DynamicSwap: System-Aware Paging of Far Memory

Paul-Andrei Aldea
University of Michigan

Pranav Bangarbale
University of Michigan

Rishi Patel
University of Michigan

Abstract

Far-memory systems, such as AIFM [1] and Infiniswap [2], provide convenient mechanisms to leverage excess memory availability across interconnected nodes. Using technologies like Remote Direct Memory Access (RDMA) or more general network communications (TCP), these systems aim to offer programmers memory flexibility with minimal increases in latency. However, the intrinsic latency of network transactions necessitates careful design to ensure that performance trade-offs are acceptable given the benefits of expanded memory capacity. Presently, nearly all far memory systems employ some prefetching mechanism, reminiscent of caches, to meet performance guarantees. However, to our best knowledge, no existing far memory system implements or affords the ability to fetch remote memory proactively to the user process' request. In this work, we introduce DynamicSwap, an extension of an existing far-memory system which monitors the availability of system resources – primarily memory – and attempts to fetch remote memory before fault time. Our evaluation shows that DynamicSwap achieves up to a 4,000-fold improvement in critical memory access latency, achieving high efficiency, throughput, and overall responsiveness, in memory-intensive applications.

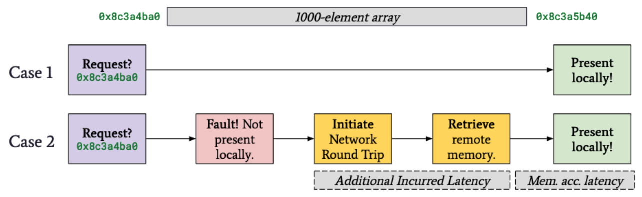


Figure 1: *The additional overheads caused by memory-intensive workloads in far memory systems.*

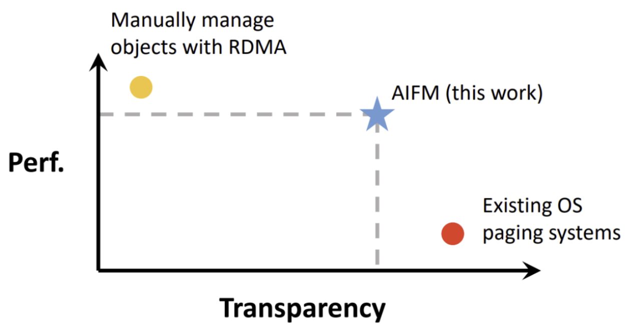
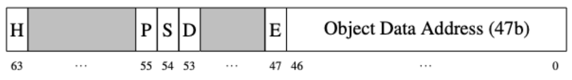


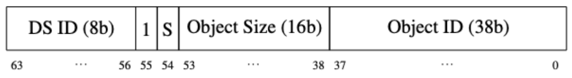
Figure 2: *High-level comparison between approaches to memory transparency.*

1 Background

The capability of offloading memory across multiple machines as "far memory" resources has been a relatively niche area of study in the systems field [3]. While the motivations for developing such systems—such as accommodating large-scale workloads and improving memory utilization—are compelling, the practical necessity for far memory has been diminished by the ever-decreasing cost of local memory. Furthermore, existing solutions necessarily leverage methods to achieve low CPU tail latency for remote memory to ensure acceptable performance and overall usability [4]. Far memory systems introduce additional dependencies and the potential for performance degradation due to network overhead. Figure 1 displays the main issue; there is an additional latency incurred by initiating a network round trip and retrieving remote memory at fault time. Thus, it is of little surprise that remote memory solutions often remain confined to the research field. The complexity of integrating these systems into real-world applications, combined with challenges such as unpredictable latency, scalability issues, and the lack of robust support for diverse workloads, frequently prevents them from transitioning into practical, widely-adopted technologies.



(a) Local object. H: hot, P: present, S: shared, D: dirty, E: evacuating.



(b) Remote (swapped-out) object. DS ID means data structure ID.

Figure 3: *AIFM far memory pointer constructions.*

Our project has been adapted to the Application Integrated Far Memory (AIFM) system [1], which strikes a balance between user transparency and fine-tuned usability. The relationship between existing solutions for memory abstraction is shown in Figure 2. AIFM is a library which is directly linked to the user program and provides API-like calls through which a programmer may explicitly manage the allocation and placement of objects within the code. In particular, in AIFM, the far-memory abstraction is much more granular than in previous solutions such as FastSwap [5] or Infiniswap [2]. Previous systems primarily operate at the page or "block" level, which can oftentimes yield undesirable performance bottlenecks, especially when processes exhibit poor spatial locality. Since objects and data-structures are allocated contiguously in virtual memory, such bottlenecks are much less prevalent. As a result of the high degree of flexibility which AIFM provides over the management of remoteable objects, we have chosen to adapt it as the foundation for our system, enabling seamless integration with diverse workloads and the ability to optimize memory management strategies for specific application requirements.

1.1 AIFM Overview

At the core of AIFM, far-memory is allocated and referenced through the use of "far-memory pointers." In the general case, such pointers are restricted to 64-bits, with the address only comprising the 47 least-significant bits, and various overhead information aligned in the upper 17 bits. However, the construction of the pointer is changed depending on whether the data to which it points is "present" or "not present" [1]. Further optimizations on far-memory pointers, such as inverting bit meanings, are performed to ensure critical-path speed; an overview of the pointer mechanism is detailed in figure 3. The state of a pointer is either implicitly managed through its access (during runtime), or explicitly placed (through "evacuators") in local or remote memory. To this end, a remote memory reference is only accessible through an AIFM-defined scope object, which ensures that any swapped-in memory is strictly "fenced" within a given scope and may be swapped-out if the reference is no longer accessible. Consequently, an access to a remote memory object outside of such scope is undefined behavior. The scope mechanism is reminiscent of "guards" in multi-threaded environments or resource allocation through initialization (RAII) mechanisms.

[1]Here we take "present" to mean locally-available, and "not present" to mean remotely-available. A pointer may exclusively only be in one such state

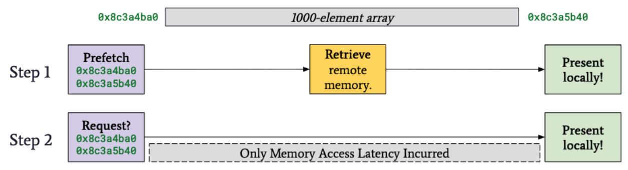


Figure 4: *Ideal execution of a remote memory prefetcher.*

In addition to the far memory management mechanism implemented by AIFM, the system also affords the ability to use pre-implemented data-structures which are inherently remoteable. Pre-constructed data-structures prevent the average user of AIFM from needing to allocate and handle multiple pointers in the common case (e.g. declaring an array, stack, hashtable, etc.). Our solution makes use of these pre-existing structures to easily construct and evaluate performance, given the relative ease of use and high degree of transparency.

Apart from the implemented data-structures and mechanisms behind far memory management, AIFM also contains a critical "prefetcher block," which aims to reduce the latency with which predictable memory accesses are made. Much like in a traditional cache, AIFM is capable of maintaining a history of pointer accesses and declare certain pointers as "hot" [2]. From preliminary destructuring of the codebase, we notice that the prefetcher block collects information on all memory accesses, but is only invoked at memory-fault time [3]. As such, AIFM is capable of achieving nanosecond latency performance on most memory accesses, and deliver more than acceptable performance when compared across several existing far-memory systems.

1.2 Problem Overview

In order to combat the problem shown by Figure 1 (incurring the entire network round-trip latency on every fault), AIFM implements a prefetcher. In the most ideal scenario, the prefetcher can bring in many remote pointers before a fault occurs, shown in Figure 4. A preliminary evaluation of AIFM,

[2]A "hot" pointer is one which is frequently accessed and should be infrequently evicted to ensure performance.

[3]We take "memory fault" to mean a memory accesses that results in the swapping of memory from a remote source to the local machine; similar to a page fault in traditional virtual memory.

Trial \ Index	62	63	64	65	66
1	107	123	**462,972**	152	105
2	198	96	**490,651**	188	71
3	122	108	**443,290**	203	289

Table 1: Latency in nanoseconds (ns) of sequential accesses to an array from indices 62 - 66 with the AIFM prefetcher enabled.

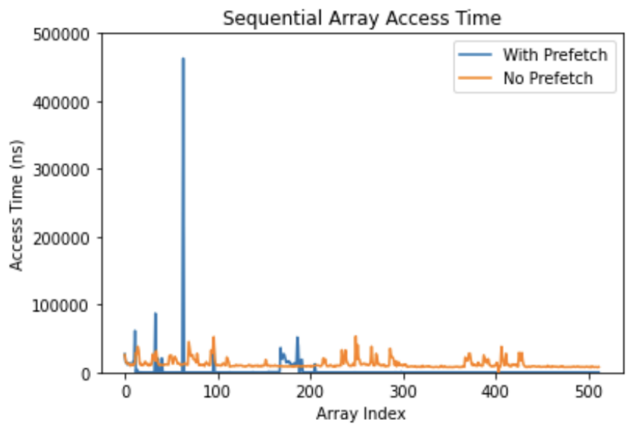


Figure 5: Visualization of memory access latency in nanoseconds for sequential array accesses with and without the AIFM prefetcher enabled.

particularly of the prefetcher block, yields a benchmark for the latency of memory accesses. When a remote memory object is locally present at memory access time, the latency incurred is mainly derived from the limitations of the hardware on the local machine [4]. Furthermore, the prefetcher block is capable of rapid speculative paging of memory, ensuring that most memory accesses are performed on local memory objects. However, when a memory access occurs for an object which is not locally present, we notice a dramatic increase in memory access latency of up to a factor of 4,000. An outline of the preliminary data can be found in table 1 and visualized in figure 5. Section 2 will expand on the reasoning behind the observed latency pattern, along with its potential impact on the usability of AIFM. Section 3 will provide a brief overview of our intended solution to address memory latency spikes, while section 4 will discuss the implementation of the solution. Finally, we present section 5 to evaluate the improvement of our approach over the base AIFM system, and conclude with a discussion about our solution and future works in sections 7 and 6, respectively.

[4]Information about the hardware used to collect evaluation data is outlined in section 5.

2 Motivation

As shown both in table 1 and figure 5, there is a clear shortcoming of the AIFM system in its ability to ensure low tail latency for general memory accesses. In particular, the memory access latency measured appears to be greater when running the far-memory system with the AIFM prefetcher enabled – a clearly counter-intuitive result. To find a cause behind the measured behavior, we turn to the actual code of AIFM. After decomposing the AIFM codebase, along with further testing, we arrive at the conclusion that the AIFM prefetcher will pre-fetch no more than $1KB$ of data at a given time, and will generally prefer to target data which exhibits high spacial locality at access time. However, even the $1KB$ upper-bound on the pre-fetch size is difficult to achieve seeing as the prefetcher is required to establish a strong access pattern to justify the decision of making such a large amount of memory locally present. Furthermore, the prefetcher will observe memory accesses which result in a "hit" on the local machine, but will never choose to swap any remote memory into the local system as a result – even if doing so might result in lower latency on a future access. To this end, the prefetcher behaves similar to a data cache, preferring only to page data in once a miss has occurred and otherwise limiting any speculative fetching.

Given this observation, the results obtained in table 1 appear less puzzling. The given sequential array accesses have caused the prefetcher to make up to $512B$ of data present in local memory. Once the series of accesses crosses the "block threshold" of the pre-fetched memory, a large spike in latency can be expected, since the memory access has faulted and must be retried after it is made locally present. The latency discrepancy between the two measurements taken in figure 5 is less intuitively explainable, though the current hypothesis relates to the possibility of more infrequent network communication occurring with the prefetcher enabled, which could lead to an increased impact of context-switching latencies within the "tcp block device."

Since the existing AIFM system is insufficient to deliver low tail-latency performance, our solution focuses on mitigating the frequency with which latency spikes are observed. By ensuring that memory access time is as consistent as possible, our approach addresses a critical barrier to the adoption of far memory systems: the unpredictable and often high latency associated with remote memory access. This improvement enhances the practicality and reliability of far memory solutions, potentially making them more accessible to a broader range of applications and increasing their adoption in real-world systems.

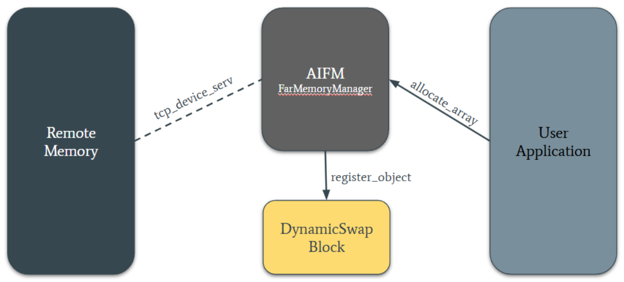


Figure 6: Example relationship between DynamicSwap, AIFM, the user application, and the far-memory node.

3 Solution Overview

As mentioned in section 1.2, AIFM experiences high latency spikes when far memory is accessed and the data associated with such access is not locally present. This observation raises a natural and important question: could the relevant data have been retained in local memory or managed in a way that avoids the need for remote placement altogether? Investigating this question involves evaluating the effectiveness of memory placement policies and the potential for optimizing data locality to minimize reliance on far memory, thereby reducing latency spikes and improving overall system performance. However, an even more fundamental observation occurs when considering the problem which far-memory placement attempts to solve in the first place: the unavailability of local memory. An argument could be made that far-memory performance can be improved by employing a hierarchical approach to memory placement (much like traditional virtual memory) and by prioritizing the local memory system over the remote one. Furthermore, even when remote placement is absolutely necessary or preferable, the availability of local system resources will naturally vary over time; as such, an optimal placement of memory at one given time-step may be suboptimal at a later time-step.

Given the variance in the availability of local resources, our solution introduces the ability to monitor the local memory usage and allotment for a process, and dynamically swap remote data into the local machine as local resource limits permit. In addition, our solution prefers to maintain data in local memory whenever possible, and evict as infrequently as possible to ensure that any remote swapping is done only in absolutely necessary contexts.

Our solution is designed to operate orthogonally to both the far-memory system and the user application, ensuring overall compatibility, without substantial modifications to the far-memory system and no modifications to the user application. Of the modifications necessary to the far-memory system, most are only limited to simple message-passing to indicate remote memory allocations and de-allocations. This allows for an internal management of both available system resources, and the current utilization of remoteable objects by the user application. A simple illustration of the intended solution can be seen in figure 6.

4 DynamicSwap

In this section, we introduce DynamicSwap, the core contribution of our work, along with a high-level implementation of its features. DynamicSwap aims to enhance existing far-memory systems such as AIFM, addressing the latency challenges associated with remote memory access. DynamicSwap is capable of independently monitoring system resource availability and preemptively fetching remote memory (when resources permit) with trivial modifications to the existing far-memory system – primarily limited to "hooking into" existing function calls. DynamicSwap is split into two blocks: the memory block and the pager block, each responsible for cooperatively achieving low-tail latency far-memory accesses when possible.

When the user programs begins to run, the AIFM "FarMemoryManager" object is constructed. When this occurs, the DynamicSwap block is notified and starts the memory and pager block, along with the necessary listeners to register the creation of remoteable objects by the user application. All notifications are carried out via function calls into the DynamicSwap codebase, and thus necessitate the altering of the AIFM codebase.

The memory block consists of logic to obtain information regarding the current placement of remoteable objects at a given time-step. For example, the memory block is capable of iterating over all of the allocated objects [5], and calculating how much local memory is utilized by the locally-present subset. The memory calculation occurs once every "memory quantum" to ensure that negligible processing power is consumed by the operation. The data is then stored in a shared buffer across the pager block to influence the pager's decisions. An outline of the memory block is given below:

[5]The "allocated objects" are strictly referring to the objects tracked by AIFM, and are subsequently capable of being moved to a remote resource.

```
uint64_t avail_mem; // atomic

int memory_block() {
    while (running) {
        avail_mem.store(kCacheSize);
        for (const auto& ref : obj_refs) {
            if (ref.ptr->meta().is_present())
                avail_mem -= ref.size;
        }

        this_thread::sleep_for(MEM_QUANT);
    }

    return 0;
}
```

The pager block consists of logic to move remote objects to the local machine, leveraging information about which pointers are already local, along with the size of each object associated with a given pointer. The size of each pointer is then considering with respect to the total amount of memory allocatable to the process along with the current amount of memory available as calculated by the memory block. The pager block will only run once every "pager quantum" amount of time to ensure minimal performance overhead for DynamicSwap. In addition, if the pager notices that the amount of memory available is less than a fraction of allocatable memory, then the pager will suspend its execution until the next iteration. This ensures that there is no thrashing over local memory, and that the decisions made by the pager block do not result in immediate evictions. Additionally, the pager block will ensure that no memory is fetched which will cause an eviction, by ensuring that the size of the referenced objects never exceeds the available amounts of local memory. An outline for the pager block is given below:

```
int pager_block() {
    while (running) {
        if (avail_mem < SWAP_THRESHOLD)
            goto sleep;
        for (const auto& ref : obj_refs) {
            if (ref.ptr->meta().is_present())
                continue;
            if (ref.size > avail_mem)
                goto sleep;
            ref.ptr->swap_in(...);
        }

        sleep:
            this_thread::sleep_for(PAGER_QUANT);
    }

    return 0;
}
```

In addition to the shown code, further consideration needed to be made to protect the shared access to data-structures, along with shared data buffers. Simple thread-safe mechanisms were utilized to ensure no race conditions within the DynamicSwap codebase are possible. Unfortunately, there is currently no way to directly integrate DynamicSwap with AIFM in a threaded context, resulting in an entirely sequential order of operations for all actions occurring in either AIFM, the user application, or the DynamicSwap codebase. Further discussion about limitations is addressed in section 4.1 and expanded upon in section 6.

4.1 Limitations

The main limitations of DynamicSwap's implementation directly stem from the existing limitations of AIFM. As previously mentioned, AIFM does not support the declaration and fetching of remote memory from different multi-threaded contexts. As a result, the pager block is currently not able to make any remote memory references locally available due to undefined behavior. Adapting DynamicSwap to a different existing far-memory system would have mitigated this issue, though this discovery was made late into the development process. A work-around to this problem would involve the modification of the AIFM codebase to either change how the existing prefetcher is functioning or allow for AIFM to be thread-safe. In our views, none of these solutions are satisfactory since they require greater integration and development of the far-memory system, which violates the desired orthogonality of our solution. If AIFM can be further developed to be thread-safe, our implementation would be functional, though this approach is a much larger undertaking than a semester-long project affords time for. We instead provide a theoretically functional pager block should AIFM allow for multi-threaded execution, and focus our evaluation on a simulation of the pager block directly within the user application. Although this is not an entirely accurate representation of the capabilities of DynamicSwap, it does provide insight into how a fully functional solution could drastically reduce the access latency of certain memory accesses by several orders of magnitude. Findings about our simulation approach are detailed in section 5.

5 Evaluation

The evaluation was performed using two CloudLab XL170 nodes, which feature the following hardware:

1. 2 x Ten-core Intel E5-2640v4 at 2.4 GHz
2. 64GB ECC Memory (4x 16 GB DDR4-2400 DIMMs)
3. Intel DC S3520 480 GB 6G SATA SSD
4. Two Dual-port Mellanox ConnectX-4 25 GB NIC

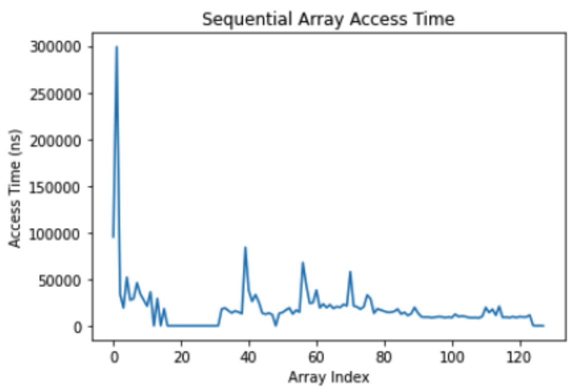


Figure 7: Sequential Array Access Times.

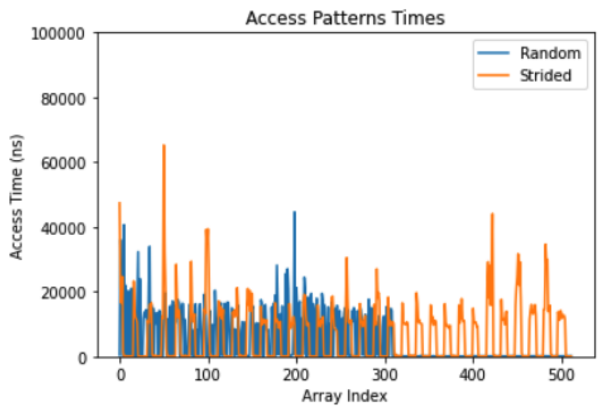


Figure 8: Random and Strided Array Access Times.

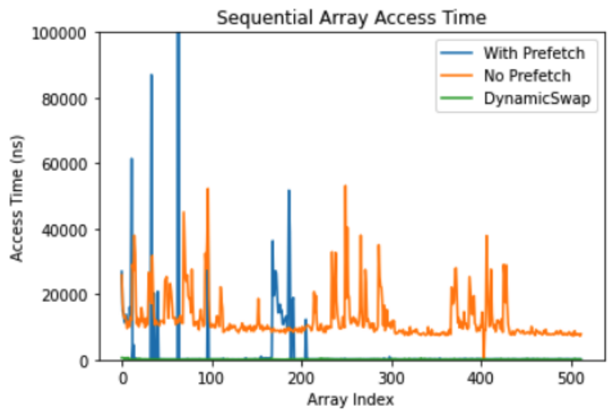


Figure 9: Sequential Access Times with and without Prefetching.

Tests were performed to evaluate some of the basic performance of AIFM. First, as seen in Figure 7, the access times by iteration are shown for a basic, sequential access pattern. There is a large spike in latency towards the beginning, as well as subsequent, smaller spikes at intermittent points. The second chart, as shown in Figure 8, shows the latencies for random and strided access patterns. Finally, Figure 9 compares the latencies with AIFM prefetching, without any prefetching, and the DynamicSwap block for sequential accesses. The prefetcher seems to incur a higher "startup" cost for pulling in remote memory, but has lower intermittent latency spikes compared to AIFM without prefetching. However, with DynamicSwap enabled, there is no initial high cost as seen with the default prefetcher, and subsequent latencies are low. By avoiding both the high initial penalty as well as moderate intermediate penalties, DynamicSwap allowed for lower and more uniform latency patterns. One caveat is that this evaluation was performed on a simple sequential array access, a task at which DynamicSwap should perform well. More benchmarking needs to be done on more real-world workloads to evaluate DynamicSwap's effectiveness.

6 Future Work

One limitation of the DynamicSwap system is that removing references from the shared data structure is extremely slow, on the order of polynomial time complexity. Each time a removal must occur, a search across the entire data structure must be done. AIFM does not notify changes in status, such as the hotness or coldness of pointers, so using the existing AIFM implementation, it is impossible to implement a priority queue or similar optimization. In the future, work could be done to improve the existing implementation of AIFM to provide notification upon pointer modification - this would allow downstream systems like DynamicSwap to operate faster.

Another limitation stemming from AIFM's underlying implementation is that AIFM implements no garbage collection for remote memory. There is similarly no communication of changes in status of remote memory back to the local node, which can cause a multitude of errors when the local node expects a pointer to be able to be brought in when it simply no longer exists. Implementing garbage collection and communication of status changes back to the local node would improve the reliability of the AIFM system.

Further, the initial goal of DynamicSwap was to work irrespective of (orthogonally to) AIFM and any user code base. The original design idea was to have a DynamicSwap thread running concurrently to both AIFM and user processes, acting when necessary. However, AIFM has no support for multithreading, and most of DynamicSwap's, AIFM's, and the user process' threads are sequential, significantly slowing down the execution of DynamicSwap. In the future, more work could be done to add multithreading support for AIFM such that other add-ons such as DynamicSwap could function in parallel. In order for DynamicSwap to be fully evaluated, better cooperation between AIFM would need to be engineered.

7 Conclusion

Our project aimed to tackle the large round-trip network latency spikes from accessing remote memory through speculative fetching. DynamicSwap was implemented orthogonally to AIFM, allowing DynamicSwap to be used on other far-memory systems with, ideally, little development effort.

Our evaluations demonstrate that DynamicSwap is capable of reducing latency spikes in some, but not all, access patterns. While DynamicSwap struggled with speculative fetching for random access, patterns such as sequential and strided access achieved observable improvements. So, while DynamicSwap should not be used for *all* problems, there are many systems where the access patterns would benefit from DynamicSwap. Further development of DynamicSwap will be necessary to allow it to run in all contexts, especially if removed from the AIFM system.

References

[1] Zhenyuan Ruan et al. "AIFM: high-performance, application-integrated far memory". In: *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. OSDI'20. USA: USENIX Association, 2020. ISBN: 978-1-939133-19-9.

[2] Juncheng Gu et al. "Efficient Memory Disaggregation with Infiniswap". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 649–667. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu.

[3] Gwendolyn Voskuilen, Arun F. Rodrigues, and Simon D. Hammond. "Analyzing allocation behavior for multi-level memory". In: *Proceedings of the Second International Symposium on Memory Systems*. MEMSYS '16. Alexandria, VA, USA: Association for Computing Machinery, 2016, pp. 204–207. ISBN: 9781450343053. DOI: 10.1145/2989081.2989116. URL: https://doi.org/10.1145/2989081.2989116.

[4] Amy Ousterhout et al. "Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads". In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 361–378. ISBN: 978-1-931971-49-2. URL: https://www.usenix.org/conference/nsdi19/presentation/ousterhout.

[5] Emmanuel Amaro et al. "Can far memory improve job throughput?" In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387522. URL: https://doi.org/10.1145/3342195.3387522.