Contention Aware DRAM Cache prefetching for CXL-enabled pooled 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

Hardware based memory pooling enabled by interconnect standards like CXL have been gaining popularity amongst cloud providers and system integrators. While pooling memory resources has cost benefits, it comes at a penalty of increased memory access latency. This paper proposes a system architecture for prefetching sub-page blocks from FAM into DRAM cache for improving the data access latency and application performance. Utility of DRAM caching is subject to bandwidth availability and contention at FAM node. To that end, we propose two contention aware enhancements to our DRAM cache prefetch mechanism. First, we consider the potential for providing additional functionality at the FAM node through weighted fair queuing of demand and prefetch requests. Second, we adapt prefetch rate at the compute-node based on observed FAM access latencies. Proposed system is evaluated in single node and multi-node configurations with applications from SPEC, PARSEC, Splash and GAP benchmark suites. Our evaluation suggests DRAM cache prefetching can improve IPC up to 7% and both proposed optimizations can further increment IPC by 7-10%, for a total IPC gain of 10-13%.

Keywords

CXL, Fabric Attached Memory(FAM), Memory pooling, DRAM cache, Prefetching, Disaggregation, Contention

1 Introduction

Compute workloads are evolving rapidly. Heavy use of Machine Learning(ML) driven workloads are placing stringent demands on system design across all platforms. ML techniques like large language models(LLM)[22, 39, 69, 74, 75], video processing [32, 37, 76] and others, rely on large amounts of data for training and sometimes for retrieval. Data center workloads, sometimes referencing Terabytes of data, with execution spread out on multiple compute nodes, exert great demands on memory systems [25, 30]. The trend towards server virtualization [11, 77], is furthering this pressure on computer memory systems.

In recent years, the performance gap between DRAM and disk has grown so large to lead system designers to eschew using storage to extend DRAM entirely in favor of over-provisioning DRAM [12, 26, 51, 57, 62, 80]. Many different techniques have been proposed to reduce the cost of data movement and page fault handling penalties. These range from employing a larger DRAM memory and running applications entirely in memory [57, 63], prefetching data blocks [24, 33, 40], HW/SW co-designed page fault handling mechanisms [47, 73], and employing remote memories [8, 29, 54].

Datacenter servers host applications with diverse memory requirements. Provisioned larger DRAM capacities can potentially ameliorate the performance needs of some applications, while not being utilized by the rest. Hence, *Memory Underutilization* is rampant in today's datacenters. Trace analysis from production clusters at Google and Alibaba revealed that 45%-60% of allocated memory to jobs is not utilized [65]. Untouched memory for virtual machine instances (VM) in Azure servers, average about 25%, while compute capacity is nearly fully used [50].

Furthermore, $/GB price of DDR memory has plateaued for last few generations [6]. As a result, cost to provision larger memory capacities in today's severs is steeply increasing with every generation. Memory contributes 37% -50% of total cost of ownership(TCO) of server fleet [3, 55]. Memory underutilization incurs substantial costs at the scale of today's datacenters.

Memory Disaggregation enables applications to avail memory from a central resource on an ad-hoc basis, freeing the memory from being tied up statically at node-level. Modern data center servers have been exploring the potential of disaggregated memories to provide a less expensive means of furnishing memory [38, 46, 65]. The disaggregated approaches have taken two parallel paths: (1) First approach accesses memory at another node as remote/far memory through RDMA and Operating System (OS) based paging mechanisms [8, 29, 54]. (2) The second approach uses CXL fabric to provide load/store access to shared common pool of memory, across multiple compute nodes [1, 7, 20, 81]. Such memory organization is referred as Fabric Attached Memory(FAM). With either approach, it is expected that the memory is used more efficiently across different workloads with divergent memory needs.

These architectural paths result in additional layers in the cache-memory hierarchy, with remote memory or FAM being the new layer beyond the DRAM. As new memory layers including disaggregated memories, far memories and non-volatile memories close the gap in speed between different layers of memory, it has become necessary to pursue lower latency approaches for accessing data from these new layers of memory [8, 48]. This paper pursues the approach of prefetching data between DRAM and the lower layers of memory such as disaggregated memory (over CXL-like interconnect) to this end.

This paper explores the potential of utilizing LLC misses that are visible at the root-complex level to build a prefetching mechanism between FAM and DRAM, utilizing a portion of local DRAM as a hardware managed cache for FAM. We call this proposed cache as DRAM cache. We employ SPP [41] based DRAM cache prefetcher as an example to demonstrate the performance gains with DRAM

cache prefetching, but other prefetchers [10, 15, 56] can be employed as well. Our DRAM cache prefetcher maintains the metadata for the cached FAM data. Root complex equipped with a prefetcher redirects requests to cached data to the DRAM cache. On a hit, the cached data will see DRAM latencies instead of FAM latencies. Unlike previous mechanisms that considered page level transfers between DRAM and lower layer memory [8, 48, 54, 78] we consider the potential for sub-page level prefetches at the hardware level.

Since multiple nodes pool memory capacities from FAM, it is imperative that the FAM bandwidth is utilized and shared across multiple nodes effectively. As previous work has shown, prefetch throttling [31, 71] is an effective mechanism to utilize the memory bandwidth. We take inspiration from this earlier work and incorporate ideas for prefetch throttling to effectively manage the FAM bandwidth across demand and prefetch streams from multiple nodes. We draw influence from network congestion algorithms [27] to develop bandwidth adaptation techniques at the source(compute node). Since CXL-connected memory devices can be enhanced with extra functionality, we evaluate the potential of employing Weighted Fair Queueing (WFQ) at the memory node and compare that approach with bandwidth adaptation at the source.

This paper makes the following significant contributions:

- Proposes an architecture for caching and prefetching FAM data into DRAM cache at the granularity of sub-pages.
- Supplements FAM node with WFQ-enabled demand/prefetch request management for effective sharing of available bandwidth.
- Enhances the DRAM prefetcher at compute nodes with bandwidth adaptation, to throttle prefetch issue rate, in response to contention at FAM.
- Evaluates the proposed prefetch mechanism along with improvements in single, multi-node system configurations with homogeneous and heterogeneous application mixes.

Through the rest of the paper we use the terms FAM, memory pool, and pooled memory interchangeably.

2 Background

2.1 CXL enabled memory pooling

Compute Express Link(CXL) [2] is a cache-coherent and a multi-protocol interconnect standard for processors to communicate with devices like accelerators and memory expanders. CXL builds upon the physical layer capabilities of PCIe(electrically compatible). CXL offers 3 kinds of protocols- CXL.cache, CXL.mem, CXL.io. Any device that connects to the host processor using CXL can use either one or more of aforementioned protocols. Based on the protocols used, CXL identifies 3 types of devices that use one or more these protocols.

Type-1 identifies devices like Network Interface Controllers(NICs) that maintain a local cache hierarchy but do not have memory attached to the device, hence uses only CXL.cache protocol. Type-2 devices like GPU, FPGA comprise both local cache hierarchy and attached memory, so these 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.

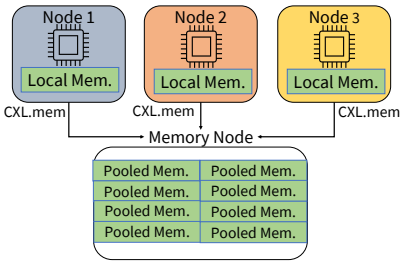

Figure 1: CXL.mem enabled memory pooling

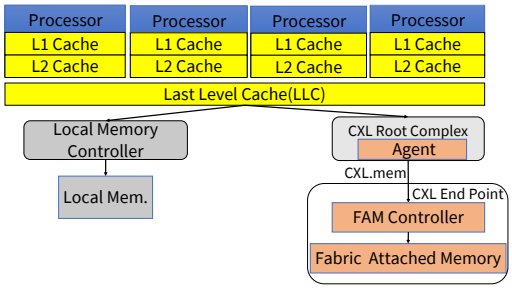

Figure 2: CXL & Fabric Attached Memory(FAM) Architecture

Our discussion in this paper is based on systems that use type-3 devices for memory pooling, through CXL.mem protocol. Fig. 1 shows compute nodes pooling memory resources from a shared memory node. 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 the compute and FAM nodes. CXL root complex of the compute node comprises of an agent that implements a given protocol(CXL.mem in our case). Agent acts on behalf of the host(CPU) handling all communication and data transfers with the CXL end point. In our system CXL end point comprises of a FAM device and a FAM Controller. FAM Controller directly interfaces with the agent, translating CXL.mem commands into requests that can be understood by the FAM device(eg: DDR commands).

As illustrated, load misses and writebacks from LLC are handled either by the local memory controller or CXL root complex based on the physical addresss. The address decoding is implemented in 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.

2.2 Memory Prefetching & Signature Path Prefetcher(SPP)

Data prefetching techniques to hide access latency across different levels of memory hierarchy, are well studied in the literature. Prefetchers typically use learning based approaches to predict the future memory access addresses[10, 40]. Most common features include address delta correlation, program counter(PC) of instruction resulting in cache misses, and access history. Recent work

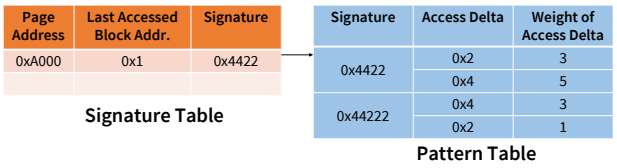

Figure 3: Signature and Pattern tables of SPP

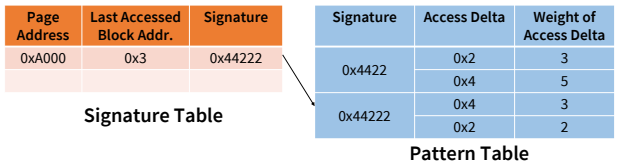

Figure 4: State of SPP after access at an address 0xA003

has applied sophisticated mechanisms like neural networks[67], reinforcement learning [14, 67] to prefetching.

2.2.1 Signature Path Prefetcher(SPP). In this work, we use SPP [40] as base architecture for our DRAM cache prefetcher. SPP uses signatures as a means to keep track of memory access patterns of the workload. Signatures are a compact representation of history of memory access delta's. Architecturally SPP comprises of 2 tables - Signature Table, Pattern Table. Fig. 3 shows the organization of SPP with these tables.

Signature Table is indexed by the physical page address and each entry comprises of of the last cache miss address(within the same physical page), and current signature. Pattern table maps signature to address delta's of future memory accesses. Each pattern table entry has the following entries.

(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.

On a cache access, the physical page address of the access is used to index into the signature table. Output from the signature table consists of the signature. With this output, we can calculate the delta and update the signature as per formulae shown below.

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

Now the generated signature is used to index into the pattern table, which gives us the ordered pairs of delta and the corresponding weights. Delta with the highest weight is used to generate the address for the prefetch requests. Additionally, speculative signature can be formed by combining the current iteration signature(newly formed) and the obtained delta, according to the above formulation.

Speculative signature can further be used to index into the pattern table, to generate another address delta. This recursive indexing into the pattern table can be continued desired number of times or till the pattern table was not able to provide any more delta's. Once the prefetch request generation is complete, weight corresponding to the current delta(output from signature table) is updated in the pattern table. Signature table is updated with current block address and current signature. Fig. 4 shows the state of SPP after an example memory access. Additionally, SPP maintains global history table that bootstraps the learning of access history, when the data access stream moves from one page to another.

Minor changes are made to baseline SPP design, to make it a functional DRAM cache prefetcher(§4.1). We also use the baseline SPP prefetcher, as L2 cache prefetcher in §7.1. *We emphasize that our proposals in this paper are not specific to SPP prefetcher and other prefetcher designs like [10, 15, 56] can be employed with suitable modifications in our system.* Our focus in this work is to demonstrate the usefulness of sub-page level prefetching to hide FAM access latency and to present adaptive optimizations to cater to the usage model of pooled memory. SPP based prefetcher was just taken as an example for a pattern based DRAM prefetcher.

3 Motivation

The primary motivation behind memory pooling in data-centers, is reduction in Total Cost of Ownership(TCO). Cost savings come in the form of decreased memory costs at compute nodes that are using the memory pool. But memory pools aren't a free resource. Establishing memory pools involve costs in terms of, on-board FAM controller, memory devices, fabric re-timers, networking costs etc. Memory pooling makes sense only when it results in costs savings or breaks even.

Recent study from Google [49] argues that CXL memory pools are impractical because of high break even costs. The study considers a 16-node pool, where each node uses 16× PCIe5 lanes to connect to memory pool. To handle memory requests from 16 nodes, FAM controller needs to process data at approximately 1TBps(64 GBps*16). Given the functional similarity to an Ethernet switch, study claims such controller could cost about $38500.

Based on these costs, we perform break even analysis for memory pooling for 16 node system. We consider the memory provisioning model of cloud providers, in which the memory capacity of the system is indicated by Giga Byte per virtual CPU(GB/vCPU). Typical cloud scale processors contain 96-384 vCPU's per node. To calculate memory costs, DDR5 price model of 3$/GB is used. Fig. 5 shows the percentage of memory to pool in order to achieve cost break even, represented across different vCPU's per node and GB/vCPU configurations.

Our analysis reveals percentage of memory that is to be pooled, can vary from 19-99%, depending on the system configuration. For certain vCPU configurations, memory pooling can make sense only with high memory configurations(larger GB/vCPU) e.g., 96 and 128 vCPU's. For a given machine configuration, fraction of memory that is to be pooled decreases with increase in GB/vCPU size.

Based on the above analysis, For memory pooling to be economical, certain system configurations require majority of their memory capacity requirements to be satisfied by the memory pool. Workloads running on these systems will have their resident memory divided between local and pooled memory, according to the

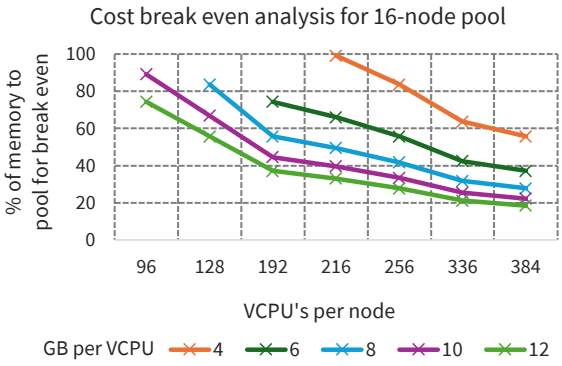

Figure 5: Breakeven analysis for 16-node system

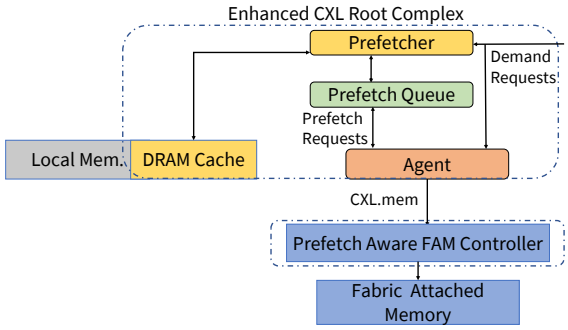

Figure 6: System architecture of root complex with DRAM cache and prefetcher

pooling percentage. Hence, applications running on systems with higher pooled memory fractions, would see longer access latency, resulting in decreased performance.

While application analysis and careful location of frequently accessed or latency sensitive data in DRAM and the rest of the data in FAM could reduce some of the performance loss from longer latencies, such analysis may not always be possible when the workloads are not owned by the cloud provider [46, 50].

To mitigate the performance loss due to using FAM memory pools, we propose DRAM caching of FAM data in local DRAM. This extra layer of caching decreases the number of accesses that see relatively long FAM access latency. Consequently, helping us achieve a balance between performance and cost savings due to memory pooling.

4 System Architecture

In this section we describe the system level architectural enhancements that we propose. Through the rest of the paper, the demands and prefetches that we refer to, are LLC misses and DRAM cache prefetches. DRAM cache prefetches are different from the prefetch requests issued by the per-core cache prefetcher (core prefetches).

4.1 Enhanced Root Complex

DRAM caching/prefetching is implemented through enhancements to the root complex. We add a prefetcher and a prefetch queue, to facilitate the issue of both prefetch and demand requests to FAM. Fig. 6 details the enhanced root complex. We explain the significance of each of the component in detail below.

4.1.1 Prefetcher. As stated earlier, we use a slightly modified SPP as our DRAM cache prefetcher. We make design changes to SPP to operate with sub-page blocks, instead of 64 byte cache blocks. Our prefetcher trains on node physical address of LLC misses. Based on observed patterns, prefetcher generates addresses that are aligned with sub-page block size. LLC misses can occur due to either demand requests or core prefetch requests(L1/L2 cache prefetcher). Our design here doesn't distinguish between types of requests that miss in LLC. Hence for training the DRAM cache prefetcher, core prefetch misses are treated like demand misses.

Prefetcher implementation at DRAM allows for a larger storage overhead relative to the core prefetcher. So, our DRAM cache prefetcher uses tables that are twice as large as the tables of the original design. Storage overhead for our DRAM prefetcher is 11kB(2× to that SPP [41]).

Note that, for memory access to complete, node physical addresses need to be translated to FAM local address. We assume that this translation is handled by the elements lower in the memory hierarchy(HDM decoders). This level of translation is orthogonal to training of DRAM prefetcher with node physical addresses.

4.1.2 Prefetch Queue. Along with the prefetcher, we added a fixed-length prefetch queue to the root complex. For every read miss in LLC that is headed to FAM, the prefetcher generates a predefined number of prefetch requests(at the maximum), we call this number the prefetch degree. Each outgoing prefetch request should have a vacant position in the queue. The prefetch request will be held in the queue until the respective response is received. Since, the queue houses the prefetch requests in progress, the queue provides an easy way to check if a demand request address belongs to any prefetch in progress. In a sense, prefetch queue functionality is similar to MSHR(Miss Status Handling Register) in processor caches. When the prefetch queue is full, no further prefetches are issued until a prefetch response is received.

We should note that the fixed length prefetch queue itself could control the rate of prefetch requests issued to FAM. As we will show later, such static approaches work well for a few applications while leading to wasteful prefetching for several applications. Prefetch bandwidth optimizations adapt the prefetch issue rate beyond the fixed length queue.

Core prefetchers that fetch data into the on-chip processor caches, share queue with the demand requests in MSHR's. In our design, DRAM cache prefetcher cannot use LLC MSHR due to the difference in block size. While it is possible to use multiple entries in the LLC MSHR, it is not a resource efficient approach.

Prefetch requests that are leaving the root complex are tagged. Architectural components in the fabric or at the FAM node, can likely take advantage of this tagging to enforce priority/QoS schemes. We leverage this tagging for WFQ priority enforcement across demand and prefetech requests at FAM(§14)

4.2 DRAM Cache

DRAM cache is explicitly managed in hardware without intervention of the operating system(OS). OS, specifically the memory allocator, only plays a role during initialization phase. The memory allocator should partition the local memory physical address space and expose a contiguous physical address range to be used as a DRAM cache. Memory allocator should also take care of not allocating the physical address range encompassing DRAM cache to any application. We assume that such support exists in the OS. In this implementation, we manage the DRAM cache as a set-associative cache, with replacement policy being LRU. The meta-data to implement the DRAM cache lookup and replacement will be stored outside the DRAM cache, in the prefetcher state(SRAM buffers). Since FAM address space can be large compared to the DRAM cache,

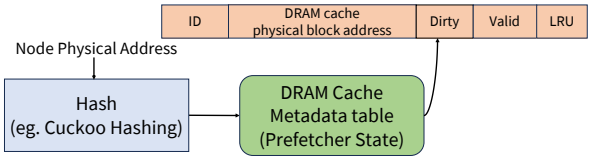

Figure 7: DRAM Cache Metadata format and retrieval

we manage DRAM cache metadata by hashing the FAM addresses into a small number of slots. To minimize hash collisions it is possible to employ techniques such as Cuckoo hashing [58]. Like in cpu caches, tag comparison ensures the correctness of hashing, in an event of collision. The allocated FAM block address is noted in this slot during allocation. The format of the DRAM cache metadata and its retrieval is shown in Fig. 7.

We should note the that metadata cost for DRAM cache management increases linearly with the number of blocks. For a given DRAM cache size, the number of blocks decreases as the block size increases. Hence, an advantage of using larger block sizes for DRAM cache is that the meta-data overhead is reduced. On the flip side, using a larger block size can cause increased latencies at FAM. Given the DRAM cache sizes, metdata management is practical. For example, when managed as a fully associative cache, 16MB cache with 256B block size would require approx. 450KB(64K*7B) of metadata to cover 48-bit physical address space, which is less than 5% of DRAM cache size.

4.3 Demand & Prefetch requests handling mechanism

Fig. 8 explains the flow of FAM memory read requests with DRAM cache and prefetcher. For every outgoing FAM memory request, the prefetcher is consulted to check if the requested block is present in the DRAM cache. If the requested block is present in the DRAM cache, we call it a DRAM cache hit, else we call it a DRAM cache miss. On a DRAM cache hit, a proxy request is created with the DRAM cache block address(obtained from the metadata) and sent to the local memory controller. FAM request waits for the response of this proxy request and will return with the respective response. Subsequently, the metadata(LRU field) of the DRAM cache block that served this proxy request is updated.

On a DRAM cache miss, the requests continue as per usual to FAM. DRAM cache hit requests experience local memory access latency, while DRAM cache misses experience relatively longer FAM latency. FAM read requests can be either generated by application or core prefetcher(L1/L2 cache), both of these request classes are served by the DRAM cache. In addition to read requests, DRAM cache should also handle writeback requests from LLC. Writebacks follow similar datapath as read requests, but in an event of DRAM cache hit, the dirty bit of the corresponding DRAM cache block's metadata is set. During invalidation of DRAM cache blocks, if the block is dirty, it is written to FAM.

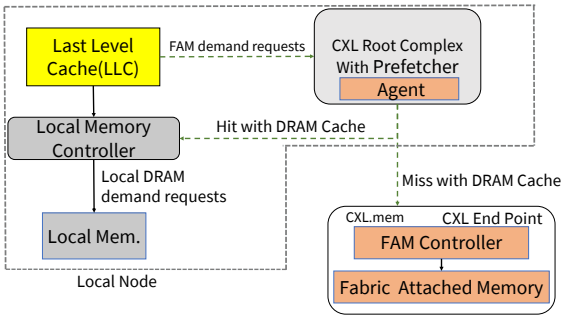

Figure 8: FAM demand request flow with prefetcher and DRAM cache

Irrespective of the DRAM cache hit status, prefetcher generates prefetch requests for every outgoing FAM read request. Before sending out each of the prefetch requests, prefetch queue and DRAM cache metadata are checked to see if generated request is redundant. If not, prefetch requests continue to issue stage. When in the issue stage, prefetch request can be dropped, if the prefetch queue is full or if the occupancy is at a pre-defined threshold(eg: 95%). Given the queue is sufficiently empty, prefetch requests are issued to FAM.

On receiving the response for a prefetch request, metadata is checked to see if there is any vacancy in the DRAM cache. If there is a vacancy, the prefetch response would be directed to appropriate block of the DRAM cache directly. If there is no vacancy, prefetcher issues an invalidation for the LRU block first in DRAM cache and then the corresponding position will be replaced by the incoming prefetch block. Our replacement policy prefers clean blocks over dirty blocks to minimize insertion delays from dirty DRAM cache block flushes to FAM. Fig. 9 shows the algorithmic depiction of demand and prefetch request flows.

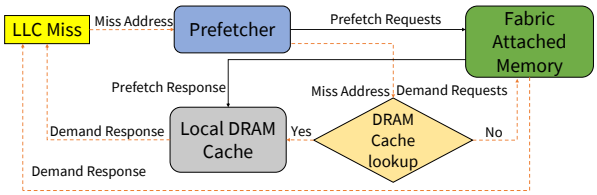

Figure 9: Request flow. Dotted orange line indicates demand requests flow, Solid black line indicate prefetch request flow.

4.4 FAM Controller

The task of FAM controller is to convert incoming cxl.mem protocol requests to DDR requests that could be ultimately handled by the FAM. In a real system, FAM Controller could adapt a port based design, with each port supporting 8×/16× lanes of PCIe/CXL at the front-end, while supporting multiple channels of DDR4/DDR5 memory in the back-end. Network on chip(NoC) like architectural structures might be present in order to route the incoming requests to appropriate queues that feed into DDR channels [50]. Vertical scaling of such controllers might be essential to support large number of PCIe lanes/memory channels depending on the pool size and desired number of memory channels.

We abstract the functionality of FAM Controller as a component that moves incoming requests to DDR channels. All the incoming requests from multiple nodes are filled into the input queue. We assume that FAM controller is aware of the maximum memory bandwidth across all its supported DDR channels. Hence, the controller scans the input queues at appropriate rate and issues the requests to the respective DDR channel.

With the addition of prefetch to the compute node's root complex, FAM controller now receives two types of requests - demand and prefetch. In the baseline design, we implement a single input queue, with FIFO scheduling. Requests(both demand and prefetch) from multiple nodes are dispatched to FAM in the order of their arrival. Later, we explain how the demand and prefetch requests can be given different treatment at the FAM through such mechanisms as Weighted Fair Queuing (WFQ).

4.5 Sub-page block size vs. latency trade-off analysis

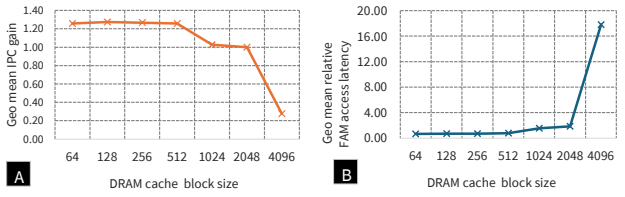

Figure 10: Subfigures A & B represent geo. mean IPC gain and relative FAM access latency across different prefetch block sizes(Both wrt to baseline)

We performed an exploratory analysis to understand the block size vs. latency tradeoff for sub-page block DRAM cache prefetching. We observe the IPC and average FAM access latency by varying the DRAM cache block size. Fig. 10 show our analysis.

As DRAM cache block size increases from 64-512B, IPC gain stays mostly constant with marginal improvements for 128B, 256B block sizes. Beyond 512B, average IPC decreases due to increase in FAM block latency. Moving a FAM page on touch to DRAM cache (4096B block size) observes about 17 × increase in relative FAM latency, resulting in substantial IPC decrement. Based on this analysis, we consider 128, 256, 512 as block sizes for DRAM cache. DRAM cache block size that is a multiple of CPU cache block size(64B) will amortize the delays due to flit packaging/serialization at CXL fabric, as well as reduce the storage overhead of DRAM cache metadata management.

5 Prefetch Optimizations

Our optimizations to the DRAM cache prefetcher are aimed at mitigating the interference between demand and prefetch requests, there by enhancing the utility of FAM bandwidth. To achieve this we propose two approaches - Weighted Fair Queuing at memory node, Prefetch Bandwidth adaptation at compute node. We describe the design and implementation of both of these optimizations below.

5.1 Weighted Fair Queuing(WFQ)

We consider WFQ as a generic means for providing priority to demand requests over prefetches at the FAM. By giving a higher weight to demand requests, we can provide a higher slice of FAM bandwidth to demand requests. Under congestion, WFQ ensures demands are served with priority over prefetches, there by potentially mitigating queuing delays for demands due to prefetches.

We enhance the baseline queuing implementation of the FAM controller, by replacing the single input queue with two input queues, one each for demand and prefetch. Double queue implementation enables us to issue prefetches and demands at independent rates. We take advantage of the prefetch request tagging by the prefetcher at the compute node, to identify the placement of the incoming requests into their respective queues. Both core prefetches and DRAM cache prefetches are placed in the prefetch queue.

We use a WFQ scheduler to issue requests from the two queues to the FAM. We use work-conserving deficit weighted round-robin (DWRR)[68] algorithm to select the queue from which the request should be issued. W-weight is used to indicate how much weight demand requests are given relative to the prefetch requests. Pseudo code of the our algorithm is as shown in Alg.1

For every issue cycle, the IssueRequests() function is called to see if either of demand or prefetch requests can be issued. We use the current_round variable to track the round number within a W+1 round window, when the prefetches and demands are served in 1:W ratio. Prefetch requests are preferred in only one round of the window. During the remaining rounds, we prefer to issue demand requests. Due to the scheduler being work conserving in nature, if the preferred choice of requests are not available, we issue the other type of requests when available. When its the demand request's turn, if the demand_deficit did not exceed maximum permissible deficit(max_demand_deficit), we increment the demand_deficit. If the demand queue is non-empty and demand_deficit is greater than 0, then we issue a demand request, resulting in decrement of the demand_deficit by 1. If either demand queue is empty or demand does not have enough deficit, we try to issue prefetch requests, which is again subject to prefetch queue non-emptiness and status of the prefetch deficit.

In order to account for the difference in prefetch and demand block sizes. We enforce that prefetch_deficit needs to be at least the ratio of prefetch block size and demand block size, for a prefetch to be issued to the FAM. Our proposed DRAM prefetcher works in conjunction with core prefetcher. So, WFQ needs to handle both CPU cache block core prefetches as well as sub-page block DRAM cache prefetches. In our implementation, when its the prefetch

```
Function IssueRequests():
    current_round += (current_weight+1)%(W+1);
    demand_queue_status = CheckDemandQueue();
    prefetch_queue_status = CheckPrefetchQueue();
    r = prefetch_block_size/demand_block_size;
    if current_round != 0 then
        if demand_deficit < max_demand_deficit then
            demand_deficit += quantum;
        if demand_queue_status and demand_deficit>0 then
            IssueDemandRequests();
            demand_deficit = demand_deficit-1;
        else if prefetch_queue_status and prefetch_deficit > r
         then
            IssuePrefetchRequests();
            prefetch_deficit = prefetch_deficit-r
    else
        if prefetch_deficit < max_prefetch_deficit then
            prefetch_deficit += quantum
        if prefetch_queue_status and prefetch_deficit>r then
            IssuePrefetchRequests();
            prefetch_deficit = prefetch_deficit-r;
        else if demand_queue_status and demand_deficit > 0
         then
            IssueDemandRequests();
            demand_deficit = demand_deficit-1
    end
```

Algorithm 1: Demand/Prefetch Issue Algorithm

turn, based on the available deficit we issue either a core prefetch or DRAM cache prefetch. Block size is taken into account when updating the deficit after issuing the given prefetch request.

5.1.1 Prefetch Queue Promotion. While deprioritization of prefetch requests promotes effective bandwidth usage by demands, it can also potentially lead to increased demand wait times. For instance, demand requests for a block might arrive at the root complex while the prefetch request for the same block is in flight. To avoid duplicate requests and inconsistencies, demands could be made to wait for the completion of the prefetch request. With WFQ at FAM, prefetch request completion time, and the resultant wait time of demand can be non-deterministic.

To minimize such waiting delays, our scheduler implements a prefetch queue promotion policy. With this policy, when the demand request arrives at root complex for a prefetch block in flight, a promotion request for the prefetch block is created and sent to the FAM. On receiving a promotion request at the FAM controller, scheduler retrieves the block address and looks for address match with requests waiting in the prefetch queue. If there is a match, the corresponding requests is promoted from prefetch to demand queue. Else, it means the prefetch requests is already issued to FAM and the promotion request is dropped. This ensures that outstanding prefetch requests for the same block do not delay later arriving demand requests for that block.

Event counter	Description
`demand_requests_issued`	demand requests issued to the FAM
`demand_requests_returned`	demand requests return from FAM
`demand_requests_total`	Total demand requests arrive at the prefetcher
`prefetch_requests_issued`	prefetch requests issued to FAM
`minimum_demand_latency`	minimum demand read latency in recent history.

Table 1: Description of event counters

5.2 Prefetch Bandwidth Adaptation

In the baseline prefetcher, we generate a fixed number of prefetch requests(prefetch degree) for every LLC miss and issue those requests depending on the prefetch queue availability. When the FAM device is saturated due to high number of demand requests, issuing prefetch requests would increase demand latency due to queuing at FAM, hurting overall performance. Under such conditions, it is better to dial back the prefetch request issue rate and wait till the FAM has enough bandwidth to accommodate prefetch requests. To incorporate such system level feedback, we implement prefetch bandwidth adaptation at the source.

We take a sampling based approach, to adapt the prefetch issue bandwidth. To incorporate FAM congestion awareness, we add event counters to the root complex's prefetcher state. Each counter stores two values, instantaneous value and average value. The instantaneous values of counters are scanned and reset during the start of each sampling cycle. Average value of the event counter stores the exponential moving average of the respective values. The descriptions of event counters that are stored in the prefetch state are as shown in Table.1.

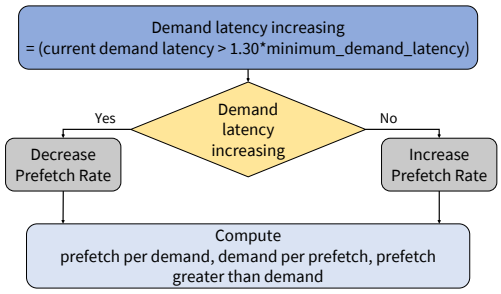

Figure 11: Flow chart for prefetch BW adaptation algorithm

Abstract-level logical flowchart of our bandwidth adaptation algorithm is shown in Fig.11. The algorithm is executed every sampling cycle. Our key idea here is to reduce the prefetch issue rate when FAM is experiencing congestion. We use growing demand latency as an indicator for congestion. Onset of this indication, we decrement the prefetch issue rate.

To determine if the demand latency is increasing, we compare the measured demand read latency with achievable minimum demand read latency. Through run-time, minimum demand latency

is unknown and influenced by several aspects like locality of memory requests, fabric topology, and queuing delays at fabric components(like switches, re-timers). So, we approximate minimum demand latency to lowest value of average demand latency seen by the node(root complex) in the recent history. By tuning the length of this history, we can tweak the agility of prefetch throttling. In our implementation, we set the history length to 8 sampling cycles.

Our algorithm perceives congestion, if the measured demand latency is above 130% of minimum demand read latency. We use 30% as heuristic to eliminate noise in demand latency measurement due to row hits/misses. Noise margin can be tweaked based on the underlying device organization.

We employ Multiplicative Increase and Multiplicative Decrease (MIMD) [21] technique for adjusting the prefetch rate. In our implementation, we set the increase factor to 1.125(12.5% over prev. value). The decrease factor is determined dynamically based on the observed prefetcher accuracy. Our algorithm penalizes prefetch request stream with high accuracy at slower rate and that of lower prefetch accuracy at faster rate. We expect this optimization to result in more accurate prefetches to be issued when multiple applications are competing for bandwidth at FAM. In addition, we mimic RED [17, 27] at the source and make the decrease factor linearly dependent on the difference of observed latency and minimum demand latency.

While both WFQ and bandwidth adaptation are trying to address the same problem, we evaluate relative merits of both the schemes. CXL memory nodes could provide additional functionality and our evaluation of WFQ is intended to understand the implications of augmenting a memory node with WFQ scheduler. On the other hand, prefetch bandwidth throttling at the source could decrease the prefetch requests issued possibly alleviating stress on both FAM and fabric components.

6 Discussion

6.1 Cache Coherency with DRAM cache

Architecturally, DRAM cache is located outside the CPU caches. Requests to DRAM cache sub-system, can be either LLC read misses or writebacks(due to invalidation of cache block). Both of these request types are handled as reads and writes respectively, at the main memory level, irrespective of the coherency protocol of the CPU. Hence, DRAM cache exists outside the coherency domain of the CPU and its operation is orthogonal to coherency protocol implementation.

6.2 Overhead of Prefetch Optimizations

Implementation of prefetch optimizations incurs minimal storage overhead. Bandwidth adaptation requires additional 32B storage to track event counter and control states. Adaptation Algorithm implementation requires few arithmetic and logic blocks, which is affordable at the root-complex level. With possibility of compute capabilities at the CXL node, WFQ scheduler implementation is straight forward, and requires minimal resources.

6.3 FAM Controller Assumptions

FAM controller in real systems supports a given PCIe/CXL configuration like, 16× or 8× lanes. Hence, peak input data rates that need to be handled by FAM controller can be easily deducted. Based on this peak data rate, vendors typically configure the transfer rate of each memory channel and number of memory channels, that would be managed by FAM controller(e.g. Astera Labs Leo Controllers[4]). With the channel count and transfer rate known, the scanning frequency of input request queues can be calculated. Therefore, it is reasonable for us to assume that FAM controller scans input queues at appropriate frequency based on the supported memory channel configuration.

6.4 Data sharing between nodes

Hardware coherency for memory accesses to pooled FAM, is not supported by CXL.mem 1.0, 2.0 protocols. Hence, shared memory implementation with these protocols would require additional middleware support. But recently announced CXL 3.0 specification, supports hardware coherency mechanisms for FAM data accesses, across multiple nodes[34].

In our proposed DRAM cache design, we did not consider the use case of FAM data sharing between different nodes. But implementation of such feature would require integration of DRAM cache with the agent that manages coherency between multiple nodes, DRAM cache block status(for every node in pool) need to updated with this coherency agent in order for the requesting node to obtain the correct(or updated) data block. Such kind of implementation is scope for future work.

Processor	8 out-of-order cores clock: 3.3 GHz, 6 issue/cycle max pending transactions : 16
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) [41]
L3 cache	8 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/direction Min. Latency: 70ns
Per-Node prefetch queue size	256
Pooled FAM	DDR4-2400 2 channels, 2 ranks

Table 2: Simulated system configuration

7 Evaluation

7.1 Methodology

We evaluate DRAM cache prefetcher along with optimizations, using SST[60] simulation components. We used Ariel, a pin-tool based processor front-end simulator to simulate compute nodes. Ramulator[42], a detailed memory simulator has been used to model both local memory(DRAM) and pooled FAM devices. Opal[43] was

Benchmark suite	Workload	Memory footprint
SPEC17	603.bwaves_s	824 MB
	607.cactuBSSN_s	257 MB
	619.lbm_s	1.55 GB
	628.pop2_s	590 MB
	649.fotonik3d_s	587 MB
	654.roms_s	245 MB
	657.xz_s	561 MB
Splash 3	LU	515 MB
	FFT	625 MB
GAP	bfs	864 MB
	cc	802 MB
	bc	593 MB
	sssp	545 MB
PARSEC	dedup	868 MB
	facesim	188 MB
	canneal	849 MB
NPB	mg	431 MB
	is	1 GB
XSBench	XSBench	611 MB

Table 3: Benchmark configurations

used to emulate the role of operating system's memory allocator and page fault handler. Opal allows for configurable memory footprint distribution between local DRAM and pooled FAM. CXL network is simulated by provided flit based network model, with programmed delay and bandwidth.

All of the simulation components are tightly integrated and distributed as a framework [5]. We found the simulation results to be deterministic and consistent across multiple simulation runs. SST based simulation methodology was used by prior works [70, 72].

We evaluated 19 memory intensive workloads from benchmark suites like SPEC[19], PARSEC[18], GAP[13], Splash3[64], and NPB[9]. Modern servers contain compute nodes that comprise 64-128 processors and few 100's of GB of main-memory. Simulating such a system, is impractical given the simulation speeds. For realistic simulation schedules, we simulate a system with scaled down configuration, that runs regions of interest(ROI) within each benchmark. We expect that our simulator and the corresponding performance characteristics to scale to a larger configuration, with no issues. The simulated system configuration is detailed in Table.2. Evaluated applications and their respective memory footprints are shown in Table 3. Workloads are configured have their memory footprint atleast 4-8 times the size of LLC+DRAM cache, in order to ensure that they stress the main memory.

Both single and multi-node system configurations are evaluated. Since modelled FAM has 2 DDR4 channels, we consider upto 4 compute nodes in the multi-node system. For multi-node systems, we ran copies of the same application on different nodes, as well as different applications(mix) on different nodes. We evaluated 7 such workload mixes on a 4-node system, details of the workload mixes are stated in Table. 4. In multinode systems, we expect the higher loads at FAM to result in tighter availability of bandwidth and hence possibly higher congestion.

Mix	node 0	node 1	node 2	node 3
mix1	is	bfs	657.xz_s	FFT
mix2	628. pop2_s	XSBench	cc	mg
mix3	canneal	bc	607. cactuBSSN_s	dedup
mix4	654. roms_s	cc	619. lbm_s	LU
mix5	603. bwaves_s	facesim	is	XSBench
mix6	cc	XSBench	657.xz_s	649. fotonik3d_s
mix7	sssp	607. cactuBSSN_s	649. fotonik3d_s	XSBench

Table 4: Details of multi-node workload mixes

Below we define figures of merit, terms, and configurations that we use in discussion through the rest of the section

(1) *Core pretcher* - Each node in our system, comprises a multi-core CPU. Within each core, a prefetcher is at the L2 cache level.
(2) *baseline configuration* - System with both core prefetchingand DRAM cache prefetching, disabled.
(3) *core prefetch configuration* - System with core prefetching enabled, but DRAM cache prefetching disabled.
(4) *all-local configuration* - Workload running with core prefetching enabled along with entirety of its memory footprint residing in local DRAM(FAM is not used).
(5) *Non-adaptive DRAM prefetch configuration* - Workload running with core prefetching enabled and DRAM prefetch enabled, with FIFO scheduling at FAM and no bandwidth adaptation.
(6) *allocation ratio(X)* - Workload's memory footprint divided between FAM and DRAM in ratio of X:1 respectively.
(7) *IPC gain* - Ratio of IPC for a given config. to that of workload in baseline config.(Higher the better)
(8) *Relative FAM latency* - Ratio of the average FAM access latency for a workload in a given configuration to that of workload running in core prefetch configuration.(Lower the better)
(9) *Relative DRAM prefetch requests issued* - Ratio of DRAM cache prefetches issued for a given config to that of DRAM cache prefetches issued with non-adaptive DRAM prefetch configuration .
(10) *Demand hit fraction* - Fraction of demand requests that miss LLC, that hit in DRAM cache.
(11) *Core Prefetch hit fraction* - Fraction of core prefetch requests that miss in LLC, that hit in DRAM cache.

7.2 Performance gain with Prefetch Bandwidth Adaptation

For this analysis, we run workloads in 1,2, and 4 node system configurations. Each workload ran in 3 prefetch configurations - Core Prefetcher (core prefetch), Core+DRAM cache prefetcher(non-adaptive DRAM prefetch), Core+DRAM prefetch+prefetch BW adaptation. Each workload has memory allocation ratio of 8.

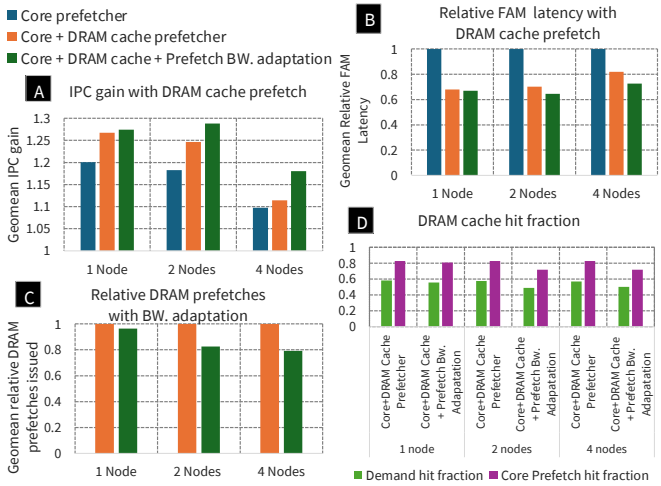

Figure 12: Evaluation of DRAM cache prefetcher with prefetch bandwidth adaptation

Fig. 12 outlines the results of our experimentation. 12A shows geometric mean (geomean) IPC gain of all benchmarks, for each prefetch configuration, across the 3 node configurations. Across the board, DRAM prefetching improves overall performance compared to core prefetching. Core prefetching resulted in IPC gain of 1.20, 1.18, 1.10 for 1,2,4 node systems. With non-adaptive DRAM prefetch, the same IPC gains increased to 1.26, 1.24, 1.11 respectively. *So, non-adaptive DRAM prefetch resulted in IPC improvement 6%, 6%, 1% for 1,2,4 node systems respectively, over core-prefetching.*

Performance improvement comes from reduction in FAM access latency, as indicated by relative FAM latency in Fig. 12B. Non-adaptive DRAM prefetch reduced average FAM access latency by 29%, 30%, 19% for 1,2,4 node systems respectively.

Prefetch BW adaptation further enhanced the performance of DRAM cache prefetching, for 2 and 4 node systems. *BW adaptation resulted in 4% and 8% IPC improvement over non-adaptive DRAM prefetch for 2 and 4 node systems.* Non-adaptive DRAM prefetch config. performed poorly in 4 node system configuration, with minimal IPC improvement over core prefetch, thus emphasising the importance of BW adaptation in bandwidth constrained systems. BW adaptation resulted in 5% and 9% decrement of relative FAM latency over non-adaptive DRAM prefetch.

Fig. 12C presents the relative no. of DRAM prefetch requests. Results indicate that bandwidth adaptation caused 18% and 21% less DRAM cache prefetches to be issued to FAM, in 2-node, 4-node systems respectively. Decreased DRAM prefetch issue rate resulted in decreased demand and core-prefetch hit rate, according to analysis presented in Fig. 12D. For instance, BW adaption reduced the demand and core-prefetch hit fraction from 57% and 83% to 50% and 72%, for 4-node system. Performance improvement despite hit fraction decrement in 4-node system reveals that prefetch requests are causing considerable queuing delays at FAM.

Further, we analyze the IPC gain with prefetch bandwidth adaptation for 4-node system across different benchmarks, as shown in Fig. 13. Non-adaptive DRAM prefetch significantly improves IPC for applications like dedup, LU, 628.pop2_s, mg, is, and facesim.

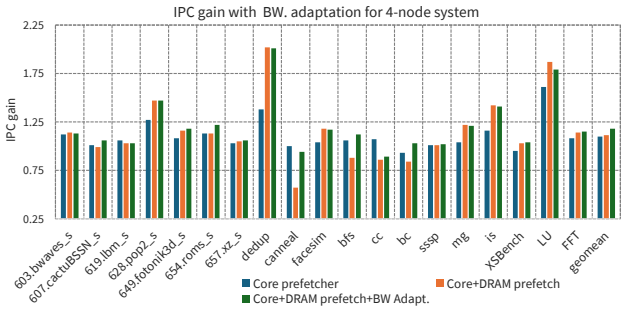

Figure 13: IPC gain due to BW. adaptation for 4-node system. Geomean from this analysis is represented in "4 Nodes" in Fig. 12A

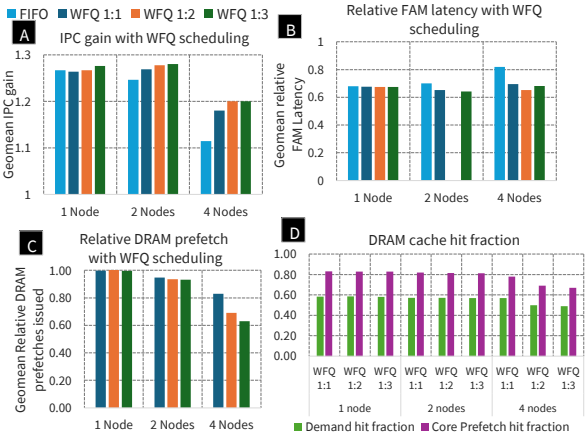

Figure 14: Evaluation of DRAM cache prefetcher with WFQ scheduling

Workloads like canneal, bfs, cc, and bc saw IPC decrement with non-adaptive DRAM prefetch, possibly due to increased FAM latency. BW adaptation was able to improve the IPC substantially for these applications, expect for cc.

BW adaptation would mitigate congestion only when the DRAM prefetches are responsible for creating it. This is because our algorithm implementation throttles only DRAM cache prefetch issue rate. BW adaptation would be of little help if core prefetches are responsible for congestion. Future implementations of our prefetch throttling algorithm can relay the occurence of congestion to the CPU cache controllers, to enable dynamic throttling of CPU prefetch requests.

7.3 Performance gain with WFQ scheduling

We evaluate our WFQ scheduling algorithm with 3 weights-1,2,3 (weight of 3 indicates demands and prefetches are served in 3:1 ratio). Each workload is run in 1,2,4 node system configuration, with WFQ scheduling at the FAM controller, and with different weights. Performance of WFQ with different weights is compared to non-adaptive DRAM prefetch(FIFO as indicated in figures). We

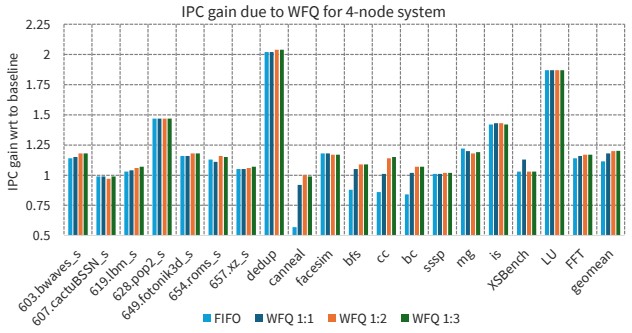

Figure 15: IPC gain due to WFQ for 4-node system. Geomean from this analysis is represented in "4 Nodes" in Fig. 14A

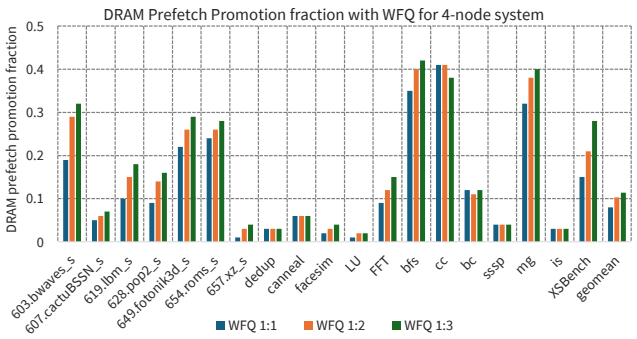

Figure 16: Fraction of DRAM prefetch requests that promoted from prefetch to demand queue at FAM controller

use FAM/DRAM allocation of 8 for this analysis. Fig. 14 shows our results.

Fig. 14A shows geomean of IPC gain across all benchmarks,for WFQ algorithm with different weights, across node configurations. WFQ improves the performance over non-adaptive DRAM prefetch for 2,4 node systems. *Weights 1,2,3 improve the average IPC by 8%(3%), 9%(4%), 9%(4%) over non-adaptive DRAM prefetch for a 4(2) node system.* Again, the increase in IPC is due to decrement in FAM access latency. For a 4(2) node system, average relative FAM latency is reduced by 17%(6%). Given that BW adaptation resulted in 7% IPC improvement over FIFO scheduler, WFQ marginally performs better.

WFQ also resulted in less number of DRAM prefetches to be issued. For a 4 node system, WFQ with weights 1,2,3 resulted in 17%, 31%, 37% decrement in average relative DRAM prefetches issued. Such behavior is expected because, as the weight to demands increases, prefetch request latency increases, filling the prefetch queue, subsequently causing less number of prefetches to be issued. Fig. 14D shows the demand and core-prefetch hit fraction with WFQ across different node configurations and weights. For 4 node system, hit fractions drop as the weight increase, this can be perhaps due to less prefetch requests being issued.

Additionally, we analyze the IPC gain with WFQ for 4-node system across different benchmarks, as shown in Fig. 15. Workloads that benefited from BW. adaptation, benefited from WFQ as well. Interestingly, cc application sees IPC improvement with WFQ but not with BW. adaptation. Due to placement of both core prefetch and DRAM cache prefetch requests into the same queue, WFQ can potentially mitigate congestion due to either of the prefetch request type. This can be the reason why WFQ performs marginally better in comparison to bandwidth adaptation.

Fig. 16 shows the DRAM prefetch promotion fraction for evaluated benchmarks across different weights of WFQ scheduler. Promotion fraction is significant in applications like, 603.bwaves_s, 649.fotonik3d_s, 654.roms_s, bfs, cc, mg, and XSBench. For most of these application, the promotion fraction increases with increase in WFQ weight, expect cc. On an average, across the evaluated workloads, DRAM prefetch promotion fraction is 8%, 10%, 11% for weights 1,2 and 3 respectively. This analysis reveals that a sub-set

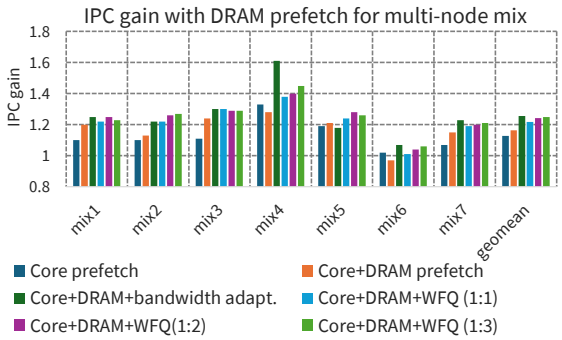

Figure 17: Performance gain with different configurations of DRAM prefetch across 7 multi-node workload mixes

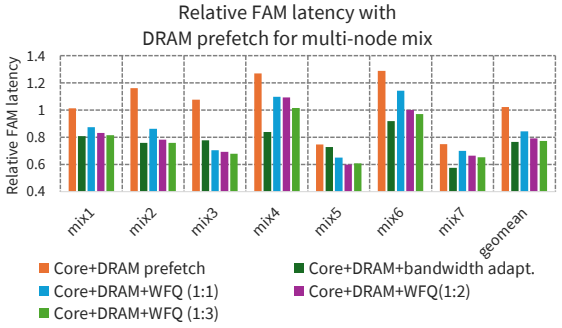

Figure 18: Relative FAM latency with different configurations of DRAM prefetch across 7 multi-node workload mixes

of applications benefit from prefetch queue promotion policy that was implemented as part of our WFQ scheduler.

7.4 Performance analysis of multi-node workload mixes

Combining the methodology of §7.2 and §7.3, we evaluated our 7 multi-workload mixes with a total of 5 prefetch configurations- Core+DRAM prefetch(non-adaptive DRAM prefetch), Core+DRAM prefetch+Bandwidth. Adaptation, Core+DRAM prefetch+WFQ[1,2,3]).

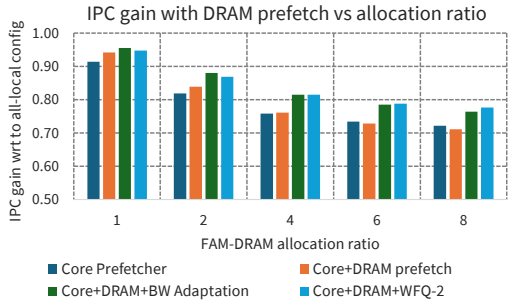

Figure 19: IPC gain wrt to all-local configuration

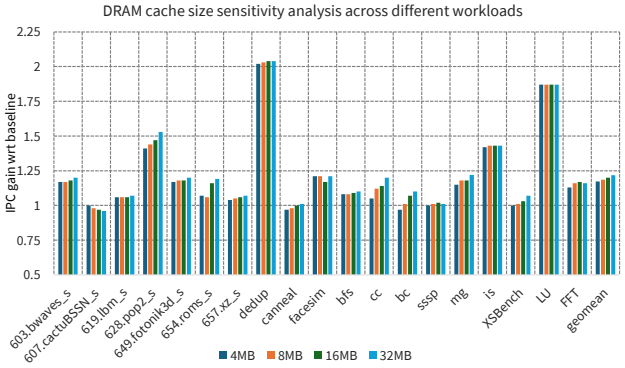

Figure 20: DRAM cache size sensitivity analysis, with DRAM cache sizes varied from 4-32MB

Allocation ratio is set to 8. Each mix is run on a 4 node system. Fig. 17 and Fig. 18 show the IPC gain and relative FAM latency for all the workload mixes.

BW adaptation and WFQ provide equivalent IPC improvement over non-adaptive DRAM prefetch for mix1, mix3, mix6 and mix7. Over non-adaptive DRAM prefetch, both of optimizations equivalently resulted in IPC gain of 5%, 6%, 10%, 8% for mix1, mix3, mix6, and mix7 respectively.

Mix5 saw slight IPC decrement with BW adaptation, but a performance improvement of 7% with WFQ. BW adaptation outperformed WFQ by 16% for mix4. WFQ outperformed BW adaptation by 5% for mix2. On an average, BW adaptation and WFQ resulted in 10% and 9% IPC gain over non-adaptive DRAM prefetch. Similarly, BW adaptation and WFQ reduced the FAM latency by 26% and 25%.

This analysis reveals to us that both these prefetch optimizations are useful in mitigating congestion at FAM. But the relative performance gain due to either of these approaches depends not just on the workload alone, but also on co-existing workloads. Bandwidth adaptation might perceive congestion due to co-existing workloads as self caused and accordingly respond with prefetch request throttling. WFQ being implemented at FAM, applies static priority to demands over prefetches for all the request streams, which might or might not result in optimal performance.

7.5 Performance improvement across allocation ratios

We study the impact of DRAM cache prefetch along with proposed optimizations across different allocation ratios. We considered 4 prefetch configurations for this experiment - Core Prefetcher(core prefetch), Core+DRAM prefetch(non-adaptive DRAM prefetch), Core+DRAM+BW. adaptation, Core+DRAM+WFQ(2). We vary the allocation ratio from 1(50%) to 8(88%) and measure the IPC gain with respect to all-local configuration for each benchmark, for a 4-node system. Fig. 19 shows the geo. mean of IPC gains of all benchmarks, with a given allocation ratio, across different DRAM prefetch configurations.

This experimentation reveals that utility of core prefetching decreases as FAM usage increases. Core prefetch allowed 9% IPC decrement when 1/2 of data is resident in FAM and 28% IPC decrement when 8/9th of data is in FAM.

DRAM prefetch along with the presented optimizations, improves the IPC by an average of 5%-6% across all the allocation ratios. These improvements help bridge the performance gap between pooled memory configuration and all-local configuration. For allocation ratio of 1 (50%), DRAM prefetch equipped system allowed only 4% performance loss compared to all-local configuration. These results indicate that our proposed DRAM prefetch(with both optimizations) efficiently exploits bandwidth at FAM for performance enhancement.

7.6 Sensitivity Analysis

7.6.1 DRAM Cache Size. To analyze the sensitivity of our design to DRAM cache size, we evaluated a 4-node system, with WFQ scheduler(weight 2) at FAM controller, and allocation ratio of 8. We run same copies of the given workload, with varying DRAM cache sizes. Fig. 20 shows the results of these experiments. Benchmarks like 628.pop2_s, 654.roms_s, cc, bc, XSBench showed sensitivity to DRAM cache size, their respective IPC gains increased with increase in DRAM cache size. On average, DRAM cache size of 4MB, 8MB, 16MB, 32MB resulted in IPC gain of 1.17, 1.19, 1.20, 1.22. IPC improved by 5% as the DRAM cache size increased from 8 to 32MB on average.

7.6.2 CXL Fabric Latency. As discussed earlier, CXL fabric latency(round trip) is implementation specific. Here we evaluate sensitivity of our DRAM cache prefetcher to different CXL fabric latencies. We consider 3 latency values for this experimentation -70 ns, 150 ns, 210 ns. We evaluate a 4-node system, with WFQ scheduler(weight 2) at FAM controller, and allocation ratio of 8.

Fig. 21 shows IPC gain for evaluated applications with DRAM cache prefetcher, across different CXL fabric latencies. For most of the workloads, performance improvement due to DRAM caching, increase with increase in fabric latency. 657.xz_s presents an interesting case, where the performance gain remains constant with increasing CXL fabric latency. On an average, DRAM prefetch resulted in IPC gain of 10%, 18%, 24% for CXL fabric latencies of 70 ns, 150 ns, 210 ns respectively.

7.6.3 Prefetch Block size. To evaluate the sensitivity of our DRAM cache to the prefetch block size. We run all the workloads in 4-node system, by changing the programmed prefetch block sizes. We consider 3 prefetch configuration for this analysis, Core+DRAM

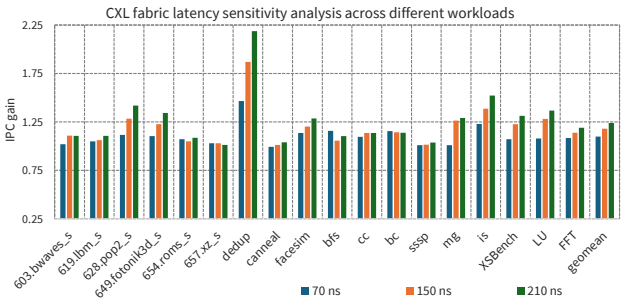

Figure 21: IPC gain with DRAM cache prefetcher for different CXL fabric latency

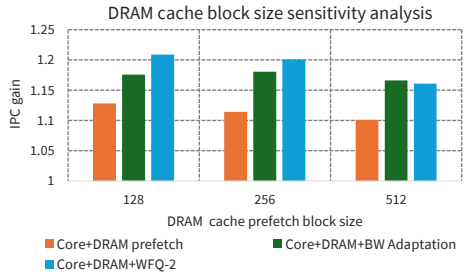

Figure 22: IPC gain with DRAM cache prefetcher for different prefetch block size

prefetch (non-adaptive DRAM prefetch),Core+DRAM+Bandwidth adaptation, Core+DRAM+WFQ-2. Allocation ratio is set to 8. Fig. 22 shows the geomean of IPC gain of all benchmarks, executed with given DRAM prefetch configurations, across 128, 256, 512 block sizes. Non-adaptive DRAM prefetch saw slight IPC decrement as the prefetch block size is increased from 128 to 512. But with the optimizations turned ON, performance improvement with 128, 256 block sizes are similar. For both of the block sizes, WFQ-2 the performance of non-adaptive DRAM prefetch by 8-9%. Even with bandwidth adaptation and WFQ, 512 block size performed poorly, with IPC trailing by 5% compared to 128, 256 block sizes.

8 Related Work

8.1 Memory Tiering

Pond [50] by Microsoft, uses machine learning algorithm to classify the applications into memory latency sensitive/insensitive categories. Based on the classification, an allocation policy that places the application's data in slow memory tier(FAM),is enforced. Similar data placement mechanism is proposed by Google [46]. Both of these approaches assume strong knowledge of application behavior. While this might be true for hyperscaler specific workloads, it might not be possible for all system use cases.

Page placement and movement strategies have been extensively explored in [52, 53, 55, 79]. These approaches heavily rely on linux kernel's page access tracking mechanisms, to create a hot/cold profile of application's memory footprint. Such profiling enables, informed placement and timely movement, of pages across different memory tiers. Our proposed DRAM cache prefetching is orthogonal to such page level approaches. Our prefetcher predicts the future memory accesses using on the fly learning of application's access patterns and prefetches sub-page blocks into DRAM cache.

8.2 Prefetch optimizations

FDP [71] has considered feedback to control the prefetching rate from DRAM into LLC. Our proposal differs from this work in significant ways. We consider prefetching from FAM into DRAM, employ more sophisticated prefetchers, implements separate queues & priorities for demand & prefetch requests.

Separate queues for prefetch & demand requests, and fair queuing, have been studied earlier in the context of multiprocessor systems where resources are shared [23, 36]. We leverage this earlier work in context of memory pooling between multiple compute nodes.

Blue[61] and APT-GET[35], considers timeliness of prefetches in order to issue prefetches sufficiently ahead. However, these works do not consider dynamic latency at the memory system. Additionally, our proposed system implements de-prioritization of prefetch requests at FAM, which can potentially complicate timeliness estimations. Timeliness optimizations that accounts for dynamic latency at FAM, can be scope for future work.

In systems with limited bandwidth, Criticality aware prefetchers drops the prefetch requests that are less critical(useful), there by reducing stalls at memory device. CLIP [59] uses a two stage design that considers ROB occupancy and prefetcher accuracy to filter out loads for prefetch generation. Our proposed system operates outside the processor and doesn't have access to such information. so we use latency measurements and DRAM prefetch accuracy to throttle(drop) prefetch requests.

8.3 CXL-FAM and Remote Memory

Prefetching from far/remote memories has received significant attention recently [8, 24, 29, 54]. These works predominantly aim to optimize the cost of moving pages from one node to another while allowing memory to be shared across nodes connected through InfiniBand network. Our work in this paper, takes motivation from aforementioned works, but considers hardware-based prefetching of sub-page blocks in a more tightly integrated environments like CXL-attached FAM.

Recent work, Johnny Cache [48] has shown that it is possible to avoid cache conflicts in fast memory when fast memory is completely used as a cache for FAM. Our work employs only part of the fast memory as a cache, while using rest of fast memory to satisfy application memory needs.

PreFAM [44] proposes prefetching at a distance from FAM into local DRAM cache. Albeit the similarities in the system architecture, PreFAM doesn't consider resource contention at FAM due to other participating nodes. Moreover, we propose dynamic adaptation of prefetch bandwidth based on perceived FAM latency in this work, while PreFAM considers changing distance at which prefetch requests are generated.

Latency overhead for LLC miss due to CXL fabric access is implementation dependent. Intel's CXL implementation [66] states that CXL fabric could add 70 ns to memory access latency. Direct CXL

[28] implements a memory pooling solution leveraging CXL.mem protocol, reporting that their CXL fabric adds around 200 ns to access latency. DRAM cache prefetching along with bandwidth optimizations that we proposed in our work are agnostic to CXL fabric latency.

Cache in Hand [45] proposes a system that uses CXL attached SSD's, as a cost effective memory capacity expander. Cache in Hand implements a prefetcher on the expander side(FAM side), which directly forwards the prefetch data to LLC using Data Direct IO(DDIO) mechanism. Our work in this paper focuses on effective bandwidth utilization of FAM pools across multiple pooling nodes. Nevertheless, DRAM caching expands the effective cache capacity of the CPU beyond LLC, potentially reducing off-node accesses.

Hotbox [16] has suggested that the employment of large pages may not be universally beneficial in disaggregated memory systems. Our work here considers the movement of sub-page data blocks between DRAM and FAM to reduce the access latencies at FAM.

9 Conclusion

This paper proposed a prefetching mechanism for caching sub-page blocks from FAM in a portion of the DRAM. The prefetcher learns from LLC misses heading to FAM and issues prefetch requests to bring FAM data to DRAM to reduce the latency of future accesses. We considered two optimizations to mitigate congestion: compute-node based issue rate management mechanism based on observed latencies, and a memory-node based weighted fair queuing mechanism. We show that both of these mechanisms can improve the IPC of non-adaptive DRAM cache prefetch by 7-10%. Our evaluation reveals that both of the approaches are effective in improving performance in different system configurations and workloads.

10 Acknowledgement

Portions of this research were conducted with the advanced computing resources provided by Texas A&M High Performance Research Computing.

References

[1] [n. d.]. Compute Express Link. https://www.computeexpresslink.org accessed 23-Aug-2022.
[2] [n. d.]. Compute Express Link 3.0. https://www.computeexpresslink.org/spec-landing accessed 23-Aug-2022.
[3] [n. d.]. CXL AND GEN-Z IRON OUT A COHERENT INTERCONNECT STRATEGY. https://www.nextplatform.com/2020/04/03/cxl-and-gen-z-iron-out-a-coherent-interconnect-strategy/ Accessed May 2024.
[4] [n. d.]. Leo memory controllers documentation. https://www.asteralabs.com/wp-content/uploads/2024/05/231220_Product_Brief_Leo_CXL_Smart_Memory_Controllers.pdf Accessed June 2024.
[5] [n. d.]. SST Elements. github.com/sstsimulator/sst-elements accessed Jun 23, 2024.
[6] [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.
[7] Neha Agarwal and Thomas Wenisch. 2017. Thermostat: Application transparent page management for two-tiered main memory. *ACM SIGARCH Computer Architecture News* (2017).
[8] E. amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker. Apr.2020. Can far memory improve job throughput? *Proc. of ACM Eurosys Conf.* (Apr.2020).
[9] 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.
[10] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad. 2019. Bingo spatial data prefetcher. *Proc. of IEE HPCA Conf.* (2019).
[11] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review* 37, 5 (2003), 164–177.
[12] Sathwika Bavikadi, Purab Ranjan Sutradhar, Khaled N. Khasawneh, Amlan Ganguly, and Sai Manoj Pudukotai Dinakarrao. 2020. A Review of In-Memory Computing Architectures for Machine Learning Applications. In *Proceedings of the 2020 on Great Lakes Symposium on VLSI* (Virtual Event, China) *(GLSVLSI '20)*. Association for Computing Machinery, New York, NY, USA, 89–94. https://doi.org/10.1145/3386263.3407649
[13] Scott Beamer, Krste Asanović, and David Patterson. 2017. The GAP Benchmark Suite. arXiv:1508.03619 [cs.DC]
[14] R. Bera, K. Kanellopoulos, A. Nori, T. Shahroodi, S. Subramoney, and O. Mutlu. Oc. 2021. Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning. *Proc. of IEEE MICRO Conf.* (Oc. 2021).
[15] R. Bera, A. Nori, O. Mutlu, and S. Subramoney. 2011. DSPatch: Dual Spatial Pattern Prefetcher. *ACM ISCA* (2011).
[16] S. Bergman, P. Faldu, B. Grot, L. Vilanova, and M. Silberstein. June 2022. Reconsidering OS Memory Optimizations in the Presence of Disaggregated Memory. *ACM Int. Symp. on Memory Management* (June 2022).
[17] S. Bhandarkar, A. L. N. Reddy, Y. Zhang, and D. Loguinov. Oct. 2007. Emulating AQM from end hosts. *ACM Sigcomm Conf.* (Oct. 2007).
[18] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors.* Ph. D. Dissertation. Princeton University.
[19] 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. https://doi.org/10.1145/3185768.3185771
[20] I. Calciu, MT Imran, I. Puddu, S. Kashyap, HA Maruf, O. Mutlu, and A. Kolli. 2021. Rethinking software runtimes for disaggregated memory. *Proc. of ACM ASPLOS Conf.* (2021).
[21] D. M. Chiu and R. Jain. 1989. Analysis of increase and decrease algorithms for congestion avoidance in computer networks. *Journal of Computer Networks and ISDN Systems* (1989), 1–14.
[22] J. Devlin, M-W. Chang, K. Lee, and K. Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv:1810.04805* (2019).
[23] E.Ebrahimi, C. K. Lee, O. Mutlu, and Y. Patt. 2011. Prefetch-Aware Shared-Resource Management for Multi-Core Systems. *ACM ISCA* (2011).
[24] Viacheslav Fedorov, Jinchun Kim, Mian Qin, A. L. Narasimha Reddy, and Paul Gratz. Oct. 2017. Speculative Paging for Future NVM and SSD. In *Proceedings of the 2017 International Symposium on Memory Systems.*
[25] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A.D. Popescu, A. Ailamaki, and B. Falsafi. 2012. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems.* ACM, 37–48.
[26] Brad Fitzpatrick. 2004. Distributed Caching with Memcached. *Linux J.* 2004, 124 (Aug. 2004), 5–. http://dl.acm.org/citation.cfm?id=1012889.1012894
[27] S. Floyd and V. Jacobson. 1993. Random Early Detection Gateways for Congestion Avoidance. *ACM/IEEE Trans. on Networking* (1993).
[28] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 287–294. https://www.usenix.org/conference/atc22/presentation/gouk
[29] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. 2017. Efficient Memory Disaggregation with INFINISWAP. *Proc. of USENIX NSDI Conf.* (2017).
[30] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. 2011. Toward Dark Silicon in Servers. *IEEE Micro* 31, 4 (july-aug. 2011), 6 –15.
[31] W. Heirman, K. DuBois, Y. Vandriessche, S. Eyerman, and Ibrahim Hur. 2018. Near-side prefetch throttling: Adaptive prefetching for high-performance many-core processors. *ACM PACT Conf.* (2018).
[32] C-C Hung, G. Ananthanarayanan, P. Bodik, L. Golubchik, M. Yu, P. Bahl, and M. Philipose. 2018. VideoEdge: Processing Camera Streams using Hierarchical Clusters. *IEEE/ACM Symposium on Edge Computing* (2018).
[33] Akanksha Jain and Calvin Lin. 2013. Linearizing irregular memory accesses for improved correlated prefetching.. In *MICRO*. 247–259.
[34] Sunita Jain, Nagaradhesh Yeleswarapu, Hasan Al Maruf, and Rita Gupta. 2024. Memory Sharing with CXL: Hardware and Software Design Approaches. arXiv:2404.03245
[35] S. Jamilan, T. A. Khan, G. Ayers, B. Kasikci, and H. Litz. Mar. 2022. APT-GET: profile-guided timely software prefetching. *Proc. of Eurosys Conf.* (Mar. 2022).
[36] N. Jerger, E. Hill, and M. Lipasti. 2006. Friendly Fire: Understanding the Effects of Multiprocessor Prefetches. *ISPASS* (2006).
[37] J. Jiang, G. Ananthanarayanan, P. Bodik, S. Sen, and I. Stoica. 2018. Chameleon: scalable adaptation of video analytics. *Proc. of ACM SIGCOMM Conf.* (2018).

[38] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan. 2017. Heteroos: OS design for heterogeneous memory management in datacenter. *Proc. of ACM ISCA Conf.* (2017).

[39] O. Khattab and M. Zaharia. 2020. ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT. *arXiv:2004.12832* (2020).

[40] J. Kim, S. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti. 2016. Path Confidence based Lookahead Prefetching. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[41] 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. https://doi.org/10.1109/MICRO.2016.7783763

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

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

[44] 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 *The International Symposium on Memory Systems* (Washington, DC, USA) *(MEMSYS 2020)*. Association for Computing Machinery, New York, NY, USA, 323–334. https://doi.org/10.1145/3422575.3422804

[45] Miryeong Kwon, Sangwon Lee, and Myoungsoo Jung. 2023. Cache in Hand: Expander-Driven CXL Prefetcher for Next Generation CXL-SSD. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems* (Boston, MA, USA) *(HotStorage '23)*. Association for Computing Machinery, New York, NY, USA, 24–30. https://doi.org/10.1145/3599691.3603406

[46] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, G. Thelen, K. A. Yurtsever, Y. Zhao, and P. Ranganathan. 2019. Software-defined far memory in warehouse-scale computers. *Proc. of ACM ASPLOS Conf.* (2019).

[47] Gyusun Lee, Wenjing Jin, Wonsuk Song, Jeonghun Gong, Jonghyun Bae, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. 2020. A Case for Hardware-Based Demand Paging. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 1103–1116. https://doi.org/10.1109/ISCA45697.2020.00093

[48] B. Lepers and W. Zwzenepoel. 2023. Johnny Cache: the End of DRAM Cache Conflicts (in Tiered Main Memory Systems). *USENIX OSDI* (2023).

[49] Philip Levis, Kun Lin, and Amy Tai. 2023. A Case Against CXL Memory Pooling. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks* (<conf-loc>, <city>Cambridge</city>, <state>MA</state>, <country>USA</country>, </conf-loc>) *(HotNets '23)*. Association for Computing Machinery, New York, NY, USA, 18–24. https://doi.org/10.1145/3626111.3628195

[50] 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. https://doi.org/10.1145/3575693.3578835

[51] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. 2015. SparkBench: A Comprehensive Benchmarking Suite for in Memory Data Analytic Platform Spark. In *Proceedings of the 12th ACM International Conference on Computing Frontiers* (Ischia, Italy) *(CF '15)*. ACM, New York, NY, USA, Article 53, 8 pages. https://doi.org/10.1145/2742854.2747283

[52] Y. Li, S. Ghose, J. Choi, J. Sun, H. Wang, and O. Mutlu. 2017. Utility-based hybrid memory management. *Proc. of IEEE Int.l Conf. on Cluster Computing (CLUSTER)* (2017).

[53] M. Maas, D. Andersen, M. Isard, M. Mahdi, K. McKinley, and C. Raffel. 2022. Combining machine learning and lifetime-based resource management for memory allocation and beyond. *Communications of ACM* (2022).

[54] H. A. Maruf and M. Chowdhury. 2020. Effectively Prefetching Remote Memory with Leap. *Proc. of In USENIX ATC* (2020).

[55] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan. June 2022. TPP: Transparent Page Placement for CXL-Enabled Tiered Memory. *arxiv.org2206.0287v1* (June 2022).

[56] A. Navarro-Torres, B. Panda, J. Alastruey-Benede, P. Ibanez, V. Vinals-Yufera, and A. Ros. 2022. Berti: an accurate local-delta data prefetcher. *ACM/IEEE MICRO* (2022).

[57] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, and Mendel Rosenblum. 2010. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *ACM SIGOPS Operating Systems Review* 43, 4 (2010), 92–105.

[58] R. Pagh and F. F. Rodler. 2001. Cuckoo Hashing. *Algorithms, Lecture Notes in CS* (2001).

[59] B. Panda. 2023. CLIP: Load Criticality based Data Prefetching for Bandwidth-constrained Many-core Systems. *ACM/IEEE MICRO* (2023).

[60] 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. https://doi.org/10.1145/1964218.1964225

[61] A. Ros. June 2021. Blue: A timely IP-based data prefetcher. *1st ML-based Data Prefetching Competition* (June 2021).

[62] Kaushik Roy, Indranil Chakraborty, Mustafa Ali, Aayush Ankit, and Amogh Agrawal. 2020. In-Memory Computing in Emerging Memory Technologies for Machine Learning: An Overview. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC18072.2020.9218505

[63] S. Rumble, A. Kejriwal, and J. Ousterhout. Feb. 2014. Log-structured Memory for DRAM-based Storage. *Proc. of Usenix FAST Conf.* (Feb. 2014).

[64] 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. https://doi.org/10.1109/ISPASS.2016.7482078

[65] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. 2018. Legoos: A disseminated, distributed OS for hardware resource disaggregation. *Proc. of USENIX OSDI* (2018).

[66] Debendra Das Sharma. 2022. Compute Express Link®: An open industry-standard interconnect enabling heterogeneous data-centric computing. In *2022 IEEE Symposium on High-Performance Interconnects (HOTI)*. 5–12. https://doi.org/10.1109/HOTI55740.2022.00017

[67] Z. Shi, A. Jain, K. Swersky, M. Hashemi, P. Ranganathan, and C. Lin. Apr. 2021. A hierarchical neural model of data prefetching. *Proc. of ACM ASPLOS Conf.* (Apr. 2021).

[68] M. Shreedhar and G. Varghese. 1996. Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on Networking* 4, 3 (1996), 375–385. https://doi.org/10.1109/90.502236

[69] Susav Shrestha, Narasimha Reddy, and Zongwang Li. 2024. ESPN: Memory-Efficient Multi-vector Information Retrieval. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on Memory Management* (Copenhagen, Denmark) *(ISMM 2024)*. Association for Computing Machinery, New York, NY, USA, 95–107. https://doi.org/10.1145/3652024.3665515

[70] Dimitrios Skarlatos, Umur Darbaz, Bhargava Gopireddy, Nam Sung Kim, and Josep Torrellas. 2020. BabelFish: Fusing Address Translations for Containers. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 501–514. https://doi.org/10.1109/ISCA45697.2020.00049

[71] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. 2007. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. 63–74. https://doi.org/10.1109/HPCA.2007.346185

[72] Jovan Stojkovic, Namrata Mantri, Dimitrios Skarlatos, Tianyin Xu, and Josep Torrellas. 2023. Memory-Efficient Hashed Page Tables. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 1221–1235. https://doi.org/10.1109/HPCA56546.2023.10071061

[73] C. Tirumalasetty, C-C. Chou, A. L. N. Reddy, P. Gratz, and A. Abouelwafa. 2021. Reducing minor page fault overheads through enhanced page walker. *ACM TACO* (2021).

[74] 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).

[75] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. 2017. Attention is all you need. *Proc. of NIPS Conf.* (2017).

[76] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J Franklin, B. Recht, and I. Stoica. 2017. Drizzle: Fast and adaptable stream processing at scale. *Proc. of ACM SIGOPS Conf.* (2017).

[77] Guohui Wang and TS Eugene Ng. 2010. The impact of virtualization on network performance of amazon ec2 data center. In *INFOCOM, 2010 Proceedings IEEE*. IEEE, 1–9.

[78] W.P.Chen, A. Rudoff, and R. Agarwal. Mar. 2022. Dynamic Multilevel Memory System. *US Patent 20220229575* (Mar. 2022).

[79] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee. 2019. Nimble page management for tiered memory systems. *Proc. of ACM ASPLOS Conf.* (2019).

[80] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (San Jose, CA) *(NSDI'12)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2228298.2228301

[81] Q. Zhang, P. Bernstein, D. Berger, and B. Chandramouli. Dec. 2021. Redy: remote dynamic memory cache. *Proc. of VLDB* (Dec. 2021).