# Limited Access: The Truth Behind Far Memory

Anil Yelam*, Stewart Grant*, Enze Liu, Radhika Niranjan Mysore†,
Marcos K. Aguilera†, Amy Ousterhout and Alex C. Snoeren

UC San Diego †VMware Research

## ABSTRACT

Memory capacity in data centers is becoming a scarce resource. To address this issue, emerging runtimes enable applications to supplement their local memory with additional tiers of compressed, non-volatile, or far memory, often accessed via OS-supported paging. In these systems, minimizing page faults is crucial for good performance. Yet, there is little common understanding of which parts of application code are responsible for triggering page faults. In this paper, we analyze page-fault behavior across a suite of 26 applications and find that the vast majority of page faults are triggered by a very small number of lines of application code. In the light of this and related observations, we discuss the feasibility of several ways to reduce page faults.

**ACM Reference Format:**
Anil Yelam*, Stewart Grant*, Enze Liu, Radhika Niranjan Mysore†,, Marcos K. Aguilera†, Amy Ousterhout and Alex C. Snoeren, . 2023. Limited Access: The Truth Behind Far Memory. In *4th Workshop on Resource Disaggregation and Serverless (WORDS '23), October 23, 2023, Koblenz, Germany*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3605181.3626288

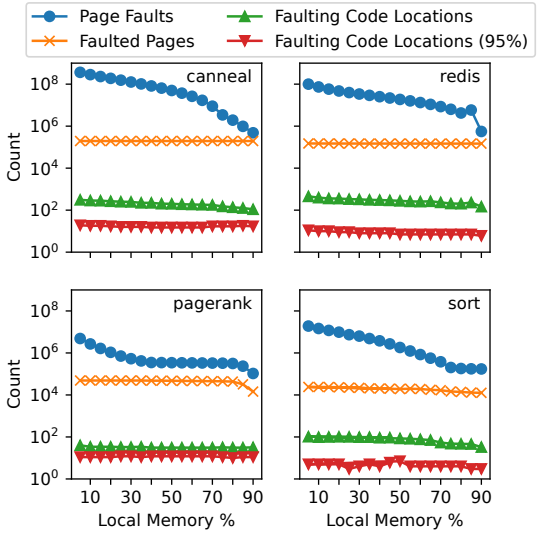


**Figure 1:** In four representative applications, the number of unique code locations that trigger the vast majority (95%) of page faults (red) remains small, orders of magnitude less than the number of overall page faults (blue), unique pages accessed (orange), or total locations that generate faults (green).

## 1 INTRODUCTION

Memory capacity—once an abundant resource—is the limiting factor for many applications in data centers today. This shift is due to the confluence of three trends: increasing demand for in-memory computing over multi-terabyte datasets [14, 17], stagnating improvements in DRAM density and cost [22, 24], and increasing per-CPU core counts [19, 20]. As a result, applications increasingly supplement their local DRAM with additional tiers of memory. A common approach is to use OS-based swapping to page out (cold) application data to a tier of compressed memory [23], high-capacity non-volatile memory [28], or a pool of far memory shared by a rack of servers [11, 18, 34]. All of these approaches can bring a page into memory within several microseconds, enabling applications to access a much larger pool of memory without the latency of swapping pages from SSD or disk.

Despite the promise of new memory tiers, when applications operate with only a small fraction of their working set present locally, page faults are abundant and performance can significantly degrade [11, 18]. To avoid page faults, prior work has studied memory access patterns in an attempt to predict which pages are likely to be accessed in the near future to hide access

*These authors contributed equally.


*WORDS '23, October 23, 2023, Koblenz, Germany*

ACM ISBN 979-8-4007-0250-1/23/10.
https://doi.org/10.1145/3605181.3626288

latency [10, 13, 34]. In general, however, it is difficult to make accurate predictions without application-specific knowledge, which has led others to suggest that programmers should annotate or redesign their code to optimize memory usage [29], which may seem like an intractable task for sophisticated applications. In this work, we take a step back and ask, from how many different *locations in application code* do page faults actually arise in practice?

To answer this question, we develop a tool that records page faults (under a configurable paging policy) and identifies the specific line of code that triggered each fault (§2). We use this tool to study page-fault behavior across a suite of 26 applications (§3). Surprisingly, we find that while both the frequency of page faults and the number of faulted-on pages can be quite large, the number of unique lines of source code that are responsible for generating the majority of these faults is quite small—often fewer than 10. Figure 1 illustrates this trend for four applications across a range of local memory sizes.

We believe that this observation suggests new opportunities for reducing page faults in far memory systems, by focusing optimization efforts on the small number of code locations that are responsible for generating page faults. We discuss how these insights might be leveraged in, e.g., application-guided prefetching and page-fault-aware programming (§4). Most significantly, we believe this observation suggests programmer-aided or other manual techniques may be far more promising than previously imagined.

## 2 METHODOLOGY

In this section, we briefly describe how we record page faults while an application runs and identify which line of source code generated each fault (§2.1). Then, we describe the suite of applications that we analyze (§2.2).

### 2.1 Fault Annotation

Our goal is to track the page faults an application would generate if placed under memory pressure—without actually deploying any particular swapping system. While Linux's `perf-trace` tool [6] can record system-wide page faults, we are interested only in anonymous pages (excluding memory-mapped files) of a particular application. We also desire a flexible swap backend with configurable reclaim behavior instead of limiting ourselves to the default Linux swap policy.

**Trace collection.** To achieve these goals, we implement a simple page-fault tracing tool, *fltrace*, based on Userfaultfd [4] which lets us handle the page faults within the tool. Designed for C/C++ applications, a user links fltrace against an application at runtime using `LD_PRELOAD` and specifies the maximum amount of local memory it can use. Fltrace then interposes on all heap allocations and forwards them to Userfaultfd. It only ever maps pages below the local memory limit and keeps the rest of the heap unmapped. When accessed, the unmapped pages result in page faults forwarded back to the tool, which then uses signals to backtrace on the faulting thread and save the stack traces of instruction pointers. For this study, fltrace evicts pages on a simple first-in-first-out basis.

| Category | Count | LoC | Max RSS |
|---|---|---|---|
| **K**ey-value stores [31] | 4 | 7–260K | 0.5–1.3 GB |
| **G**raph [9] | 10 | 0.2–1.2K | 0.1–300 MB |
| **P**arsec [12] | 9 | 10–32K | 110–800 MB |
| **C**oreutils [2] | 3 | 1–3K | 44–98 MB |

**Table 1: Application Test Suite**

**Log processing.** Post execution, we filter out zero-page faults (i.e., first-ever faults on a mapped page) because they occur in any setting regardless of memory pressure. Next, we convert the log of stack traces (composed of instruction pointers) to lines of code using `addr2line` [1], which requires us to disable address space randomization and compile the applications with `gcc -g -no-pie -fno-pie` to ensure that code locations are binary independent and debug information is available. Some applications cannot be compiled with these flags if they contain shared libraries which must be independent; in these cases, we analyze traces by the instruction pointers alone.

We are interested in the distribution of faults across unique faulting code locations. However, in some applications these locations are in standard libraries, which provides no insight into what application code caused the fault. For example, if an application uses the C++ standard-vector library to load data, most faults may appear to occur from the same line in the library even though it is called from different lines of application code. The reverse case, in which one line of application code can trigger faults in multiple different locations in the library, is also possible. To identify the fault-triggering line of code in the application, we extract the backtrace and remove shared library locations to record only the top-most function in the stack that belongs to the application itself.

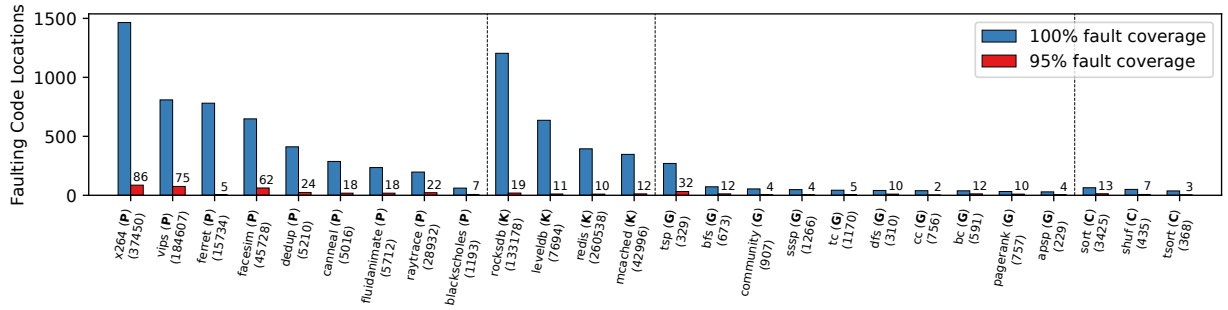


**Figure 2:** The number of unique lines of source code that account for 100% (blue) and 95% (red) of the page faults in 26 different applications. Local memory is configured to hold only 10% of the working set. The $x$-axis labels indicate the total number of lines of code in each application and the application's category (bolded letter).

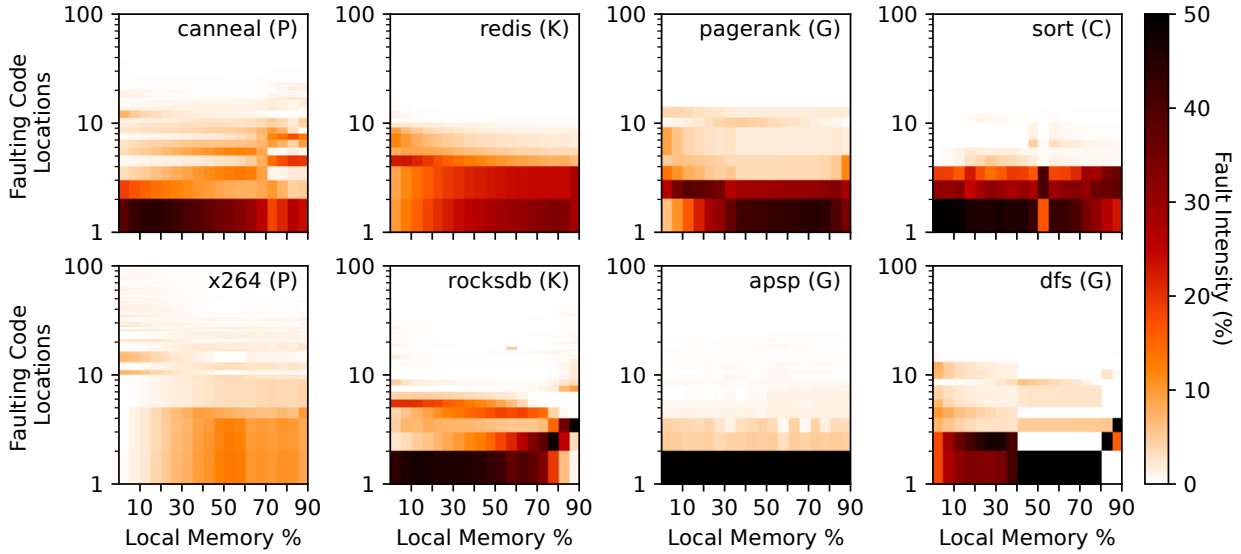


**Figure 3:** Heat maps showing the fault intensity of each faulting code location, for eight example applications. Most of the faults concentrate in the first several locations and the same locations remain the heavy-hitters across different local memory settings.

## 2.2 Application Suite

Our analysis includes applications drawn from different benchmark suites to cover a range of application categories listed in Table 1. We run each application with its standard benchmarking workload to ensure similar code coverage as in realistic scenarios. We first run each application with a large amount of local memory to observe the size of its maximum resident set size (RSS). We then use fltrace to restrict the amount of local memory available in our experiments to a varying fraction of the maximum RSS and record the resulting page faults. The working sets for these applications are not particularly large as our experiments with even these modest memory footprints already generate very large fault traces (tens of GBs); however we do not expect the faulting behavior to vary significantly for larger working sets as the behavior depends more on the ratio of allowed resident set size to working set size than the absolute size of the working set itself.

**KVS:** We evaluate four key-value store applications: two in-memory caches (Redis [7] and Memcached [5]) and two persistent data stores (Rocksdb [8] and LevelDB [3] with memory caches). We use the workloads from LK_PROFILE [31] but increase the number of operations to at least a million, which yields a proportionally larger memory footprint.

**Graph:** We consider the CRONO graph benchmark [9], which consists of 10 independent multi-threaded graph algorithms. We use the inputs provided with the benchmark suite and run each program with four threads.

**PARSEC:** PARSEC [12] is a benchmarking suite of compute-intensive applications designed to stress parallel performance on multi-core CPUs. We include an HPC-focused subset of these applications. We run each with eight threads and its *native* dataset.

**Coreutils:** For breadth we analyze a subset of the GNU `coreutils` library [2], as a representative set of programs with a small code base and small memory footprint. Most coreutils with the exception of `sort` have small working sets which are agnostic to the input, as they are designed to work on streaming data. We chose three which accept large files as input.

## 3 FINDINGS

Our experiments yield three key findings. The first two observations are based on analysis of just the unique instruction pointers (which are frequently in library code) that cause page faults. Our third observation results from tracing the call stacks of the faulting instruction back to their origins in application code.

The majority of page faults originate from a small number of code locations. Figure 2 shows for each application both the total number of code locations that trigger faults and the number of code locations responsible for 95% of faults when the application can hold only 10% of its maximum RSS in local memory. In most cases, a small number of locations cover 95% of faults—12 locations at the median and fewer than 32 for all but four applications.

Faulting code locations remain stable under a wide range of memory pressure. Figure 3 shows the intensity of each faulting code location under different amounts of local memory, where the "intensity" of a code location is the percentage of page faults at that local memory percentage that originate from this code location. The most-frequently faulting locations are shown at the bottom of each graph. While the intensity of a given location shifts as the local memory percentage changes, the set of heavy-hitter locations remains largely the same. This suggests that a user can profile an application at one specific local memory percentage to discover the faulting code locations, and that this set of locations will generalize well across different degrees of memory pressure.

Faults concentrate even more in application code. The previous two graphs are based on unique instruction pointers, so the faulting locations may be in library code.

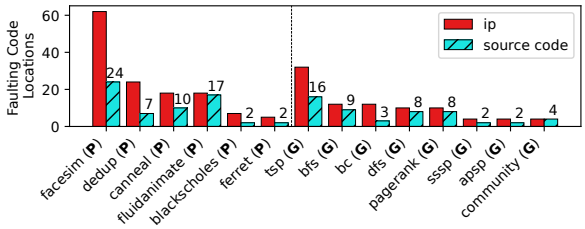


**Figure 4:** Code locations responsible for 95% of faults using instruction pointer analysis (including library code) vs. application source code analysis.

Figure 4 shows that when we trace the faults back to the line of application code that led to the fault, even fewer unique lines of code are implicated: less than 10 in most cases. (Because this analysis is labor intensive, we present results for only a subset of applications.)

## 4 IMPLICATIONS

The number of important faulting code locations in the applications we analyze is well within the range of manual inspection. This provides an opportunity to reduce the number of page faults for far memory applications not by predicting memory access patterns, which is known to be challenging, but rather by carefully handling this small number of faulting locations.

### 4.1 Better Prefetching

App-aware Prefetching: Prefetching performs poorly when accesses are random, and at the OS level predictive prefetches can become muddled by the seemingly random accesses of many threads and processes. Access patterns at specific faulting locations may follow a more predictable pattern, which could be exploited to improve prefetching accuracy. Language-level annotations at the faulting locations may reveal simple access patterns like strides, which were previously hidden from the underlying systems like Leap and 3PO [10, 13]. For example, a managed-language application with regular accesses running with garbage collection enabled may exhibit a near random access pattern from the OS perspective due to muddled accesses [33]. Labeling application and the garbage-collection accesses as such could enable a prefetcher to isolate the application accesses and run the prefetching on just these accesses with better accuracy.

When to fetch? Researchers have long focused on identifying what data to prefetch, and prior systems have incorporated a wide range of hints to try and improve accuracy rates [25, 27, 30, 32]. Our observation addresses a different dimension, which is *when to fetch*. Even if a prefetcher knows what to fetch, an untimely request

may consume limited memory bandwidth restricting application throughput. Just like modern CPUs do branch prediction to determine which code paths are likely to be executed (and results committed), speculation about the upcoming set of instructions could provide input to the prefetcher regarding when to dispatch requests. Similarly runtimes like Javascript, Java, Go, and Python that have knowledge of code flow could leverage this foresight to hide access latency.

## 4.2 Manual Hinting

The main faulting locations provide a great place for a variety of hints beyond prefetching to let the application guide the memory management without requiring significant application changes as in AIFM [29].

Alleviating page fault overheads: Page-fault handling incurs significant overhead to transition between user and kernel space, motivating solutions like AIFM [29] and DiLOS [35], which handle remote accesses in userspace. In that spirit, a few programmer-added hints at the frequent faulting locations could avoid the cost of page faults by servicing them in userspace.

Latency hiding: Latency of page-fault handling can be hidden by scheduling another thread to run while the page fault is serviced. Typically, rescheduling occurs at the kernel, but this is too slow when context switch time is on the order of the time to service a fault on a fast device as in remote memory (2–3 $\mu$s). Instead, with userspace hints about where page faults might occur, we can use userspace schedulers like Shenango [26] to reschedule execution in nanoseconds.

Informing memory-management decisions: Annotating important faulting locations could provide rich information to a user-level paging runtime, similar to AIFM [29]. For example, each faulting code location may point to a specific data structure in the program. Based on offline profiled hotness of data structures, memory reclamation could prioritize the pages (or data) faulted in at certain code locations over others.

## 4.3 Fault-aware Programming

When developers know that the number of faulting code locations is small and also know their locations and frequencies, it becomes feasible for them to manually improve program behavior around them, in a few ways.

Optimizing data structures to reduce faults: In some cases, refactoring data structures can reduce the number of page faults. Consider the paradigm of inlining metadata with data. Iterations over this sparse metadata triggers faults proportional to the size of the data. Placing metadata into a separate small structure can dramatically reduce the number of faults, especially if it enables data that is commonly written to be separated from read-only data, thereby reducing the number of write-protect faults. Other faults can be reduced by minimizing the amount of data processed or adjusting data-structure alignment. Data structures may be shrunk by considering data types carefully (e.g., replacing `int64` with `int16`). Just as some programming languages provide keywords to facilitate cache line alignment, languages could provide support for aligning data structures to page granularities. Ultimately, understanding faulting locations may lead us to better data representations and algorithms.

Code Optimization: Important faulting code locations are typically near the code that processes high volumes of data. This neighborhood of code is ripe for optimization as improvements will have a proportionally large impact.

## 4.4 Others

VM Memory Density: As the ratio of memory to compute continues to shrink, servers tend to get packed with a large number of VMs with little memory and high page-fault rate. Because a host can only serve a few million page faults per second, page faults become a scarce resource. Accordingly, developers can be given budgets on page faults, and knowing the key faulting code locations can help them manage their budget.

Mitigating side channels: Page faults create side channels because they expose information about how often data is accessed (infrequent data have a much higher access latency). Accordingly, information about important faulting code locations can help developers remove these side channels (e.g., by ensuring the time to execute the code remains the same whether the fault occurs or not).

# 5 RELATED WORK

We are not aware of recent profiling studies of page fault behavior, but there is prior work in adjacent topics.

**Architecture-level profiling:** CPU profiling of application code is a mature topic, with both open-source and commercial tools available (e.g., GNU `perf` [6], Intel vTune, AMD $\mu$prof). These tools can identify instructions that cause cache misses, which are analogous to page faults. But the tools are quite different from page fault profiling because of the different time scales involved: processor caches operate in the nanosecond scale, and so profiling needs hardware help (e.g., performance counters), while page faults operate in the microsecond scale, which is amenable to software profiling. Despite these

differences, CPU cache profiling is driven by a similar insight as ours: unoptimized code may have a small number of instructions that cause a large number of misses.

**Hot and cold pages:** Prior work has looked into techniques to classify pages as hot and cold to determine the best candidates for swapping out (e.g. LRU, 2Q LRU [21], multi-generational LRU [15], etc). These methods are based on the insight that applications have working sets [16] with few hot pages relative to cold pages. This insight differs from ours because the hot or cold pages refer to the data being accessed, not code locations.

**Far memory:** Recent work proposes ways to improve swapping performance in the kernel for far memory [10, 11, 18, 34]. These techniques are orthogonal to our work since they are application agnostic, while our work provides insights into applications. Other systems access far memory through software constructs such as remoteable pointers [29, 36], rather than transparently through page faults. These systems could benefit from our insights (e.g., code profiling could allow developers to optimize remoteable pointers).

## 6 CONCLUSION AND FUTURE WORK

In this paper we study the origins of page faults in far memory systems. We find that, surprisingly, the majority of page faults originate from a small number of locations in application source code, and that these code locations are relatively stable across different amounts of local memory. Our study is limited in several ways: we focus on a modest number of applications, our application working set sizes are limited due to the overheads of post processing traces, and we only evaluate one workload for each application; we leave overcoming these limitations to future work. Despite the limitations of our study, we believe that it suggests promising new opportunities for reducing page faults in far memory systems via fault-aware programming, improved prefetching, and—perhaps most saliently—manual hinting.

## ACKNOWLEDGEMENTS

This work was conducted under a sponsored research agreement between UC San Diego and VMware Research. S. Grant is supported by a Meta Research PhD Fellowship.

## REFERENCES

[1] 2023. addr2line. https://man7.org/linux/man-pages/man1/addr2line.1.html.
[2] 2023. GNU core utilities. https://www.gnu.org/software/coreutils/.
[3] 2023. Leveldb - An open-source on-disk key-value store. https://en.wikipedia.org/wiki/LevelDB.
[4] 2023. Linux Userfaultfd. https://www.kernel.org/doc/html/latest/admin-guide/mm/userfaultfd.html.
[5] 2023. Memcached - Free, open source, high-performance, distributed memory object caching system. https://memcached.org/.
[6] 2023. perf-trace. https://man7.org/linux/man-pages/man1/perf-trace.1.html.
[7] 2023. Redis - an open source, in-memory data store. https://redis.io/.
[8] 2023. Rocksdb - A persistent key-value store for fast storage environments. http://rocksdb.org/.
[9] Masab Ahmad, Farrukh Hijaz, Qingchuan Shi, and Omer Khan. 2015. CRONO: A Benchmark Suite for Multithreaded Graph Algorithms Executing on Futuristic Multicores. In *Proceedings of the IEEE International Symposium on Workload Characterization*. 44–55.
[10] Hassan Al Maruf and Mosharaf Chowdhury. 2020. Effectively Prefetching Remote Memory with Leap. In *Proceedings of the USENIX Annual Technical Conference* (Virtual Event).
[11] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can far memory improve job throughput?. In *Proceedings of the 15th European Conference on Computer Systems* (Heraklion, Greece).
[12] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph. D. Dissertation. Princeton University.
[13] Christopher Branner-Augmon, Narek Galstyan, Sam Kumar, Emmanuel Amaro, Amy Ousterhout, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2023. 3PO: Programmed Far-Memory Prefetching for Oblivious Applications. *arXiv preprint arXiv:2207.07688* (2023).
[14] Kindra Cooper. 2021. OpenAI GPT-3: Everything You Need to Know. https://www.springboard.com/blog/data-science/machine-learning-gpt-3-open-ai/.
[15] Jonathan Corbet. 2021. Multi-generational LRU: the next generation. https://lwn.net/Articles/856931.
[16] Peter J. Denning. 1968. The working set model for program behavior. *Commun. ACM* 11, 5 (May 1968), 323–333.
[17] Google. 2023. The Size and Quality of a Data Set. https://developers.google.com/machine-learning/data-prep/construct/collect/data-size-quality.
[18] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient memory disaggregation with infiniswap. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation* (Boston, MA). 649–667.
[19] HPC Wire. 2022. AMD's Genoa CPUs Offer Up to 96 5nm Cores Across 12 Chiplets. https://www.hpcwire.com/2022/11/10/amds-4th-gen-epyc-genoa-96-5nm-cores-across-12-compute-chiplets/.
[20] Intel. 2023. Intel Launches 4th Gen Xeon Scalable Processors, Max Series CPUs. https://www.intel.com/content/www/us/en/newsroom/news/4th-gen-xeon-scalable-processors-max-series-cpus-gpus.html.
[21] Theodore Johnson and Dennis Shasha. 1994. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the International Conference on Very Large Databases* (San Francisco, CA). 439–450.
[22] Uksong Kang, Hak-Soo Yu, Churoo Park, Hongzhong Zheng, John Halbert, Kuljit Bains, S. Jang, and Joo Sun Choi. 2014. Co-architecting Controllers and DRAM to Enhance DRAM Process Scaling. In *Proceedings of the Memory Forum*.

[23] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. 2019. Software-Defined Far Memory in Warehouse-Scale Computers. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 317–330.

[24] Seok-Hee Lee. 2016. Technology Scaling Challenges and Opportunities of Memory Devices. In *IEEE International Electron Devices Meeting*.

[25] Todd C. Mowry, Angela K. Demke, and Orran Krieger. 1996. Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications. *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*.

[26] Amy Ousterhout, Joshua Fried, Jonathan Behrens, and Adam Belay. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation* (Boston, MA). 361–377.

[27] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. 1995. Informed Caching and Prefetching. In *Proceedings of the ACM SIGOPS 15th Symposium on Operating Systems Principles* (Copper Mountain, CO).

[28] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany). 392–407.

[29] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, application-integrated far memory. *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, 315–332.

[30] Andrew Tomkins, R. Hugo Patterson, and Garth Gibson. 1997. Informed Multi-Process Prefetching and Caching. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (Seattle, WA).

[31] Muhammed Ugur, Cheng Jiang, Alex Erf, Tanvir Ahmed Khan, and Baris Kasikci. 2023. One Profile Fits All: Profile-Guided Linux Kernel Optimizations for Data Center Applications. *SIGOPS Oper. Syst. Rev.* 56, 1 (June 2023), 26–33.

[32] S. VanDeBogart, C. Frost, and E. Kohler. 2009. Reducing Seek Overhead with Application-Directed Prefetching. In *Proceedings of the USENIX Annual Technical Conference* (San Diego, CA).

[33] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. 2022. MemLiner: Lining up Tracing and Application for a Far-Memory-Friendly Runtime. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation* (Carlsbad, CA). 35–53.

[34] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Yiying Zhang, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2023. Canvas: Isolated and adaptive swapping for multi-applications on remote memory. In *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation* (Boston, MA).

[35] Wonsup Yoon, Jinyoung Oh, Jisu Ok, Sue Moon, and Youngjin Kwon. 2021. DiLOS: Adding Performance to Paging-Based Memory Disaggregation. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems* (Hong Kong, China). 70–78.

[36] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. 2022. Carbink: Fault-Tolerant Far Memory. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation* (Carlsbad, CA). 55–71.