PreFAM: Understanding the Impact of Prefetching in Fabric-Attached Memory Architectures

Vamsee Reddy Kommareddy
University of Central Florida
Orlando, Florida, USA
vamseereddy8@knights.ucf.edu

Jagadish Kotra
Advanced Micro Devices
USA

Clayton Hughes
Sandia National Labs
New Mexico, USA
chughes@sandia.gov

Simon David Hammond
Sandia National Labs
New Mexico, USA
sdhammo@sandia.gov

Amro Awad
University of Central Florida
Orlando, Florida, USA
amro.awad@ucf.edu

ABSTRACT

With many recent advances in interconnect technologies and memory interfaces, disaggregated memory systems are approaching industrial adoption. For instance, the recent Gen-Z consortium focuses on a new memory semantic protocol that enables fabric-attached memories (FAM), where the memory and other compute units can be directly attached to fabric interconnects. Decoupling of memory from compute units becomes a feasible option as the rate of data transfer increases due to the emergence of novel interconnect technologies, such as Silicon Photonic Interconnects.

Disaggregated memories not only enable more efficient use of capacity (minimizes under-utilization) they also allow easy integration of evolving technologies. Additionally, they simplify the programming model at the same time allowing efficient sharing of data. However, the latency of accessing the data in these Fabric Attached disaggregated Memories (FAMs) is dependent on the latency imposed by the fabric interfaces. To reduce memory access latency and to improve the performance of FAM systems, in this paper, we explore techniques to prefetch data from FAMs to the local memory present in the node. We realize that since the memory access latency is high in FAMs, prefetching a cache block (64 bytes) from FAM can be inefficient, since the possibility of issuing demand requests before the completion of prefetch requests, to the same FAM locations, is high. Hence, we explore predicting and prefetching FAM blocks at a distance; prefetching blocks which are going to be accessed in future but not immediately. We show that, with prefetching, the performance of FAM architectures increases by 38.84%, while memory access latency is improved by 39.6%, with only 17.65% increase in the number of accesses to the FAM, on average. Further, by prefetching at a distance we show a performance improvement of 72.23%.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MEMSYS 2020, September 28-October 1, 2020, Washington, DC, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8899-3/20/09…$15.00
https://doi.org/10.1145/3422575.3422804

CCS CONCEPTS

• **Computer systems organization → Heterogeneous (hybrid) systems**.

ACM Reference Format:
Vamsee Reddy Kommareddy, Jagadish Kotra, Clayton Hughes, Simon David Hammond, and Amro Awad. 2020. PreFAM: Understanding the Impact of Prefetching in Fabric-Attached Memory Architectures. In *The International Symposium on Memory Systems (MEMSYS 2020), September 28-October 1, 2020, Washington, DC, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3422575.3422804

1 INTRODUCTION

For over a decade, servers, wherein hardware resources are tightly coupled, have been the linchpin for implementing data centers. In such systems each server maintains various hardware resources, mainly processing elements, memory and storage, exclusively. This restricts the growth of distinct resources since the capabilities of the resources are complimentary to each other. To overcome this a new revolutionary architecture design, *disaggregated systems* is explored and is becoming promising considering both academic and industrial adoption. HPE Labs' *The Machine* [11], Facebook's Disaggregated Rack [58], Intel's Rack Scale Architecture [37] are some of the prominent prototypes for disaggrgatted memory systems. In such architectures, different resources are decoupled and are connected using fast interconnects [9, 13, 15, 22, 54]. For instance, Intel's Omnipath [9] allows for packing of CPU and NIC within the same chip to connect to the remove memory or storage over fabric networks pioneering memory-centric, fabric attached memory (FAM), systems. This allows for easy integration of evolving technologies, effective memory sharing avoiding memory under-utilization, has potential to simplify programming model and, most importantly, permits expansion of resources autonomously.

Emerging Non-Volatile Memories (NVMs) are considered among the best candidates for building memory-centic systems for several reasons. NVMs are expected to have capacities in terabytes per processor socket [38, 39, 42, 52]. NVMs can be used to host directly-accessible filesystems, e.g., Linux's Direct Access for Files (DAX) support[2] allowing efficient operation on shared files. With scaling up the total memory capacity of data centers to petabytes, idle power becomes a major concern, and thus DRAM becomes a less practical solution given its significant refresh power and hence high cooling and operational costs. Also, due to the slow and limited

endurance nature of NVM writes, they are expected to be used as an additional layer in the memory hierarchy. Hence, having small DRAM-based local memory within nodes while NVM being used as a lower-tier memory attached over fabric is a more natural design point [11, 42–44].

Although FAM architectures improve memory utilization and reduce maintenance cost, given that the memory is attached over fabric network and also NVM is considered as a best candidate for the remote memory, the latency in accessing memory is a prime concern for such architectures. Further, FAM is accessed by a number of computing units and hence the latency in accessing memory is also dependent on the number of computing units accessing FAM.

To hide memory access latency and to improve the performance of the system, in this paper, we explore prefetching for FAM architectures (PreFAM). A prefetcher is implemented in the local node's memory controller to prefetch data from the off-node FAM. We dedicate a small part of the local memory to store prefetched FAM data (i.e., as a prefetch buffer). Prefetching in FAM architectures pose unique challenges due to the nature of shared global memory and fabric; thus conservative prefetching schemes are required. Moreover, the long latency for prefetching data from the farbic-attached global memory can easily render many prefetching schemes ineffective, i.e., fail to hide reasonable amount of latency before the actual access is triggered. Thus, FAM-friendly prefetching schemes need to be simple and accurate.

The accuracy of the prefetcher depends on the prefetching mechanism that detects possible future memory accesses. Address map pattern prefetcher [29] proposed in the past is accurate and successful in detecting predictable address streams. Hence we adopted address map pattern prefetcher in the memory controller to detect multiple memory access patterns of the benchmarks. To increase the prefetching accuracy, rather than choosing the nearest prefetchable candidate among various candidates, as done in address map pattern prefetcher, we choose the best prefetchable candidate. This is achieved by training the prefetcher before issuing prefetch requests.

Even though the accuracy of the prefetcher is high, we observe that because of high memory access latency, the benchmark might issue demand requests to the prefetched locations before the prefetch request is completed. Thus, the usage of the prefetched blocks decreases. To solve this, we explored distance factor based prefetching. In distance factor prefetching scheme, the prefetcher brings FAM data blocks at a distance of which they will be accessed in the near future (at a distance) instead of bringing the block that will be accessed next[1].

Prefetching data before the actual memory access and placing it in the faster memory is an approach explored intensively in previous studies [40, 49, 50]. Although prefetching is widely explored at both cache side [49], memory side [50] and also for different memory architectures [40], to the best of our knowledge, this is the first paper to study and investigate the impact of prefetching in FAM architectures, and discuss the effectiveness of current prefetching schemes and if suitable for FAM systems. Moreover, we discuss how to tune the parameters of different state-of-the-art prefetching schemes to better suit FAM architectures.

To evaluate our approach, we use the Structural Simulation Toolkit (SST)[51], a publicly available architectural simulator. We utilize the existing decoupled memory model Opal[35], to model our optimization. We developed a prefetcher component and attached it to the memory controller. For every memory access, the prefetching component is fed with the accessed memory address and this component either trains or issues prefetch requests to FAM. Also before forwarding the request to the FAM we modified the memory component unit of SST to check if the data is already prefetched to the local memory.

Our contributions in this paper are as follows:

- We investigate the impact of prefetching blocks from FAM memory to local memory, and effectiveness of current techniques.
- Further, we analysed potential drawbacks of prefetching in FAM, and proposed distance factor prefetching to improve the timeliness of prefetching.
- On average, with prefetching, we improve the memory access latency by 39.6% and improve the performance of the FAM system by 38.84% with 17.65% increase in the number of accesses to the FAM. Further, by prefetching at a distance we show a performance improvement of 72.23%.

The organization of the paper is as follows. First, we discuss disaggregated memory and prefetching models background in Section 2. Section 3 explores our proposed prefetching design for FAM architectures (PreFAM). Methodology and applications evaluated are discussed in Section 4. Section 5 analyses results. We provide Section 6 to discuss possible alternatives to our approach. Finally, we discuss the related work in Section 7 and conclude in Section 8.

2 BACKGROUND

In this section firstly, we briefly discuss drawbacks of in-node memory architectures then we give a brief primer on FAMs covering its feasibility and benefits. Finally, since this paper is about techniques for effective prefetching in FAMs, we shed some light on the prior prefetching schemes.

2.1 Tightly Coupled In-node Resources

Traditionally, resources are tightly coupled within the nodes, and the resources are dedicated to the components within the node. However tightly coupled resources restrict capabilities of individual resources. The performance and scalability of such systems is limited to an inefficient resource within the node. For instance, although the computation capability of the processing elements in a node are improved significantly, memory is still a bottleneck. Available memory technologies like DRAM are not suitable to scale up the memory considering the power and real estate consumed by such memory types [41]. Also, many DRAM modules are prevented from being connected to the memory channel for signal integrity [34] preventing memory to scale. Moreover, memory utilization relies on application requirements. Recent studies show that almost 80% of the tasks overestimate their memory requirements on HPC systems[3]. Hence, the behavior of the applications might lead to

[1]Note that the distance factor based prefetching mentioned here differs from distance prefetching[33]

under-utilization of memory if a large amount of memory is dedicated for the applications [46]. In addition, such tightly coupled in-node memory architectures require complicated programming models to access remote memory.

2.2 Decoupled Fabric Attached Resources

In contrast to the tightly coupled resource architectures, recent studies show that decoupling of resources as a potential solution[4, 5, 11, 36, 43–45, 53] to address the challenges when resources are tightly coupled. In such architectures, various resources are decoupled from each other and are managed independently (Figure 1). Memory regions of the decoupled memory are allocated on demand and is allowed to be utilized by the co-located nodes avoiding memory under-utilization. Further, it simplifies programming models by providing load/store access to the decoupled memory. Migrating jobs is one of the crucial requirements for hybrid cloud systems. With disaggregation, such requirement can be easily achieved by simply transferring the process metadata to the target node, avoiding expensive data transfers associated with memory migration.

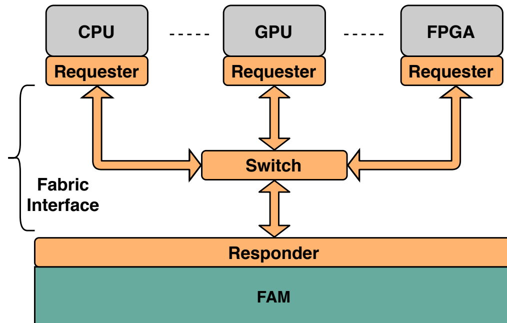


Figure 1: Fabric attached memory architecture.

The essential prerequisite to make such memory-centric decoupled memory architecture feasible are a) fast memory semantic interconnect and b) scalable memory.

a) Memory-semantic fast fabric interconnect: To connect decoupled resources to each other a fast and reliable fabric interconnect is required. In recent times more attention is given to develop fast interconnects using fabrics. Various fabric providers released specifications discussing different aspects to provide a standard fabric protocol, Gen-Z [14], CCIX [13], CXL [54] etc. For instance, Gen-Z is a new interconnecting standard which provides memory semantic interface allowing resources to be attached directly to the computing systems and accessing such fabric attached remote memory using traditional load/store operations. Fabric interconnect also provides reliability by providing multi-path connectivity to the remote memory.

b) Memory Scalability: Another important requirement for FAM is dense memory and ease of scalability, since FAM serves requests from various computing resources, unlike in-node architectures wherein memory is granted only to the tightly coupled nodes. To support scalability, memory is organized in a rack-scaled manner [1]. This allows easy integrating of additional memory pools to scale up memory. Further such a memory organization and the universal nature of the fabric interconnects supports different memory types. However, NVMs in their different variants i.e, 3D Xpoint, PCM, ReRAM [25, 59, 60] are suitable as main memory due to high density and endurance. Moreover, NVMs are suitable to host direct access filesystems (DAX) [2] which provides efficient file sharing between the compute units and avoids costly page caching to process files [16].

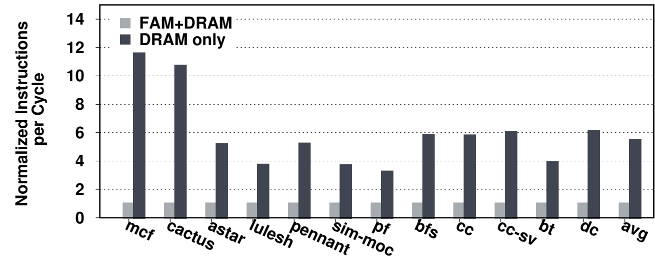


Figure 2: Maximum improvement possible if all the memory requests are served by the local memory (DRAM as main memory).

FAM architectures suffer from severe performance overhead. We evaluated and compared the performance of the system with DRAM as the main memory (`DRAM only`) and dedicated DRAM with shared NVM over a fabric interconnect as the main memory (`FAM+DRAM`), Figure 2. On an average, we observe that compared to systems which has only dedicated DRAM as a main memory, FAM architectures suffer a performance overhead of 5.48x (maximum of 11.58x for *mcf* benchmark), according to our evaluation configuration[2].

2.3 Prefetching Schemes

Prefetching is a mechanism to avert expensive FAM accesses by timely bringing the data from FAM in to local memory. Previously prefetching has been explored extensively at various levels including at the cache and at the memory [6, 30, 49, 50]. Prefetching is explored both at software and hardware levels. In software, prefetching is achieved by placing dedicated prefetch instructions into the code [10, 48]. Software prefetchers, though have control over prefetching they, are often agnostic to the resource contention in the hardware. On the other hand, hardware prefetching schemes require separate hardware structures to predict the prefetchable candidates. There are different variants of hardware prefetchers.

Immediate next block prefetcher: This scheme prefetches next immediate block (cacheline) for each memory access without any prior verification, if the block will be accessed in future or not. Next block prefetcher provides performance for applications which access memory sequentially. However, since there is no prior verification, false positives are higher.

Verification based prefetcher: Unlike prefetching immediate next blocks for every memory access, verification based prefetchers issues prefetch requests based on the previous accesses. Stride prefetching [19], stream buffers [32], distance (delta) prefetching

[2]Refer to Section 4 for evaluation methodology.

[33] and address map pattern prefetching [29] are prominent verification based prefetching schemes. These schemes store recent history of memory accesses and triggers prefech requests when a specific pattern is observed. For instance, assuming 'A' as an address of a block, if 'A' and 'A+3' blocks are accessed contiguously, the stride prefetcher assumes that the memory is accessed in multiples of 3 (stride 3) and prefetches '$A + 3 * N'$ address blocks, where N is the degree of the prefetcher. Distance prefetching is a correlation prefetcher which is based on Markov prefetching[31]. It maintains a correlation table to store the history of address differences (deltas) of any two consecutive misses. When the prefetcher receives any two consecutive misses with a delta difference, then the prefetcher verifies the history with similar delta and issues prefetching requests after calculating addresses using deltas right after the current delta. Address map pattern matching prefetcher identifies multiple prefetchable candidates based on the memory access patterns. Memory access map table is a bit map which stores memory access patterns. Memory is divided into multiple zones and memory access map table records access patterns of only hot zones. Each entry has bits to store access states of the blocks within the hot zone (2 bits per block). Based on the previous accessed block states, prefetchable candidates are chosen. For instance, if blocks 'A', 'A+2', 'A+4' and 'A+5' accessed previously and if 'A+6' is the current accessed block with in the hot zone, the prefetcher identifies two potential candidates to prefetch, 'A+7' and 'A+8' blocks with strides 1 and 2 respectively based on the previous accesses. Among these the nearest candidate is chosen, 'A+7' in this case.

Feedback based dynamic prefetcher: Based on the accuracy of the prefetcher, for each application, the feedback based prefetcher dynamically adjusts prefetch parameters. For instance, the prefetching aggressiveness is varied based on the accuracy of the prefetcher [28, 30, 57].

Trained prefetcher: In the trained prefetcher, first the prefetching algorithm is trained to evaluate the prefetchable candidates. For instance, Sandbox prefetcher [49] issues prefetchable addresses and stores them in the bloom filter during candidate evaluation. For every LLC miss the bloom filter is checked if the prefetcher would have prefetched the current block or not. If the current access would have been prefetched previously, the scores of the prefetchable candidate is incremented. Prefetch requests are issued during prefetch action, if the candidate scores are high.

3 DESIGN

In this section, we describe our prefetching policy for FAMs. Then we explain prefetching blocks at a distance and brief about prefetch buffers. Finally, we discuss the feasibility of memory contiguity in FAMs to enable prefetching blocks beyond a page. However, before diving into the respective topics we first explain the request flow in PreFAM.

3.1 Memory Access Request Flow in PreFAM

In the baseline, for every memory access the memory controller verifies if the request is for the local memory or for the FAM. This is done by verifying the block address of the requester event. Since FAM provides a flat address space we dedicated the lower addresses to the local memory. If the request address falls within the local memory address region, it is identified as a local memory request. When accessing the local memory the request is directly forwarded to the local memory without any delay. If accessing FAM, the memory controller forwards the requests to the media controller of the FAM, Figure 3. In PreFAM, while serving the current memory access the requested block address is also sent to the FAM prefetching unit. This unit verifies and identifies any prefetchable candidates and issues prefetch requests. The media controller unit of the FAM maintains separate queues for prefetch and demand (actual requests sent by the benchmarks) requests, to avoid demand requests being throttled by the prefetch requests. The media controller serves both the prefetch and demand request and responds back to the requesting node. Upon receiving a response from the FAM, the memory controller verifies if the response is a demand response or a prefetch response. A demand response is forwarded to the last level cache and a prefetch block is stored in the temporary prefetch buffer of the memory controller. This prefetched block is stored in the temporary buffer until a new block is prefetched. For the next memory access, the memory controller first inspects the prefetch buffer and if the accessed FAM block is found in the prefetch buffer, the memory controller responds back to the last level cache without issuing a FAM requests. If not found in the prefetch buffer, the request is forwarded to the media controller of the FAM over the fabric network. Concurrently, prefetchable candidates are issued by the prefetching unit if found, based on the current memory access. When the prefething unit receives a response, it moves previously prefetched blocks from the temporary prefetch buffer to the the local memory prefetch buffer and stores the recently prefetched block in the temporary prefetch buffer. Section 3.3 details about temporary and local memory prefetch buffers. While moving the previously prefetched block to the local memory, the prefetch buffer table (PBT) is updated with the location of the local memory where the prefetched block is stored. PBT size depends on the local memory prefetch buffer size. If the size of PBT is large as shown in Figure 3 we move the PBT to the local memory and cache part of the PBT in the memory controller. Henceforth, for every FAM access, the memory controller first inspects the temporary prefetch buffer, then probes PBT to verify if the FAM block is prefetched to the local memory prefetch buffer. If found in PBT, the request address is updated with the local memory address and is forwarded to the local memory. The request is forwarded to the FAM only if the block is not found in the temporary prefetch buffer and local memory prefetch buffer.

3.2 Our FAM Prefetcher

As shown in Figure 3, FAM prefetcher is attached to the memory controller to perform prefetching. Since pattern matching prefetcher is considered as one of the best candidates as it stores memory access states and issues multiple prefetch requests. Address map pattern prefetcher maintains a table to store the states of memory accesses (access, init and prefetch). In our prefetching scheme, memory is divided into multiple zones and the memory map table maintains an entry for each hot zone. This limits prefetchable candidates to only within the hot zones. Similarly, we also divided memory into zones, with 64 blocks per zone. However, instead of storing the states of only hot zones, we store memory access

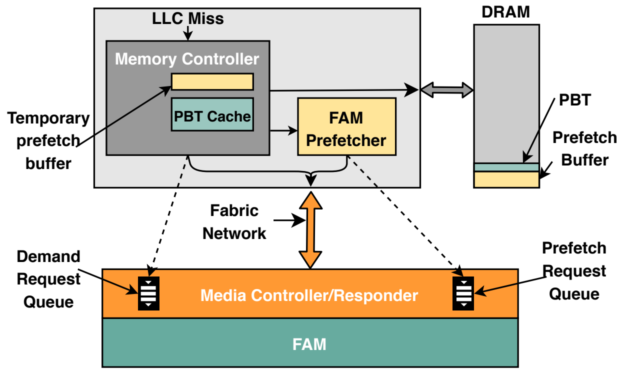
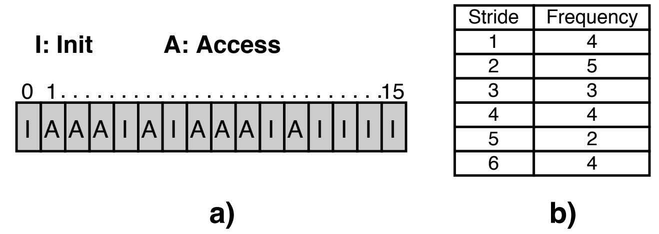


Figure 3: Overview of prefetching in FAM architectures.

states of the entire memory and rely on replacing memory map table entries if not found. Prefetching aggressiveness is defined as the degree of the prefetcher. The number of prefetch requests a prefetcher issues for every memory access to the FAM is based on the prefetch degree.

In FAMs, prefetcher false positives are very costly because the FAM is shared by multiple nodes and an incorrectly calculated prefetched block might starve demand requests from other nodes. Also, part of the local memory is used as a prefetch buffer and the prefetched data has to be stored (written) in the local memory. Hence an incorrectly prefetched block pollutes local memory prefetch buffer. Further, prefetch buffer victim entries are written back to the shared FAM pool, if the victim entry is modified. Therefore, the prefetching accuracy should be higher in PreFAM. To improve the prefetching accuracy we first train the prefetcher and then issue prefetch requests. To do so, our prefetching mechanism constitute of two phases a) training phase and b) prefetching phase.

3.2.1 Observation/training Phase: During the training phase, for every Last Level Cache (LLC) miss the prefetcher trains the memory map table of address map pattern prefetcher and book-keeps specific patterns which are frequently observed to be prefetchable. For instance, considering 16 blocks per entry in memory map table, if the memory accesses are as shown in Figure 4(a) from left to right, we note down the detected prefetchable candidate strides. That is when accessing block 1 the prefetcher checks the previous access with stride 1. Since block 0 is not accessed the prefetcher identifies no prefetchable candidates. Number of strides verified is dependent on the *stride limit* configured during initialization. When block 1 is accessed and when the stride limit is higher than 1, the status of the blocks from the preceding memory map table entry are checked. For now we are assuming that all the preceding entry blocks are in not accessed state. When block 2 is accessed, the prefetcher identifies prefetchable candidate at stride 1. When accessing block 3, the prefetcher identifies prefetchable candidates at stride 1 and 2 based on the previous accesses. The frequency of the prefetchable candidate strides are book-kept in prefetchable stride frequency table, Figure 4(b). After the training phase is completed, during the prefetch phase the prefetcher looks for prefetchable candidates starting with the most frequently identified strides during the training phase. The aggressiveness of the prefetcher and the frequency of the training period are dynamically adjusted based on the accuracy and the usage of the prefetched blocks.



Stride	Frequency
1	4
2	5
3	3
4	4
5	2
6	4

Figure 4: (a) Memory map table entry and (b) prefetchable stride frequency table with stride limit 6

3.2.2 Prefetching Phase: During the prefetching period the prefetcher issues prefetch requests. The blocks to be prefetched are dependent on memory access pattern of the memory map table and identified prefetchable strides book-kept in the frequency table during the observation period. The current memory access map pattern for a memory region is as shown in the Figure 5. For such an access pattern, when block P is accessed, based on the previous accesses, block P_1 and P_2 are identified as prefetchable candidates. Among these, the block which is nearest to the accessed block, P_1, is prefetched in address map pattern matching prefetcher. However, to make sure that the prefetched block is most likely used by the node, the prefetcher in PreFAM, prefetches blocks only with identified prefetched strides during the observation period. In the case of Figure 5, P_2 is prefetched first since stride 2 is identified most frequently (5 times) prefetchable stride, as depicted in Figure 4(b). At that instant, if the prefetch degree is more than 1, next most frequently prefetchable stride (stride 1) is also prefetched, P_1. When multiple strides are identified frequently prefetchable, 1,4 and 6 from Figure 4(b), block with nearest stride is selected. Also, when only fewer blocks in the memory region are accessed, even though a stride is identified frequently prefetchable compared to other strides, it is unlikely that the block is used if prefetched. To eliminate such prefetchable strides, we only consider strides whose frequency is greater than *stride frequency threshold* percentage of the observation period. For instance, if *stride frequency threshold* is set to 25% then according to Figure 4(b) only stride 2 is identified as useful. This is because the frequency of stride 2 of 5 times, which is greater than 25% of observation period (16 memory accesses). Even though the prefetching degree is more than 1, the prefetcher does not prefetch any other blocks if the identified prefetchable strides are as shown in Figure 4(b).

3.2.3 Distance Factor Prefetching. When address map pattern prefetcher, with training and prefetch phases, is directly deployed in FAM systems, the benefits of the prefetcher are limited. This is because the prefetcher predicts and prefetches the next immediate accessible block and due to high memory access latency in FAMs, a demand request might be issued before the prefetch request, issued to the same memory location, is completed. Hence the prefetched block might not be used by the benchmarks and the accuracy of the prefetched blocks decreases.

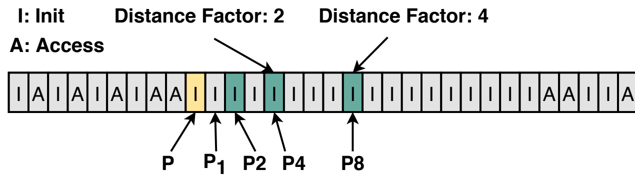


Figure 5: Memory map table entry and prefetchable blocks with and without distance factor.

Hence, to make effective use of prefetching in FAMs we explore prefetching blocks at a distance. That is, instead of prefetching next immediate accessible block we prefetch blocks which might be accessed by the benchmarks in future (at a distance) but not immediately. We achieve this by introducing *distance factor*. For instance, if the distance factor is 2 and the most frequently identified prefetchable stride is 2, according to Figure 4, the prefetcher prefetches block P_4 and if the distance factor is 4, block P_8 is prefetched instead of P_2, Figure 5.

3.3 Prefetch Buffers and Metadata Structures

As shown in Figure 3, the prefetched data is stored at two buffers a) local memory buffer and b) temporary buffer.

Local memory prefetch buffer: We carve out some space from the local memory for storing prefetched blocks and the respective metadata. This carving is done in the OS and the carved out space is managed by the hardware. PBT manages the metadata of the prefetch buffer. Both PBT and local memory prefetch buffer are managed as set-associative caches in the local memory. Each PBT entry contains multiple FAM addresses which are prefetched to the local memory. If the FAM address is found in PBT, the local memory address is calculated and request is forwarded to the local memory prefetch buffer. While adding a prefetch block to the local memory, PBT replacement bits (LRU) are used to replace an entry in the PBT and the buffer. Since PBT is checked for every FAM access, for a faster access to PBT it is cached in the memory controller.

Temporary prefetch buffer: This is a buffer maintained in the memory controller to store prefetched blocks temporarily before they are written to the prefetch buffer in the local memory. Since the prefetched blocks are supposed to be accessed by the benchmarks next, after prefetching is completed, instead of immediately storing them in the local memory prefetch buffer, it would be beneficial to store them temporarily in the memory controller. The temporary buffer is used for this case. The number of entries in the temporary buffer are limited to prefetch degree (4). When the prefetcher prefetches newer blocks the temporary buffer entries are moved to the local memory prefetch buffer.

3.4 Media Controller Modifications

Media controller abstracts memory media of FAM. It maintains a queue to execute requests one at a time and returns responses back to the requester. With FAM prefetcher the media controller request queue also hosts prefetch requests. This throttles on-demand requests. Hence, to avoid this, we added a separate prefetch queue to host prefetch requests coming from different nodes. Further to avoid demand request throttling, priority is given to the demand requests i.e, prefetch requests are served only when all the demand requests are served.

3.5 Feasibility of Prefetching Blocks over Page Boundary

Prefetchers usually disable prefetching blocks over the page boundary. However, since prefetching is implemented at the memory side, some of the benchmarks access memory with strides beyond page limit. For instance, for a block size of 64 (Bytes), a 4KB page has 64 blocks. With such a configuration when the pages are allocated contiguously, according to our evaluation setup, we observe that 46% of memory accesses are with stride 66 for cactus benchmark. Also when prefetching blocks over page boundary the prefetcher can identify more prefetchable candidates. To enable prefetching blocks beyond the page size the pages should be contiguously allocated. For instance, virtual pages vpn0 and vpn1 should translate to physical pages ppn0 and ppn1 (ppn1 is immediate next page to ppn0).

To allocate 4KB pages contiguously, first option is to allocate huge pages. However, since local memory is limited, allocating huge pages will host lesser number of applications. This will also have a negative impact on the performance of the applications which rely mostly on the local memory. Further, this wastes memory since for most of the time the entire allocated large page is not used by the benchmarks. Second method is to allocate memory at 4KB pages and relying on Linux memory allocator to allocate contiguous pages. Previous work [47] showed that Linux memory buddy allocator aims at allocating physical pages that are close to each other on each allocation request. Using such memory allocators there is a high chance that the subsequent physical pages are related. Third option is to allocate local memory at 4KB page size and allocating FAM regions (2MB for instance).

In all the above cases if page migration is enabled [36, 44], physical pages will no longer be contiguous with respect to the virtual pages after the pages are migrated. However, we note that FAM architectures require a second level address translation to access FAM [43] i.e, virtual addresses are converted to node addresses and node addresses are converted to the FAM addresses (node to physical address translator), Figure 6. Hence for such systems, when the pages are migrated to the local memory or moved from one location to the other location within the FAM, only node to FAM address mapping is modified but virtual to node address mapping is unchanged. For instance, when node physical address (npn2) is migrated to the local memory the node to physical address mapping is modified from npn2:ppn2 to npn2:ppnX, in the translation unit. However the virtual address to node address mapping, vpn2:npn2, is not modified. Hence memory contiguity is preserved with node addresses even when the pages are migrated. In this case the prefetcher operates on node addresses instead of actual FAM physical addresses i.e, the prefetcher keeps status of previous memory accesses tagged with node addresses and also predicts and prefetches blocks tagged with node addresses. For virtual address to node address contiguity we rely on buddy allocator of the Linux operating system to preftch blocks over the page boundary.

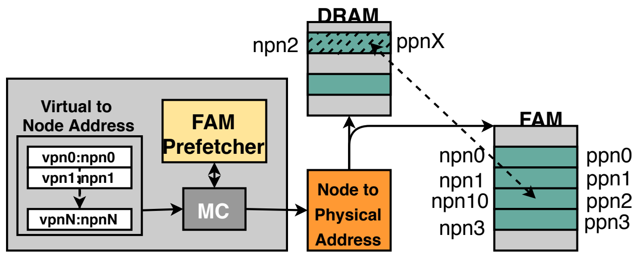


Figure 6: FAM prefetcher schematic with two-stage address translation based FAM design.

4 METHODOLOGY

We evaluated our design in Structural Simulation Toolkit (SST) [51]. SST is a cycle accurate event based architectural simulator. SST is a modular based design in which each component is implemented as a module and the modules are linked to construct and evaluate the system. Kommareddy et al. evaluated dis-aggregated memory model by developing a centralized memory manager model, Opal [35]. This model emulates memory management for scaled architectures, [12, 21]. We utilize this memory manager to implement our design. We modeled prefetcher component and attached it to the memory controller component.

System configuration used in our evaluation is shown in Table 1. The system is emulated with 4 cores and two instructions are served per cycle. Three levels of caches L1, L2 and L3 are evaluated with sizes 32KB, 256KB and 1MB, respectively. 256MB of DRAM is used as local memory [53] and 16GB of NVM is used as FAM. In real systems global FAM is in TBs or PBs and local memory is in GBs. Considering the simulation speeds, we evaluated our proposal with reduced memory sizes. However, although the performance and the results vary in numbers in the real system, our approach will still show similar trends. After recent projections on the fabric network interconnects [13, 15] and considering fabric latency assumed by recent works [20, 53, 56], we simulated fabric network to connect to dis-aggregated NVM with a network latency of 500ns. Local memory and FAM are allocated at 1:4 ratio wherein 80% of the memory required by the benchmarks is allocated in the FAM.

In our design, we used a part of the local memory to act as prefetch buffer. 1MB of local memory (<0.5%) is reserved for prefetch buffer. Number of entries (or) size of PBT is dependent on the size of the prefetch buffer and prefetching granularity. For the given prefetch buffer size (1MB), and for the prefetching granularity of cacheline size (64B), the prefetch buffer can store 16K prefetch blocks and the number of PBT entries required is 16K. In this case, each PBT entry is 64 bits, for storing the prefetched FAM address, replacement and dirty bits. With such a prefetch buffer size and prefetch granularity, the metadata, PBT, required is 128KB. Hence 128KB of local memory is also reserved for PBT. To reduce the search latency, PBT is maintained as a set-associative cache and part of it is cached in the media controller (1KB) Memory map table size is 1KB. During the training period for every access 128 strides are verified which cover up to 2 pages. Epoch frequency to dynamically adjust prefetch parameters is 1000 memory accesses.

Table 1: System Configuration

Node	
CPU	4 Out-of-Order cores, 2GHz, 2 issues/cycles, 32 max. outstanding requests
L1	Private, 64B blocks, 32KB, LRU
L2	Private, 64B blocks, 256KB, LRU
L3	Shared, 64B blocks, 1MB, LRU
Local memory	DRAM, Size: 256MB
Number of nodes	up to 8
Fabric Network	
Latency	500ns
Fabric Attached Memory	
Size	16GB
Latency	Read 60ns, Write 150ns
Outstanding requests	256
Number of FAM pools	up to 8
Prefetching Unit	
Local memory prefetch buffer	1MB
Prefetch buffer table	128KB
Stride frequency threshold	50%
Degree	4
Training period	100 memory accesses

Since we are focusing on HPC benchmarks, we evaluated 12 HPC benchmarks from various benchmark suits. Mcf, Cactus and Astar benchmarks are from SPEC 2006 benchmark suite[26] which is widely used in both industry and academia. Lulesh and Pennant [18] are mini-apps for unstructured hydrodynamics and mesh physics implemented for advanced architecture research. SimpleMoC (*simmoc*) [23] is another mini-app which characterises the performance and demonstrates the feasibility of the Method of Characteristics (MOC) in 3D neutron transport calculations for full scale light water reactor simulation. Path finder (*pf*) [27] is from Mantevo benchmark suite, which searches for signatures within the directed and cyclic graphs. Breadth-first search (*bfs*) and connected components; based on Afforest sub-graph sampling algorithm (*cc*) and Shiloach-Vishkin algorithm (*cc-sv*), benchmarks are from Intel GAP[8] benchmark suite [55]. Data cube (*dc*) is from NAS benchmark suite [7] which showcases grid capabilities to handle large distributed data set. Each of these benchmarks are evaluated for a minimum of 100 million instructions per core. Since 4 cores are evaluated per node, in total each benchmark is evaluated for up to 400 million instructions.

5 EVALUATION

In this section, we first show the impact of prefetching on memory access latency and off-node traffic. Then we discuss prefetching accuracy, possible improvement by prefetching using distance factor. Finally, we discuss overall performance improvement with one and multiple nodes.

5.1 Impact of Prefetching on Average Memory Access Latency

When FAM blocks are prefetched to the local memory, the memory latency while accessing (both writes and reads) the prefetched FAM locations eliminates fabric network latency and also eliminates NVM latency. Figure 7 shows normalized delay per request to fetch the data to the LLC. The baseline is the latency observed without prefetching. With PreFAM, the delay in fetching the data to the LLC reduces to 0.71x on an average. The maximum reduction is observed for sequential memory access benchmarks *lulesh*, *pennant*, *cc* and *bfs* benchmarks (0.54x, 056x, 0.56x and 0.58x respectively). These benchmarks are moderately memory intensive with sequential accesses to the memory. For such a memory access the accuracy of the prefetcher is high, Section 5.4, and hence most of the memory accesses are served by the prefetch buffer. *cc-sv* benchmark also accesses memory sequentially and the prefetching accuracy is high, however, since the memory is accessed intensively, the reduction in memory access latency is 0.73x.

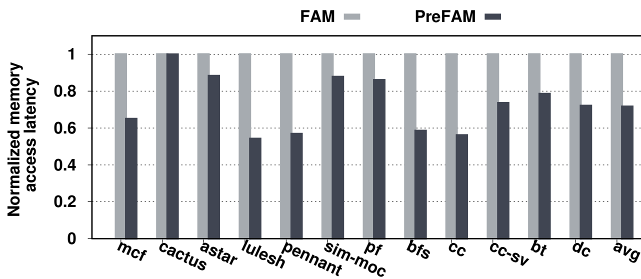


Figure 7: Average memory access latency observed at the LLC of the node.

5.2 Impact of Prefetching on Off-Node Traffic

As asserted in Section 3, for every FAM access by the benchmarks, based on the prefeching degree we either issue one or more prefetch requests, in prefetching phase, when multiple strides are identified as useful. Thus, although memory access delay is improved, the number of total requests (demand and prefetch) sent to the FAM and hence off-node traffic might increase and the prefetcher should be conservative while issuing requests to the FAM. Figure 8 shows off-node traffic generated by a node while accessing FAM. We observe that for the evaluated benchmarks, on an average, the number of requests sent to the FAM increases by 17.65%. Hence, PreFAM increases off-node traffic by 17.65% to improve the memory access latency by 39.60% as shown in Figure 7.

5.3 Number of Demand Requests Issued for ongoing Prefetches

While prefetching FAM blocks to the local memory, it is possible that the application might issue demand requests, before the prefetch request to the same location is completed. Specifically in FAMs, since the memory access latency is higher, the number of such scenarios can be higher. Also, to avoid throttling demand requests, priority is given to the demand requests over prefetch requests

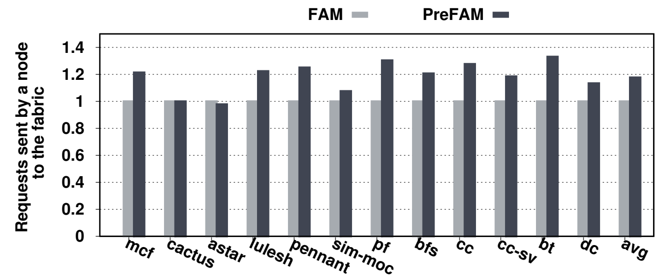


Figure 8: Off-node traffic. Total number of requests (demand and prefetch) sent by a node to the FAM via fabric interconnect.

in the media controller. This increases the latency to prefetch a block and hence increases the chances of issuing demand requests before the prefetch request to the same location is completed. Figure 9 shows the fraction of demand requests issued to FAM whose locations (blocks) are being prefetched but not yet completed. For instance, for *astar*, we observe that with PreFAM this fraction is 0.70 i.e, for every prefetch request issued there are 70% of the chances that the demand request will be issued before the prefetch requests is completed. This is one of the reasons for not able to achieve significant reduction in memory access delay.

To reduce such a fraction we prefetch FAM blocks at a distance from the current identified prefetchable FAM location using distance factor. 'DF' indicates distance factor. For instance 'DF:2' indicates PreFAM with distance factor 2. We varied the distance factor from 2 to 16 and we note that as the distance factor increases this fraction decreases. For instance, for *astar* benchmark this fraction reduces from 0.7 to 0.1, with just PreFAM and PreFAM with distance factor 16, respectively. However, as the distance factor increases although this fraction decreases, for some workloads, the usage of the prefetched blocks might decreases, Section 5.4.

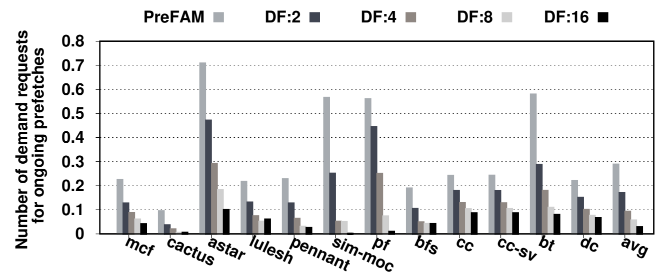


Figure 9: Probablity of a demand request issued to FAM before the prefetch request to the same memory location is completed.

5.4 Usage of Prefetched Blocks

Prefetching FAM blocks improves the performance or is beneficial only when the prefetched blocks are used by the benchmarks. Figure 10 shows the usage of the prefetched blocks. For instance, for *pennant* benchmark the chances of utilizing a prefetched block is 75.6% with PreFAM. However, when prefetcing blocks at a distance,

using distance factor, the probability of using a prefetched block decreases to 74%, 66.29%, 54.17% and 47.81% when the distance factor is 2, 4, 8 and 16 respectively for *pennant* benchmark. This decrease is because, since we are far fetching the blocks the prefetching unit accuracy is less. Specifically *lulesh* benchmark for most of the time access memory blocks with stride 64. And when the distance factor is 16, the prefetching unit, prefetches block $16 * 64$ from the current accessed block. This eliminates performance improvements achieved by not introducing the distance factor. Distance factor is a variable which works differently for different benchmarks. That is, for benchmarks *astar,sim-moc* and *bfs*, when distance factor is 2 the usage of the prefetched blocks is more. This has a positive impact on the performance. But when the distance factor is further increased the prefetched block usage decreases.

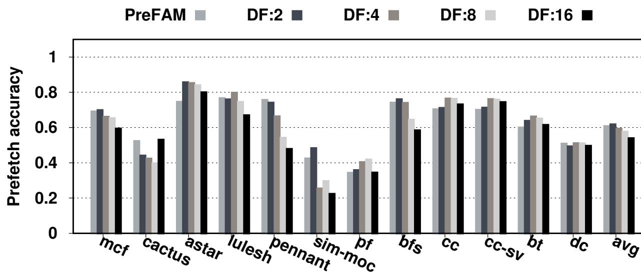


Figure 10: Probablity of a prefetched FAM block being used atleast once.

5.5 Overall Performance Improvement

Figure 11, shows overall performance improvement with PreFAM and with PreFAM plus distance factor. On an average the performance improves by 38.84% with PreFAM (maximum of 90.9% for *cc* benchmark). As the distance factor increase the average performance also increases because a) the prefetched block usage increases with distance factor. b) the number of total requests issued to FAM increases when the distance factor increases. When the distance factor is small, for two consecutive LLC misses the prefetcher might generate repeated prefetch candidates and the repeated requests are cancelled. However when the distance factor is high the repetitions are less and, hence, the number of prefetch requests issued to FAM are more. Thus, the performance increases from 38.84%, with PreFAM only, to 72.23%, with a distance factor 16. This increase in performance is at the cost of more number of requests issued to the FAM, 17.65% without distance factor to 33.64% with distance factor 16. A maximum performance improvement of 3.24 is achieved for *cc* benchmark with 45.56% increase in requests sent to FAM.

5.6 Temporary Prefetch Buffer Hit Rate

Since the prefetched block is supposed to be accessed immediately, based on the prediction, we maintained a temporary prefetch buffer in the memory controller to avoid accesses to DRAM. The size of the temporary prefetch buffer is equal to the degree of the prefetcher (4). The hit rate of the temporary prefetch buffer is shown in Figure 12. The hit rate is very small (4.15% on an average) because this buffer

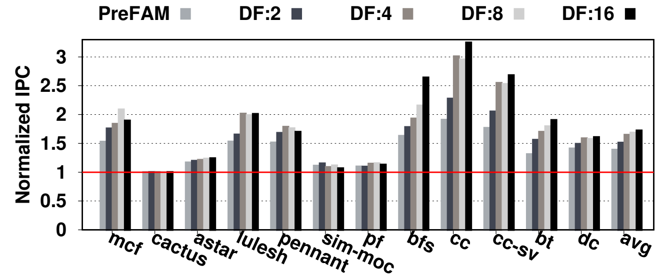


Figure 11: Performance improvement using prefetching with respect to the baseline.

is very small and is very frequently replaced by the prefetched requests.

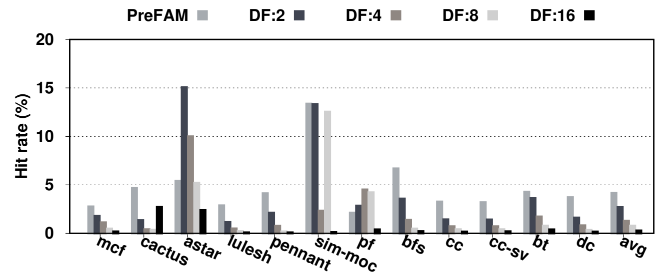


Figure 12: Temporary prefetch buffer hit rate.

5.7 Sensitivity Analysis

5.7.1 Impact of Memory Allocation Policy. Performance of the applications depends on the number of pages allocated in the local memory and in the FAM. When more pages are allocated in the local memory the performance will be better. We allocated local memory and FAM at 1:4 ratio. However, in this section we evaluate performance improvement with PreFAM when memory allocation ratio is varied, Figure 13. We varied the ratio from 1:1 to 1:16. When more pages are allocated from FAM (1:16) the latency in accessing the data increases and hence the performance degrades. When the data is prefetched the performance improvement increases as the number of pages allocated from FAM increases. On an average, the performance improvement increases from 1.3x to 1.43x with PreFAM when memory allocation is varied from 1:1 to 1:16, as shown in Figure 13. With distance prefetching the performance improves further. With distance factor 8 we observe a maximum performance improvement of 1.8x, when local memory to FAM allocation ratio is 1:16.

5.7.2 Impact of Prefetch Buffer Size. We dedicated 1MB of the local memory to store the prefetched data. However, PreFAM can improve the improve the performance further if the size of the local memory prefetch buffer is increases. In this section we show the performance improvement with PreFAM by varying the size of the prefetch buffer in the local memory. When local memory prefetch buffer size is 4MB PreFAM improves the performance by 48.95% and with distance factor 16 the improvement is 75.6%. Also even

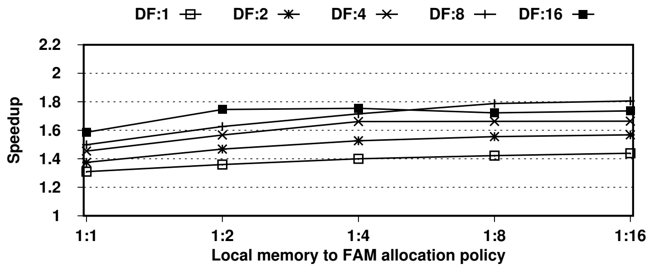


Figure 13: Impact of memory allocation policy on PreFAM.

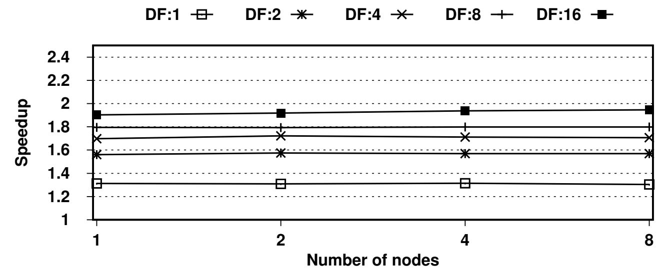


Figure 15: Impact of number of nodes on PreFAM.

when the prefetch buffer size is 256KB, PreFAM achieves an improvement of 24.6% and PreFAM with distance factor 8 achieves an improvement of 60.89%. Further increasing the distance factor will decrease the performance. With higher distance factor the number of prefetch block cancellations are less and the prefetcher tend to prefetch more blocks, Section 5.5. When prefetch buffer size is less and when more prefetch blocks are fetched the useful prefetched blocks are frequently replaced by the upcoming prefetched blocks.

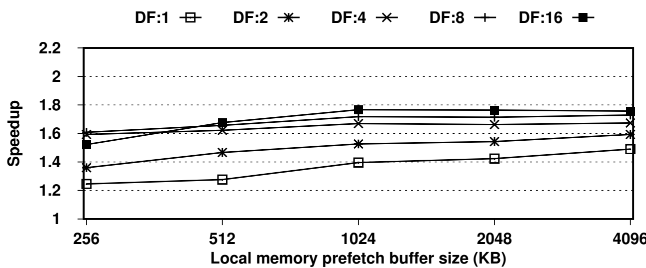


Figure 14: Impact of local memory prefetch buffer size.

5.7.3 Effect of Prefetching on Number of Node. Apart from fabric network and NVM, the memory access latency is also dependent on the number of nodes accessing FAM. By prefetching blocks to the local memory, latency when multiple nodes sharing a FAM pool can be reduced. When the number of nodes increase, the number of shared FAM pools also usually tend to increase, since it is not practical to just have one pool and assigning all the nodes to access a single pool. Hence, in our evaluation we maintained equal number of FAM pools and number of nodes. However, each FAM pool is shared by all the nodes. Figure 15 shows that as the number of nodes increase from 1 to 8 the performance improvement also increases for *bt* benchmark. When the number of nodes is 1, on an average, we observe a performance improvement of 90% with a distance factor 16 and it increases to 94% when the number of nodes in the FAM systems increases to 8.

6 DISCUSSION

In this paper, we implemented FAM prefetcher within the node. The advantages of this approach is that the prefetching unit can also consider local memory accesses by the benchmarks to study memory access patterns. Also per node prefetching units can run parallely and independent to the other nodes. This is beneficial for configurations wherein applications are not allowed to run in multiple nodes. However when the applications are allowed to run in multiple nodes this approach is not beneficial, since to learn application memory access patterns prefetching unit should also consider memory accesses coming from multiple nodes.

Prefetching unit can also be implemented in the media controller near the FAM. The advantages of implementing it in the media controller is that the prefetcher can also consider access pattern of the other nodes, when an application is allowed to run in multiple nodes. But when an application is allowed to run only in one node, multiple prefetching units (one per node) has to be maintained in the media controller, which can operate independently and parallely. However, the number of requests sent from the node to the media controller are more in this approach. For every memory access, the prefetching unit has to be informed about the current memory access, even though the memory requests is served by the local memory prefetch buffer.

In this paper, since we are focused on improving the performance of HPC benchmarks which are executed in a single node we explored and implemented prefetching unit within the node. We leaving exploring prefeching unit in the media controller for future.

7 RELATED WORK

Fabric attached disaggregated memory systems have been recently explored to address the challenges of tightly coupled in-node memory architectures [4, 5, 11, 36, 42–45, 53]. GenZ [15], CCIX [13], CXL [54] aim at providing standard fabric memory semantic protocols to access FAM. Birrittella et al, designed Intel Omni-path to enable scalable computations and resources by packing CPU or other resources like memory and storage with NIC in a single chip [9]. Gu et al. [22] and Han et al. [24] discuss and explored fast fabric network requirements to enable decoupling memory. Lim et al. [43] discussed fine-grained and page swapped remote memory access based disaggregated memory system. Shan et al. [53] proposed complete disaggregation of resources including OS. Aguilera et al. [4, 5] demonstrated the ease of using remote memory and discuss various challenges with remote memories. However, none of the previous works discuss prefetching in FAMs.

A large body of the previous work explored prefetching both at the cache and memory side to improve the performance of various systems. Pugsley et al. [49] proposed sandbox prefetching to prefetch blocks to the last level cache, after training the prefetcher. Ebrahimi et al. proposed prefetchers per core to prevent interference from other applications running simultaneously [17]. Rafique

et al. proposed memory side prefetching, conflict aware prefetching, which closely monitors the states of memory banks and prefetches data based on utilization of rows opened in the row buffer [50]. Yoon et al. stores PCM row buffers, which are frequently accessed and which incur row conflicts in the DRAM [61]. Address map pattern matching prefetcher [29], distance prefetcher [33], stride prefetcher [19] and stream prefetcher [32] are conventional and prominent prefetching types. Srinath et al. [57], Hur et al. [28] and Jiménez et al. [30] proposed adaptive prefetching schemes. The prefetch parameters are adjusted dynamically adjusted based on the efficiency.

We adopted address map pattern prefetcher to prefetch blocks from FAM. To increase the accuracy, we first train the prefetching algorithm, similar to sandbox prefetcher [49]. Then we adapt prefetch parameters; prefetch aggressiveness and frequency of the training period, based on the accuracy of the prefetcher. Also local memory is used to store prefetched blocks.

8 CONCLUSION

Disaggregating resources and accessing the decoupled resources is seemingly a promising design to address crucial challenges of tightly coupled in-node memory architectures. Such systems eliminate memory under-utilization and manage memory efficiently with less operational costs. However, these benefits comes at the cost of memory access latency. According to our evaluations, we observe that when the fabric network latency is around 500ns, the performance overhead in FAM systems is 9.77x on an average, compared to when the entire application memory is allocated in the dedicated and fast memory (DRAM). To reduce this latency in this paper we explore prefetching from FAM.

A prefetching unit is attached to the memory controller of each node. We adopted address map pattern prefetching as a prefetching scheme to prefetch FAM blocks. However, we observe that the cost of false positives is high in FAMs. This is because FAM is shared by multiple nodes and thus unused prefetched blocks not only can delay demand requests of the current node but affect all the nodes sharing the FAM. Hence, to improve the usage of the prefetched blocks and to predict future accesses accurately we first train the prefetching unit and use only most frequently identified prefetchable strides to decide and issue prefetch requests in the prefetch phase. Furthermore, we observe that due to high memory access latency, the demand requests might be issued to the FAM location which are ongoing prefetch. This also reduces the usage of the prefeched blocks. To address such cases, we proposed distance-factor based prefetching, which prefetches blocks which might be accessed in near future but not immediately. Also, a part of the local memory is dedicated to store prefetched blocks. To avoid demand request throttling, separate queues for demand requests and prefetch requests are maintained in the media controller of the FAM pool. Priority is given to the demand requests over prefetch requests. In total, with prefetching in FAMs, the performance can be improved by 38.84% and the memory access latency per request can be improved by 39.6%. However, this increases number of of accesses to the FAM by 17.65%. By introducing distance factor the performance can be further improved to 72.23%, at the cost of more accesses to FAM.

9 ACKNOWLEDGMENTS

This work has been funded through Sandia National Laboratories (Contract Number 1844457). Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International,Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

REFERENCES

[1] [n.d.]. Intel® Rack Scale Design (Intel® RSD), URL=https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html. ([n. d.]).

[2] [n.d.]. Linux Direct Access of Files (DAX). https://www.kernel.org/doc/Documentation/filesystems/dax.txt

[3] 2015. Adaptive Resource Optimizer For Optimal High Performance Compute Resource Utilization. Synopsys Inc, silicon to software, Mountain View, 1–5.

[4] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, et al. 2018. Remote regions: a simple abstraction for remote memory. In *2018 {USENIX} Annual Technical Conference ({USENIX} {ATC} 18)*. 775–787.

[5] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2017. Remote memory in the age of fast networks. In *Proceedings of the 2017 Symposium on Cloud Computing*. 121–127.

[6] Jean-Loup Baer and Tien-Fu Chen. 1991. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. 176–186.

[7] David H Bailey. 2011. Nas parallel benchmarks. In *Encyclopedia of Parallel Computing*. Springer, 1254–1259.

[8] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).

[9] Mark S Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D Underwood, and Robert C Zak. 2015. Intel® Omni-path architecture: Enabling scalable, high performance fabrics. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 1–9.

[10] David Callahan, Ken Kennedy, and Allan Porterfield. 1991. Software prefetching. *ACM SIGARCH Computer Architecture News* 19, 2 (1991), 40–52.

[11] Dan Comperchio and Jason Stevens. 2014. Emerging Computing Technologies: Hewlett-Packard's "The Machine" Project. In *HP Discover 2014 conference held in Las Vegas June 10-12*. Willdan Energy Solutions, 1–4.

[12] Adaptive Computing and Green Computing. 2015. Torque resource manager. *online] http://www. adaptivecomputing. com* (2015).

[13] CCIX Consortium et al. 2017. Cache Coherent Interconnect for Accelerators (CCIX). *Online]. http://www. ccixconsortium. com* (2017).

[14] Gen-Z Consortium et al. 2017. Gen-Z–A New Approach to Data Access.

[15] Gen-Z Consortium et al. 2017. Gen-Z DRAM and Persistent Memory Theory of Operation.

[16] Gustavo Duarte. 2009. Page Cache, the Affair Between Memory and Files. *Brain Food for Hackers Blog, http://duartes. org/gustavo/blog/post/page-cache-the-affair-between-memory-and-files* (2009).

[17] Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N Patt. 2009. Coordinated control of multiple prefetchers in multi-core systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. 316–326.

[18] Charles R Ferenbaugh. 2015. PENNANT: an unstructured mesh mini-app for advanced architecture research. *Concurrency and Computation: Practice and Experience* 27, 17 (2015), 4555–4572.

[19] John WC Fu, Janak H Patel, and Bob L Janssens. 1992. Stride directed prefetching in scalar processors. *ACM SIGMICRO Newsletter* 23, 1-2 (1992), 102–110.

[20] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network requirements for resource disaggregation. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 249–264.

[21] Hugh Greenberg, Michael Lang, Latchesar Ionkov, and Sean Blanchard. [n.d.]. REDfISh—REsilient Dynamic dIstributed Scalable System Services for Exascale. ([n. d.]).

[22] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient memory disaggregation with infiniswap. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 649–667.

[23] Geoffrey Gunow, John Tramm, Benoit Forget, Kord Smith, and Tim He. 2015. Simplemoc-a performance abstraction for 3d moc. (2015).

[24] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. 2013. Network support for resource disaggregation in next-generation datacenters. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. ACM, 10.

[25] Jim Handy. 2015. Understanding the Intel/Micron 3D XPoint memory. In *Proc. SDC*.

[26] John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (sep 2006), 1–17. https://doi.org/10.1145/1186736.1186737

[27] Michael Heroux and Richard Barrett. 2016. Mantevo project.

[28] Ibrahim Hur and Calvin Lin. 2006. Memory prefetching using adaptive stream detection. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 397–408.

[29] Yasuo Ishii, Mary Inaba, and Kei Hiraki. 2011. Access map pattern matching for high performance data cache prefetch. *Journal of Instruction-Level Parallelism* 13, 2011 (2011), 1–24.

[30] Victor Jiménez, Roberto Gioiosa, Francisco J Cazorla, Alper Buyuktosunoglu, Pradip Bose, and Francis P O'Connell. 2012. Making data prefetch smarter: Adaptive prefetching on POWER7. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 137–146.

[31] Doug Joseph and Dirk Grunwald. 1997. Prefetching using Markov predictors. In *Proceedings of the 24th annual international symposium on Computer architecture*. 252–263.

[32] Norman P Jouppi. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *ACM SIGARCH Computer Architecture News* 18, 2SI (1990), 364–373.

[33] Gokul B Kandiraju and Anand Sivasubramaniam. 2002. Going the distance for TLB prefetching: an application-driven study. In *Proceedings 29th Annual International Symposium on Computer Architecture*. IEEE, 195–206.

[34] Uksong Kang, Hoe-Ju Chung, Seongmoo Heo, Duk-Ha Park, Hoon Lee, Jin Ho Kim, Soon-Hong Ahn, Soo-Ho Cha, Jaesung Ahn, DukMin Kwon, et al. 2009. 8 Gb 3-D DDR3 DRAM using through-silicon-via technology. *IEEE Journal of Solid-State Circuits* 45, 1 (2009), 111–119.

[35] Vamsee Kommareddy, Clayton Hughes, Simon David Hammond, and Amro Awad. 2018. *Opal: A Centralized Memory Manager for Investigating Disaggregated Memory Systems*. Technical Report SAND2018-9199. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

[36] Vamsee Reddy Kommareddy, Simon David Hammond, Clayton Hughes, Ahmad Samih, and Amro Awad. 2019. Page migration support for disaggregated non-volatile memories. In *Proceedings of the International Symposium on Memory Systems*. ACM, 417–427.

[37] Jay Kyathsandra and Eric Dahlen. 2013. Intel rack scale architecture overview. *Proc. INTEROP, Las Vegas, NV, 2–6 May 2013* (2013).

[38] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. In *ACM SIGARCH Computer Architecture News*, Vol. 37. ACM, 2–13.

[39] Benjamin C Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. 2010. Phase-change technology and the future of main memory. *IEEE micro* 30, 1 (2010), 143–143.

[40] Junghoon Lee, Taehoon Kim, and Jaehyuk Huh. 2016. Dynamic prefetcher reconfiguration for diverse memory architectures. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*. IEEE, 125–132.

[41] Charles Lefurgy, Karthick Rajamani, Freeman Rawson, Wes Felter, Michael Kistler, and Tom W Keller. 2003. Energy management for commercial servers. *Computer* 36, 12 (2003), 39–48.

[42] Zhongqi Li, Ruijin Zhou, and Tao Li. 2013. Exploring high-performance and energy proportional interface for phase change memory systems. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 210–221.

[43] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K Reinhardt, and Thomas F Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. In *ACM SIGARCH Computer Architecture News*, Vol. 37. ACM, 267–278.

[44] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F Wenisch. 2012. System-level implications of disaggregated memory. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*. IEEE, 1–12.

[45] Hugo Meyer, Jose Carlos Sancho, Josue V Quiroga, Ferad Zyulkyarov, Damian Roca, and Mario Nemirovsky. 2017. Disaggregated computing. an evaluation of current trends for datacentres. *Procedia Computer Science* 108 (2017), 685–694.

[46] Gagandeep Panwar, Da Zhang, Yihan Pang, Mai Dahshan, Nathan DeBardeleben, Binoy Ravindran, and Xun Jian. 2019. Quantifying Memory Underutilization in HPC Systems and Using it to Improve Performance via Architecture Support. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 821–835.

[47] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. Colt: Coalesced large-reach tlbs. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 258–269.

[48] Allan Kennedy Porterfield. 1989. *Software methods for improvement of cache performance on supercomputer applications*. Ph.D. Dissertation.

[49] Seth H Pugsley, Zeshan Chishti, Chris Wilkerson, Peng-fei Chuang, Robert L Scott, Aamer Jaleel, Shih-Lien Lu, Kingsum Chow, and Rajeev Balasubramonian. 2014. Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 626–637.

[50] Muhammad M Rafique and Zhichun Zhu. 2018. CAMPS: Conflict-Aware Memory-Side Prefetching Scheme for Hybrid Memory Cube. In *Proceedings of the 47th International Conference on Parallel Processing*. 1–9.

[51] Arun F Rodrigues, K Scott Hemmert, Brian W Barrett, Chad Kersey, Ron Oldfield, Marlo Weston, Rolf Risen, Jeanine Cook, Paul Rosenfeld, E CooperBalls, et al. 2011. The structural simulation toolkit. *ACM SIGMETRICS Performance Evaluation Review* 38, 4 (2011), 37–42.

[52] Andy Rudoff. 2013. The Impact of the NVM Programming Model. In *Storage Developer Conference*.

[53] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. Legoos: A disseminated, distributed {OS} for hardware resource disaggregation. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 69–87.

[54] DD Sharma. 2019. Compute express link. *CXL Consortium White Paper.[Online]. Available: https://docs. wixstatic. com/ugd/0c1418 d9878707bbb7427786b70c3c91d5fbd1. pdf* (2019).

[55] Yossi Shiloach and Uzi Vishkin. 1980. *An O (log n) parallel connectivity algorithm*. Technical Report. Computer Science Department, Technion.

[56] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. 2019. Shoal: A network architecture for disaggregated racks. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 255–270.

[57] 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*. IEEE, 63–74.

[58] Jason Taylor. 2015. Facebook's Data Center Infrastructure: Open Compute, Disaggregated Rack, and Beyond, In Optical Fiber Communication Conference. *Optical Fiber Communication Conference*, W1D.5. https://doi.org/10.1364/OFC.2015.W1D.5

[59] H-S Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E Goodson. 2010. Phase change memory. *Proc. IEEE* 98, 12 (2010), 2201–2227.

[60] Yuan Xie. [n.d.]. Emerging NVM Memory Technologies. *Online]. https://web.engr.oregonstate.edu/ sllu/xie.pdf* ([n. d.]).

[61] HanBin Yoon, Justin Meza, Rachata Ausavarungnirun, Rachael A Harding, and Onur Mutlu. 2012. Row buffer locality aware caching policies for hybrid memories. In *2012 IEEE 30th International Conference on Computer Design (ICCD)*. IEEE, 337–344.