Exploring multi-level cache prefetching for fabric attached memory

Chandrahas Tirumalasetty
chandrahas996@tamu.edu
Dept. of Electrical & Computer Engineering
Texas A&M University
College Station, TX, USA

Narasimha Reddy
reddy@tamu.edu
Dept. of Electrical & Computer Engineering
Texas A&M University
College Station, TX, USA

Abstract

Memory disaggregation in data centers has been approaching practicality, owing to the maturity of interconnect standards like Compute Express Link (CXL) [3]. CXL presents a hardware centric approach for multiple compute nodes to pool memory capacities from a shared Fabric Attached Memory (FAM) node, on a per need basis. Using FAM for memory provisioning can potentially mitigate resource underutilization and yield in cost savings, but can cost the application it's performance due to relatively longer access latency.

Modern processors attempt to hide memory access latency by employing sophisticated cache prefetchers. While resourceful, current cache prefetching techniques can be further optimized, in light of the long access latency of CXL FAM. To that end, we consider multi-level cache prefetcher that adds additional layer of prefetching at Last Level Cache (LLC). Our multi-level prefetching scheme increases the fraction of requests that hit in LLC, potentially decreasing the sensitivity of workload to FAM latency. We implemented our multi-level cache prefetcher using SST simulation components [27], and evaluated it with workloads from standard benchmarks suites in single and multi-node system configuration. Our evaluation reveals that, comparing to using only per-core prefetcher, multi-level prefetcher resulted in performance improvement of 2-7% , with LLC hit fraction increasing by 13%.

Keywords

Memory, Compute Express Link (CXL), DRAM

1 Introduction

Workloads are evolving rapidly. Proliferate use of techniques like Machine Learning (ML) and Natural Language Processing (NLP) in applications at scale (ex: large language models(LLM) [14, 33, 34]), is warranting memory organization, that is large in capacity, and offers high performance. More importantly, memory capacity needs of data center workloads are growing at steady pace. For instance, parameter count of Transformer models is increasing by 410 × for every 2 years, and consequently necessitating increasing memory capacity needed to serve these models [15].

Increased data needs of such emergent workloads, is prompting cloud providers/integrators to increase the memory capacity in their systems, for every generation. Plateaued $/GB price of DDR memory [2] means that memory provisioning costs are a significant contributor to Total Operation Cost (TCO) of datacenters; Memory contributes about 37% -50% to TCO of today's server fleet [1, 25].

Cloud servers support applications with diverse memory requirements and use cases. Large DRAM capacities while warranted by a set of applications, are not being fully utilized by the rest, resulting in *Memory Underutilization*. Analysis from production clusters at Google and Alibaba reveals that 45%-60% of allocated memory to jobs is not utilized [29]. Microsoft reports that about 25% of machines across the Azure server fleet, have their compute capacity fully used by virtual machine instances (VM) while having untouched memory, leading in *stranding* of memory resources [23]. Sub optimal usage of memory resources cannot be tolerated given the scale and infrastructure costs involved in today's datacenters.

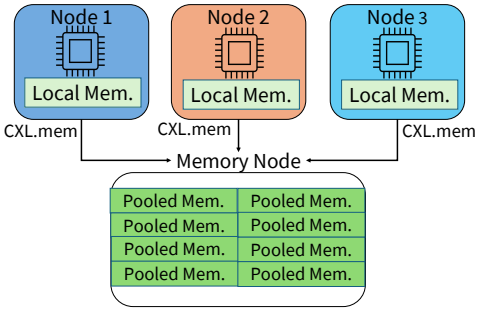

Figure 1: Logical view of multiple nodes pooling capacities from memory node using CXL.mem protocol

Memory Disaggregation presents an alternative approach to provision large memory capacities while mitigating the underutilization problem. With disaggregation, applications allocate memory from a central resource on a per-need basis, freeing the memory from being tied up statically at node-level. Prior systems research have explored the potential of resource disaggregation in datacenters [17, 22, 29].

Compute Express Link (CXL) enables multiple nodes to share a common pool of memory that can be accessed through load/store instructions without the need to rewrite application software. Fig. 1 shows compute nodes pooling capacities from a memory node using CXL.mem protocol. We refer to memory attached to CXL fabric as Fabric Attached Memory(FAM). Accessing FAM from pooling nodes involves traversing off-node fabric and perhaps switches, resulting in access latency that is as high as few 100's of nanoseconds. Thus, relying on FAM to suffice application's memory needs although resource efficient, comes at a penalty in the form of performance degradation.

Cache prefetching techniques are prevalent among current processors to hide main memory access latency. Prefetcher typically reads ahead memory resident data into on-chip CPU caches to capitalize on low latency & high bandwidth characterstics of caches. Prefetchers track application's historic memory access patterns to predict addresses of future memory requests. Although instrumental in hiding the memory access latency, prefetching can be further refined to better manage the long latency of CXL FAM pools.

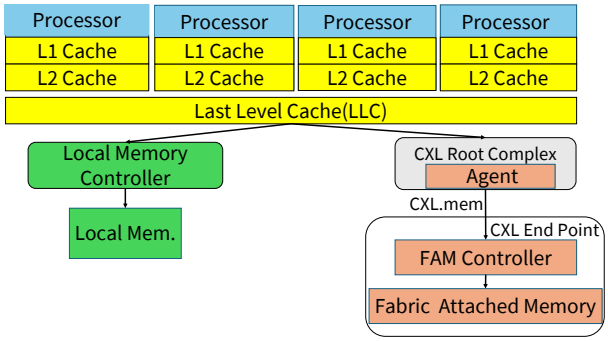

Figure 2: CXL & Fabric Attached Memory(FAM) Architecture

In this paper, We explore the design of a multi-level cache prefetcher that augments the CPU with a prefetcher at last level cache (LLC) to work in conjunction with per-core prefetcher. We evaluate the impact of this multi-level prefetcher on overall performance using workloads from standard benchmark suites, running in single and multi-node system configurations, and with different CXL fabric latencies. Our analysis reveals that multi-level prefetcher has resulted in IPC to be improved by 2.2%-6.9% and LLC hit fraction to be improved by 13%.

2 Background

2.1 CXL enabled memory pooling

Compute Express Link (CXL) [3] is a cache-coherent interconnect standard for CPU to communicate with peripheral devices like accelerators and memory expanders. CXL is physically compatible with PCIe offering 3 kinds of protocols- CXL.cache, CXL.mem, CXL.io. Devices that connect to the host processor using CXL can use either one or more of these protocols, depending on use case, and the operation model.

CXL protocol classifies devices into 3 types. Type-1 devices like Network Interface Controllers (NICs) that maintain a local cache hierarchy but do not have any attached memory use only CXL.cache protocol. Type-2 devices like GPU, FPGA comprise both local cache hierarchy and attached memory, so they use both CXL.cache and CXL.mem protocols. Type-3 devices like memory expanders do not have a local cache hierarchy, but have device attached memory, hence use CXL.mem protocol only. Our discussion in this paper is based on systems that use type-3 devices for memory pooling, through CXL.mem protocol, as shown in Fig. **??**. We call the memory attached to the processor using CXL as Fabric Attached Memory(FAM). Fig. 2 shows the architectural components that enable memory pooling within each of the compute and FAM nodes. CXL root complex of the compute node comprises of an agent that implements all communication and data transfers with the the end point in compliance with the CXL.mem protocol. In our system CXL end point comprises of a FAM device and a FAM Controller. FAM Controller serves as a translation layer to convert CXL.mem commands into requests that can be understood by the FAM device(eg: DDR commands).

As illustrated, LLC load misses and writebacks requests are handled either by the local memory controller or CXL root complex based on the physical addresss. The address decoding is handled by Host managed device memory (HDM) decoders. During the device enumeration phase, HDM decoders are programmed for every CXL.mem device and their contribution to flat address space of the processor. Device identification and enumeration is handled in CXL kernel driver [4].

CXL 3.0 supports multi-level switching, implying that FAM access now includes traversing network switches which further increases latency as high as few 100's of nano seconds. depending on switch topology. Thus, in context of our discussion in this paper, we consider 3 CXL fabric latency configurations - 70, 140, 210 nano seconds.

2.2 Prefetching & Signature Path Prefetcher (SPP)

Data prefetching techniques to hide access latency across different levels of memory hierarchy, are well studied in the literature [6, 7, 10, 11, 26]. Cache prefetchers typically use learning based approaches to predict addresses of future memory accesses, fetch them ahead in time into the CPU caches. Most common learning features include address delta correlation, program counter (PC) of instruction resulting in cache misses, and access history. Recent work [9, 30] has applied sophisticated mechanisms like neural networks, reinforcement learning to cache prefetcher.

2.2.1 Signature Path Prefetcher (SPP). In this paper, we use SPP [18] as a cache prefetcher. SPP uses signatures as a compact representation of address deltas [1] across workload's memory accesses. Architecturally, SPP comprises of 2 tables- Signature Table and Pattern Table.

The Signature Table is indexed by the physical page address, and each entry in this table, stores the last cache miss address(within the same physical page), and current signature. Pattern table maps the signature obtained from the signature table to address delta. Each entry in the pattern table has the following fields.

(1) Signature - Obtained from the signature table. Serves as an index to this table.
(2) Signature weight - Counts the number of times the corresponding signature has been accessed since the creation of entry.
(3) delta, weight [4] - Address delta that comprise the signature and their corresponding access count. 4 ordered pairs.

Learning process of SPP is as follows. On a cache access, the physical page address of the access is used to index into the signature table. Based on the output of signature table, we can calculate the delta and update the signature as per formulae shown below.

$$delta = (Miss\ Address_{current} - Miss\ Address_{previous})$$
$$signature = (signature << 4) \oplus delta$$

The generated signature is used to index into the pattern table, which gives us the ordered pairs of delta and the corresponding confidence (ratio of weights). Furthermore, speculative signature can be formed by combining the current iteration signature (just generated) and the obtained delta, according to the above formulation. Thus, through this recursive calculation of signature and

[1] Address deltas are difference between consecutive accessed memory address. Cache Prefetchers typically predict the delta for future accesses based on the current address.

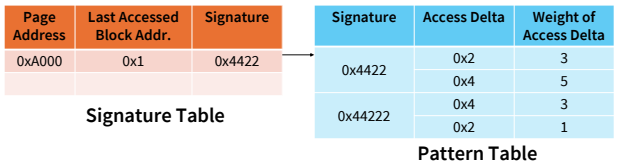

Figure 3: Signature and Pattern tables of SPP

indexing the pattern table, future access delta's can be obtained desired number of times or till there is no valid entries in the pattern table for the indexing signature.

For learning access patterns, SPP updates the weight corresponding to the current delta (output from signature table). Signature table is also updated with current block address and current signature. Fig. 3 shows the tables of SPP. Additionally, SPP maintains global history table that bootstraps the learning of access history, when the data access stream moves from one page to another.

SPP is typically used as L2 cache. For every prefetch request generated, SPP predicts the confidence value. Prefetch cache blocks with high confidence are placed in L2, on the other hand ones with lower confidence are placed in Last Level Cache (LLC). Confidence based prefetch block placement achieves two goals, first it minimizes cache pollution due to low confidence prefetch blocks in smaller capacity caches like at L2. Second, it takes complete advantage of large capacity of LLC.

2.3 Related work

Previous work has considered establishing a low latency access path to FAM, by using a portion of local memory as a hardware managed cache for FAM [21, 32]. Such memory system approaches are orthogonal to work we present this paper.

Intel's Flat memory is a hardware based memory tiering technique [36]. With flat memory, CPU is presented with a flat address space divided among DRAM and FAM in 1:1 ratio. Based on application's access pattern, specialized hardware moves the hot cache lines into DRAM and cold cache lines to FAM, updating the corresponding metadata. Similarly, Software based memory tiering approaches like Pond by Microsoft [23] uses machine learning based techniques to manage memory tiers, predicting the optimal memory pages to be allocated from FAM for a given virtual machine (VM). Cache prefetchers work with memory tiers irrespective of whether implemented in hardware or software.

Recent research have explored the possibility of expanding available memory bandwidth using CXL connected FAM [16, 24, 31, 35]. Cache prefetcher enhancements that we present in this paper are independent of CXL enabled memory bandwidth expansion.

3 Multi-level cache prefetching

Data prefetching mechanisms are typically implemented on a per-core basis (at L1/L2 caches). Additional layer of prefetching can be added at last level cache (LLC) to further improve overall performance. Such multi-level prefetching schemes offers the following advantages.

(1) Promotes prefetching deeper into future access stream, potentially covering demand requests that would otherwise result in cache misses.
(2) Capitalize on relatively large capacity of LLC to aggressively prefetch data from main memory.
(3) Increases the hit fraction for requests at LLC, potentially minimizing performance degradation of workloads using CXL FAM pools which have longer access latencies.
(4) Better support higher level caches and their respective prefetchers by reading ahead both demand and prefetch blocks that might be needed in the future.

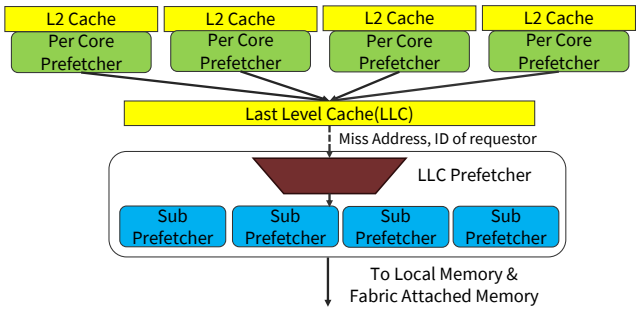

Figure 4: Multi-level prefecher design - at L2 (per-core) & LLC

Fig. 4 shows the design of our multi-level prefetching scheme with an added LLC prefetcher. LLC prefetcher is logically organized as an array of sub-prefetchers, each for tracking access patterns for a single core. This design takes the core-ID and access address of the request as inputs. The core-ID is used to select a sub-prefetcher from the array, and the access address is passed as an input to the selected sub-prefetcher. For a given input, the sub-prefetcher updates its internal tracking state and subsequently generate addresses for prefetch requests.

Our LLC prefetcher solates access patterns on a per-core basis, enabling applications (processes) running on different cores to be tracked without interference. Capacity and logic overhead requirements for our design would scale linearly with number of cores in the CPU.

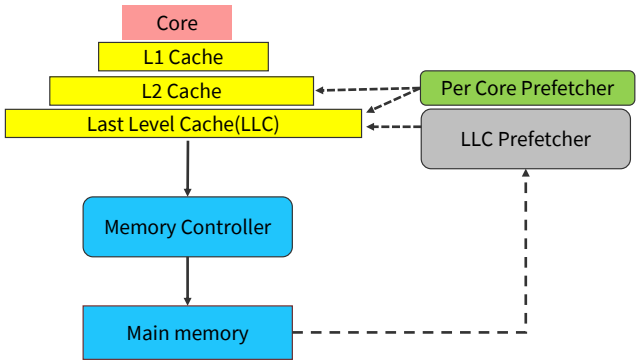

Figure 5: Prefetch Block placement with multi-level prefetching at L2 and LLC

Prefetch block placement with L2 and LLC prefetchers is shown in Fig. 5. L2 prefetcher places block either into L2 or LLC, based on the calculated confidence, while the LLC prefetcher places all of the blocks it brings into LLC. In this work, we made a design choice to train the LLC prefetcher only on addresses of the requests(both prefetch and demands) that miss in LLC.

Alternative design considerations like, training LLC prefetcher only on demands and/or training on accesses irrespective of hit status, are possible. The intuition behind our design choice is that the LLC prefetcher needs to cover requests that are not covered by prefetch blocks placed in LLC by per-core (L2) prefetcher. Hence it is logical that we train only on address that miss in LLC, instead of all access addresses. Heuristically, this design choice can reduce the number of redundant requests generated by the LLC prefetcher.

Prefetch request generation & placement with L2 and LLC prefetchers is as follows. Both L2 demand misses and prefetch requests follow the same flow

(1) L2 prefetcher is activated for every cache access to produce prefetch requests. Addresses of the generated prefetch requests are checked for redundancy in L2 cache and its MSHR. Requests with blocks that are present in either of these structures are dropped and rest proceed to LLC.
(2) At LLC, we repeat the same redundancy checks. If requested prefetch block is absent in LLC and its MSHR, the request is sent to main memory controller, along with forwarding the address & core-id of prefetch request to train the LLC prefetcher.
(3) In the LLC prefetcher, we use the core-ID pick one sub-prefetcher from the array. Selected sub-prefetcher trains on the address that missed in LLC to further generate prefetch requests.
(4) Similar to L2 prefetch requests, LLC prefetch requests are checked for redundancy before being sent to the memory controller.
(5) While the prefetch requests are in flight in an inclusive cache hierarchy, Each of the L2 prefetch blocks that are to be placed in L2 (high-confidence) will have an entry created in both L2 & LLC MSHR, whereas L2 prefetch blocks that are to be placed in LLC (low-confidence) will have an entry only in LLC MSHR. LLC prefetch requests that are in flight will have their entries created in LLC MSHR.

In our implementation, we use SPP as prefetcher at L2, and as sub-prefetcher(in the array) at LLC. Other designs can be implemented. For both instances of SPP, we configure 75% as high confidence threshold and 25% as minimum confidence threshold for generated prefetch requests. Prefetch requests (both L2 and LLC) with confidence less than 25% are dropped immediately after generation. Data from L2 prefetch requests with confidence greater than 25% and less than 75% are placed in LLC, while those prefetch requests that are greater than 75% are brought to L2. LLC prefetch requests that are not dropped are placed in LLC.

4 Evaluation

4.1 Methodology

We use simulation components built using SST [27] to evaluate our multi-level cache prefetcher for CXL FAM. Ariel - pin based processor simulator was used to model the multi-core CPU. Ramulator [19] was used to model memory modules in local memory and FAM pool. Opal [20] was used to emulate the role of FAM-aware memory allocator of the operating system. Opal allows for configuration of workload's memory to be divided among local DRAM and pooled FAM. CXL network is modeled using a flit-based network model with configurable delay and bandwidth.

Processor	8 OOO cores clock: 3.3 GHz, 6 issue/cycle
L1 cache	32 KB, 4 ways 4 cycle access latency
L2 cache	256 kB, 8 ways 12 cycle access latency
L2 Cache Prefetcher	Signature Path Prefetcher(SPP) [18]
LLC	32 MB, 16 ways 30 cycle access latency
Local memory	DDR4-3200 2 channels, 2 ranks
Nodes	1-4
CXL Network	256B flit-size, Min-packet size: 28B Bandwidth: 128 GB/s Min. Latency: 70-210 ns
Per-Node prefetch queue size	256
Pooled FAM	DDR4-2400 2 channels, 2 ranks

Table 1: Simulated system configuration

We evaluated 19 memory bound workloads from standard benchmark suites like SPEC [13], PARSEC [12], GAP [8], Splash3 [28], and NPB [5]. Considering the simulation speeds, we simulate a scaled down configuration of a representative server system, that runs regions of interest (ROI) within each benchmark. We expect that observed performance characteristics through our evaluation to scale to a system with realistic configuration. Plus, Our simulation methodology yielded in deterministic and consistent performance metrics across different simulation runs. Simulated system configuration is shown in Table. 1. Workloads are configured have their memory footprint atleast 4-8 times the size of LLC, in order to ensure that they are memory bound.

We evaluate both single and multi node system configurations, we simulate up to 4 nodes sharing 2 DDR4 memory channels at FAM. In multi-node systems, we ran copies of the same application on different nodes, as well as different applications on different nodes (mixes). We expect applications in the mix to have diverse memory access patterns, and subsequently each mix to behave differently when sharing available bandwidth at FAM. We evaluated 8 such workload mixes on a 4-node system.

We parametrize memory allocations from FAM in the form of *FAM/DRAM allocation ratio*. FAM/DRAM allocation ratio of x indicates that memory pages are allocated b/w local DRAM and pooled FAM in ratio of 1:x. For example, FAM/DRAM allocation ratio of 0.5 indicates that 66% of application's pages are allocated in DRAM and 33% in FAM. Similarly, FAM/DRAM allocation ratio of 1, indicates workload's pages are equally divided between local DRAM and FAM. In this paper, we consider FAM/DRAM allocations of 0.125, 0.25, 0.5, 1.

We use the following figures of merit in the discussion through the rest of this section

(1) *IPC gain* - Amount of increase in Instructions per Cycle (IPC) due to multi-level prefetching compared to using prefetcher at only L2 cache.
(2) *Relative LLC hit fraction* - Ratio of hit fraction for loads at LLC with multi-level prefetcher to hit fraction for accesses at LLC with L2 prefetcher.
(3) *Relative off chip prefetch* - Ratio of prefetch requests that are issued to main memory with mulit-level prefetcher to that of prefetch requests issued to main memory with per-core L2 prefetcher. Indicates the increase in amount of data being prefetched from main memory into on-chip to CPU caches.
(4) *Relative LLC miss latency* - Ratio of miss latency for LLC misses (main memory accesses) with multi-level prefetcher to miss latency for LLC misses with L2 prefetcher. Quantifies the potential downside of aggressive/multi-level prefetching.

4.2 Evaluation

4.2.1 Single and Multi-node systems. We evaluate a single node system running an application with portion of its memory pages allocated from FAM. Fig. 6a shows IPC gain due to addition of LLC prefetcher, across different FAM/DRAM allocation ratio for different benchmarks. Average IPC gain for these workloads across allocation ratio is 2.2%(0.125), 2.8%(0.25), 2.7%(0.5) and 3.6%(1) respectively. As we can see, average IPC gain increases with increase in memory allocation fraction for FAM. Benchmarks like bfs, LU, 654.roms_s, mg show linear increasing characteristic of performance gain with allocation fraction, while IPC gain for 603.bwaves_s, 619.lbm_s is not monotonically dependent on FAM/DRAM allocation ratio. 607.cactuBSSN_s interestingly have its performance gain slightly reduced with increase in FAM/DRAM allocation ratio, likely due to increase of off-chip LLC prefetch requests with increase in FAM/DRAM allocation ratio, while LLC hit fraction increases marginally by 1%.

Fig. 6b shows the relative LLC hit fraction for these workloads in a single node system, across different FAM/DRAM allocation ratios. Multi-level prefetcher results in LLC hit fraction to be increased by 13%. Canneal has the highest increase in requests that hit in LLC, by about 70-80%, while 607.cactuBSSN_s has the lowest increment only by 1-2%. Canneal benchmark despite having highest increment in LLC hit fraction, only has modest performance improvement indicating that it is insensitive to LLC miss latency.

Fig. 6c show relative off chip prefetch request analysis for benchmarks in single node system across different allocation ratios. Multi-level prefetcher has resulted 17-20% more prefetch requests to be issued for main memory. XSBench has relatively more LLC prefetch requests to be issued despite having no performance gain or LLC hit fraction improvement, likely showing inefficient use of memory bandwidth. Fig. 6d shows the relative LLC miss latency analysis for this system configuration. Addition of LLC prefetcher has resulted in increment of miss latency by 4.9%(0.125), 4.3%(0.25), 3.1%(0.5), 2.5%(1). Latency increment due to aggressive prefetching becomes less significant with increase in FAM/DRAM allocation fraction.

Similar to the performance analysis presented for a single node, we evaluate applications running on multiple nodes (4), with a fraction of their memory footprint in FAM. Fig. 7a shows the IPC gain for multi-node system for different applications across different allocation ratios. LLC prefetcher addition has resulted in IPC improvement of 2%(0.125), 3%(0.25), 3%(0.5) and 2%(1) across the configured FAM/DRAM allocation ratios. Performance trends are similar to that of a single node system configuration. The corresponding relative LLC hit fraction analysis in shown in Fig. 7b. LLC hit fraction improved by an average of 13% across different FAM/DRAM allocation fractions.

Fig. 7c shows the relative off chip requests for evaluated benchmarks across different allocation fractions in multi-node configuration. On average, each node issued 19% more prefetch requests to main memory due to addition of a prefetcher at LLC. Relative miss latency analysis presented in Fig. 7d indicates that miss latency has increased by about 3.9% - 4.7%, tad higher than single node system configuration.

4.2.2 Multi-node workload mixes. 8 shows our workload mix evaluation. We further characterize the impact of multi-level prefetching with workload mixes running in a multi-node system. Fig. 8a shows IPC gains for workload mixes across different FAM/DRAM allocation ratios. For these mixes, additional prefetching resulted in IPC improvement of 2.5%, 2.0%, 2.1%, 3.3% for allocation fraction of 0.125, 0.25, 0.5 and 1 respectively. Mixes 2 & 4 have about 5-9% IPC improvement due to multi-level prefetching, while mix1 has slight performance decrement of about 1-3% for some allocation ratios.

Fig. 8b shows the corresponding relative LLC hit fraction for these workload mixes. Due to our multi-level cache prefetcher, hit fraction at LLC has increased by an average of 14-15%. Mix7 has highest increment in LLC hit fraction, of about 23%, while mix8 has the least increment of about 8-10%.

Fig. 8c shows relative off chip prefetch request analysis for these mixes. On average, 16-18.7% more prefetch requests are issued to main memory, due to multi-level cache prefetching. Mix2 and 3 have the highest increase in memory bound prefetch requests of about 23-40%. Fig. 8d shows the LLC relative miss latency for these multi-node mixes has increased by an average of 9-10%. Mix8 has the highest increment in LLC miss latency, by about 12.1-22.5%. All of the measured metrics across different workload mixes are related to FAM/DRAM allocation ratio non-deterministically.

4.2.3 CXL Fabric Latency. In this experiment, we evaluate the multi-level prefetcher in a single node configuration with different CXL fabric latencies. We consider 70 ns, 140 ns, 210 ns as fabric latencies, Fig. 9 shows our analysis. Fig. 9a shows IPC gain due to our multi-level prefetcher across different CXL fabric latencies for evaluated workloads, Across different fabric latencies, addition of LLC prefetcher has resulted in performance gain of 3.8%(70), 5.4%(150), 6.9%(210). Most of the evaluated workloads (excluding 607.cactuBSSN_s, dedup, facesim, bfs, sssp, and XSBench), have seen larger IPC gain with multi-level prefetching for longer latency configuration, emphasizing the importance of additional layers of cache prefetching for CXL memory pools. Fig. 9b shows the corresponding relative LLC hit fraction analysis for evaluated workloads. LLC hit fraction has consistently improved by about 14% with our multi-level prefetching across different CXL fabric latencies.

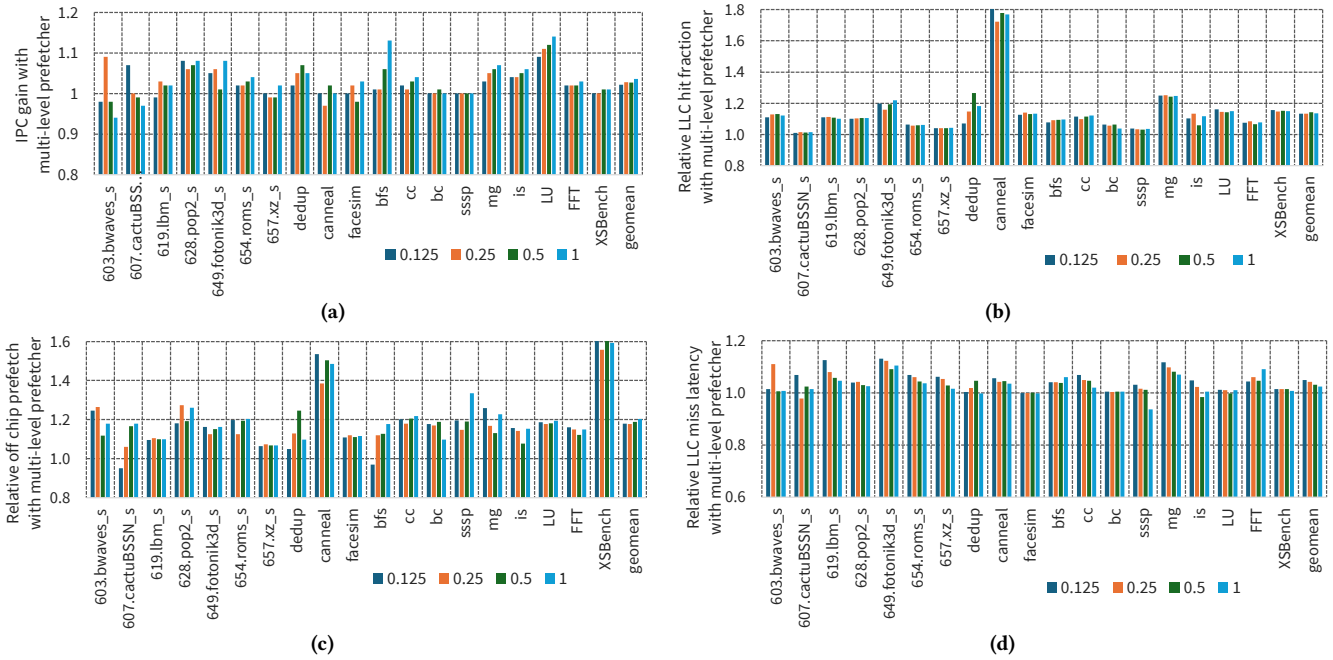

Figure 6: Evaluation of workloads running in single node system configuration across different FAM/DRAM allocation fractions

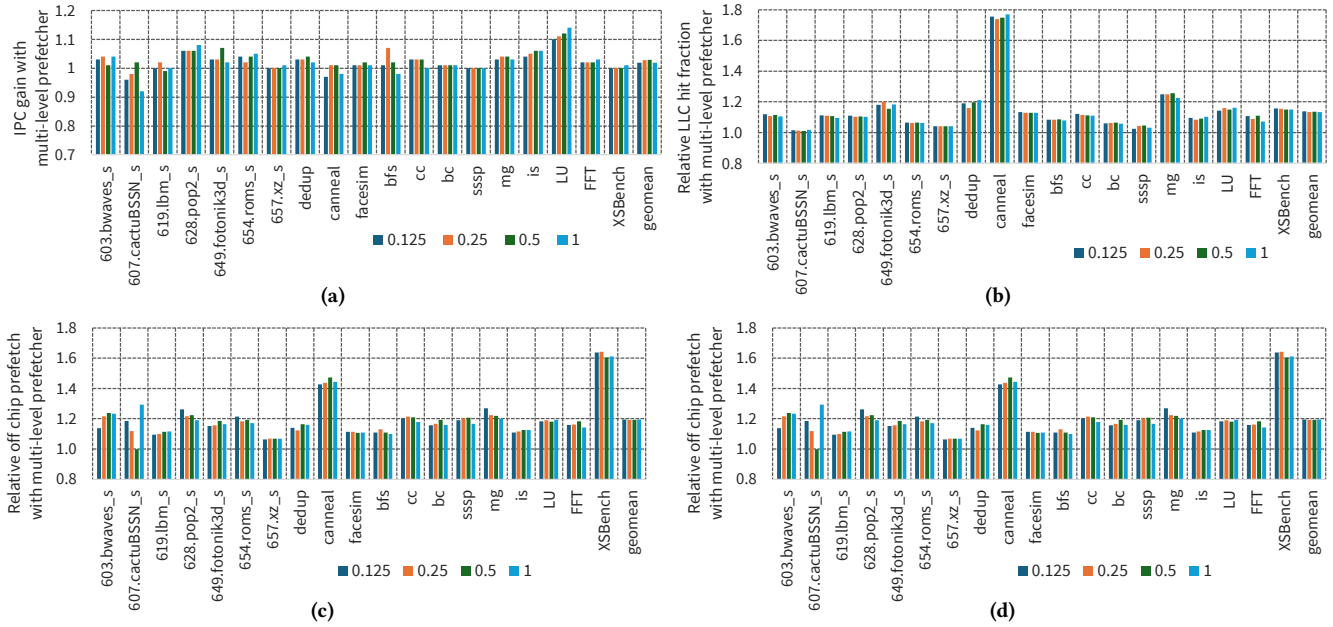

Figure 7: Evaluation of workloads running in a 4 node system configuration across different FAM/DRAM allocation fractions

5 Conclusion

In this paper, we explored the design of a multi-level cache prefetcher design that issues prefetch requests from both L2 cache and LLC. Additional layer of prefetching in our design is aimed at reducing the sensitivity of application to long access latency of CXL enabled FAM pools.

Our evaluation of multi-level cache prefetcher with different workloads and system configurations suggest that it can result in, IPC improvement upto 6.9%, and LLC hit fraction improvement by 13%. Moreover, multi-level prefetching resulted in 16-20% more prefetch requests to be issued to memory, resulting in LLC miss latency to be increased by 5-10%. Overall performance improvement despite increase in LLC miss latency suggests that prefetching from multiple cache levels can be a viable architectural approach to extract more performance from a memory system that consists of FAM.

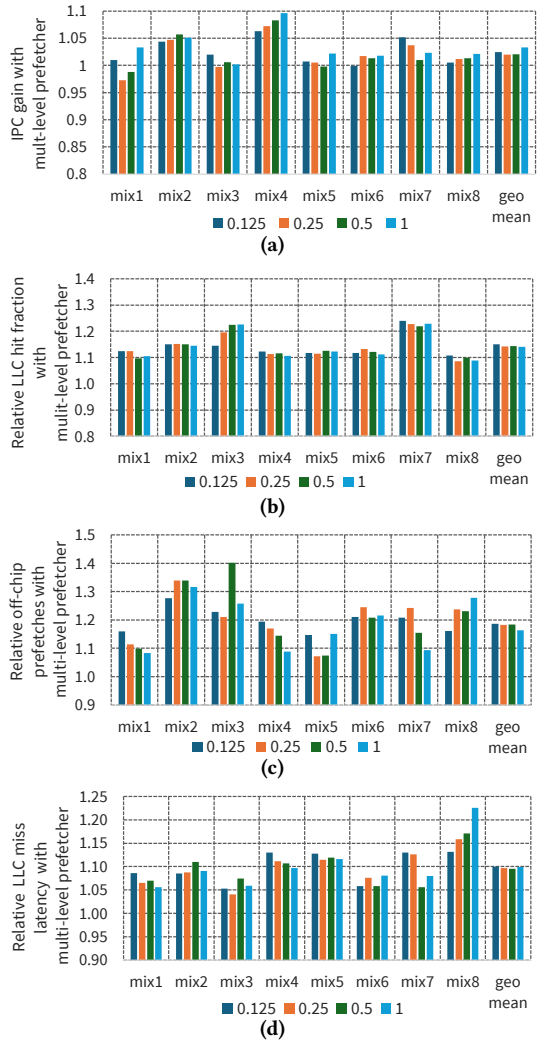

Figure 8: Workload mixes evaluation in a 4 node system configuration, with different FAM/DRAM allocation fractions

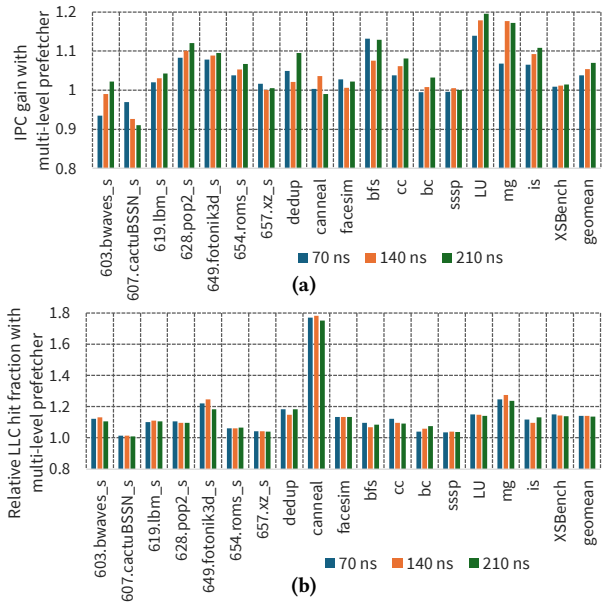

Figure 9: Performance analysis of multi-level prefetcher across different CXL fabric latencies

Likewise, our work in this paper motivates future research into LLC prefetchers with intricate design to track and predict main memory accesses.

References

[1] [n. d.]. CXL AND GEN-Z iron out a coherent interconnerct stratergy. https://nextplatform.com/2020/04/03/cxl-and-gen-z-iron-out-a-coherent-interconnect-strategy. Accessed May 2024.

[2] [n. d.]. WHAT DO WE DO WHEN COMPUTE AND MEMORY STOP GETTING CHEAPER? https://www.nextplatform.com/2023/01/18/what-do-we-do-when-compute-and-memory-stop-getting-cheaper/. Accessed May 2024.

[3] 2025. *Compute Express Link protocol specification.* https://computeexpresslink.org/cxl-specification/ Accessed April-2025.

[4] 2025. *CXL driver in linux kernel.* https://www.kernel.org/doc/html/v6.1/driver-api/cxl/memory-devices.html Accessed June-2025.

[5] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. 1995. *The NAS parallel benchmarks 2.0.* Technical Report. Technical Report NAS-95-020, NASA Ames Research Center.

[6] Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2018. Domino Temporal Data Prefetcher. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA).* 131–142. doi:10.1109/HPCA.2018.00021

[7] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Bingo Spatial Data Prefetcher. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA).* 399–411. doi:10.1109/HPCA.2019.00053

[8] Scott Beamer, Krste Asanović, and David Patterson. 2017. The GAP Benchmark Suite. arXiv:1508.03619 [cs.DC]

[9] Rahul Bera, Konstantinos Kanellopoulos, Anant Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. 2021. Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21).* Association for Computing Machinery, New York, NY, USA, 1121–1137. doi:10.1145/3466752.3480114

[10] Rahul Bera, Anant V. Nori, Onur Mutlu, and Sreenivas Subramoney. 2019. DSPatch: Dual Spatial Pattern Prefetcher. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO-52).* Association for Computing Machinery, New York, NY, USA, 531–544. doi:10.1145/3352460.3358325

[11] Eshan Bhatia, Gino Chacon, Seth Pugsley, Elvira Teran, Paul V. Gratz, and Daniel A. Jiménez. 2019. Perceptron-based prefetch filtering. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19).* Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3307650.3322207

[12] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: characterization and architectural implications. In *PACT '08.* ACM, Toronto, Ontario, Canada, 72–81.

[13] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering* (Berlin, Germany) *(ICPE '18).* Association for Computing Machinery, New York, NY, USA, 41–42. doi:10.1145/3185768.3185771

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL] https://arxiv.org/abs/1810.04805

[15] Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, and Kurt Keutzer. 2024. AI and Memory Wall. https://arxiv.org/abs/2403.14123. arXiv:2403.14123 [cs.LG]

[16] Wentao Huang, Mo Sha, Mian Lu, Yuqiang Chen, Bingsheng He, and Kian-Lee Tan. [n. d.]. Bandwidth Expansion via CXL: A Pathway to Accelerating In-Memory Analytical Processing. *Proceedings of the VLDB Endowment. ISSN* 2150 ([n. d.]), 8097.

[17] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. 2017. HeteroOS — OS design for heterogeneous memory management in datacenter. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture*

(ISCA). 521–534. doi:10.1145/3079856.3080245

[18] Jinchun Kim, Seth H. Pugsley, Paul V. Gratz, A.L. Narasimha Reddy, Chris Wilkerson, and Zeshan Chishti. 2016. Path confidence based lookahead prefetching. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. doi:10.1109/MICRO.2016.7783763

[19] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters* 15, 1 (2016), 45–49. doi:10.1109/LCA.2015.2414456

[20] Vamsee Kommareddy, Clayton Hughes, Simon David Hammond, and Amro Awad. 2018. Opal: A Centralized Memory Manager for Investigating Disaggregated Memory Systems. (8 2018). doi:10.2172/1467164

[21] Vamsee Reddy Kommareddy, Jagadish Kotra, Clayton Hughes, Simon David Hammond, and Amro Awad. 2021. PreFAM: Understanding the Impact of Prefetching in Fabric-Attached Memory Architectures. In *Proceedings of the International Symposium on Memory Systems* (Washington, DC, USA) *(MEMSYS '20)*. Association for Computing Machinery, New York, NY, USA, 323–334. doi:10.1145/3422575.3422804

[22] 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 Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 317–330. doi:10.1145/3297858.3304053

[23] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 574–587. doi:10.1145/3575693.3578835

[24] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S. Berger, Marie Nguyen, Xun Jian, Sam H. Noh, and Huaicheng Li. 2025. Systematic CXL Memory Characterization and Performance Analysis at Scale. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 1203–1217. doi:10.1145/3676641.3715987

[25] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 742–755. doi:10.1145/3582016.3582063

[26] Agustín Navarro-Torres, Biswabandan Panda, Jesús Alastruey-Benedé, Pablo Ibáñez, Víctor Viñals Yúfera, and Alberto Ros. 2023. Berti: An Accurate Local-Delta Data Prefetcher. In *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture* (Chicago, Illinois, USA) *(MICRO '22)*. IEEE Press, 975–991. doi:10.1109/MICRO56248.2022.00072

[27] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis, and B. Jacob. 2011. The Structural Simulation Toolkit. *SIGMETRICS Perform. Eval. Rev.* 38, 4 (mar 2011), 37–42. doi:10.1145/1964218.1964225

[28] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. 2016. Splash-3: A properly synchronized benchmark suite for contemporary research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 101–111. doi:10.1109/ISPASS.2016.7482078

[29] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: a disseminated, distributed OS for hardware resource disaggregation. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 69–87.

[30] Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin. 2021. A hierarchical neural model of data prefetching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 861–873. doi:10.1145/3445814.3446752

[31] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 105–121. doi:10.1145/3613424.3614256

[32] Chandrahas Tirumalasetty and Narasimha Reddy Annapareddy. 2024. Contention aware DRAM caching for CXL-enabled pooled memory. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '24)*. Association for Computing Machinery, New York, NY, USA, 157–171. doi:10.1145/3695794.3695808

[33] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *arXiv:2302.13971* (2023).

[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[35] Midhul Vuppalapati and Rachit Agarwal. 2024. Tiered Memory Management: Access Latency is the Key!. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 79–94. doi:10.1145/3694715.3695968

[36] Yuhong Zhong, Daniel S. Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. 2024. Managing memory tiers with CXL in virtualized environments. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 3, 20 pages.