Tolerate It if You Cannot *Reduce* It: Handling Latency in Tiered Memory

Musa Unal Vishal Gupta Yueyang Pan Yujie Ren Sanidhya Kashyap
EPFL

Abstract

Current memory tiering systems mitigate asymmetric latency through page migration between tiers. While this approach effectively hides latency, it overlooks scenarios where latency could potentially be tolerated. We propose that an efficient system should integrate both latency reduction (migration) and latency tolerance strategies. Our research demonstrates the effectiveness of prefetchers in tolerating latency within such systems, highlighting their importance in the design of high-performance memory tiering solutions.

Keywords

Memory tiering, prefetching, compiler, runtime, CXL

1 Introduction

Memory has emerged as one of the most critical resource in modern data centers as applications demands continue to grow [41]. Recent studies have highlighted this challenge, particularly as data-intensive workloads become more prevalent [19, 31, 36, 41]. The emergence of interconnect technologies, such as Compute Express Link (CXL) [17], presents a promising solution for expanding DRAM capacity beyond traditional limits. CXL provides low latency, cache-coherent load/store access to memory over the PCIe interface, effectively circumventing the constraints of the physical DIMM slots. Studies have shown that combining conventional DRAM with CXL-attached memory creates multiple memory tiers with varying performance characteristics in terms of latency and bandwidth [33, 40].

Achieving close-to-DRAM application performance across these memory tiers requires efficient bandwidth and latency utilization. This challenge mirrors research from the 1990s and early 2000s that focused on managing latency. Earlier works differentiated between **reducing** latency, primarily through cache hierarchies and locality optimizations [7, 12, 21, 34], and **tolerating** latency, often using data prefetching and multithreading [7, 12, 21, 34].

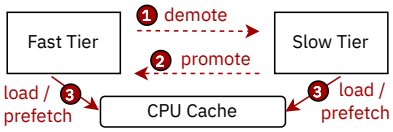

Figure 1: Current tiering systems are trying to optimize load time (3) by demoting (1) and promoting (2) between the nodes. However not each load equally created, there is an opportunity to (3) prefetch into the cache before accessing to hide the access latency of slower tier.

Today's tiered memory systems primarily reduce latency via dynamically migrating data between tiers based on access frequency, often referred to as *hotness*. Figure 1 illustrates this process, which entails promoting frequently accessed (hot) data to fast tier while demoting infrequently accessed (cold) data to the slower tier. Current tiered memory systems primarily focus on either the mechanisms for migrating data [33, 46] or policies to detect hotness [3, 30, 33, 37, 42]. While migrating pages effectively clusters them by their hotness, this approach fundamentally relies on exploiting the locality properties inherent in the application's data accesses, also known as its working set [18].

An alternative approach for managing latency is tolerating it by exploiting CPU cache prefetching. Prefetchers attempt to predict future memory access patterns and preemptively load data from memory to CPU cache, thereby reducing effective access latencies by minimizing cache misses. While prefetching becomes particularly crucial in tiered systems to hide slow-tier access latencies, existing hardware prefetchers [10, 11, 24] and software solutions [4, 28] are designed for homogeneous memory architectures. We find that directly applying these conventional prefetching approaches to tiered memory systems can actually degrade application performance by approximately 19% in real-world scenarios on CXL-attached memory, as analyzed shortly.

In this paper, we argue that a comprehensive approach to latency management in tiered memory systems should include both latency reduction and tolerance techniques. As a part of this proposal, we are designing a compiler and runtime to enable data prefetching in tiered memory systems. The compiler is responsible for detecting the prefetchability of memory regions, while the runtime is responsible for

HOTOS 25, May 14–16, 2025, Banff, AB, Canada

ACM ISBN 979-8-4007-1475-7/25/05.
https://doi.org/10.1145/3713082.3730376

enforcing multi-dimensional policies (discussed in §3.4) for efficient memory tiering.

2 Prefetching on Tiered Memory

Current prefetching techniques, both hardware and software, focus on DRAM [38]. To understand the impact of prefetchers in a tiered memory system, we evaluate existing data-intensive applications using both DRAM and CXL-attached memory (CXL 2.0 SMART CXA-4F1W equipped with 4x64GB DDR5-5600 RDIMMs [1]). In our system, local DRAM achieves an idle-latency of 112 ns with a bandwidth of 271 GB/s, while CXL has an idle-latency of 237 ns and a bandwidth of 46 GB/s. From now on we will refer to CXL-attached memory as CXL.

2.1 Hardware Prefetching

Hardware prefetchers aim to reduce cache misses by proactively predicting and preloading data from DRAM into the processor cache. A key limitation, imposed by constrained CPU die size, is that hardware implementations can only effectively manage a restricted set of predictable access patterns. As a result, significant research effort has been dedicated to developing and analyzing specific hardware prefetching heuristics, such as stride [8, 16, 24, 26, 29], spatial access [11, 14, 15], and temporal access [9, 10].

Impact of hardware prefetching on CXL memory. To understand the impact of prefetching, we evaluate the GAP benchmark suite [13]. We run tests allocating application memory entirely on DRAM or CXL. We observe that when we allocate application memory on DRAM, 18 out of 20 applications benefit from hardware prefetchers, achieving an average speedup of 1.47×, with a maximum of 3.43×. When memory is allocated on CXL, 17 out of 20 applications benefit, with an average speedup of 1.26×, reaching up to 2.27×. Another key observation is that DRAM did not exhibit significant performance degradation with prefetching enabled, whereas CXL experienced a 19% performance degradation.

To further understand the impact of CPU core count, we select two applications: betweenness centrality (*BC*) with the *twitter* graph (using Brandes algorithm) and XSBench, which represents Monte Carlo neutron transport algorithm. Figure 2 shows that prefetching is more effective when using CXL memory compared to DRAM at low number of cores. This is because CXL has a higher latency, which presents a greater opportunity for prefetchers to hide access delays. However, on increasing the number of cores, we observe that the effectiveness of prefetching on CXL starts to decrease and eventually begin to hurt the performance for both applications.

The primary reason hardware prefetching in CXL yields limited benefits is due to its limited bandwidth (5.9× lower

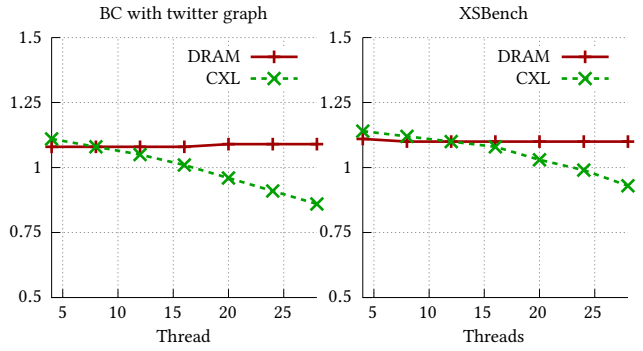

Figure 2: When the number of threads increases the prefetching effectiveness of CXL decreases while DRAM being constant, and for high number of threads prefetching hurts the performance on CXL.

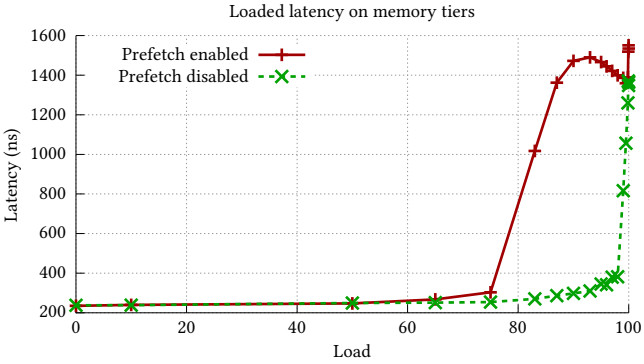

Figure 3: Under high load, prefetching causes latency to increase dramatically at lower loads compared to when prefetching is disabled, resulting in up to 6.3× higher latency.

than DRAM). Excessive prefetching requests causes contention on the CXL link (due to memory expander), leading to performance degradation. To quantify the impact of bandwidth contention on latency due to hardware prefetching, we run the *MLC* [44] loaded latency test. It measures latency under different load scenarios where higher loads corresponds to shorter delays between requests (starting from 10K cycles to 0 cycles). As shown in Figure 3, unnecessary prefetching can increase latency by an order of magnitude, reaching 1,500 ns for CXL. In contrast, DRAM latency increases only by 2× due to its higher bandwidth.

2.2 Software Prefetching

Software prefetching offers greater flexibility and potential for application-specific optimization compared to hardware prefetching, as it involves inserting prefetch instructions directly into the target program's code. Implementations typically rely on compilers [4, 6, 28] or profile-guided optimization (PGO)[28, 39, 47]. Moreover, recent works have focused on understanding how hardware and software prefetching

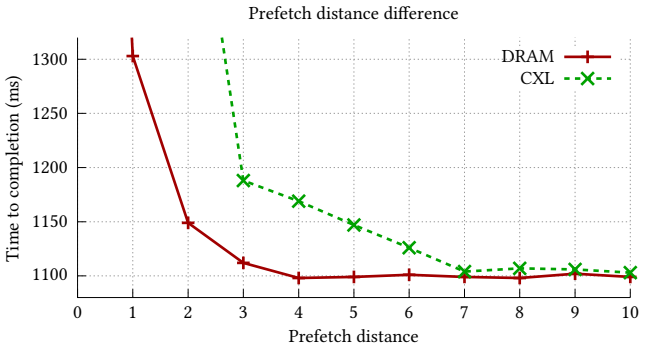

Figure 4: DRAM and CXL have different optimal prefetch distances. For example, in the scan microbenchmark, DRAM performs best with a prefetch distance of 4, while CXL requires a longer distance of 7 due to its higher latency.

can be combined effectively, especially for applications running at data center scale [27]. Software prefetching is effective when prefetch instructions are issued at the correct time, known as *timeliness*. Timeliness is controlled through *prefetch distance*, which specifies how far in advance data should be prefetched [4]. Setting the correct distance is critical: fetching too early risks data eviction from the cache before use, while fetching too late fails to hide latency and negates the potential benefit [28].

Impact of prefetch distance on CXL memory. In a tiered memory system with low-latency DRAM and high-latency CXL, determining the optimal prefetch distance is challenging due to dynamic data migration between tiers. To evaluate the impact of prefetch distance, we developed a microbenchmark that executes a scan operation on a large array of data with a constant stride. In each scan, we systematically varied the software prefetch distance (k) from 1 to 16 elements ahead. Moreover, we disabled hardware prefetching during this experiment. We observe a distinct optimal prefetch distance for each tier: $k = 4$ for DRAM, while $k = 7$ for CXL. This difference confirms that CXL's higher latency requires prefetches to be initiated further in advance (via a larger prefetch distance) to effectively hide the longer access time.

Summary. We summarize the following insights according to our experiments.

- Data prefetchers can effectively tolerate CXL latency.
- Hardware prefetchers can degrade performance under bandwidth contention.
- Software prefetchers must use tiering-aware prefetch distances to be effective.

3 Proposal

We propose Linden: a system to reduce data access latency by efficiently managing data placement between memory tiers and tolerating the latency by exploiting data prefetchers. Figure 5 shows the main components of Linden: compiler,

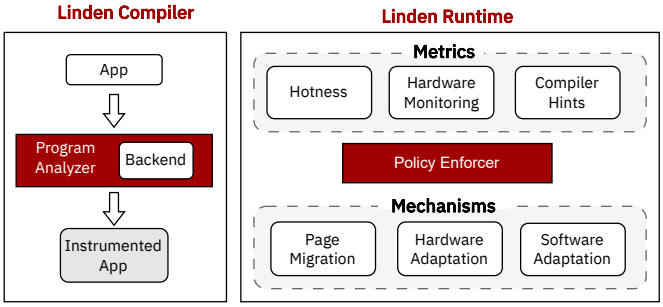

Figure 5: Linden consists of a compiler and runtime. Compiler takes a program and finds the prefetchable regions in the program. Runtime is responsible for detecting hotness, hardware monitoring, and compiler hints to enforce different policies. Runtime able to migrate pages between the tiers, enable/disable hardware prefetchers and change the software behavior in terms prefetchability.

and a runtime. Linden compiler detects *prefetchable regions*, where prefetching is beneficial in tolerating the data access latency. Linden runtime first combines information about prefetchable regions with page hotness and hardware contention metrics. It later enforces policies to reduce and tolerate the data access latency by migrating pages between tiers, enabling/disabling hardware prefetchers, and modifying software prefetches. The policy enforcement has two objectives:

- *Reduce*: Minimize data access latency by increasing the locality.
- *Tolerate*: Hide the data access latency by prefetching.

We start by defining the prefetchable region (§3.1) and how Linden compiler detects them (§3.2). Later, we describe the Linden runtime's mechanism and policies (§3.3).

3.1 Prefetchable Regions

A prefetchable region is a set of memory pages, potentially non-contiguous, that are amenable to hardware or software prefetching. The goal of a prefetchable region is to hide latency for the loads within that region and to minimize it by issuing accurate and timely prefetches. Each region has the following properties:

- **Memory region** (*region*): List of pages that shares a similar access pattern for a particular thread. The size of this list is configurable.
- **Prefetchability ratio** (*ratio*): This metric quantifies the potential percentage of load access latency within a region that can be hidden by effective prefetching. Regions with more predictable access patterns yield higher prefetchability ratios. The value ranges from 0 (no latency hidden) to 1 (all latency hidden).
- **Type of access pattern** (*pattern*): How are loads accessed within a region? Simple patterns like next line and stride can be prefetched either by hardware

Thread	*region*	*ratio*	*type*	*freshness*	*target*
T_0	A	0.9	Sequential	200	l_1, d_1
T_1	A	0.05	None	1	None
T_2	B	0.7	Chase	50	l_2, d_2

Table 1: An example of a *prefetchability table*.

prefetchers or software prefetchers. However, irregular and temporal access patterns require software prefetch instructions.

- **Target** (*target*): Which prefetch instruction is targeting which load? This includes the prefetch address and the prefetch distance. This helps in issuing timely prefetches when memory access latency changes.
- **Freshness** (*freshness*): Time passed after the last $ratio$ update on a particular region. It starts from 0 and increments until a user-defined threshold. This helps to give different policy/scheduling decisions to runtime.

Table 1 outlines the data tracked by the Linden runtime for each prefetchable region. Consider two memory regions, A and B, and three worker threads: T_0, T_1, and T_2. T_0 sequentially scans A, resulting in a high *prefetchability ratio* due to its linear access pattern. In contrast, T_1 acts as a transaction handler, accessing A randomly, which leads to a low *prefetchability ratio* because of random accesses. Meanwhile, T_2 performs pointer chasing within B, which can be effectively prefetched using software prefetching. By linking access patterns to memory regions, we can better assess the advantages of prefetching in mitigating latency.

3.2 Detecting Prefetchable Regions

Depending on whether the application source code is available, prefetchable regions can be detected either by the Linden compiler or the Linden runtime.

Compiler support. The Linden compiler takes an application as an input and produces a modified binary. This binary is instrumented so that, at runtime, it can inform the associated runtime system about identified prefetchable regions. To identify these regions, the compiler utilizes pluggable backends capable of analyzing data access patterns; these backends can leverage existing techniques like specialized compiler passes [4, 28] or machine learning approaches [22]. For example, upon detecting a predictable access pattern (e.g., within a loop) via a backend, Linden instruments the program by inserting specific helper functions. At runtime, these helper functions execute and communicate key information, such as the accessing thread ID, the memory region identifier (*region*), and the detected access pattern type (*pattern*), to the runtime system. This runtime information is collected to populate the prefetchability table (Table 1).

Hardware sampling. When application source code is unavailable for compile-time instrumentation, the Linden runtime can use hardware sampling. It uses specific hardware performance counters to assess the effectiveness of the built-in hardware prefetchers for accessed memory pages [25]. Based on this sampling data, the runtime identifies regions that appear to be adequately prefetched by the hardware. In addition, just-in-time compilation [2] offer another runtime alternative where software prefetches can be added to the code on the fly.

```
metrics = [hotness, hw_info, sw_info]
# Reduce
reduce(state, metrics.hotness, metrics.hw_info)
# Check for any contention
if (metrics.hw_info < contention_threshold):
   tolerate(state, metrics.sw_info)
   # Adapt software after tolerating if needed
   adapt_sw(state, RELAXED)
else:
   # Adapt hardware to avoid perf degradation
   adapt_hw(state)
   adapt_sw(state, CONSERVATIVE)
```

Listing 1: Algorithm for policy enforcement in the Linden runtime.

3.3 Linden Runtime

Linden runtime consists of two parts: (1) monitoring utility to collect information about page hotness, compiler hints for prefetchability, and hardware statistics to check memory contention; (2) mechanisms like page migration, adapting hardware prefetcher or changing software prefetches. They enforce policies to *reduce* and *tolerate* the data access latency.

Metric monitoring.

We now discuss various metrics that the *metric monitor* collects and processes.

- **Hotness**. Prior works have explored various forms of policies and mechanisms to detect hotness, including LRU- and histogram-based approaches [30, 33, 37]. Similar to prior works, Linden uses Intel PEBS [30, 37] or page table scanning [33, 46] for maintaining hotness metrics.
- **Compiler hints**. Linden runtime gets information from the instrumented application code about the prefetchability of the memory regions, and eventually it generates *prefetchability table*.
- **Hardware monitoring** is important for both *reduce* and *tolerate* approaches. As discussed in §2.1, application performance degrades under high memory system load due to factors like ineffective prefetching, bandwidth limitations, or memory tier congestion, the latter also being highlighted by recent work [42]. Thus, to effectively analyze and react to these conditions, Linden monitors three categories of hardware metrics: (1) Prefetcher-related hardware counters, such as L2 hardware prefetches (`L2_RQSTS.ALL_HWPF`) and useless prefetches (`L2_LINES_OUT.USELESS_HWPF`), (2) Bandwidth

and latency measurements for each memory tier, (3) Tier-specific congestion indicators. These measurements rely on sampling-based techniques with overhead directly proportional to sampling frequency. To minimize this overhead, we plan to adjust the sampling frequency dynamically, similar to Memtis [30].

Mechanisms. Linden's runtime mechanisms include:

- **Page migration**. Similar to hotness detection, the runtime uses existing approaches for page migration. For instance, it promotes and demotes pages in the background similar to Memtis [30] to avoid any performance degradation.
- **Hardware adaptation**. The runtime controls hardware prefetchers (stream, stride) through MSR registers [25]. It provides fine-grained control to enable/disable hardware prefetching for specific cores to avoid any performance degradation due to excessive prefetching requests.
- **Software adaptation**. The runtime follows two approaches to modify software prefetch instructions inserted by the Linden compiler: (1) It modifies the *prefetch distance* by moving the prefetch instruction earlier or later in the instruction stream. (2) It adds new prefetch instructions by using JIT compilers [2].

3.4 Policy Enforcer

The runtime uses *reduce* and *tolerate* methods (Listing 1) to enforce the policies. The *reduce* method improves locality by piggybacking existing memory tiering policies. Then, depending on the hardware state (e.g. contention on memory), it can either increase or decrease the amount of prefetches. When contention is low, it uses the *tolerate* method to maximize latency hiding opportunities by increasing the aggressiveness of software prefetching. Under high contention, it disables hardware prefetching and makes software prefetching more conservative. Linden can support multiple policies using the *reduce* and *tolerate* functions. We now look at three policies in detail:

1. Improve performance (*tolerate if you cannot reduce*). Previous works on memory tiering focus on *reduce* and migrating data between tiers to increase locality. One of the major challenges in tiered memory is managing a scenario where the hot data resides in both fast and slow tiers. Previous works suggest that in such scenarios it is better to disable tiering [45]. However, there is still an opportunity to use *tolerate* to increase system's performance. The main insight for this policy is that if a region is hot and prefetchable, we might migrate that region into the slower tier since prefetchers may hide the latency to access this region.

We test this policy with a microbenchmark, where we allocate 10 GB of memory: 5 GB from DRAM and 5 GB from CXL (1:1), and we break down memory into 20 regions. The microbenchmark consists of a worker thread that issues loads to different memory regions with different access patterns (either sequential or random). We reran the experiment by changing the allocation site (DRAM or CXL) of the regions that are sequentially accessed. We observe that in such scenarios, if the region is hot and prefetchable, we gain up to 7% improvement by moving it into the slow tier, even for only 20% sequential regions.

2. Control throttling (*do not tolerate every time*). Prefetching via *tolerate* should happen optimally as aggressive prefetching can hurt the application's performance [27] and the effect is even worse with CXL (Figure 2). Linden runtime offers different policies to throttle prefetch requests depending on the bandwidth characteristics of the tiering system. A policy like Limoncello [27], which disables hardware prefetching under high memory utilization, leads to suboptimal performance in the tiering system. Specifically, contention might happen on the slow tier, but the threads accessing the fast tier will suffer as prefetching is also disabled for them. Linden leverages its *prefetchability table* to track which cores access which memory tiers. It can then selectively disable hardware prefetching only for cores accessing the slow tier, while implementing conservative software prefetching as needed.

3. Control timeliness (*tolerate when there is some time*). Ensuring prefetch timeliness is critical, as fetching too early risks cache eviction, while fetching too late fails to hide latency completely [4, 28]. Tiered memory systems complicate this timeliness (§2.2) because migrating pages change the effective memory latency, thus altering the optimal prefetch distance required for different tiers. Linden addresses this dynamically: when a page containing a software prefetch instruction migrates to a different tier, the runtime adjusts the associated prefetch distance to match the optimal value for the new tier. For instance, the prefetch distance would be updated to 7 if the data is demoted into the CXL and vice-versa, as shown in Figure 4. Linden can use JIT [2] to change the prefetch distance based on the latency of the slower tier. Moreover, this operation occurs reactively based on relevant page migration events, not at fixed time intervals.

4 Discussion

Linden tradeoffs. The compiler must balance several tradeoffs when identifying prefetchable memory regions:

1) Instrumentation vs. size. Entries in the prefetchability table are generated for each memory region. Smaller regions add additional performance and memory overhead due to excessive instrumentation and extra entries in the table.

2) Accuracy vs. size. Too big or small region can be less predictable to prefetchers, resulting in either inaccurate or untimely prefetches. We plan to evaluate this with a wide range of benchmarks.

3) Coverage vs. compilation time. A machine learning based approach can provide higher coverage and in turn, the prefetchability ratio, compared to simple stride or irregular access pattern but at the cost of spending more time in finding them.

Exploring the synergy between *tolerate* and *reduce*. The compiler has the capability to detect both prefetchable and non-prefetchable regions. Thus, inferring prefetchable regions can be considered as an opportunity for page migration. In particular, the prefetchable region can capture the set of pages, which can also be used for minimizing tracking information [35]. On the other hand, for applications with non-prefetchable regions [6], we can extend Linden to run *scavenger jobs* (background jobs such as batch jobs that can harvest the idle cycles) similar to MSH [32]. These jobs execute while the application code is waiting for data to be fetched from memory. The application can switch to scavenger jobs for non-prefetchable regions through compiler-inserted coroutines [23, 32], which can be integrated into the compiler. Applying this technique can help Linden to *tolerate* latencies even for non-prefetchable regions.

Extensions for other memory systems. We can extend Linden for systems with multiple memory tiers that consist of DRAM, CXL, and RDMA-based memory. Prefetching showed promising results for RDMA-based memory systems [5, 20]. Similar to RCMP [43], Linden's *reduce* and *tolerate* can be extended to support these systems. The complexity lies in designing policies that can work across multiple tiers with varying definitions of prefetching and memory characteristics. As a future work, memory pooling can be explored; however, these systems require more support as multiple applications can be accessing the memory at the same time with different access pattern resulting in different prefetchability regions. Combining this information along with congestion across multiple machines will be interesting to explore.

Future landscape of CXL. PCIe5 supports 4GB/s per lane while PCIe6 will support 8GB/s, which will increase the CXL's memory bandwidth up to 128GB/s (assuming x16 lanes), and interleaving between multiple CXL devices might eliminate the bandwidth problem. However, the latency problem will still be there. Effectively prefetching the regions from CXL can hide the latency difference between CXL and DRAM. Moreover, for higher bandwidths prefetching can be done in a more aggressive way.

5 Conclusion

For an efficient tiered memory system we need to consider both latency reduction and latency toleration. With this proposal, we showed how data prefetching can be used as a latency toleration technique in the tiered memory, what are the current challanges and how can we solve them.

6 Acknowledgment

We thank the anonymous reviewers, and our shepherd, Marcos K. Aguilera, for their helpful feedback.

References

[1] CXL memory expansion. https://www.smartm.com/ [Accessed: 2025-01-16].

[2] A.-R. Adl-Tabatabai, R. L. Hudson, M. J. Serrano, and S. Subramoney. Prefetch injection based on hardware monitoring and object metadata. *ACM SIGPLAN Notices*, 39(6):267–276, 2004.

[3] N. Agarwal and T. F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 631–644, 2017.

[4] S. Ainsworth and T. M. Jones. Software prefetching for indirect memory accesses. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 305–317. IEEE, 2017.

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

[6] G. Ayers, H. Litz, C. Kozyrakis, and P. Ranganathan. Classifying memory access patterns for prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 513–526, 2020.

[7] A.-H. A. Badawy, A. Aggarwal, D. Yeung, and C.-W. Tseng. Evaluating the impact of memory system performance on software prefetching and locality optimizations. In *Proceedings of the 15th international conference on Supercomputing*, pages 486–500, 2001.

[8] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Supercomputing '91:Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pages 176–186, 1991. doi: 10.1145/125826.125932.

[9] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad. An efficient temporal data prefetcher for l1 caches. *IEEE Computer Architecture Letters*, 16(2):99–102, 2017. doi: 10.1109/LCA.2017.2654347.

[10] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad. Domino temporal data prefetcher. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 131–142, 2018. doi: 10.1109/HPCA.2018.00021.

[11] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad. Bingo spatial data prefetcher. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 399–411, 2019. doi: 10.1109/HPCA.2019.00053.

[12] A. Bakshi, J.-L. Gaudiot, W.-Y. Lin, M. Makhija, V. K. Prasanna, W. Ro, and C. Shin. Memory latency: to tolerate or to reduce. In *12th Symposium on Computer Architecture and High Performance Computing. Invited paper*, 2000.

[13] S. Beamer, K. Asanović, and D. Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[14] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Stealth prefetching. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, page 274–282, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595934510. doi: 10.1145/1168857.1168892. URL

https://doi.org/10.1145/1168857.1168892.

[15] C. Chen, S.-H. Yang, B. Falsafi, and A. Moshovos. Accurate and complexity-effective spatial pattern prediction. In *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, pages 276–287, 2004. doi: 10.1109/HPCA.2004.10010.

[16] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE transactions on computers*, 44 (5):609–623, 1995.

[17] T. C. Consortium. Compute Express Link Specification. https://www.computeexpresslink.org/ [Accessed: 2025-01-16].

[18] P. J. Denning. Working set analytics. *ACM Computing Surveys (CSUR)*, 53(6):1–36, 2021.

[19] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao. Who Limits the Resource Efficiency of My Datacenter: An Analysis of Alibaba Datacenter Traces. In *2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS)*, pages 1–10, 2019. doi: 10.1145/3326285.3329074.

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

[21] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber. Comparative evaluation of latency reducing and tolerating techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 254–263, 1991.

[22] M. Hashemi, K. Swersky, J. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan. Learning memory access patterns. In *International Conference on Machine Learning*, pages 1919–1928. PMLR, 2018.

[23] Y. He, J. Lu, and T. Wang. Corobase: coroutine-oriented main-memory database engine. *arXiv preprint arXiv:2010.15981*, 2020.

[24] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham. Effective stream-based and execution-based data prefetching. In *Proceedings of the 18th Annual International Conference on Supercomputing*, ICS '04, page 1–11, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138393. doi: 10.1145/1006209.1006211. URL https://doi.org/10.1145/1006209.1006211.

[25] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel, 2025. URL https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. Volumes 1–3.

[26] Y. Ishii, M. Inaba, and K. Hiraki. Access map pattern matching for data cache prefetch. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, page 499–500, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605584980. doi: 10.1145/1542275.1542349. URL https://doi.org/10.1145/1542275.1542349.

[27] A. Jain, H. Lin, C. Villavieja, B. Kasikci, C. Kennelly, M. Hashemi, and P. Ranganathan. Limoncello: Prefetchers for scale. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 577–590, 2024.

[28] S. Jamilan, T. A. Khan, G. Ayers, B. Kasikci, and H. Litz. Apt-get: Profile-guided timely software prefetching. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 747–764, 2022.

[29] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, pages 364–373, 1990. doi: 10.1109/ISCA.1990.134547.

[30] T. Lee, S. K. Monga, C. Min, and Y. I. Eom. Memtis: Efficient memory tiering with dynamic page classification and page size determination. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 17–34, 2023.

[31] C. Lu, K. Ye, G. Xu, C.-Z. Xu, and T. Bai. Imbalance in the cloud: An analysis on alibaba cluster trace. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2884–2892. IEEE, 2017.

[32] Z. Luo, S. Son, S. Ratnasamy, and S. Shenker. Harvesting memory-bound {CPU} stall cycles in software with {MSH}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 57–75, 2024.

[33] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. 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.

[34] T. C. Mowry. Tolerating latency in multiprocessors through compiler-inserted prefetching. *ACM Transactions on Computer Systems (TOCS)*, 16(1):55–92, 1998.

[35] S. Park, Y. Lee, and H. Y. Yeom. Profiling dynamic data access patterns with controlled overhead and quality. In *Proceedings of the 20th International Middleware Conference Industrial Track*, Middleware '19, page 1–7, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450370417. doi: 10.1145/3366626.3368125. URL https://doi.org/10.1145/3366626.3368125.

[36] R. J. Pfitscher, M. A. Pillon, and R. R. Obelheiro. Customer-oriented diagnosis of memory provisioning for iaas clouds. *SIGOPS Oper. Syst. Rev.*, 48(1):2–10, may 2014. ISSN 0163-5980. doi: 10.1145/2626401.2626403. URL https://doi.org/10.1145/2626401.2626403.

[37] A. Raybuck, T. Stamler, W. Zhang, M. Erez, and S. 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.

[38] B. Saglam, N. Ho, C. Falquez, A. Portero, F. Schätzle, E. Suarez, and D. Pleiter. Data prefetching on processors with heterogeneous memory. In *Proceedings of the International Symposium on Memory Systems*, pages 45–60, 2024.

[39] H. Shen, K. Pszeniczny, R. Lavaee, S. Kumar, S. Tallam, and X. D. Li. Propeller: A profile guided, relinking optimizer for warehouse-scale applications. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 617–631, 2023.

[40] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, C. Song, J. Huang, H. Ji, S. Agarwal, J. Lou, I. Jeong, et al. Demystifying cxl memory with genuine cxl-ready systems and devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 105–121, 2023.

[41] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the tenth european conference on computer systems*, pages 1–17, 2015.

[42] M. Vuppalapati and R. Agarwal. Tiered memory management: Access latency is the key! In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 79–94, 2024.

[43] Z. Wang, Y. Guo, K. Lu, J. Wan, D. Wang, T. Yao, and H. Wu. Rcmp: Reconstructing rdma-based memory disaggregation via cxl. *ACM Transactions on Architecture and Code Optimization*, 21(1):1–26, 2024.

[44] T. Willhalm, S. Sakthivelu, S. Srikanthan, V. Viswanathan, and K. Kumar. Intel® Memory Latency Checker v3.11. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html, 2021.

[45] L. Xiang, Z. Lin, W. Deng, H. Lu, J. Rao, Y. Yuan, and R. Wang. Nomad: Non-Exclusive memory tiering via transactional page migration. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 19–35, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/xiang.

[46] Z. Yan, D. Lustig, D. Nellans, and A. 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.

[47] Y. Zhang, N. Sobotka, S. Park, S. Jamilan, T. A. Khan, B. Kasikci, G. A. Pokam, H. Litz, and J. Devietti. Rpg2: Robust profile-guided runtime prefetch generation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 999–1013, 2024.