Data Prefetching on Processors with Heterogeneous Memory

Berk Saglam
Nam Ho
Carlos Falquez
Antoni Portero
Jülich Supercomputing Centre, Institute for Advanced Simulation, Forschungszentrum Jülich GmbH
Jülich, Germany
b.saglam@,n.ho@,c.falquez@,a.portero@fz-juelich.de

Fabian Schätzle
Central Institute of Engineering, Electronics and Analytics
ZEA-2 – Electronic Systems, Forschungszentrum Jülich GmbH
Jülich, Germany
f.schaetzle@fz-juelich.de

Estela Suarez[*]
Jülich Supercomputing Centre, Institute for Advanced Simulation, Forschungszentrum Jülich GmbH
Jülich, Germany
e.suarez@fz-juelich.de

Dirk Pleiter
Division of Computational Science and Technology, EECS, KTH Royal Institute of Technology
Stockholm, Sweden
pleiter@kth.se

Abstract

Heterogeneous memory architectures, such as a mix of High Bandwidth Memory (HBM) and Double Data Rate (DDR), offer flexible performance optimization by leveraging the high bandwidth of HBM along with the high capacity of DDR. However, these architectures present challenges in balancing bandwidth and capacity to maximize overall system performance and complicate hardware design.

In a flat memory organization mixing HBM and DDR, prefetchers must carefully reduce prefetch requests on DDR when transitioning from HBM to avoid performance degradation due to potential bandwidth saturation. Traditional hardware prefetchers, which typically assume a homogeneous memory, are unaware of this circumstance, so they may not be effective in heterogeneous memory architectures. The paper enhances the aggressiveness of prefetchers in this kind of architecture. Our technique enables a prefetcher to dynamically determine the optimal prefetch degree and distance based on memory type. It balances prefetch aggressiveness and timeliness through an adaptive strategy informed by bandwidth utilization and prefetch metrics learned for each memory type. We evaluated the technique within the Stride and Stream Prefetchers at L2 in a gem5 model of a 20-core Arm Neoverse V1-like architecture, a mix of HBM2 and DDR5. The simulation results, focusing on scientific benchmarks, showed that the technique effectively guides prefetchers to near-optimal static configurations. On HBM2, the adaptation strategy detects bandwidth availability and prefetches more aggressively to boost performance, achieving speedups of 1.3× to 2.3×. On DDR5, when faced with saturated bandwidth contention, the adaptation strategy switches to conservative prefetching mode to mitigate performance degradation.

Keywords

Hybrid Memory, Hardware Prefetcher, NUMA

[*]Also with Computer Science Department, University of Bonn.

1 Introduction

Modern data-intensive applications require up to 80% of the total computation time spent on data retrieval [9]. Thus, the speed at which data can be delivered to the CPU (Central Processing Unit) is an important performance factor, prompting the development of advanced memory technologies such as High-Bandwidth Memory (HBM) [29]. As reported by Li et al. [27], HBM is 1.5× faster and exhibits higher bandwidth (up to 5×) than DDR4, highlighting its superior performance.

Despite the increased bandwidth of high-performance memories, a key challenge is the persistent latency issues in the interface between memory tiers and the CPU, preventing the achieving maximum performance. This challenge, often exacerbated by the rapid advancements in CPU capabilities predicted by Moore's Law, has led to a significant performance bottleneck, commonly referred to as the *memory wall* [63]. The memory wall emphasizes the difference between the exponential increase in processing power and the slower rate of improvement in memory access times. Consequently, this gap has emerged as a critical barrier in computer architecture, substantially affecting system performance. Techniques implementing small, low-latency caches or memory prefetching can address these challenges. These aim to reduce latency and improve data access times, thereby enhancing overall system performance. While deeper cache levels or scaling cache capacities help address the increasingly widening gaps of the memory wall, they are expensive solutions [44]. In contrast, the memory prefetching technique, which hides the large memory latency by predicting and fetching data from slow main memory into small, fast caches to mitigate future cache misses, may offer a cost-efficient solution [10, 36]. Therefore, recent efforts continue to advance memory prefetching [13, 16, 21–23, 34, 40, 41, 43, 46, 52, 56].

A hardware prefetching technique proves highly effective when data are prefetched in highly accurate and timely manner. *Timeliness* in prefetching can be gained by tuning the *aggressiveness*; that is, the prefetcher must decide the number of prefetch requests and how far ahead the prefetch trigger is to operate. The level of aggressiveness must be in a delicate balance, as being too aggressive can lead to cache evictions, increased energy consumption, and potential performance trade-offs [10, 12, 56].

Heterogeneous memory architectures combine different memory technologies, such as hybrid memory architectures where two memory types serve different roles based on their bandwidth and capacity [55, 61]. In a flat memory organization within an HBM/DDR architecture [33], where both HBM and the conventional DRAM are mapped into the same global physical address space, data allocated in HBM's region benefits from high bandwidth and faster access but with limited capacity compared to DDR. Consequently, this architecture introduces non-uniform memory access (NUMA), that is, the varying performance characteristics (e.g., latency, bandwidth) of memory accesses from a CPU to different memory types. The hardware is typically exposed as distinct NUMA domains (or NUMA nodes), where each domain groups CPUs associated with the same memory technologies, similar to the traditional multi-socket systems that assume memory to be homogenous [1]. In multi-socket systems, it is expected to keep all traffic locally within the NUMA domain to mitigate inter-NUMA domain communication and, therefore, to achieve the lowest latency. In heterogeneous memory architectures, the advantages of memory technology are more critical, and the cost of traffic crossing NUMA boundaries may be a minor factor compared to the characteristics of memory type across NUMA domains. Thus, applications deployed on this architecture must be aware of memory allocation to a specific memory type to maximize capacity or bandwidth/latency benefits.

NUMA effects within a heterogeneous memory architecture may lower the effectiveness of aggressive prefetching. We, therefore, developed an adaptive technique that adjusts the prefetching parameters during runtime, dynamically identifying near-optimal prefetch degrees and distances according to the memory type that is being addressed. To analyze the role of prefetching in a processor with heterogeneous memory, we developed a gem5 model simulating a hybrid memory architecture mixing two new memory technologies, HBM2 and DDR5 [2], and using 20-cores Arm Neoverse V1 [4]. This model is employed to evaluate our solution. The contributions presented in this paper are as follows:

- We introduce a heterogenous memory-aware prefetching technique to minimize system latency within the NUMA-style hybrid memory architecture. The technique optimizes the balance between prefetch aggressiveness and timeliness through an adaptive strategy informed by bandwidth utilization and prefetch metrics learned for each memory type.
- We have developed a gem5-based simulation framework to model a detailed hybrid memory architecture. The architecture comprises 20 Arm Neoverse V1-like cores and is configured with 8× HBM2 and 1× DDR5 channels.
- Using the simulation framework, we explore the technique when integrated into Stream and Stride Prefetcher, analyzing their timeliness, aggressiveness, and accuracy. The results strongly show the importance of adaptive prefetching in enhancing system performance by dynamically identifying near-optimal prefetch degrees and distances across scientific workloads.
- We investigate the limitation of static configurations of aggressive prefetching, which require workload-specific optimizations that are impractical to implement universally. In contrast, our technique demonstrates notable performance enhancements, in particular on HBM2. The adaptive strategy can mitigate the system congestion on the DDR5 device, where bandwidth availability is limited for some data-intensive applications, achieving performance aligned with the best static configurations.

The paper is structured as follows. We start in Section 2 with the fundamentals of memory prefetching and our motivation. Section 3 characterizes the 20-core Neoverse V1-like architecture using HBM2/DDR5 memory devices. In Section 4, we describe the hybrid-memory aware prefetching, followed by the implementation in Stream Prefetcher. We dedicate Section 6 to describe our evaluation methodology. Next, Section 7 insights into the experimental analysis and evaluation of scientific benchmarks. Moving forward, in the related work in Section 8, existing research, methodologies, and relevant findings are compared to our approach. Lastly, in Section 9, we make conclusions and suggest potential directions for future research.

2 Background and Motivation

2.1 Prefetching terminology

Memory prefetching predicts and proactively fetches instructions and data from the slow memory subsystem levels to faster cache levels ahead of their actual needs. This approach significantly enhances overall system performance by hiding large memory latencies. [10, 36].

Memory prefetching can be deployed either in hardware or software. In the software prefetching approach, the compiler of the programmer inserts prefetch instructions into the program code to guide the prefetching process. In hardware prefetching, the CPU predicts specific access patterns in applications, e.g., spatial and temporal locality, using this knowledge to apply prefetching [10]. The simple, cost-effective hardware prefetcher *Next-Line* [54] assumes that accessing a memory location likely leads to the next one, and always preloads the next sequential memory block. *Stride* [5] and *Stream Prefetchers* [20, 56] predict future data requests based on stride access patterns, like $R[i]$, $R[i{+}Q]$, $R[i{+}2Q]$, $R[i{+}3Q]$, where Q is the stride. These hardware prefetchers are effective with the regular data access patterns often found in scientific computing [10, 36]. *Stride Prefetcher* detects stride-based streams issued by the same Program Counter (PC). The *Stream Prefetcher* detects the streams that are on the same page. More complex *Markov Prefetcher* [19] utilizes a history table that stores correlations of past accesses to predict future accesses. The Markov Prefetcher can predict complex repetitive patterns like those found in linked data structures or multidimensional arrays, which makes it efficient for database applications dominated by repetitive query patterns [19].

Prefetches are considered *useful* when the data that they prefetch is employed by future demand accesses. When the data collected by a *useful* prefetch is available in the cache ahead of the demand

[1]In a multi-socket system, each domain has its own local memory, and the CPU within the domain accesses its local memory faster than remote memory from other domains.
[2]HBM2 is used in the modest A64FX processors, while the Graviton3 CPU uses DDR5.

request, it is called a *timely* prefetch. In contrast, a useful prefetch is considered a *late* prefetch when the prefetch request has not yet finished by the time its corresponding demand access is issued, forcing it to wait. Such late prefetches do not fully hide large memory latency. Finally, *useless* prefetches are those that bring to the cache data that is not needed, potentially evicting useful data in the process, which can slow-down the application.

An effective prefetching technique relies on two key metrics: prefetch *accuracy* and prefetch *coverage*. The former measures how accurately the prefetcher predicts access patterns. The latter indicates the amount of cache misses that are eliminated by timely prefetches. Both metrics need to be high for better performance, but they are in an inverse proportional relation: achieving high coverage with improved timeliness may result in low accuracy. Hence, a good prefetcher balances *accuracy* and *coverage* by carefully managing the *aggressiveness* of the prefetcher - that is, determining how many prefetches (*prefetch degree*) and how far ahead in memory the prefetch trigger goes (*prefetch distance*), constrained with an accuracy threshold [56]. Degree and distance are the two main metrics of aggressive prefetching.

2.2 Prefetching in heterogeneous memory systems and motivation

Compared to standard DRAM, emerging memory technologies are providing orders of magnitude either more bandwidth (e.g. with die-stacked DRAM and hybrid memory cube [29, 47]) or larger density and capacity (via non-volatile technologies (NVM) [1, 25, 32]). A heterogeneous memory architecture integrating different memory technologies can provide trade-off solutions to satisfy large-capacity vs. large-bandwidth scalings. The two main ways to organize heterogeneous memory systems are: i) to expose the high-bandwidth [17, 18] or the low-latency memory [26] as a cache, or ii) to map both in the same physical address space [33, 51]. For example, the Intel Xeon Phi KNL (Knight's Landing) processors mixing stacked DRAM (MCDRAM) and DDR memory provides flexible MCDRAM configurations, both using MCDRAM as a cache (*cache mode*) or as part of the physical address space (*flat mode*) [55, 61]. For this purpose, KNL integrated SNC (Sub-NUMA Clustering) modes, which optimize latency and bandwidth by dividing the processor into several NUMA domains. MCDRAM is designed to provide high bandwidth but at a lower capacity, making it ideal for rapidly processing critical data. By contrast, DDR memory offers a larger storage capacity suitable for less frequently accessed data and comes with the trade-off of lower bandwidth.

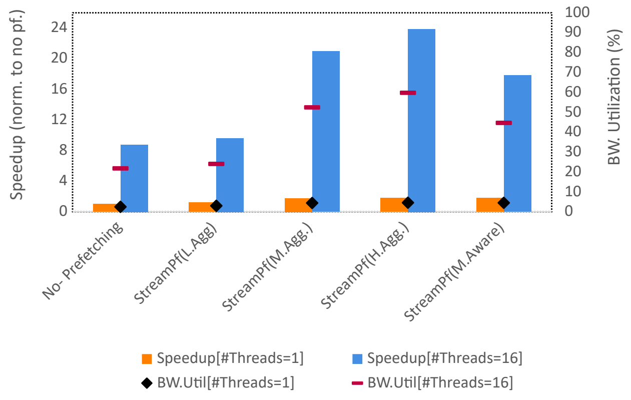
Our work focuses on enhancing aggressive prefetching of hardware prefetchers within a hybrid memory Arm-based architecture that combines HBM and DDR technologies. We examine the effects on its flat memory organization when both devices are mapped to the same physical address space, exposing them to two NUMA domains as the simulated architecture shown in Figure 2. Using the Stream benchmark [31], we measured bandwidth versus average latency of two NUMA domains, as shown in Figure 3. The bandwidth utilization for both HBM2 and DDR5 saturates at around 80% of the corresponding peak bandwidths, which are 245 GiB/s and 30.7 GiB/s, respectively. In the case of HBM2, saturation is reached using more than 16 cores, while for DDR5, using more than two cores suffices. The measurement result emphasizes the diverse bandwidths and latencies within a heterogeneous memory architecture. Traditional prefetchers assume homogenous memory. Thus, tuning aggressiveness parameters at the global level, such as those described in [12, 56], may be unsatisfactory on this architecture. Static configuration strategies for aggressive prefetching are also no longer relevant, given the lack of awareness of memory access characteristics of different memory tiers.

To investigate this further, we implemented a Stream Prefetcher, one described in [56], and evaluated the impact of different aggressive prefetching levels (*low*, *medium*, and *high*). An analysis with a sparse matrix-vector multiplication (SpMV) kernel (details in Section 6.2) is shown in Figure 1, which plots speedup (execution time normalized to the single-thread run without prefetching), and bandwidth utilization on the two memory devices. On the HBM2, where bandwidth availability is high, higher aggressiveness leads to higher speedup gains. On the DDR5, however, higher aggressiveness can degrade performance: in the 8-thread case, the execution without prefetching saturates memory bandwidth (80% of the peak), and high prefetch aggressiveness leads to application slow-down.

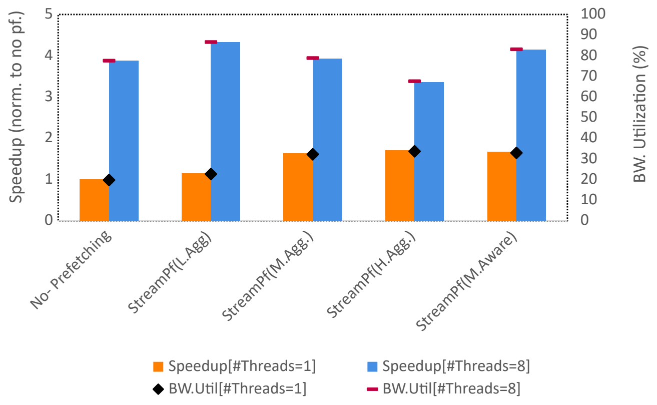
Thus, to make prefetching more efficient on heterogeneous memory architecture, the key optimization relies on enabling the prefetcher to dynamically monitor the distinct characteristics of different memory technologies and adapt its strategy at run-time. To address this, particularly within a hybrid HBM2/DDR5 architecture, we propose an adaptive aggressive prefetching technique, which guides the prefetcher to learn memory access patterns and bandwidth utilization for each device. During prefetching, this learned information is used to tune the aggressiveness parameters, thereby optimizing timeliness and accuracy for both memory devices. Our technique improves the effectiveness of the Stream Prefetcher, as illustrated in Figure 1 (**M.Aware**). It increases prefetching aggressiveness when bandwidth is available, as in the HBM2, and applies less aggressive prefetching in DDR5 when high memory bandwidth utilization is detected.

3 Baseline Hybrid-Memory Architecture

Our baseline hybrid-memory architecture is an Arm-based design inspired by the KNL processor. Figure 2 illustrates the mesh configuration of one quadrant of the simulated CPU using Arm Neoverse V1-like architecture based on the reference design [4]. The quadrant is connected to 8× HBM2 channels and 1× DDR5 memory channel. The blue routers, linked to the HBM2 channels, incorporate a pair of cores each. Orange routers, connected to the DDR5 channel, also have two cores. Serving a critical backup role, the two red routers act as a fail-safe for the other cores in cases of malfunction for various reasons, and it also contains two cores. The two purple routers are dedicated to inter-chiplet connections. In Figure 2, the **NUMA domains** are distinctly illustrated: the **HBM2 domain** is depicted in blue, consisting of 8 routers and thereby 16 cores, while the **DDR5 domain** is represented in orange, comprising 2 routers and the associated 4 cores. Finally, these configurations account for a total of 20 cores inter-connected by 2D-mesh Network-on-Chip (NoC).



(a) HBM2



(b) DDR5

Figure 1: The impact on speedup and bandwidth utilization (BW.Util.) when statically varying the aggressiveness of Stream Prefetcher - low (L.Agg.), medium (M.Agg.), and high (H.Agg.) - is compared with our memory-aware technique within the Stream Prefetcher (M.Aware). The experiments are conducted for the SpMV kernel on the simulated hybrid HBM2/DDR5 memory architecture. Speedup is computed as the execution time normalized to the single-thread run without prefetching.

We have modelled the hybrid-memory architecture in the gem5 simulator. Details for the simulation setup are described in Section 6.1. Next, we characterize the bandwidth utilization and READ latency of HBM2 and DDR5 technologies in the simulated architecture.

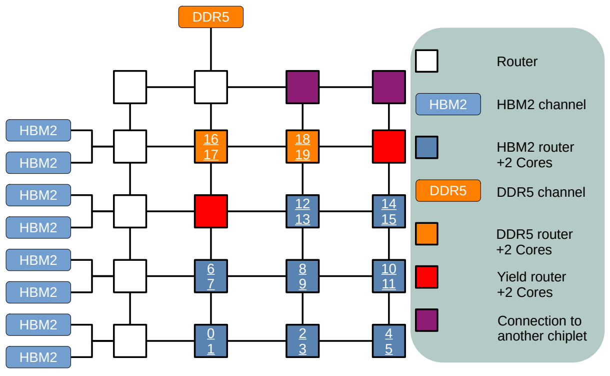


Figure 2: Simulated Quadrant architecture with Arm Core-Link CMN-650 configuration: Illustrates the allocation of 16 Neoverse V1-like cores to 8× HBM2 channels (blue) and 4 cores to 1× DDR5 channels (orange routers). The two red routers serve as yield cores, and the two purple routers enable inter-chiplet connections.

3.1 Bandwidth utilization

Queuing theory suggests that when bandwidth utilization is high, i.e., once saturated or close to peak bandwidth, the system enters into a bottleneck state and thus increases the memory latency (due to increased queuing time) [59]. Capturing the relationship between average memory latency and bandwidth helps to understand the capabilities of two memory technologies. We measured memory bandwidth utilization using the Stream benchmark (Triad kernel), and computed the average memory latency of cache misses at L2 during execution.

Figure 3 shows curves where values on the y-axis and x-axis are bandwidth utilization and average memory latency, respectively. Figure 3a plots data for HBM2, while Figure 3b plots data for DDR5. The solid curves correspond to the simulation of Stream-Triad, and the data points are the number of OpenMP threads used (1, 2, 4, 8, 16, 20). The dashed curves are generated using the M/M/1 queue model [59] to approximate the missing data points and refine the measurement results [3]. The more cores are used, the higher the memory bandwidth consumed. Once bandwidth is saturated, the memory latency increases significantly on both devices.

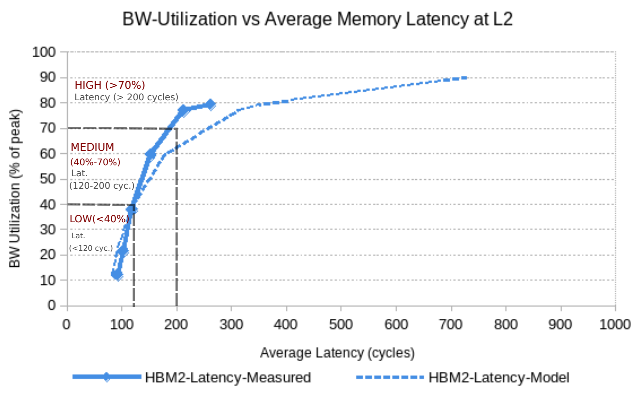
The red labels indicate splitting bandwidth utilization into *low* ($<$ 40% of peak), *medium* (40% – 70% of peak), and *high* ($>$ 80% of peak). For HBM2, with a peak bandwidth of 307.2 GiB/s, the low bandwidth threshold is at 120 cycles, the medium is between 120 and 200 cycles, and the high bandwidth is reached at more than 200 cycles. For DDR5, with a peak bandwidth of 38.4 GiB/s, the low bandwidth threshold is at 100 cycles, the medium spans from 100 to 270 cycles, and the high bandwidth is reached at 270 cycles.

The hybrid-memory-aware prefetching described in Section 4 monitors the average memory latency at runtime to estimate the system bandwidth utilization and uses that information to adapt aggressive prefetching.

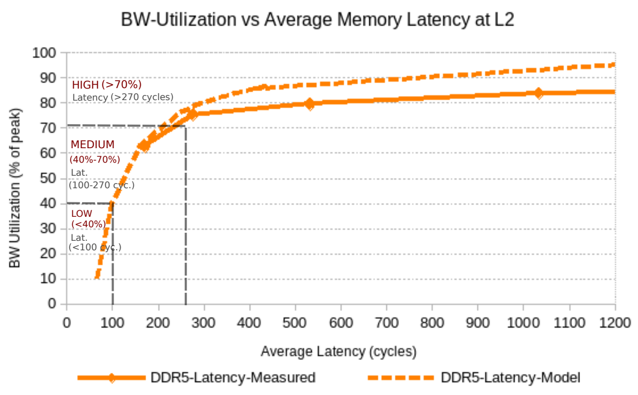
3.2 Random read system latency

We ran the Tiny-Membench [60], a well-known benchmark, to measure system latencies with dual random read operations to two

[3]The plot follows the formula noted in [15]: $L = \frac{1}{1-R\cdot S} \cdot S$, where R is the bandwidth (arrival rate), L is the average total latency, and S is the no-waiting service time. We measured S by generating memory requests without stressing the memory system. The curve is generated by varying the fraction of peak bandwidth $R \cdot S$.



(a) HBM2



(b) DDR5

Figure 3: Memory bandwidth utilization vs. average total latency curves measured with Stream-Triad (solid curves) and queuing model (dashed curves) for HBM2 and DDR5 in the baseline architecture. The red ranges partition bandwidth utilization into low, medium, and high levels, each corresponding to average memory latency thresholds.

memory domains. Our measurements are as follows: When data allocation is on HBM2, the execution is pinned on core-0, which is close to HBM2 controllers. When data allocation is on DDR5, the execution is pinned on core-19. Pinning to these two specific cores aims to avoid NUMA effects that may happen within the NoC. The results are shown in Figure 4 for varying block sizes.

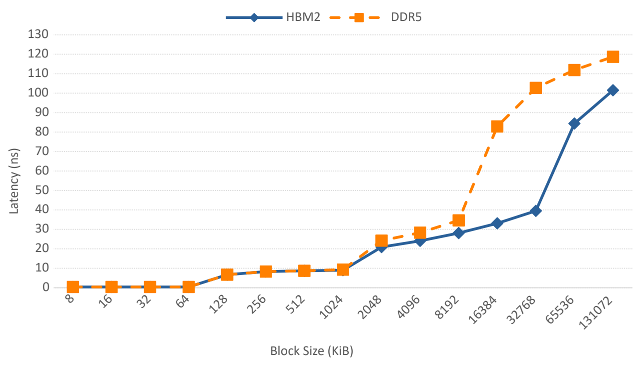


Figure 4: Random read latency measured by Tiny-Membench.

Since core-0 and core-19 have the same private L1 and L2 cache sizes in common (see Table 1), a similar latency trend is observed for the block size range 8 KiB to 1024 KiB. The core-19, which belongs to the DDR5 domain, has four assigned LLC (shared Last Level Cache) slices with a size of 4×2 MiB; thus, the impact of DDR5 latency is seen right after the block sizes 8192 KiB. Meanwhile, the core-0, which belongs to the HBM2, has 16 assigned LLC cache slices with a size of 16×2 MiB; the impact of HBM2 latency is seen right after block size 32768 KiB. The plot shows that accesses to DDR5 have a larger read latency than those to HBM2. This aligns with previous work [27], which showed that the average random access latency of DDR5 devices is larger than that of HBM due to reduced queuing delay because of a higher degree of parallelism on HBM. In our case, HBM2 has eight channels; meanwhile, DDR5 operates with a single channel.

To this end, the HBM2 domain has lower latency and higher bandwidth than the DDR5 domain. However, DDR5 offers the benefit of greater capacity over HBM2.

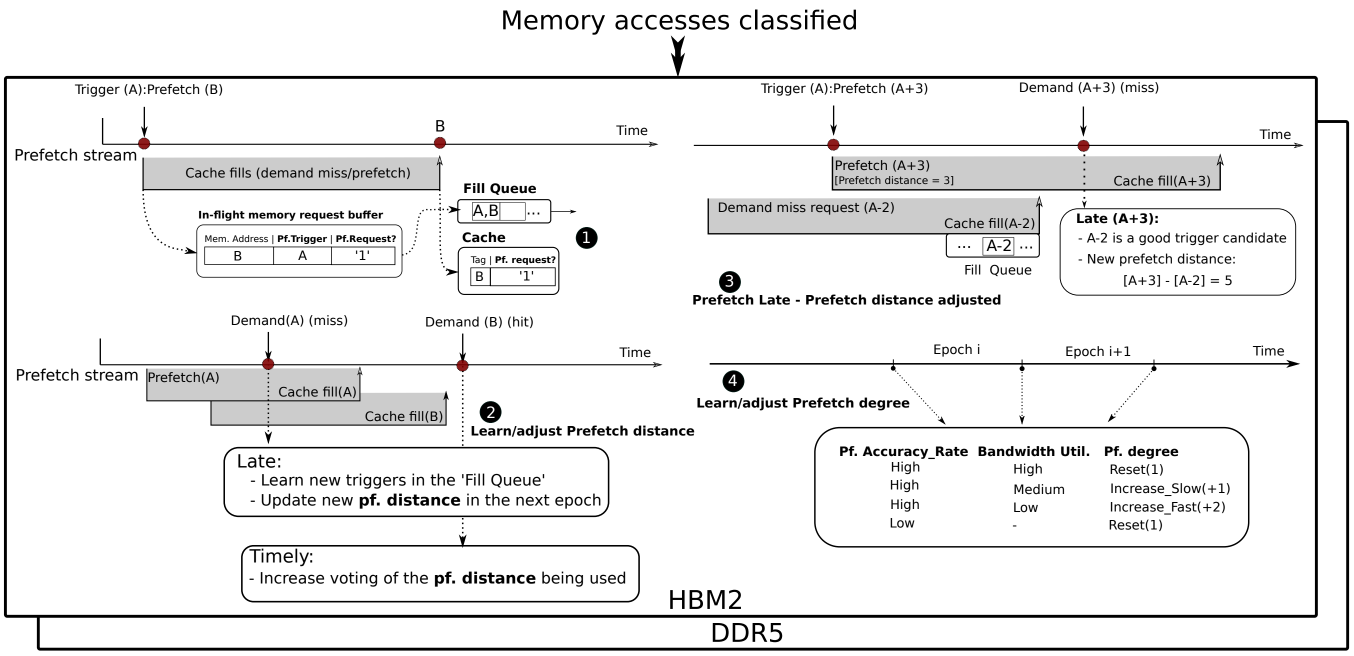
4 Hybrid-Memory-Aware Prefetching

Our work improves prefetchers in architectures using different memory technologies, focusing on the optimization aspect of aggressive prefetching. As shown in Figure 5, we propose a prefetching strategy for hybrid memories that dynamically tunes two critical prefetching parameters: *prefetch distance* (steps ❶, ❷ and ❸) and *prefetch degree* (step ❹) to match well the characteristic of each memory technology. Prefetcher extensions include the following features:

Memory Device Access Classification: For memory accesses seen by prefetchers, the first necessary action is to classify memory accesses belonging to each memory device, i.e., either accesses to HBM2 or DDR5 of the baseline architecture. The classification can be done by extracting memory device information from the access addresses via a hash function, a memory address decoder typically supported in high-end CPU architectures.

Fill Queue: Most prefetchers track and learn memory access patterns for prefetching in the forms of access streams; for example, Stride Prefetcher stores in its hash table access streams from the same PC. The prefetcher is extended by a queue dedicated to each prefetching stream to learn prefetch triggers and tune prefetch distance. In addition, the in-flight memory request buffer (IFB)[4], is extended to track whether a pending memory request is a prefetch request ('pf.request=1') or not. If there is a prefetch request, the IFB buffer also stores the prefetch trigger. Furthermore, the cache line also has a prefetch bit ('pf.request') to distinguish prefetch data. It is

[4]The mechanism handles in-flight memory requests like Miss Status Holding Register (MSHR).



Memory accesses classified
Trigger (A):Prefetch (B)
B
Time
Prefetch stream
Cache fills (demand miss/prefetch)
Fill Queue
A,B ...
In-flight memory request buffer
Mem. Address | Pf.Trigger | Pf.Request?
B | A | '1'
Cache
Tag | Pf. request?
B | '1'
❶
Trigger (A):Prefetch (A+3)
Demand (A+3) (miss)
Time
Prefetch (A+3)
[Prefetch distance = 3]
Cache fill(A+3)
Demand miss request (A-2)
Cache fill(A-2)
... A-2 ...
Fill Queue
Late (A+3):
- A-2 is a good trigger candidate
- New prefetch distance:
[A+3] - [A-2] = 5
❸
Prefetch Late - Prefetch distance adjusted
Demand(A) (miss)
Demand (B) (hit)
Time
Prefetch stream
Prefetch(A)
Cache fill(A)
Cache fill(B)
❷
Learn/adjust Prefetch distance
Late:
- Learn new triggers in the 'Fill Queue'
- Update new pf. distance in the next epoch
Timely:
- Increase voting of the pf. distance being used
Epoch i
Epoch i+1
Time
❹
Learn/adjust Prefetch degree
Pf. Accuracy_Rate | Bandwidth Util. | Pf. degree
High | High | Reset(1)
High | Medium | Increase_Slow(+1)
High | Low | Increase_Fast(+2)
Low | - | Reset(1)
HBM2
DDR5

Figure 5: Hybrid-memory-aware Prefetching Strategies: In step ❶, the in-flight request buffer and cache line are extended to store information about the prefetch trigger('pf.trigger') and whether a prefetch request is made ('pf.request'). Steps ❶, ❷, and ❸ demonstrate the distance adaptation methodology in reaction to cache events (late/timely prefetch). Step ❹ illustrates degree optimization strategy with fluctuating bandwidth utilization and prefetch accuracy rate.

used to construct prefetch statistics, determine whether prefetches are timely or late, and count the number of prefetches.

When a cache fill event occurs, indicating that the cache has been filled with data returned from memory, the corresponding pending request in the IFB is resolved. Upon the event belonging to a prefetch request, the 'pf.request' in the cache line is set. Furthermore, the pending prefetch request and its prefetch trigger stored in the IFB are extracted and queued in the Fill Queue, as illustrated (❶) in Figure 5. If the cache fill event is from a demand miss belonging to the access stream, that demand miss is also pushed into the Fill Queue since it is a prefetch trigger candidate to adjust the prefetch distance. As illustrated in Figure 5, the prefetch (marked as **B**) is initiated by the trigger access **A**. Both **B** and **A** are queued once the cache fill for **B** is done. Note that when the queue is full, the oldest is removed.

Learning/Tuning Prefetch Distance: Prefetchers learn and tune prefetch distance by adjusting the prefetch trigger upon late and timely prefetch events (❷). The prefetcher is notified of a late prefetch when the corresponding demand miss finds the prefetch request pending in the IFB, indicated by the 'pf.request' being set. This implies that the late prefetch could have been timely if it had been prefetched by a trigger stored in the Fill Queue, since memory requests stored in that queue are completed before the occurrence of the demand miss. Thus, the prefetcher searches in the Fill Queue for a new trigger, which is the most recent pushed in the queue. As demonstrated in ❸, a prefetcher operating with prefetch distance = 3 adjusts the prefetch distance for future prefetching. The late prefetch event associated with the demand miss **A**+3 causes the prefetcher to select a trigger **A**-2 in the Fill Queue and adjust the prefetch distance for the subsequent prefetch from the current value of 3 to 5.

While the new prefetch distance can be immediately applied for the next prefetches, collecting statistics of candidates and picking up the most voted or the mean value from the candidate set would be a more promising strategy. The candidate set is a mix of the recently computed distance and those in use. In the latter case, the candidates are the distances from which prefetches are timely. The prefetcher is notified of a timely prefetch when the demand access finds the 'pf.request' set in the corresponding cache line. If so, the recent trigger address is searched in the Fill Queue. If found, the distance is computed, and its voting is updated in the candidate set. Distance voting occurs over a fixed interval (epoch), similar to the prefetch degree learning described below. Once the epoch expires, the best distance (e.g., the most voted or the mean) is selected for the next prefetching. We elaborate on different voting strategies in Section 7.

Learning/Tuning Prefetch Degree: The learning and tuning strategy of the prefetch degree takes place after each epoch period as illustrated in Figure 5 (❹). After an epoch expires, prefetchers dynamically increase or decrease the prefetch degree through learning bandwidth utilization in each memory device in conjunction with prefetch accuracy rates. At runtime during epoch time, bandwidth utilization and prefetch accuracy rate are measured; when the epoch expires, they are categorized into low, medium, and high levels by comparing the measured values with the configured thresholds.

In low or medium bandwidth utilization scenarios, as the system experiences reduced pressure, prefetchers can operate more aggressively. Thus, prefetchers can increase their degree to bring a larger amount of memory blocks into the cache. Conversely, at high bandwidth utilization, it is better to maintain or even decrease the prefetch degree to avoid increasing pressure on bandwidth.

However, bandwidth utilization is not the sole determinant in the optimization strategy; prefetch accuracy also plays a crucial role. Thus, dual-parameter tuning provides a balanced and efficient prefetching strategy. For either a high prefetch *accuracy rate*, once bandwidth utilization is at medium or low levels; the decision is to increase the degree slowly or quickly up to the limited maximum value, respectively. The prefetch degree will be reset to 1 immediately when a low prefetch accuracy rate or high bandwidth utilization is observed.

Memory Bandwidth Utilization Approximation: Inside the prefetcher, gathering the information about memory bandwidth utilization at the memory controller is costly due to the physical distance between hardware components and may need an advanced performance monitoring infrastructure, such as an MPAM unit in Neoverse processors [8]. We approximate the bandwidth utilization for each memory device and categorize it into three levels (*high*, *medium*, and *low*, as shown in Figure 3), by indirectly measuring the average latency of in-flight memory requests at the cache level where the prefetcher is located. When a memory request allocates an entry in the in-flight memory request buffer, a dedicated counter starts timing and stops once the requested data returns. The latency value is calculated when the entry in the buffer is deallocated. The average value is computed once the epoch expires and then compared with thresholds determined from the bandwidth utilization vs. average latency curve (Figure 3) to estimate the bandwidth utilization of each memory device.

Tuning Statistics: The tuning strategy for the prefetch degree requires a statistic parameter, the prefetch accuracy rate. The prefetch accuracy rate is calculated as the ratio of the number of *useful* prefetches, a sum of *timely* and *late* prefetches, to the *total* number of prefetches: $\frac{\text{useful prefetches}}{\text{number of prefetches}}$. It indicates how effectively a prefetcher predicts the memory access patterns generated by the application. Thus, *timely-/late-/number-of-prefetches* counters must be supported [5].

5 Integration into Stream Prefetcher

To demonstrate the integration of our approach into existing prefetchers, we have chosen the Stride Prefetcher, available in gem5, and Stream Prefetcher, our implementation based on one described in [56]. We explain the integration within the Stream Prefetcher at the L2 cache, as shown in Figure 6. This history table must be augmented with a queue mechanism to facilitate distance adjustments. As previously described in Section 4, upon each cache fill event, the cache controller checks if the accessed data stream is recorded within the prefetcher's history table. This involves comparing the tag of the valid entries in the history table to the address observed

[5]Our counters are defined as follows: *Timely* measures timely prefetches, where a demand miss finds 'pf.request' set in the corresponding cache line. *Late* measures late prefetches, where, by the time the demand access is issued, it finds a corresponding pending prefetch in the IFB; *Number of prefetches* measures the total number of prefetch requests issued by the prefetcher.

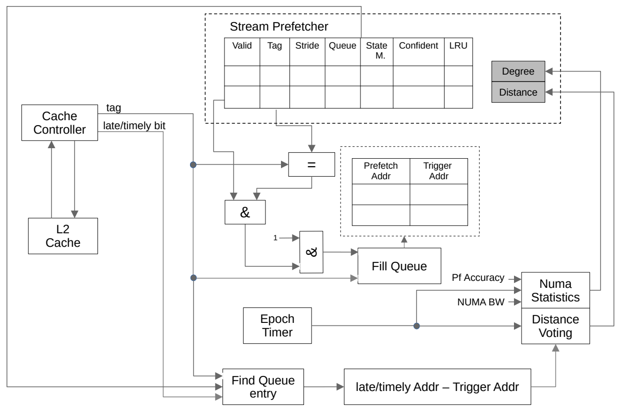


Figure 6: Block diagram illustrating the integration of our adaptive strategy into Stream Prefetcher at the L2 cache.

by the cache controller. Upon a successful match, the queue is populated with the prefetch address and the trigger address of the ongoing cache fill.

Additionally, the cache controller assumes the responsibility of assessing the timeliness of incoming prefetches, employing a 'late/timely bit' for this purpose. Depending on the outcome, the appropriate candidate within the Fill Queue is selected, followed by the calculation of a new distance. This is achieved by subtracting either the late or timely prefetch address from the trigger address found in the Fill Queue. Distance votes are collected within the fixed epoch period. After the epoch, the most frequent vote, or the mean vote, is applied. The prefetch degree adjustment process is relatively straightforward, as described in the previous section. It involves collecting the prefetch accuracy statistic and measuring system bandwidth per NUMA domain within an epoch.

Here, we describe the extension in the Stream Prefetcher, extending our approach to other stream-based prefetchers, e.g., the Stride Prefetcher, follows a similar way.

6 Evaluation Methodology

6.1 Simulation setup

We evaluated our hybrid-memory-aware prefetching using a gem5 model of the baseline architecture using Arm Neoverse V1-like cores [4] as described in Section 3. The gem5 simulator was extended to model in detail the AMBA-CHI-based Network-on-Chip (NoC) architecture [2], based on the recently released CHI Ruby protocol [38] and the Garnet model [30]. The NoC architecture implements Arm Neoverse CMM-650 Coherent Mesh Network architecture [3], where each SNOOP, DATA, RESPONSE, and ACK message type are transferred on separated link channels. Cache hit latencies are chosen empirically for overall performance optimization, though they may be more optimistic than commercial processors. The LLC's allocation is inclusive for CHI ReadShared requests and exclusive for CHI ReadUnique requests.

Table 1 lists the main architecture parameters. The HBM2 memory model is based on the previous work [50], delivering a peak

Clocks	System: 1.6 GHz; CPU: 2.4 GHz; NoC: 2.0 GHz;
CPU	#Armv8-A cores: 20; Branch prediction: BiMode; Vector Unit: 2×256-bit SVE
L1-D, L1-I	Line size: 64 B; Size: 64 KiB; Ass: 4-way; Replacement policy: LRU; TBEs: 256; Hit latency: 2-cycles (L1-D), 1-cycles (L1-I)
L2	Unified cache; Line size: 64 B; Size: 1 MiB; Ass: 8-way; Replacement policy: LRU; Clusivity policy: strictly inclusive; Hit latency: 4-cycles; TBEs: 256
LLC	Shared LLC (Last Level Cache); Size: 16× 1 MiB-slices; Line size: 64 B; Ass: 16-way; Replacement policy: LRU; Clusivity policy: inclusive/exclusive; Hit latency: 10-cycles; TBEs: 256 per slice;
NoC	Model: Garnet 3.0; Protocol: AMBA-CHI; Flit width: 64B; Router latency: 1-cycle; Link latency: 1-cycle; 2D-Mesh: 4×5; #VNETs: 4; Link configuration: 4-physical channels
Memory	8× HBM2 channel; Size: 2 GiB; BWs: 307.2 GiB/s 1× DDR5 channel; Size: 4 GiB; BWs: 38.4 GiB/s
NUMA Assignment	HBM2 domain: {core-0 ,..., core-15} DDR5 domain: {core-16 , core-17, core-18, core-19}

Table 1: Parameters in the gem5 simulation of the baseline architecture.

bandwidth of 307GiB/s. Our DDR5 model is based on the Micron DDR5 datasheet [35], providing a peak bandwidth of 38.4GiB/s. The simulated architecture has 16 distributed Shared LLC-slides/16-cores assigned for 8× HBM2 channels and 4 LLC-slides/4-cores designated for 1× DDR5 channel. The vector unit has two `256-bit SVE` engines. Each core has a private L1 cache, and the two-core tile shares a unified L2 cache. At each cache level, in gem5 the behavior of the Miss Status Holding Register (MSHR) responsible for handling in-flight memory requests is modelled by Transaction Buffer Entry (TBE) parameters, which are not only tracking the in-flight memory requests but also the transactions of the CHI cache coherency protocol [38]. The NoC operates with the AMBA-CHI protocol [2] configured for four message types transferring data on four physical-link channels.

6.2 HPC benchmarks

We have chosen four representative scientific benchmarks: three memory-bound codes (waLBerla, miniFE, Stream) and a latency-bound code (Ligra). These codes benefit from stream-based prefetchers (as analyzed in Section 6.2.1), allowing us to assess the hybrid-memory technique within Stride and Stream Prefetchers. All codes, except Ligra, support ARM Scalable Vector Extension (SVE) either through an optimized implementation using SVE intrinsics or by leveraging compiler auto-vectorization. Paralelisation is implemented via OpenMP.

waLBerla: is a specialized framework tailored for conducting computational fluid dynamics simulations based on the Lattice Boltzmann Method (LBM) [11]. We used the waLBerla benchmark (release 4.1) known as *UniformGridBenchmark*, which implements the D3Q19 model on a regular grid, to analyze the prefetch effects on stencil codes.

Locality score	**waLBerla**	**SpMV**	**Ligra-BFS**	**Triad**
Spatial ($W = 32$)	0.807	0.782	0.689	0.999
Temporal ($L = 32$)	0.032	0.130	0.114	0.000

Table 2: Spatial and temporal locality scores for window size $W = 32$ and $L = 32$.

miniFE: is a mini-app designed to represent finite element problems [49]. We evaluated the Conjugate Gradient (CG) Solver of this mini-app, which implements core numerical methods, including the Sparse Matrix-Vector (SpMV) Product kernel. Our study utilizes the SVE-ported version of the SpMV benchmark that has adopted the SIMD-friendly Sliced ELLPACK (SELL) format [24, 37], optimized for the ARMv8 architecture [7].

Ligra: is a lightweight graph processing framework designed for shared-memory multicore platforms [53]. We evaluated the Breadth-First Search (BFS) benchmark in the Ligra framework, which is used in various data analytics applications, such as studying social networks and supercomputer performance ratings. The graph input is the Kronecker symmetric graph comprising 2^{19} vertices.

Stream: This benchmark is included in our analysis due to its regular stride-1 access patterns, making it a suitable case for analyzing prefetch effects. We used the Triad kernel [31].

6.2.1 Locality analysis. To better understand the effects of prefetching, we record the traces using the gem5 model and analyze the benchmark's *spatial* and *temporal locality*. Spatial locality reveals the likelihood of nearby memory accesses, while temporal locality reflects how memory accesses are repeated shortly.

Spatial and temporal locality are computed following the scoring methodology described in [45, 62], which requires profiling stride and reuse distance distributed across all memory references[6]. A high spatial score (i.e., close to 1) implies that memory access patterns tend to exhibit regular accesses with small strides. On the other hand, a high temporal score (i.e., close to 1) means that memory access patterns contain highly repeated access sequences.

Table 2 summarizes the locality scores computed for traces of memory references up to 300 million executed instructions. The Stream-Triad kernel exhibits a spatial locality score of approximately 1 and a temporal locality score of roughly 0, indicating that it consists of entirely sequential stride-1 access patterns with no repeated sequences. High spatial scores across all kernels imply high benefits when using spatial address-correlation prefetchers such as stream-based prefetchers. Therefore, the selected benchmarks allow us to focus solely on evaluating the aggressive prefetching enhanced by our adaptive technique for these prefetchers. Low temporal scores suggest that temporal address-correlation prefetchers, which trigger prefetching based on learning address repetition, are unlikely to improve performance.

[6] *Stride* is the minimum distance between the address and its neighbors within a window of W memory references. Given a memory reference a_i, the *reuse distance* of a_i is the number of unique memory references, other than a_i, accessed since the last access a_j within a window of L memory references

	NUMA coarse-grained memory allocation	
	Kernel	**Problem size**
waLBerla	UniformGridBenchmark	Lattice size = 64^3
Ligra	BFS	Kronecker graph $[(A, B, C) = (0.57, 0.19, 0.19)]$: 2^{19} vertices
miniFE	SpMV	Lattice size = 63^3
Stream	Triad	1.5×10^6
	NUMA fine-grained memory allocation	
miniFE	CG Solver	Lattice size = 128^3
Stream	Triad	1.5×10^6

Table 3: Experimental configuration of the selected benchmarks: NUMA memory allocations are with either coarse-grained or fine-grained strategies.

6.3 NUMA execution

To achieve optimal performance of an application on a NUMA architecture, a set of cores for code executions and specific memory devices for data allocation are commonly identified, a technique known as "pinning" or "affinity". We evaluate different NUMA execution scenarios as follows: Core affinity ('GOMP_CPU_AFFINITY') is utilized to pin thread execution on a specific core. Data allocation on HBM2 or DDR5 is done by either using the 'numactl —m' utility or using the Linux NUMA policy library ('libnuma'). We refer to the former strategy as the ***NUMA coarse-grained memory allocation*** since the total memory footprint of application kernels is allocated on a specific memory device, and no code modification is required. In contrast, the latter strategy is referred to as ***NUMA fine-grained memory allocation*** since code modifications are needed to gain better data allocation.

Table 3 sketches the experimental configuration of benchmark kernels for NUMA execution. The problem sizes are chosen large enough, e.g., exceeding the total LLC size, to ensure meaningful assessments.

7 Experimental Results

We evaluate four prefetchers using full-system (FS) mode in gem5. The **Stride** Prefetcher and the **Stream** Prefetcher. The other two are the newly developed hybrid-memory aware Prefetchers: the Timely-Aware Stride Prefetcher (**TiA**) and the Aggressive Prefetcher (**Agg**). The TiA and the Agg Prefetchers correspond to our integration of hybrid-memory aware prefetching techniques within the Stride and Stream Prefetcher, respectively. For TiA and Agg, the high prefetch accuracy rate is set to 0.8, and the epoch period is 128000 cycles. Prefetches are issued upon demand access, and prefetches whose addresses cross the page boundary (4KiB) are dropped.

7.1 Prefetch effects of NUMA coarse-grained memory allocation

7.1.1 Adaptive prefetching behavior within NUMA domain. Figure 7 demonstrates the adaptive behavior of the Agg Prefetcher compared to a static aggressiveness configuration of the Stream Prefetcher on the HBM2 domain for single-thread runs of Stream-Triad. The Agg Prefetcher (Figure 7a) dynamically tunes both the prefetch degree and distance, while, the Stream Prefetcher (Figure 7b) is configured with the prefetch distance=2 and degree=2. Note that statistical metrics are accumulated per epoch and are reset at the beginning of each new epoch. The x-axis (number of epochs) and the demand accesses are different left and right because the overall runtime of the benchmark varies when applying different prefetchers.

The upper left quadrants in both Figures (7a and 7b) reveal a correspondence between total prefetches (orange curve) and useful prefetches (green curve) per epoch, reflecting high prefetch accuracy in both prefetchers. This is expected since Stream-Triad presents stride-1 access patterns, which are straightforward for stream-based prefetchers to detect.

However, the Agg Prefetcher shows a much closer proximity between useful prefetches (green curve) and demand accesses (blue curve). That implies useful prefetches potentially avoid cache misses since demand accesses within Stream-Triad are typically demand misses without prefetching. In fact Agg Prefetcher achieves a prefetch coverage [7] of 0.95, much higher than that of 0.5 (shown on the bottom right quadrants in the figures) observed in the static configuration in Stream Prefetcher. This indicates a higher efficacy in improving prefetch timeliness via adapting aggressiveness when compared to the static configuration of the Stream Prefetcher. The variation in the degree used, observable in the bottom left quadrants in both Figures (blue curve), accounts for this discrepancy: The Agg Prefetcher, operating under high bandwidth availability (via low bandwidth utilization) and demonstrating high accuracy in pattern prediction, adjusts its degree up to 16 as per the guidelines from the prefetch degree tuning.

The investigation into late prefetches, depicted in red curves, initially shows a greater occurrence in the Agg Prefetcher, rapidly diminishing in the following epochs as a result of the implemented adaptation strategies, in contrast to the static configuration. This difference is influenced by the greater volume of prefetches issued by the Agg Prefetcher. In addition, the gap between useful and late prefetches is significantly wider in the Agg Prefetcher, indicating that the distance adjustment strategy effectively minimizes late prefetches, thus improving overall workload performance. Hence, it shows both more demand accesses per epoch and shorter epochs, outperforming the static configuration of Stream Prefetcher. This demonstrates the advantage of the adaptive strategy in adjusting to this workload demands, especially with the high bandwidth available on the HBM2 domain.

The Agg Prefetcher behavior on the DDR5 domain is shown in Figure 8 for a 4-thread execution. High bandwidth utilization (top right) indicates substantial stress on this memory device. The prefetcher detects this, stays at a low prefetch degree, and simultaneously attempts to adjust the prefetch distance for timeliness. The situation stabilizes after epoch 9500, where a prefetch distance of 34 is found optimal. It then gains better timely prefetches by showing a significantly wider gap between useful and late prefetch curves.

[7]Coverage rate is computed as $\frac{\text{useful prefetches}}{\text{useful prefetches + demand misses}}$. When more demand misses are eliminated by timely prefetches, the coverage approaches 1.0.

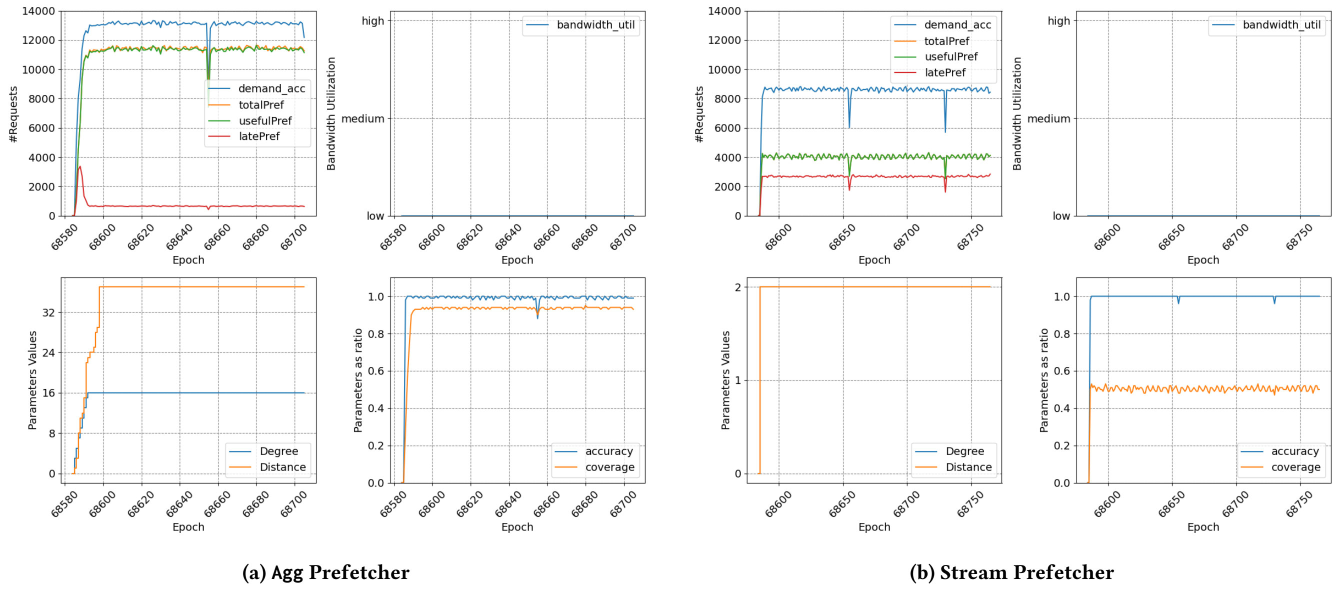


Figure 7: Stream-Triad (single thread) on HBM2 domain: 7a) Agg Prefetcher with adjustable degree, distance, and voting enabled. 7b) Stream Prefetcher in static configuration with both degree and distance = 2. The figure compares tuning operations, bandwidth utilization, prefetching accuracy, and coverage versus static prefetching configurations. The Agg Prefetcher outperforms the static configuration of Stream Prefetcher.

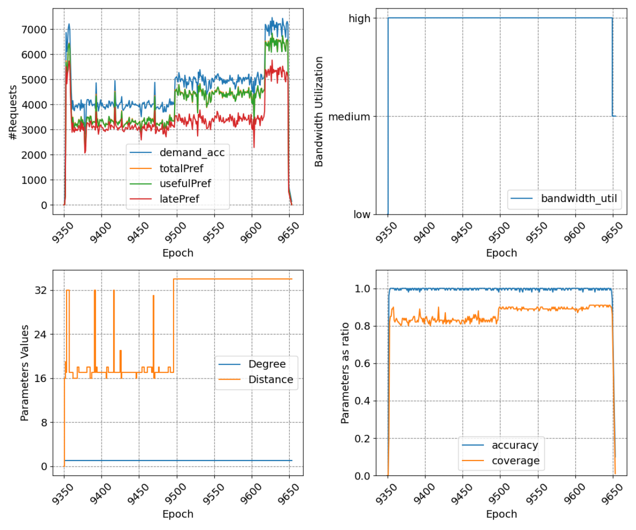


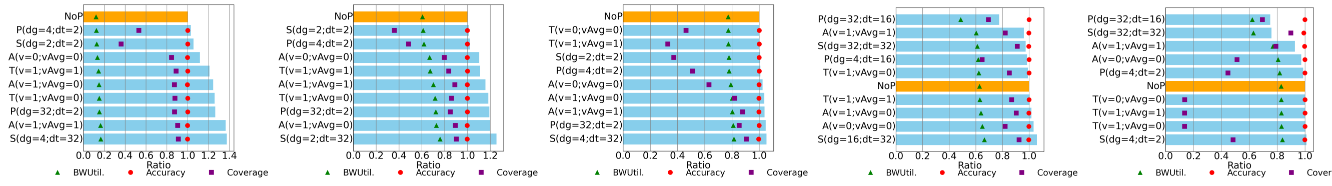
Figure 8: Stream-Triad (4-thread) in the DDR5 domain with Agg Prefetcher: The adaptation maintains a low prefetch degree to avoid bandwidth congestion in the system due to the saturated bandwidth while simultaneously prefetching with greater distance to enhance timeliness.

7.1.2 Exploration. We explore different adaptive configurations in the Agg and TiA Prefetcher, with and without the voting mechanism for distance (Figure 9). With voting ($v = 1$), the prefetch distance is set either to the most voted ($v = 1; vAvg = 0$) or the mean vote ($v = 1; vAvg = 1$) at the beginning of the epoch. Without voting ($v = 0; vAvg = 0$), the adjusted distance will be used immediately for the next prefetch request. We compared all configurations of the two prefetchers (Agg and TiA) with the various static configurations of the Stream and Stride Prefetcher, showing the speedup (ratio of execution time with prefetching vs. the baseline with no prefetching), their bandwidth utilization, prefetch accuracy, and coverage.

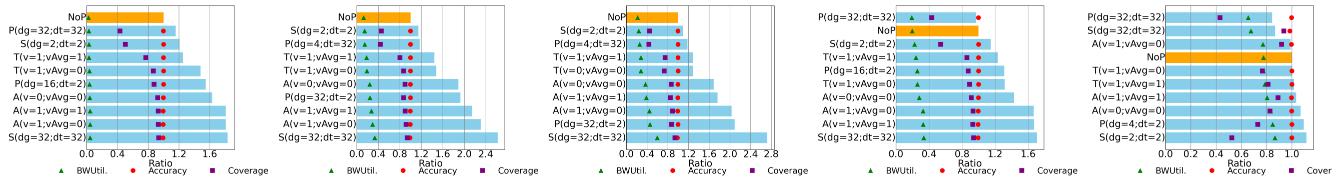
Stream-Triad. Figure 9a to Figure 9e show results of Stream-Triad for various threads on HBM2 and DDR5 domains, with very high prefetch accuracy across all prefetch configurations.

On the HBM2 domain, the single-thread execution (Figure 9a) shows the least effective static configuration of the Stream Prefetcher (degree=2 and distance=2) with a speedup of 1.05×, while the most effective configuration (degree=4 and distance=32) resulted in a speedup of 1.37× due to higher coverage from improved timeliness. The Agg Prefetcher with two voting strategies enabled, ($v = 1; vAvg = 0$) and ($v = 1; vAvg = 1$) shows speedups of 1.36× and 1.34×, respectively, closely approaching the optimal static configuration. The marginal difference between both voting strategies suggests negligible impact, indicating their near-optimal efficacy and outperforming the no-voting strategy. The TiA Prefetcher configurations show similar results.

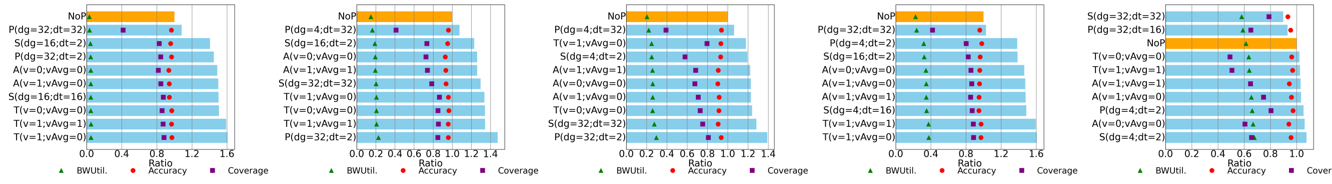
The transition to more threads introduces a new dimension of complexity and performance dynamics. The execution of more threads (Figure 9b and 9c) amplifies the demand on the HBM2 device, leading to an intricate interplay of memory requests and data traffic. Thus, we test the scalability of prefetching strategies and examine the efficacy of adaptive prefetching mechanisms when



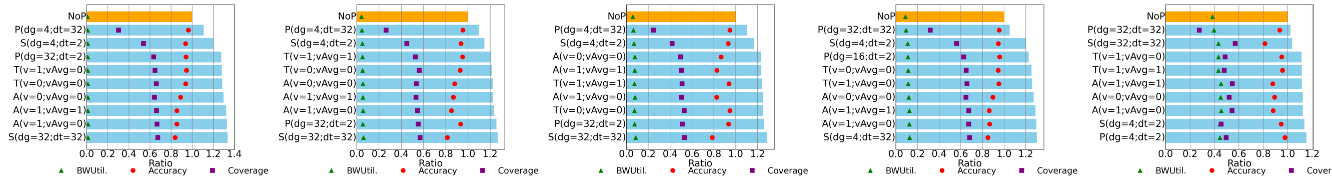
(a) Triad [HBM2;#thr=1] **(b) Triad [HBM2;#thr=8]** **(c) Triad [HBM2;#thr=16]** **(d) Triad [DDR5;#thr=1]** **(e) Triad [DDR5;#thr=8]**



(f) SpMV [HBM2;#thr=1] **(g) SpMV [HBM2;#thr=8]** **(h) SpMV [HBM2;#thr=16]** **(i) SpMV [DDR5;#thr=1]** **(j) SpMV [DDR5;#thr=8]**



(k) waL. [HBM2;#thr=1] **(l) waL. [HBM2;#thr=8]** **(m) waL. [HBM2;#thr=16]** **(n) waL. [DDR5;#thr=1]** **(o) waL. [DDR5;#thr=8]**



(p) Ligra [HBM2;#thr=1] **(q) Ligra [HBM2;#thr=8]** **(r) Ligra [HBM2;#thr=16]** **(s) Ligra [DDR5;#thr=1]** **(t) Ligra [DDR5;#thr=8]**

Figure 9: Performance evaluation of the selected benchmarks for various threads on HBM2 (three left columns) and DDR5 domains (two right columns). 'NoP': No prefetching, 'P': Stride Prefetcher, 'S': Stream Prefetcher, 'A': Agg Prefetcher, 'T': TiA Prefetcher, 'v': voting flag, 'vAvg': average vote flag, 'dg': prefetch degree and 'dt': prefetch distance.

confronted with increased memory access concurrency. For both Agg and TiA Prefetchers, the 8-thread executions with the voting mechanism activated achieved the highest speedups of 1.2× and 1.18×, respectively. Although slightly below the optimal configurations, this still underscores the adaptability of both prefetchers to environmental stress. However, due to bandwidth saturation with 16-thread execution (even without prefetching on the HBM2), no benefit of prefetching is observed despite high prefetch coverage gains.

Transitioning from HBM2 to DDR5 domain introduces a trade-off between increased capacity and reduced memory bandwidth. This trade-off is evident in the performance results illustrated in Figure 9d and Figure 9e. Compared to single-thread runs on the HBM2 domain, DDR5 shows no prefetching benefit due to limited available bandwidth. The least effective static configuration of the Stream and Stride Prefetchers (using large degree and distance) results in speedup drops of 0.97× and 0.77×, respectively. This decrease is attributed to more memory requests and consequently increases latency, leading to delayed data delivery. Conversely, maintaining a distance=32 while reducing the degree is the most efficient method. This suggests that a balanced degree is crucial for optimal prefetching performance on DDR5. The adaptive configurations of both TiA and Agg Prefetchers demonstrate performance enhancements, emphasizing the potential benefits of adjustable parameters.

Advancing to eight threads might intensify the challenge of prefetching effectiveness due to increased demands on the memory system. The least effective static configurations show a speedup drop of around 0.75×. However, TiA and Agg Prefetcher show low-performance drops, emphasizing the effectiveness of self-adjustment strategies, which are capable of adapting to diverse memory environments and securing benefits, even under adverse conditions.

SpMV. MiniFE, designed to emulate finite element generation, assembly, and solution processes for unstructured grid problems, relies on the SpMV kernel as a core component in its iterative solvers. The SpMV kernel has stride access patterns similar to Stream-Triad, along with temporal and indirect memory access patterns. High spatial locality scores (in Section 6.2.1) explain the high accuracy and effectiveness of prefetching configurations when evaluating SpMV across HBM2 and DDR5 domains (Figures 9f to Figure 9j).

The static exploration shows that when the prefetch distance is large (e.g., distance = 32), the Stride Prefetcher gains the least effectiveness, while the Stream Prefetcher gains the highest. The main difference impact lies in the fact that the Stride Prefetcher issues prefetch with addresses starting from the prefetch distance, while the Stream Prefetcher continues to issue prefetch until the monitored window is filled. Thus, fewer prefetches are sent when a large distance is configured for Stride Prefetcher.

As expected, the adaptation strategy in the `Agg` and `TiA` Prefetcher shows high coverage, nearly matching the most effective static configurations. Under single-thread evaluations on HBM2 and DDR5 with high bandwidth available, they achieve speedups close to the optimal static configurations. Moving to multithreaded executions on HBM2 still demonstrates their ability to utilize high-bandwidth memory effectively, achieving significant speedups, e.g., over 2.0× with `Agg` Prefetcher for 8- and 16-threads. The shift to eight threads on DDR5 reveals challenges due to increased bandwidth contention. Even though both `Agg` and `TiA` Prefetcher can switch to less aggressive prefetching to mitigate memory contention impacts.

waLBerla. waLBerla's exploration results (Figures 9k to Figure 9o) show trends similar to SpMV. As expected, the D3Q19 stencil computation exhibiting high spatial locality (Section 6.2.1) explains the benefits of the prefetchers. High coverage with `Agg` and `TiA` Prefetcher highlights the advantages of the adaptation strategy. WaLBerla is more sensitive to PC-based prefetching, showing higher speedups with Stride and `TiA` Prefetcher compared to Stream and `Agg`. In single-thread runs, `TiA` achieves a speedup of 1.6× on HBM2 and DDR5. However, high bandwidth utilization on DDR5 with 8-thread runs shows no significant speedup.

Ligra-BFS. Ligra-BFS exhibits high spatial locality but with a score lower than other kernels (Section 6.2.1) due to random node traversals in the graph processing patterns. Low memory bandwidth utilization on HBM2 and DDR5 makes it beneficial for all prefetching configurations, as shown in Figure 9p to Figure 9t. The random node traversals reduce prefetching accuracy, especially with high aggressiveness, as seen in the results with the static configuration of Stream (degree and distance = 32) and `Agg` and `TiA` Prefetcher. The adaptive strategies of `Agg` and `TiA` improve timeliness (higher coverage gains), maintain accuracy above 0.8, and achieve speedups between 1.15× and 1.3×, close to the most effective static configurations. Due to BFS's low bandwidth utilization, the 8-thread evaluation on the DDR5 shows significant speedup gains (e.g., 1.15× with `Agg`), which are not observed with other kernels.

The exploration results for four kernels emphasize the advantage of the adaptation strategy as it approaches the performance of the best static configurations across threading levels and memory types. Among voting configurations, the most vote strategy ($v = 1; vAvg = 0$) shows the most effective for `Agg` and `TiA` Prefetcher.

7.2 Prefetch effects of NUMA fine-grained memory allocation

We evaluated the effectiveness of the `TiA` and `Agg` Prefetcher using the most vote strategy for various NUMA fine-grained memory allocation strategies.

7.2.1 Stream-Triad, fine-grained. Table 4 presents the results of single-thread executions for the Stream-Triad, comparing the efficiency of various NUMA memory allocation strategies of vectors a,b, and c. Here, 0 indicates their allocation on the HBM2 and 1 on the DDR5. To analyze prefetching, we compare scenarios where prefetching is enabled both on HBM2 and DDR5 with cases where prefetching is solely on HBM2 or DDR5.

When prefetching is disabled, allocating vectors on DDR5 results in more significant performance drops than allocating all vectors on HBM2 (a=0,b=0,c=0 [000]), as seen in the eighth column (*Speedup w.r.t HBM (%)*). In particular, when all vectors are allocated on DDR5 (a=1,b=1,c=1 [111]), a slowdown of almost a speedup of 0.63× is observed. This happens because of the constrained bandwidth availability of the single-channel DDR5.

Once prefetching is enabled, prefetchers benefit from the higher bandwidth available on HBM2, resulting in stronger improvements when more vectors are allocated on HBM2. This explains the higher prefetching effectiveness, as noticed on the 'Only HBM' column compared with the 'Only DDR' column, where individual prefetching is activated either on HBM2 or DDR5, respectively. In particular, allocating all vectors on HBM2 (000) shows the most substantial improvement (1.25×). Conversely, allocating all vectors on DDR5 (111) saturates bandwidth on DDR5 quickly, showing no benefit from prefetching as, in this case, the prefetchers operate in a more conservative prefetching mode.

For the memory NUMA allocation strategies where vectors are allocated across both memory devices, performance improvements depend on achieving balanced prefetching effectiveness for three vectors a, b, and c due to data dependence ($a \longleftarrow C \cdot b + c$). When allocating two vectors on HBM2 (001, 010, 100), prefetchers are likely benefiting from the available bandwidth both on HBM2 and DDR5 (**HBM2**: BW-Util.(RD+WR) up to 8% of the peak; **DDR5**: BW-Util.(RD+WR) up to 38%). Meanwhile, allocating more vectors on DDR5 (011, 101, 110) reduces prefetching effectiveness due to increased bandwidth utilization on DDR5 (BW-Util.(RD+WR) up to 48%) despite the high bandwidth available on HBM2. In these cases, the number of demand requests on DDR5 memory puts more pressure on the bandwidth, leading to increased memory contention. In particular, prefetchers for the case 011 slightly decrease performance due to the high bandwidth utilization (RD) on DDR5 (43%). In this case, prefetchers likely switch between conservative or aggressive prefetching, potentially leading to adverse effects.

In some instances, enabling prefetching on both DDR5 and HBM2 brings less improvement than on HBM2 or DDR5 alone (001, 010, 101, 110). That phenomenon could be the impact of eviction or contention on cache, which needs further investigation. Finally, we prove that the low effectiveness of prefetchers on single-channel DDR5 can be avoided when doubling DDR channels for the Stream-Triad benchmark (not shown in the Table).

7.2.2 MiniFE-CG-Solver, fine-grained. The CG-Solver in miniFE uses not only SpMV but also various other kernels. Figure 10 illustrates all kernels using different colors. SpMV is marked in `multvec` in light blue as a core component. Other key kernels include `Ap.dot` in orange, `daxpy` in grey, `norm2` in yellow, and `daxpby` in green, which present streaming behavior similar to Stream-Triad. Here, the vectors representing the sparse matrix are denoted as A, the

			No Prefetching (NoP)					Prefetching Enabled (Speedup w.r.t NoP)					
			HBM:BW-Util (% peak)		DDR:BW-Util (% peak)		Speedup	HBM and DDR		Only DDR		Only HBM	
a	**b**	**c**	**RD**	**WR**	**RD**	**WR**	**w.r.t HBM**	**Agg**	**TiA**	**Agg**	**TiA**	**Agg**	**TiA**
0	0	0	9.38	2.82	0.0	0.0	1.00	1.25	1.24	1.00	1.00	1.25	1.24
0	0	1	5.74	2.45	22.96	0.0	0.92	1.08	1.04	1.00	1.00	1.11	1.06
0	1	0	5.75	2.45	22.98	0.0	0.92	1.04	1.02	1.01	1.02	1.07	1.02
0	1	1	2.70	1.93	42.87	0.0	0.86	1.00	0.96	0.98	0.96	1.03	1.00
1	0	0	4.90	0.0	19.63	18.17	0.79	1.07	1.04	1.05	1.00	1.04	1.04
1	0	1	2.04	0.0	32.61	15.64	0.65	1.03	1.01	1.05	1.03	1.01	1.01
1	1	0	2.02	0.0	32.25	15.47	0.65	1.00	0.98	1.06	1.04	1.00	1.00
1	1	1	0.0	0.0	47.25	15.32	0.63	1.02	1.00	1.02	1.00	1.00	1.00

Table 4: Stream Triad (Single-thread) for various NUMA memory allocation strategies: HBM2 and DDR5 are encoded as 0 and 1, respectively. The column 'Speedup w.r.t HBM' is the execution time for each memory NUMA allocation normalized to that of HBM2 $a = 0, b = 0, c = 0$, where all memory is allocated onto HBM2 without prefetching. The 'Agg' and 'TiA' columns show speedups, either enhancements or reductions, against the no prefetching for each specific NUMA memory allocation. RD=Read, WR=Write.

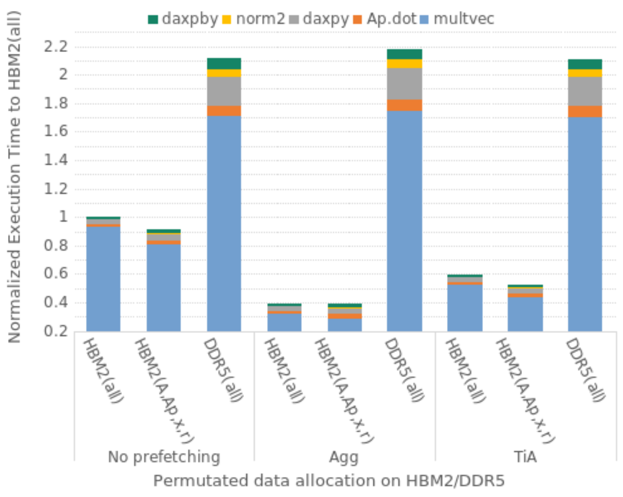


Figure 10: 16-thread evaluation of miniFE-CG-Solver NUMA version.

resulting dense vector as Ap, and x, r, p, and b serve as vectors utilized for the kernels. To investigate the impact of utilizing different memory technologies and the effectiveness of our prefetcher in a NUMA-optimized workload operation, we allocate these vectors across two NUMA domains. The figure presents the outcomes of a 16-threaded evaluation, where lower values are better.

The notation "HBM2(all)" indicates that all vectors are allocated on the HBM2, whereas "HBM2(A,Ap,x,r)" specifies which vectors are allocated on the HBM2, with the remaining vectors allocated on the DDR5. Similarly, "DDR5(all)" denotes that all vectors are allocated on the DDR5. The x-axis illustrates the memory allocation scenarios, differentiating between no prefetching, `Agg`, and `TiA` Prefetcher. The y-axis depicts the execution time normalized to HBM2.

When excluding prefetching, it becomes evident that allocating vectors leveraging spatial locality on HBM2 while assigning others to DDR, such as those involving indirect accesses ("HBM2(A,Ap,x,r)"), results in enhanced performance as it increases the overall available bandwidth. Conversely, allocating all vectors solely to DDR5 leads to a performance deterioration of 2.1× compared to HBM2.

Our optimization strategies, as implemented in the `Agg`, have shown their benefit by significantly reducing the runtime to 0.4× for all vectors allocated on HBM2. This benefit persists even with partial memory allocation across HBM2 and DDR5, as observed previously. It is noteworthy that while certain kernels, such as SpMV, benefit from this partial allocation due to indirect memory accesses, others like `daxpby` and `Ap.dot` show small drawbacks. Despite these variations in kernel performance, the overall runtime remains unchanged. Allocating all vectors exclusively on DDR5 results in a performance degradation of 2.17× due to the reduction in available memory bandwidth. The TiA Prefetcher notably reduces the runtime by a factor 0.58× and 0.52× when allocating all vectors or partially allocating them on HBM2, respectively. Similar to `Agg`, allocating all vectors exclusively on DDR5 results in a performance decline. Overall, while the Agg Prefetcher exhibits slightly more influence on this workload, both prefetchers effectively enhance performance by intelligently adapting their parameters to the utilized memory technologies and the measured statistics.

8 Related Work

One of the most notable techniques for dynamically adapting the aggressiveness of the prefetcher is the Feedback-Directed Prefetching Mechanism (FDP) [57]. The FDP Prefetcher analyzes prefetch accuracy, timeliness, and cache pollution during execution, allowing immediate adjustments to prefetch aggressiveness and cache insertion policies. Moreover, the paper presents a windowing technique designed to dynamically monitor memory accesses within a defined range for Stream Prefetcher. Our prefetcher draws inspiration from this methodology. Further efforts have also aimed to dynamically control the aggressiveness of the prefetcher. For example, the techniques described in [12, 64] target improving timeliness, and the bandwidth-aware techniques described in [6, 39] aimed at avoiding high-contention bandwidth consumption. Our work aligns in the same direction; however, we focus on improving prefetching for hybrid memory scenarios.

Some recent prefetchers support dynamic adjustments for aggressive prefetching [16, 22, 23, 34, 40]. For instance, the Berti Prefetcher is a local-delta prefetcher attached to the L1 data cache [40]. Berti focuses on refining prefetch decisions based on the difference between successive cache line addresses initiated by the same instruction. By differentiating between deltas and strides, Berti's methodology mirrors the approach of learning timeliness by employing a history table to discover new prefetch triggers and utilizing latency measurements to issue prefetch requests. T-SKID Prefetcher decouples the timing of prefetch operations from their address predictions [23]. T-SKID leverages temporal correlation between load instruction Program Counters (PCs) to pick the best prefetch trigger and gain more timely prefetches. Access Map Pattern Matching (AMPM) Prefetcher [16] utilizes a memory access map and pattern-matching logic to predict and prefetch data. AMPM learns about bandwidth consumption and adjusts the prefetch degree dynamically to avoid high-bandwidth contention at runtime. Our work aligns with the AMPM approach. However, ours focuses on classifying memory access and tuning the prefetch degree for each memory device. The SPP Prefetcher [22] adaptively throttles only prefetch degree. The Best-Offset Prefetcher [34] attempts to fine-tune the prefetch offset, which is the difference between successive addresses. Both techniques aim at achieving more timely prefetches. Arm recently introduced the Completer Busy (CBusy), which detects system congestion and dynamically guides the aggressiveness of advanced prefetching engines in Neoverse processors [8, 48]. Our technique can combine with Cbusy to adapt aggressiveness for heterogeneous memories.

In heterogeneous memory architecture, we noticed a few works aimed at improving prefetchers' effectiveness. In [28], a stream-based prefetcher located at the main memory controller is proposed to improve the long latency of flash memory within a hybrid DRAM/NAND flash memory architecture. The prefetcher tracks multi-stream of page access and prefetch pages in NAND flash into DRAM to mitigate the long latency of the flash accesses. The prefetcher implements a Global History Buffer (GHB) structure [42] to store missed pages and uses the historical info to select better candidates for future prefetching. Our multi-stream prefetcher also supports multi-stream tracking, but our work is dedicated to on-cache prefetching. In our proposal, the 'Fill Queue' is also an implemented GHB structure; it is not globally used. Instead, each queue is dedicated to each prefetch stream.

Barrera [58] introduced a NUMA-aware L1 Stride Prefetcher within a two-socket NUMA system. This prefetcher tunes the prefetch degree parameter for access streams targeting local or remote memory locations. A prefetch request to the remote location, where the request address is near the trigger address, will fetch data directly in L1. Meanwhile, the prefetch, where the address is far from the trigger address, places data in L3. By doing that, high latency for remote access can be mitigated. More closely related to our work, an effort to optimize timeliness for stream-based prefetchers within a NUMA architecture is described in [14]. In this approach, the aggressiveness associated with each memory device is tuned using a prefetch degree threshold, and prefetching is throttled by monitoring the wrong prefetch fraction. Our approach differs from both these techniques in that the prefetcher learns timeliness and bandwidth consumption at run-time, dynamically adjusting both prefetch distance and degree parameters for either more aggressively prefetching or not.

9 Conclusion

In this paper, we optimized hardware prefetchers in a mixing HBM2 and DDR5 memory architecture using a gem5 model with 20 Arm Neoverse V1-like cores. We analyzed the characteristics of HBM2 and DDR5 memory technologies and their impacts on hardware prefetchers within a global physical address configuration. The exploration of static configurations of prefetchers showed that memory-bound kernels (SpMV, waLBerla, Stream) benefited from increased aggressiveness when accessing data on HBM2, while higher latency and lower bandwidth on DDR5 make a low aggressiveness strategy necessary to avoid performance slow-downs. That emphasizes the need to dynamically adjust prefetching aggressiveness for each memory technology used in the architecture.

We have introduced a hybrid-memory-aware prefetching technique that dynamically tunes the aggressiveness of prefetchers, leveraging run-time bandwidth feedback for prefetch degree adjustment and the queuing structure for prefetch distance adjustment specific to each memory type. This technique has demonstrated significant advantages when integrated into Stride and Stream Prefetchers. Evaluation with four HPC kernels, either with coarse-grain or fine-grain NUMA optimized codes, shows our technique's ability to guide both prefetchers to near-optimal static configurations and thus improves overall system performance. Looking ahead, our future work includes investigating prefetching across page boundaries potentially for higher timeliness gain, examining the hardware cost of implementation, and evaluating the technique integrated into more hardware prefetchers.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. Special thanks to Tiago Muck from Arm Research for supporting and granting early access to the gem5 AMBA-CHI implementation. This work has received funding from the European High Performance Computing Joint Undertaking (JU) under Framework Partnership Agreement No 800928 and Specific Grant Agreement No 101036168 EPI-SGA2. The JU receives support from the European Union's Horizon 2020 research and innovation programme and from Croatia, France, Germany, Greece, Italy, Netherlands, Portugal, Spain, Sweden, and Switzerland.

References

[1] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, Alexander Driskill-Smith, and Mohamad Krounbi. 2013. Spin-transfer torque magnetic random access memory (STT-MRAM). *J. Emerg. Technol. Comput. Syst.* 9, 2, Article 13 (may 2013), 35 pages. https://doi.org/10.1145/2463585.2463589

[2] Arm. 2020. *AMBA® 5 CHI architecture specification.* https://developer.arm.com/documentation/ihi0050/ea/,2020

[3] Arm. 2021. *Arm® Neoverse™ CMN-650 Coherent Mesh Network Technical Reference Manual.* https://developer.arm.com/documentation/101481/0200/?lang=en

[4] Arm. 2021. *Arm® Neoverse™ V1 reference design - software developer guide.* https://developer.arm.com/documentation/PJDOC-1779577084-33214/RelG?lang=en

[5] Jean-Loup Baer and Tien-Fu Chen. 1995. Effective Hardware-Based Data Prefetching for High-Performance Processors. *IEEE Trans. Comput.* 44, 5 (may 1995), 609–623. https://doi.org/10.1109/12.381947

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

[7] Bine Brank, Stepan Nassyr, Fatemeh Pouyan, and Dirk Pleiter. 2020. Porting Applications to Arm-based Processors. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. 559–566. https://doi.org/10.1109/CLUSTER49012.2020.00079

[8] Magnus Bruce. 2023. Arm Neoverse V2 platform: Leadership Performance and Power Efficiency for Next-Generation Cloud Computing, ML and HPC Workloads. In *2023 IEEE Hot Chips 35 Symposium (HCS)*. 1–25.

[9] Hariharan Devarajan, Anthony Kougkas, and Xian-He Sun. 2020. Hfetch: Hierarchical data prefetching for scientific workflows in multi-tiered storage environments. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 62–72.

[10] Babak Falsafi and Thomas F Wenisch. 2022. *A primer on hardware prefetching.* Springer Nature.

[11] Christian Feichtinger, Jan Götz, Stefan Donath, Klaus Iglberger, and Ulrich Rüde. 2009. *WaLBerla: Exploiting Massively Parallel Systems for Lattice Boltzmann Simulations.* Springer London, London, 241–260. https://doi.org/10.1007/978-1-84882-409-6_8

[12] Wim Heirman, Kristof Du Bois, Yves Vandriessche, Stijn Eyerman, and Ibrahim Hur. 2018. Near-side prefetch throttling: adaptive prefetching for high-performance many-core processors. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques* (Limassol, Cyprus) *(PACT '18)*. Association for Computing Machinery, New York, NY, USA, Article 28, 11 pages. https://doi.org/10.1145/3243176.3243181

[13] Wim Heirman, Ibrahim Hur, Ugonna Echeruo, Stijn Eyerman, and Kristof Du Bois. 2020. Apparatus, method, and system for enhanced data prefetching based on non-uniform memory access (NUMA) characteristics. US Patent 10,621,099.

[14] Wim Heirman, Ibrahim Hur, Ugonna Echeruo, Stijn Eyerman, and Kristof du Bois. U.S. Patent 11, 256, 626 B2, Feb. 22, 2022. Apparatus, method, and system for enhanced data prefetching based on non-uniform memory access (NUMA) characteristics.

[15] Mark D. Hill. 2019. Three Other Models of Computer System Performance. *CoRR* abs/1901.02926 (2019). arXiv:1901.02926 http://arxiv.org/abs/1901.02926

[16] 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.

[17] Djordje Jevdjic, Gabriel H. Loh, Cansu Kaynak, and Babak Falsafi. 2014. Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 25–37. https://doi.org/10.1109/MICRO.2014.51

[18] Djordje Jevdjic, Stavros Volos, and Babak Falsafi. 2013. Die-stacked DRAM caches for servers: hit ratio, latency, or bandwidth? have it all with footprint cache. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 404–415. https://doi.org/10.1145/2485922.2485957

[19] D. Joseph and D. Grunwald. 1999. Prefetching using Markov predictors. *IEEE Trans. Comput.* 48, 2 (1999), 121–133. https://doi.org/10.1109/12.752653

[20] Norman P. Jouppi. 1990. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. *SIGARCH Comput. Archit. News* 18, 2SI (may 1990), 364–373. https://doi.org/10.1145/325096.325162

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

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

[23] Toru Koizumi, Tomoki Nakamura, Yuya Degawa, Hidetsugu Irie, Shuichi Sakai, and Ryota Shioya. 2022. T-SKID: predicting when to prefetch separately from address prediction. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1389–1394.

[24] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. 2014. A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units. *SIAM Journal on Scientific Computing* 36, 5 (2014), C401–C423. https://doi.org/10.1137/130930352

[25] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. 37, 3 (jun 2009), 2–13. https://doi.org/10.1145/1555815.1555758

[26] Hyung Gyu Lee, Seungcheol Baek, Chrysostomos Nicopoulos, and Jongman Kim. 2011. An energy- and performance-aware DRAM cache architecture for hybrid DRAM/PCM main memory systems. In *2011 IEEE 29th International Conference on Computer Design (ICCD)*. 381–387. https://doi.org/10.1109/ICCD.2011.6081427

[27] Shang Li, Dhiraj Reddy, and Bruce Jacob. 2018. A performance & power comparison of modern high-speed dram architectures. In *Proceedings of the International Symposium on Memory Systems*. 341–353.

[28] Ing-Chao Lin, Da-Wei Chang, Wei-Jun Chen, Jian-Ting Ke, and Po-Han Huang. 2020. Global Clean Page First Replacement and Index-Aware Multistream Prefetcher in Hybrid Memory Architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 9 (2020), 1750–1763. https://doi.org/10.1109/TCAD.2019.2925404

[29] Gabriel H. Loh. 2008. 3D-Stacked Memory Architectures for Multi-core Processors. In *2008 International Symposium on Computer Architecture*. 453–464. https://doi.org/10.1109/ISCA.2008.15

[30] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jerónimo Castrillón, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Marjan Fariborz, Amin Farmahini Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc S. Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. 2020. The gem5 Simulator: Version 20.0+. *CoRR* abs/2007.03152 (2020). arXiv:2007.03152 https://arxiv.org/abs/2007.03152

[31] John D. McCalpin. [n. d.]. *STREAM: Sustainable Memory Bandwidth in High Performance Computers.* https://www.cs.virginia.edu/stream/

[32] Jagan Singh Meena, Simon Min Sze, Umesh Chand, and Tseung-Yuen Tseng. 2014. Overview of emerging nonvolatile memory technologies. *Nanoscale Research Letters* 9, Article 526 (2014). https://doi.org/10.1186/1556-276X-9-526

[33] Mitesh R. Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H. Loh. 2015. Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 126–136. https://doi.org/10.1109/HPCA.2015.7056027

[34] Pierre Michaud. 2016. Best-offset hardware prefetching. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 469–480. https://doi.org/10.1109/HPCA.2016.7446087

[35] Inc Micron Technology. 2022. *DDR5 SDRAM Product Core Data Sheet.* https://www.micron.com/products/memory/dram-components/ddr5-sdram/part-catalog/part-detail/mt60b2g8hb-48b-a,2022

[36] Sparsh Mittal. 2016. A survey of recent prefetching techniques for processor caches. *ACM Computing Surveys (CSUR)* 49, 2 (2016), 1–35.

[37] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. 2010. Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. In *High Performance Embedded Architectures and Compilers*, Yale N. Patt, Pierfrancesco Foglia, Evelyn Duesterwald, Paolo Faraboschi, and Xavier Martorell (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 111–125.

[38] Tiago Mück. 2021. *CHI.* https://www.gem5.org/documentation/general_docs/ruby/CHI/

[39] Carlos Navarro, Josué Feliu, Salvador Petit, Maria E. Gómez, and Julio Sahuquillo. 2020. Bandwidth-Aware Dynamic Prefetch Configuration for IBM POWER8. *IEEE Transactions on Parallel and Distributed Systems* 31, 8 (2020), 1970–1982. https://doi.org/10.1109/TPDS.2020.2982392

[40] Agustín Navarro-Torres, Biswabandan Panda, Jesús Alastruey-Benedé, Pablo Ibánez, Víctor Viñals-Yúfera, and Alberto Ros. 2022. Berti: an Accurate Local-Delta Data Prefetcher. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 975–991.

[41] Agustín Navarro-Torres, Biswabandan Panda, Jesús Alastruey-Benedé, Pablo Ibáñez, Víctor Viñals-Yúfera, and Alberto Ros. 2022. Berti: an Accurate Local-Delta Data Prefetcher. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 975–991. https://doi.org/10.1109/MICRO56248.2022.00072

[42] K.J. Nesbit and J.E. Smith. 2004. Data Cache Prefetching Using a Global History Buffer. In *10th International Symposium on High Performance Computer*

Architecture (HPCA'04). 96–96. https://doi.org/10.1109/HPCA.2004.10030

[43] Nuno Neves, Pedro Tomás, and Nuno Roma. 2020. Compiler-assisted data streaming for regular code structures. *IEEE Trans. Comput.* 70, 3 (2020), 483–494.

[44] Anant Vithal Nori, Jayesh Gaur, Siddharth Rai, Sreenivas Subramoney, and Hong Wang. 2018. Criticality Aware Tiered Cache Hierarchy: A Fundamental Relook at Multi-Level Cache Hierarchies. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 96–109. https://doi.org/10.1109/ISCA.2018.00019

[45] Geraldo F. Oliveira, Juan Gómez-Luna, Lois Orosa, Saugata Ghose, Nandita Vijaykumar, Ivan Fernandez, Mohammad Sadrosadati, and Onur Mutlu. 2021. DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks. *IEEE Access* 9 (2021), 134457–134502. https://doi.org/10.1109/ACCESS.2021.3110993

[46] Biswabandan Panda. 2023. CLIP: Load Criticality based Data Prefetching for Bandwidth-constrained Many-core Systems. In *2023 56th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 714–727.

[47] J. Thomas Pawlowski. 2011. Hybrid memory cube (HMC). In *2011 IEEE Hot Chips 23 Symposium (HCS)*. 1–24. https://doi.org/10.1109/HOTCHIPS.2011.7477494

[48] Andrea Pellegrini. 2021. Arm Neoverse N2: Arm's 2nd generation high performance infrastructure CPUs and system IPs. In *2021 IEEE Hot Chips 33 Symposium (HCS)*. 1–27.

[49] Mantevo Project. [n. d.]. *miniFE Finite Element Mini-Application*. https://github.com/Mantevo/miniFE

[50] Yasir Mahmood Qureshi, William Andrew Simon, Marina Zapater, Katzalin Olcoz, and David Atienza. 2021. Gem5-X: A Many-core Heterogeneous Simulation Platform for Architectural Exploration and Optimization. *ACM Trans. Archit. Code Optim.* 18, 4, Article 44 (jul 2021), 27 pages. https://doi.org/10.1145/3461662

[51] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. 2011. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing* (Tucson, Arizona, USA) *(ICS '11)*. Association for Computing Machinery, New York, NY, USA, 85–95. https://doi.org/10.1145/1995896.1995911

[52] Sudhanshu Shukla, Sumeet Bandishte, Jayesh Gaur, and Sreenivas Subramoney. 2022. Register file prefetching. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 410–423. https://doi.org/10.1145/3470496.3527398

[53] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory *(PPoPP '13)*. Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/2442516.2442530

[54] A. J. Smith. 1978. Sequential Program Prefetching in Memory Hierarchies. *Computer* 11, 12 (dec 1978), 7–21. https://doi.org/10.1109/C-M.1978.218016

[55] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. 2016. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro* 36, 2 (2016), 34–46. https://doi.org/10.1109/MM.2016.25

[56] 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

[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] Isaac Sánchez Barrera. 2022. *Exploiting data locality in cache-coherent NUMA systems*. Ph. D. Dissertation. Universitat Politècnica de Catalunya. http://hdl.handle.net/2117/367546

[59] Y. C. Tay. 2013. *Analytical Performance Modeling for Computer Systems* (2nd ed.). Morgan & Claypool Publishers.

[60] Tinymembench. [n. d.]. *Simple benchmark for memory throughput and latency*. https://www.cs.virginia.edu/stream/

[61] Andrey Vladimirov and Ryo Asai. 2016. *Clustering Modes in Knights Landing Processors: Developer's Guide*. https://colfaxresearch.com/knl-numa/

[62] Jonathan Weinberg, Michael O. McCracken, Erich Strohmaier, and Allan Snavely. 2005. Quantifying Locality In The Memory Access Patterns of HPC Applications. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC '05)*. IEEE Computer Society, USA, 50. https://doi.org/10.1109/SC.2005.59

[63] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News* 23, 1 (1995), 20–24. https://doi.org/10.1145/216585.216588

[64] Huaiyu Zhu, Yong Chen, and Xian-He Sun. 2010. Timing local streams: improving timeliness in data prefetching. In *Proceedings of the 24th ACM International Conference on Supercomputing* (Tsukuba, Ibaraki, Japan) *(ICS '10)*. Association for Computing Machinery, New York, NY, USA, 169–178. https://doi.org/10.1145/1810085.1810110