. We defer further exploration of tiering to future work and believe that more robust interfaces, including userfaultfd and `madvise` options for direct page

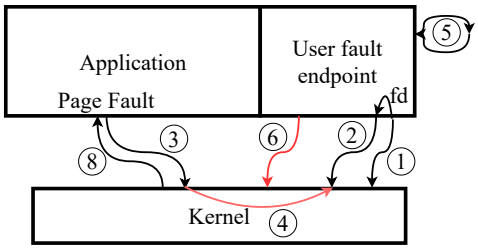

Figure 1: Userfaultfd method: 1. Application registers a memory area with the kernel and receives a file descriptor. 2. A handler thread calls a select/poll system call on the file descriptor. 3. A thread triggers a page fault and context switches into the kernel. 4. After verifying the faulting address, kernel submits the fault information to the file descriptor and blocks the faulting thread. 5. The user-level handler receives the fault, and performs the necessary IO and data preparation. 6. A user level makes a system call to map the new page and unblock the faulting thread. 7. Faulting thread goes back to continue the execution. The red arrows are inter-process-communication.

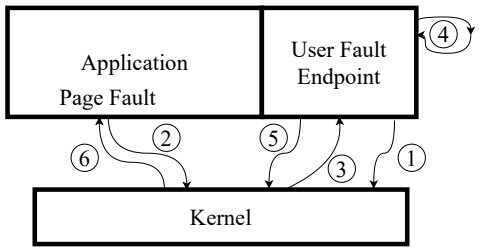

Figure 2: Upcall method based on Linux signal path: 1. Application registers a memory area with the kernel and registers an upcall handler function. 2. A thread triggers a page fault and context switches into the kernel. 3. Kernel makes an upcall by registering the upcall handler context on the faulting thread's stack 4. The faulting thread is back in userland, and executes handler code to perform the necessary IO and data preparation. 5. Faulting thread makes a system call to map the new page and return from the upcall. 6. Faulting thread goes back to continue the execution. No IPC is necessary.

allocation on specific nodes, can be developed for improved performance. Tiering interfaces are under active development in the kernel community [5].

3.2.1 Handling page faults

The primary challenge for ExtMem is effectively handling page faults in user space. In current architectures, page fault triggers a context switch to the kernel. There are two main approaches for forwarding these page faults to user space: IPC (Inter-Process Communication) and upcalls.

Microkernels, such as sel4 [50], use IPC both to transmit fault information to a user-space fault handler and to return a response. Linux's userfaultfd [10] uses a similar approach. When a program registers a memory area with userfaultfd, it receives a file descriptor. When a thread faults in that area, the kernel sends the fault information to the file descriptor and blocks the thread. A handler thread monitors the file descriptor and handles the received page faults, waking up the faulting thread via `ioctl`.

While userfaultfd has been adopted in recent memory managers [23, 76, 90], it exhibits two inherent performance issues. First, the communication between faulting and handler threads essentially constitutes an IPC mechanism, incurring considerable overhead and burdening the scheduler. Second, userfaultfd becomes a point of serialization, limiting scalability. Even if multiple handler threads are active, they must synchronize over the file descriptor and the wait queue of faulting threads. Consequently, userfaultfd excels only in scenarios where page faults are infrequent, not time-sensitive, or already require IPC.

Another approach for user-space, page-fault handling is via upcalls. Exokernels [32, 43] and multi-kernels [19] use upcalls to handle page faults in the same process in user space. This approach, known as *self-paging*, offers advantages in terms of flexibility [47], scalability [19], and Quality of Service (QoS) [43]. The closest element to an upcall inside the Linux kernel is the signal handling path. The kernel can direct signals such as SIGBUS to the same thread that caused the fault and set up a new context on the thread's stack to handle it in user space. The thread jumps to the signal handler function, resembling an upcall.

We adopt this SIGBUS pathway to implement upcalls. Using the existing signal path instead of implementing a new upcall means that ExtMem requires only a few isolated changes to the kernel. To that end, we modified userfaultfd to not block the faulting thread upon a pagefault, but to generate a SIGBUS instead. Our core layer provides a corresponding signal handler, which eventually calls the policy layer.

Unfortunately, the SIGBUS path produces contention on the per-process signal handler structure in the kernel and limits the signal handler to async-safe functions, only. To address contention, we introduce an additional signal handler structure within the Linux task struct, localized to each thread (task), exclusively used for handling the SIGBUS forced by the same task in page faults. Consequently, tasks no longer contend for access to a shared structure. Although this method incurs some overhead compared to a pure upcall, it exhibits the same scalability as a traditional upcall.

The signal/upcall handler must, in the general case, only execute functions that are async-safe, i.e., functions that can be asynchronously interrupted and re-entered. To address these constraints, ExtMem (1) disallows user-level page faults in itself (i.e., we never register ExtMem memory for user-level fault handling), 2) never manipulates user memory that hasn't been locked by the kernel due to a page fault, 3)

does not maintain global state and (4) refrains from using stateful libc functions such as `malloc` and `printf`.

3.3 Observability Layer

One essential tool used by memory managers is tracking memory accesses to differentiate frequently accessed (hot) and infrequently accessed (cold) data, enabling efficient cache replacement, page promotion, and prefetching strategies. ExtMem provides access to: 1) the faulting addresses of page faults, 2) MMU access and dirty bits (via new `ioctl` calls), and 3) hardware counters. Specialized `ioctl` system calls for accessing these bits in user space are pending in the upstream kernel [3].

3.4 Policy Layer

Within the policy layer, developers implement custom policies that identify cold pages for eviction or demotion and select potential pages for prefetching or promotion. To implement parts of ExtMem, we adopted some pieces of API from HeMem [76]. The policy layer maintains a list of free pages and returns them to faulting threads for allocation or swap operations. ExtMem tracks pages using page structs, and a policy can choose to maintain additional metadata as necessary.

To demonstrate the ease of development provided by ExtMem, we implemented a 2Q-LRU page eviction policy similar to the Linux kernel policy. Like Linux, our implementation uses page table access bits to track access recency and maintains active and inactive pages using two FIFO lists. A user-level `kswapd` thread scans page access bits and reclaims inactive pages. When a specific threshold is reached, user `kswapd` awakens evictor threads to write the least recently accessed pages to swap. Our implementation required only about 300 lines of localized code, compared to over 500 scattered across many sub-systems in Linux.

Additionally, we implemented a straightforward sequential page prefetcher (in 200 LOC) that, upon a page fault, fetches not only the faulting page but also asynchronously fetches the next n pages in the virtual address space, and a custom memory manager for graph applications using the compressed sparse row (CSR) layout (described in the next section).

4 Evaluation

Our evaluation answers the following questions: 1) How does ExtMem's method of handling page faults in user space perform compared to native Linux and userfaultfd? 2) How well does ExtMem scale with increasing thread counts? 3) How does the performance of our implementation of 2Q-LRU compare to that of the native Linux implementation? 4) What are the performance advantages of our application-specific memory manager?

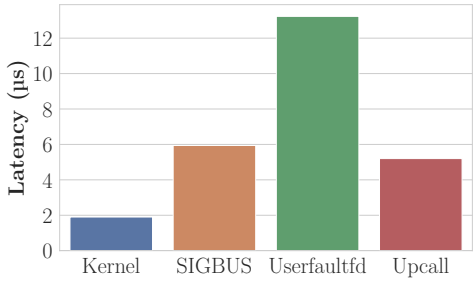

Figure 3: Average latency of resolving a single minor fault in a single-threaded execution

We ran all experiments on a 2-socket machine (16 cores, 32 hardware threads each) with 2.30GHz Intel Xeon 5218 processors and 198GB of DDR4 DRAM, running a modified Linux 5.15. We used an NVMe SSD disk with 2700 MB/s read rate as our storage/swap backend. We use the Linux swap system as our baseline and use Cgroups to control the amount of available physical memory for the baseline. ExtMem explicitly controls available memory.

4.1 Upcall Performance

We evaluate our upcall performance under a high fault rate (Figure 3, Figure 4). A varying number of threads continually access pages in a newly allocated memory area to generate minor faults. Minor page faults occur when a page is in memory but is not mapped in process page tables. Therefore, it does not require any IO or data movement and represents the basic fault resolution cost. We record the end-to-end latency for each memory access and compute the average page fault latency. As expected, the performance of userfaultfd (UFFD) scales poorly due to the round-trip IPC in the fault handling path. While the default SIGBUS approach exhibits better scalability, it suffers from a serialization bottleneck. In contrast, our upcall method exhibits superior scalability, closely mirroring the kernel's performance; both methods ultimately bottleneck on mmap lock contention. Ongoing work in lockless paging [9,27,59] should benefit both in-kernel and user-level self-paging.

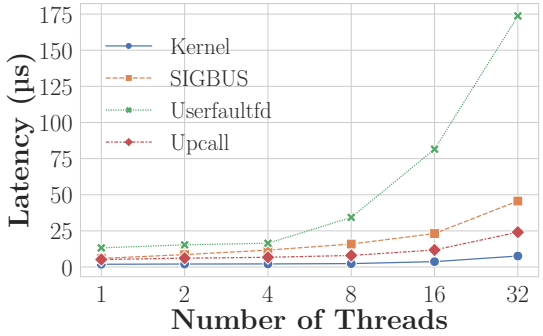

Figure 4: Average latency of resolving a minor fault in a multi-threaded system under pressure

4.2 Microbenchmarks

For the remaining tests, we use the `mmap` microbenchmark derived from Crotty et al [28]. The benchmark initializes a 16 GB region of memory and updates bytes in that region using different patterns. We limit the amount of available RAM to 8 GB, to encourage page faults.

Random Access: First, we update the bytes uniformly at random, so 50% of the memory is paged out, and each access has a 50% chance of causing a major page fault. Figure 5a shows the memory update throughput as a function of the number of threads. We see that the 2Q-LRU policy implemented in ExtMem performs better than Linux and scales well with multiple threads. The ExtMem implementation evicts pages more quickly than Linux does, because its eviction code path is simpler, thereby explaining its performance advantage. Policy and prefetching have no effect on this test since the access pattern is uniformly random.

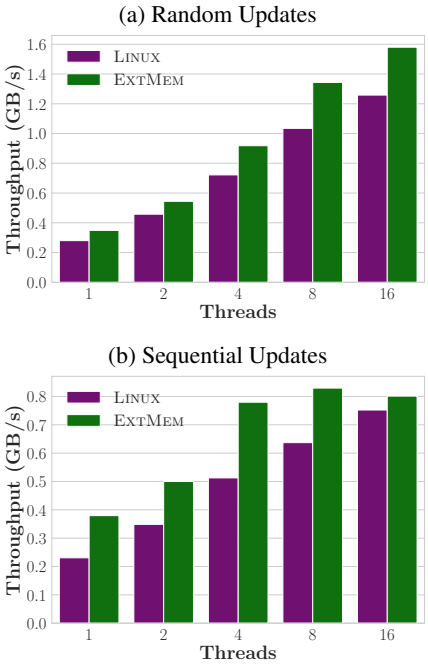

Figure 5: Throughput of the system making byte-sized updates when 50% of the working set fits in memory.

Sequential Access: Next, we evaluate our sequential prefetcher used in conjunction with the 2Q-LRU in ExtMem. Here we use a sequential update pattern. Figure 5b shows that ExtMem outperforms Linux, because we prefetch more aggressively. Throughput levels off at about four threads in ExtMem, due to expensive userfaultfd's write-protect and `madvise` operations required when evicting pages to storage. These operations operate on only one page at a time, hold the `mmap` lock, and are generally slow. A faster page unmapping path can resolve this in the future.

Working Set Access: Our last microbenchmark is a variant of the random one; rather than updating bytes uniformly at random, we update them with a skewed distribution in which 90% of updates are to 10% of the memory, and the remaining 10% of updates are randomly distributed in the whole region. As in the other experiments, we use the 2Q-LRU in ExtMem. This experiment illustrates how ExtMem efficiently identifies and keeps the working set in memory, much like Linux. Figure 6 shows system performance as a function of time, when we run with eight threads. In both systems, performance climbs quickly and then levels off, demonstrating that both Linux and ExtMem maintain the hot working set in memory, while swapping less frequently accessed data. As before, ExtMem is faster than Linux, due to its simpler eviction code path.

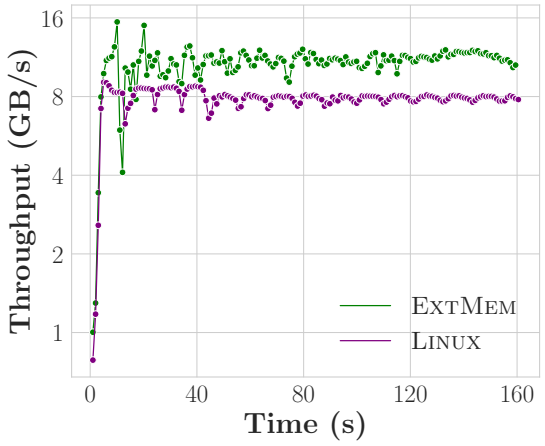

Figure 6: Throughput over time of ExtMem and Linux for the working set access microbenchmark both implementing the 2Q-LRU policy. ExtMem-2Q-LRU efficiently keeps working set in memory.

4.3 Application Study

Graph processing under memory constraints is an example of an application that can benefit from ExtMem with application-specific memory management. As a proof of concept, we implemented custom memory management for a graph processing system that uses the compressed sparse row layout to store the graph. Our goal is to achieve good runtime performance using in-memory data structures, even if the graph exceeds available memory.

Compressed Sparse Row (CSR) [29] is a prevalent data structure used for in-memory graph analytics. It consists of two arrays: a vertex array and an edge array. The vertex array stores a pointer to the data structure of vertex attributes and a pointer to the starting position of the vertex edges in the edge array. The edge array stores all edges, grouped by source vertex ID, and sorted by destination vertex ID within each group. Many graph algorithms process vertices' neighborhoods sequentially [35] (e.g., PageRank), so CSR's sequential

layout is attractive.

We developed a custom memory manager based on the following intuition. First, edges account for the majority of space consumed by a graph, because graphs typically have many more edges than vertices. Second, many algorithms repeatedly iterate over neighborhoods. Our policy keeps the vertex array, arrays that store vertex attributes, and intermediate values (e.g., the rank of vertices in a PageRank computation) in memory. In contrast, we store only a sliding window of the edge array in memory. In each iteration over the edge array, we move this sliding window, evicting the pages that are no longer necessary and prefetching the next window of pages in the background. We implemented this custom policy over the existing default policy by changing fewer than 100 lines of code. This implementation is available in the published artifact.

Figure 7 demonstrates the benefit of the custom memory management policy for computing PageRank (ExtMem-Pr). We ran 10 iterations of the algorithm using the standard GAP [20] benchmarking suite on the Twitter graph [51], limiting memory to 50% of the total graph size. Figure 7 shows that using ExtMem with an algorithm almost identical to that of Linux (ExtMem-2QLru) produces a modest improvement, but when we deploy the custom algorithm (ExtMem-Pr), we obtain a speedup over 2×. Perhaps even more surprisingly, using our user-level custom paging policy, when only 50% of the graph fits in memory, our runtime is only slightly over twice that of a pure in-memory solution.

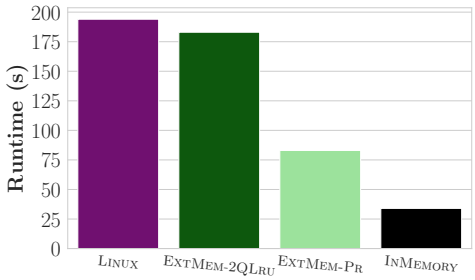

Figure 7: Runtime of 10 iterations of pagerank on the Twitter dataset using GAP benchmark suite.

5 Conclusion

We introduced ExtMem, a versatile framework tailored for application-specific memory management in user space. ExtMem seamlessly integrates into Linux-based environments, offering developers a flexible platform with high code velocity while maintaining isolation by running in the application's address space. Using an upcall approach, it establishes a scalable self-paging architecture, resolving shortcomings of previous user-space paging systems. By reusing the signal handling code path for upcalls, ExtMem requires only small isolated changes to Linux.

ExtMem still faces limitations stemming from scalability issues within the underlying operating system's virtual memory system, particularly the `mmap` lock. Despite these limitations, ExtMem empowers users to harness the performance advantages of application-aware memory management in real-world scenarios. As we continue to explore the workload and hardware-aware memory management policies, ExtMem provides a promising framework for future developments in this domain.

6 Acknowledgement

We acknowledge the support of Intel and the Natural Sciences and Engineering Research Council of Canada (NSERC). Nous remercions le Conseil derecherches en sciences naturelles et en génie du Canada (CRSNG) de sonsoutien. We appreciate the constructive feedback from the reviewers and thank the artifact reviewers for their collaborative efforts in verifying the functionality and reproducibility of our results. We also want to thank David Holland for sharing his insights on kernel internals and his valuable feedback. We thank Hemem authors for their functional artifact which helped us in implementing ExtMem.

References

[1] Cxl consortium — computeexpresslink.org. https://www.computeexpresslink.org. [Accessed 09-01-2024].

[2] Database Concepts — docs.oracle.com. https://docs.oracle.com/en/database/oracle/oracle-database/19/cncpt/memory-architecture.html.

[3] Implement IOCTL to get and optionally clear info about PTEs [LWN.net] — lwn.net. https://lwn.net/Articles/934580/. [Accessed 11-01-2024].

[4] lpc.events. https://lpc.events/event/11/contributions/896/attachments/793/1493/slides-r2.pdf. [Accessed 09-01-2024].

[5] Memory-management changes for CXL [LWN.net] — lwn.net. https://lwn.net/Articles/931416/. [Accessed 11-01-2024].

[6] Memory-management patch review [LWN.net] — lwn.net. https://lwn.net/Articles/718212/.

[7] Readahead: the documentation I wanted to read [LWN.net] — lwn.net. https://lwn.net/Articles/888715/.

[8] syscall_intercept. https://github.com/pmem/syscall_intercept.

[9] The ongoing search for mmap lock — lwn.net. https://lwn.net/Articles/893906/.

[10] Userfaultfd(2). https://man7.org/linux/man-pages/man2/userfaultfd.2.html.

[11] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: a system for Large-Scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[12] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. Flatflash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 971–985, 2019.

[13] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote memory in the age of fast networks. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 121–127, 2017.

[14] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 843–857, 2020.

[15] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[16] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. Disaggregation and the application. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[17] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[18] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. *The datacenter as a computer: Designing warehouse-scale machines*. Springer Nature, 2019.

[19] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44, 2009.

[20] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[21] Andrew Borg, Andy Wellings, Christopher Gill, and Ron K Cytron. Real-time memory management: Life and times. In *18th Euromicro Conference on Real-Time Systems (ECRTS'06)*, pages 11–pp. IEEE, 2006.

[22] Irina Calciu, M Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 79–92, 2021.

[23] Blake Caldwell, Sepideh Goodarzy, Sangtae Ha, Richard Han, Eric Keller, Eric Rozner, and Youngbin Im. Fluidmem: Full, flexible, and fast memory disaggregation for the cloud. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 665–677. IEEE, 2020.

[24] Pei Cao, Edward W Felten, Anna R Karlin, and Kai Li. A study of integrated prefetching and caching strategies. *ACM SIGMETRICS Performance Evaluation Review*, 23(1):188–197, 1995.

[25] Josiah Carlson. *Redis in action*. Simon and Schuster, 2013.

[26] J Bradley Chen and Brian N Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 120–133, 1993.

[27] Austin T Clements, M Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using rcu balanced trees. *ACM SIGPLAN Notices*, 47(4):199–210, 2012.

[28] Andrew Crotty, Viktor Leis, and Andrew Pavlo. Are you sure you want to use mmap in your database management system. In *CIDR 2022, Conference on Innovative Data Systems Research. https://db. cs. cmu. edu/papers/2022/p13-crotty. pdf*, 2022.

[29] J. Dongarra, P. Koev, X. Li, J. Demmel, and H. van der Vorst. *10. Common Issues*, pages 315–336.

[30] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.

[31] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, et al. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 727–741, 2023.

[32] Dawson R Engler, M Frans Kaashoek, and James O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. *ACM SIGOPS Operating Systems Review*, 29(5):251–266, 1995.

[33] Shai Bergman1 Priyank Faldu and Boris Grot3 Lluís Vilanova4 Mark Silberstein. Reconsidering os memory optimizations in the presence of disaggregated memory. 2022.

[34] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[35] Per Fuchs, Domagoj Margan, and Jana Giceva. Sortledton: a universal, transactional graph data structure. *Proceedings of the VLDB Endowment*, 15(6):1173–1186, 2022.

[36] Saugata Ghose, Kevin Hsieh, Amirali Boroumand, Rachata Ausavarungnirun, and Onur Mutlu. The processing-in-memory paradigm: Mechanisms to enable adoption. *Beyond-CMOS Technologies for Next Generation Computer Design*, pages 133–194, 2019.

[37] Christina Giannoula, Kailong Huang, Jonathan Tang, Nectarios Koziris, Georgios Goumas, Zeshan Chishti, and Nandita Vijaykumar. Daemon: Architectural support for efficient data movement in fully disaggregated systems. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 7(1):1–36, 2023.

[38] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. GraphX: Graph processing in a distributed dataflow framework. In *11th USENIX symposium on operating systems design and implementation (OSDI 14)*, pages 599–613, 2014.

[39] Mel Gorman. *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2004.

[40] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, 2017.

[41] Zhiyuan Guo, Zijian He, and Yiying Zhang. Mira: A program-behavior-guided far memory system. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 692–708, 2023.

[42] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 417–433, 2022.

[43] Steven M Hand. Self-paging in the nemesis operating system. In *OSDI*, volume 99, pages 73–86, 1999.

[44] Gernot Heiser and Kevin Elphinstone. L4 microkernels: The lessons from 20 years of research and deployment. *ACM Transactions on Computer Systems (TOCS)*, 34(1):1–29, 2016.

[45] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. ghost: Fast & flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 588–604, 2021.

[46] Congfeng Jiang, Yitao Qiu, Weisong Shi, Zhefeng Ge, Jiwei Wang, Shenglei Chen, Christophe Cérin, Zujie Ren, Guoyao Xu, and Jiangbin Lin. Characterizing colocated workloads in alibaba cloud datacenters. *IEEE Transactions on Cloud Computing*, 10(4):2381–2397, 2020.

[47] M Frans Kaashoek, Dawson R Engler, Gregory R Ganger, Héctor M Briceno, Russell Hunt, David Mazieres, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 52–65, 1997.

[48] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. Exploring the design space of page management for Multi-Tiered memory systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 715–728, 2021.

[49] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an os microkernel. *ACM Transactions on Computer Systems (TOCS)*, 32(1):1–70, 2014.

[50] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, 2009.

[51] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600, 2010.

[52] Miryeong Kwon, Sangwon Lee, and Myoungsoo Jung. Cache in hand: Expander-driven cxl prefetcher for next generation cxl-ssd. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*, pages 24–30, 2023.

[53] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, et al. Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 317–330, 2019.

[54] Mario Lanza, Abu Sebastian, Wei D Lu, Manuel Le Gallo, Meng-Fan Chang, Deji Akinwande, Francesco M Puglisi, Husam N Alshareef, Ming Liu, and Juan B Roldan. Memristive technologies for data storage, computation, encryption, and radio-frequency communication. *Science*, 376(6597):eabj9979, 2022.

[55] Niel Lebeck, Arvind Krishnamurthy, Henry M Levy, and Irene Zhang. End the senseless killing: Improving memory management for mobile operating systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 873–887, 2020.

[56] Sekwon Lee, Soujanya Ponnapalli, Sharad Singhal, Marcos K Aguilera, Kimberly Keeton, and Vijay Chidambaram. Dinomo: an elastic, scalable, high-performance key-value store for disaggregated persistent memory. *Proceedings of the VLDB Endowment*, 15(13):4023–4037, 2022.

[57] Seok-Hee Lee. Technology scaling challenges and opportunities of memory devices. In *2016 IEEE International Electron Devices Meeting (IEDM)*, pages 1–1. IEEE, 2016.

[58] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. Mind: In-network memory management for disaggregated data centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 488–504, 2021.

[59] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. Virtual-memory assisted buffer management. *Proceedings of the ACM on Management of Data*, 1(1):1–25, 2023.

[60] Philip Levis, Kun Lin, and Amy Tai. A case against cxl memory pooling. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, pages 18–24, 2023.

[61] Huaicheng Li, Daniel S Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst*, 2023.

[62] Jochen Liedtke. On micro-kernel construction. *ACM SIGOPS Operating Systems Review*, 29(5):237–250, 1995.

[63] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. Disaggregated memory for expansion and sharing in blade servers. *ACM SIGARCH computer architecture news*, 37(3):267–278, 2009.

[64] Jimmy Lin and Alek Kolcz. Large-scale machine learning at twitter. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 793–804, 2012.

[65] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Scale and performance in a filesystem semi-microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 819–835, 2021.

[66] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 399–413, 2019.

[67] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 742–755, 2023.

[68] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. Processing data where it makes sense: Enabling in-memory computation. *Microprocessors and Microsystems*, 67:28–41, 2019.

[69] Joel Nider, Craig Mustard, Andrada Zoltan, John Ramsden, Larry Liu, Jacob Grossbard, Mohammad Dashti, Romaric Jodin, Alexandre Ghiti, Jordi Chauzi, et al. A case study of Processing-in-Memory in off-the-Shelf systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 117–130, 2021.

[70] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, 2019.

[71] Shaurya Patel, Sidharth Agrawal, Alexandra Fedorova, and Margo Seltzer. Cheri-picking: Leveraging capability hardware for prefetching. In *Proceedings of the 12th Workshop on Programming Languages and Operating Systems*, pages 58–65, 2023.

[72] Ivy Peng, Marty McFadden, Eric Green, Keita Iwabuchi, Kai Wu, Dong Li, Roger Pearce, and Maya Gokhale. Umap: Enabling application-driven optimizations for page management. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 71–78. IEEE, 2019.

[73] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[74] 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*, pages 31–39, 1987.

[75] Richard F Rashid. From rig to accent to mach: The evolution of a network operating system. In *Proceedings of 1986 ACM Fall joint computer conference*, pages 1128–1137, 1986.

[76] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 392–407, 2021.

[77] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. AIFM:High-Performance,Application-Integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 315–332, 2020.

[78] rwestMSFT. Memory Management Architecture Guide - SQL Server — learn.microsoft.com. https://learn.microsoft.com/en-us/sql/relational-databases/memory-management-architecture-guide. [Accessed 09-01-2024].

[79] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid Aref, Marcelo Arenas, Maciej Besta, Peter A Boncz, et al. The future is big graphs: a community view on graph processing systems. *Communications of the ACM*, 64(9):62–71, 2021.

[80] 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), pages 69–87, 2018.

[81] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R Lyu. FUSEE: A fully Memory-DisaggregatedKey-Value store. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 81–98, 2023.

[82] Michael Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, 1981.

[83] Michael Stonebraker and Ariel Weisberg. The voltdb main memory dbms. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[84] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Ipoom Jeong, Ren Wang, and Nam Sung Kim. Demystifying cxl memory with genuine cxl-ready systems and devices. *arXiv preprint arXiv:2303.15375*, 2023.

[85] Bijan Tabatabai, Mark Mansi, and Michael M Swift. Fbmm: Using the vfs for extensibility in kernel memory management. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 181–187, 2023.

[86] Andrew S Tanenbaum and Albert S Woodhull. *Operating systems: design and implementation*, volume 68. Prentice Hall Englewood Cliffs, 1997.

[87] Thomas N Theis and H-S Philip Wong. The end of moore's law: A new beginning for information technology. *Computing in science & engineering*, 19(2):41–50, 2017.

[88] Shin-Yeh Tsai and Yiying Zhang. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 306–324, 2017.

[89] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A Memory-Disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 261–280, 2020.

[90] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Canvas: Isolated and adaptive swapping for Multi-Applications on remote memory. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 161–179, 2023.

[91] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, et al. Tmo: transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 609–621, 2022.

[92] Luna Xu, Min Li, Li Zhang, Ali R Butt, Yandong Wang, and Zane Zhenhua Hu. Memtune: Dynamic memory management for in-memory data analytic platforms. In *2016 IEEE international parallel and distributed processing symposium (IPDPS)*, pages 383–392. IEEE, 2016.

[93] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 331–345, 2019.

[94] Ting Yang, Emery D Berger, Scott F Kaplan, and J Eliot B Moss. Cramm: Virtual memory support for garbage-collected applications. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 103–116, 2006.

[95] Wonsup Yoon, Jisu Ok, Jinyoung Oh, Sue Moon, and Youngjin Kwon. Dilos: Do not trade compatibility for performance in memory disaggregation. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 266–282, 2023.