Rheinische
Friedrich-Wilhelms-Universität Bonn

Master Thesis

Heterogeneous Memory Aware Prefetching on High Performance Arm Processors

Author:
Berk Saglam

Supervisor:
Dr. Nam. Ho

First Examiner:
Prof. Dr. Estela Suarez

Second Examiner:
Dr. Sven Mallach

Date: April 10, 2024

Declaration

I hereby declare that I am the sole author of this thesis and that none other than the specified sources and aids have been used. Passages and figures quoted from other works have been marked with appropriate mention of the source.

Bonn, 10.04.2024

Place, Date

Signature

Abstract

Modern computing often sees up to 80% of computation time spent on data retrieval, emphasizing the importance of prefetching for enhancing CPU data delivery speeds by moving data from slower storage to faster caches. Balancing timeliness and aggressiveness is crucial for reducing access times. Utilizing heterogeneous memory, in this context HBM2 and DDR5, serve different roles due to their bandwidth and capacity trade-offs, underscoring the need for balanced memory management and awareness while prefetching.

This work focuses on developing prefetching strategies for heterogeneous memory configurations in high-performance Arm processors, targeting a system architecture comprising 20 cores, with 16 cores dedicated to HBM2 and 4 cores dedicated to DDR5 memory. The primary objective is to reduce latency and improve system performance by introducing two innovative optimization strategies for prefetching. These strategies meticulously balance timeliness and aggressiveness by adaptively tuning the prefetch degree and distance. These strategies adapt dynamically to the specific memory type and available bandwidth with consideration of the prefetch accuracy, optimizing prefetching operations for enhanced performance and efficiency. The Prefetcher are integrated with the L2 cache and its performance is rigorously assessed through Gem5 simulations. These evaluations compare the effectiveness of adaptive optimization strategies for both Stream and PC-based Stride Prefetchers, utilizing the Arm Neoverse V1 as the computational model.

Findings reveal adaptive prefetching is boosting system performance, notably with HBM2 and DDR5 Memory, while facing memory contention on DDR5. This research advances prefetching strategies with the understanding of heterogeneous memory, advocating further exploration to enhance high-performance computing efficiency and performance.

Acknowledgments

My heartfelt gratitude goes out to all those who have supported and contributed to the completion of this thesis. Foremost, Dr. Nam Ho, my dedicated supervisor, deserves special recognition. His invaluable guidance, expertise, and unwavering support throughout the research process have been significant in this work. Dr. Nam Ho's commitment to my success has made a profound impact, and for that, I am deeply grateful. The members of the institute Forschungszentrum Jülich played a pivotal role in facilitating my research. Their provision of resources and essential facilities, along with their assistance and cooperation, were crucial in conducting experiments and collecting data. Acknowledgment is also due to the generous financial support received, without which this research would not have been possible. Appreciation is extended to all the members of the EPI, NOVAS Meetings for their invaluable assistance in searching for solutions to problems encountered during the course of this research. Their collective efforts were indispensable. I appreciate my second examiner, Dr. Sven Mallach, for accepting this thesis as an examiner. Lastly, profound gratitude is expressed to the first examiner, Prof. Dr. Estela Suarez, whose belief in my capabilities and the opportunity provided allowed me, as a student of the University of Bonn, to conduct this thesis at Forschungszentrum Jülich. This has been pivotal in my academic journey. This thesis owes its existence to the collective support and encouragement of all these individuals and organizations. Thank you for being integral to this transformative journey.

Contents

1. Introduction

In the area of computer science, data-intensive processing stands as a critical cornerstone. Simulations, in particular, represent indispensable tools that significantly enhance insights across various research domains. Their role extends beyond fostering innovation, often leading to groundbreaking scientific advancements, such as the recent developments in hardware. In such computations, the access of data in time is a driving factor of the total computation time. In some modern applications, up to 80% of the total computation time is spent on data retrieval. Hence, the speed at which data can be delivered to the CPU (Central Processing Unit) has become a major determinant, prompting the evolution of faster storage devices [12].

A key challenge in this evolution has been the persistent latency issues in the interface between storage solutions and the CPU [3, 12, 20]. This challenge, often exacerbated by the rapid advancements in CPU capabilities as predicted by Moore's Law, has led to a significant performance bottleneck, commonly referred to as the 'Memory Wall'. The Memory Wall underscores the disparity 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 and efficiency.

To address these challenges, advancements in memory hierarchy, including the development of low-latency caches and prefetching techniques, have been explored. These innovations aim to reduce latency and improve data access times, thereby enhancing overall system performance.

Prefetching enhances system performance by proactively fetching data from slower storage to faster caches, reducing access times significantly [12, 18, 27, 33]. However, it is crucial that prefetching delivers data timely and accurately to be effective. The level of prefetch aggressiveness is a delicate balance, as being too aggressive can lead to cache evictions, increased energy consumption, and potential performance trade-offs [16, 18, 22, 33]. "Aggressiveness" in this context denotes the number of prefetches sent by the Prefetcher w.r.t. the prefetch degree and distance. Under ideal conditions, prefetching eliminates cache misses, making it a standard feature in high-performance processors [25].

Therefore, balancing timeliness and aggressiveness is reducing access times. Utilizing heterogeneous memory, in this context HBM2 (High-Bandwidth Memory 2) and DDR5 (Double Data Rate 5), serve different roles due to their bandwidth and capacity trade-offs, underscoring the need for balanced memory management, like shown in the work [36]. By following the recommended strategies for memory allocation, thread pinning, and selecting the appropriate clustering mode, developers can significantly enhance application performance, especially in scenarios demanding high parallelism and memory bandwidth. Therefore, the integration of memory-awareness into prefetching strategies emerges as a promising approach to leverage these benefits further, laying the groundwork for this research.

This work focuses on developing memory-aware Prefetcher and strategies to minimize system latency across multi-node architectures with heterogeneous memory environments. These Prefetcher are designed to optimize the balance between prefetch aggressiveness and timeliness, utilizing adaptive strategies informed by metrics such as bandwidth utilization and accuracy. Focusing on an implementation of the Arm (Advanced RISC Machines) Instruction Set Architecture, provided by [15]. The study explores Stream and PC-based Stride Prefetcher, analyzing their timeliness, aggressiveness, and accuracy w.r.t. the developed adaptive strategies. The research utilizes the Gem5 Simulator. Results of this study underscore the importance of adaptive prefetching in enhancing system performance by dynamically identifying near-optimal prefetch degrees and distances across diverse workloads. This approach contrasts with static configurations, which require workload-specific optimizations that are impractical to implement universally. Furthermore, the introduced adaptive optimization strategies have led to notable performance enhancements, most prominently with high-bandwidth memory types like HBM2. However, the effectiveness of these strategies varies across different memory types. Specifically, DDR5 encounters distinct obstacles under conditions of high memory usage for memory-intensive applications, such as Simple Triad. This is attributed to memory contention issues arising from elevated demand requests to the DDR5 device.

To provide a comprehensive understanding, this thesis is structured methodically. Starting with “Theoretical Fundamentals” chapter (2), delving into essential concepts such as Memory Systems, Prefetching Strategies, the High-End Arm CPU Architecture, and introduce the Gem5 Simulator. This foundational chapter lays the groundwork for the subsequent chapters. The heart of the research lies in the “Methodology and Implementation” chapter (3). Here, the implemented Arm architecture is briefly analyzed. Furthermore, basic implementation ideas and optimization strategies for the implemented Prefetcher are elucidated. In the “Experimental Analysis and Evaluation” chapter (4), insights into the experimental environment and the used benchmarks for evaluation

are given. In addition, the performance of the implemented prefetching techniques are analyzed, and their effectiveness evaluated. Moving forward, in the "Related Work" chapter (5), existing research, methodologies, and findings relevant to this thesis are meticulously compared to the approaches presented in this work. Lastly, the "Conclusions and Future Directions" chapter (6) summarizes the key findings and draws conclusions from the analyses and suggests potential directions for future research in this field.

2. Theoretical Fundamentals

This chapter is a foundational exploration essential for grasping the subsequent sections of this thesis. Beginning with exploring the concept of memory systems and prefetching, it is followed by the evolutionary trajectory and profound significance of Arm architecture. Additionally, this chapter introduces the Gem5 simulation tool, an instrumental asset for analyzing and evaluating diverse computer architectures.

2.1. Memory System

2.1.1. Basic Computer Architecture

The fundamental architecture of a computer can be described through the Von Neumann model, which illustrates several key components: an I/O (Input/Output) interface, a Processor (also known as the Central Processing Unit or CPU), and the Main Memory, as depicted in Figure 2.1.

At the heart of the computer lies the CPU, containing an Arithmetic Unit and a Control Unit. In more advanced systems, multiple Arithmetic and Control Units are present, each known as a “Core”. Here, the terms “Processor” and “CPU” refer to the entire chip.

Serving as a bridge for data exchange, the I/O interface connects the computer with external devices. It enables interaction with various peripherals, such as mice, keyboards, and monitors.

For executing computational operations, the Arithmetic Unit is tasked with everything from basic arithmetic to complex mathematical functions. The Control Unit, on the other hand, manages the operations of the CPU, directing the flow of data within the system and managing the execution of instructions.

The Main Memory plays a pivotal role in this architecture. It temporarily stores data and instructions that are in active use, allowing access by the CPU. The efficient functioning of the Main Memory is important for the overall performance of the computer, as it directly impacts the speed at which data can be processed and retrieved. In this context, “latency” refers to the duration required to transfer data from the main memory to the CPU. Lower

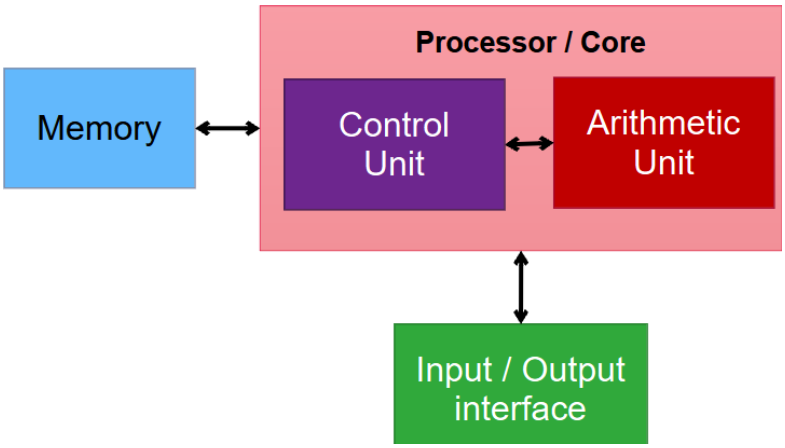


Figure 2.1.: Von Neumann Architecture: 1) Processor - Executes instructions and processes data. 2) Arithmetic Unit - Performs mathematical operations. 3) Control Unit - Manages the execution of instructions. 4) Main Memory - Stores data and instructions for quick access. 5) Input/Output Interface - Communication with external devices like keyboards, mice, and monitors. Adapted from [35].

latency leads to a more rapid delivery of data and instructions, which are stored in the main memory, to the CPU. This acceleration is key to enhancing the performance of any given workload. A bottleneck in this process can lead to significant performance degradation, as it hinders the timely delivery of essential data needed for computation. Such a situation often arises when the maximum bandwidth of the system is reached, leading to increased latency [14, 35].

2.1.2. The Memory Wall

While storage solutions have seen improvements in both capacity and overall bandwidth speed, they still grapple with latency issues when interfacing with the CPU [3, 12, 20]. This latency gap has been a persistent challenge since 1970, a phenomenon represented in Moore's Law, which states that the number of transistors on a computer chip doubles approximately every two years, thereby rapidly advancing computational power. However, memory and storage devices, like RAM and various storage drives, have not scaled comparably to processing units. Despite advancements in memory capacity and speed, the latency between these devices remains too high, creating a performance bottleneck. This challenge is commonly known as the "Memory Wall" [12, 35] and can be seen in Figure 2.2.

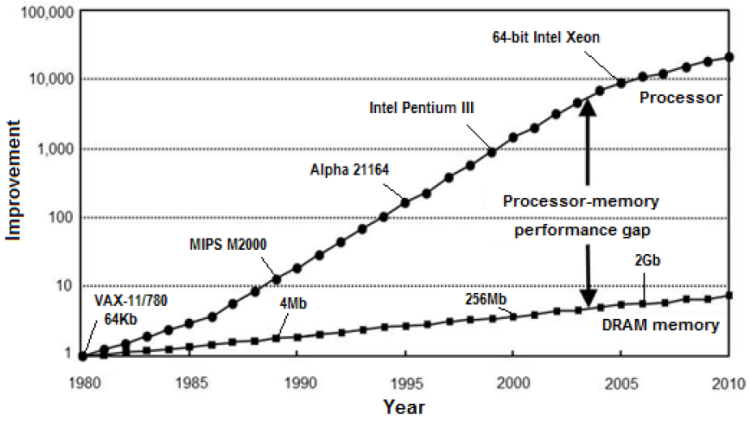


Figure 2.2.: The figure represents Moore's Law, where the number of transistors on a chip doubles every two years, significantly enhancing computational power. However, memory performance lacks behind this rapid growth, resulting in a visible gap between transistor density and memory advancement. Adapted from [13].

2.1.3. Memory Hierarchy

To solve the "Memory Wall" problem, the best solution would be a low latency, high capacity memory device, which is realizable but too expensive to implement. However, an observation is that workloads to the CPU often access only a small proportion of the memory at a time. Temporal locality is a term used for access of memory that is likely to be accessed again soon. On the other hand, spatial locality means that nearby memory regions are likely to be accessed soon [20, 35]. Given this knowledge, the ideal solution is to have a memory hierarchy, where storage that is further away from the CPU has higher latency but greater capacity, but storage that is closer to the CPU, like caches, has low latency but smaller capacity. The idea hereby is to load data to the next level of hierarchy if the data locality is likely to be exploited by the workload [3, 35]. Figure 2.3 illustrates this idea. Every layer of the memory hierarchy is explained from the lowest level to the highest in the subsequent sections.

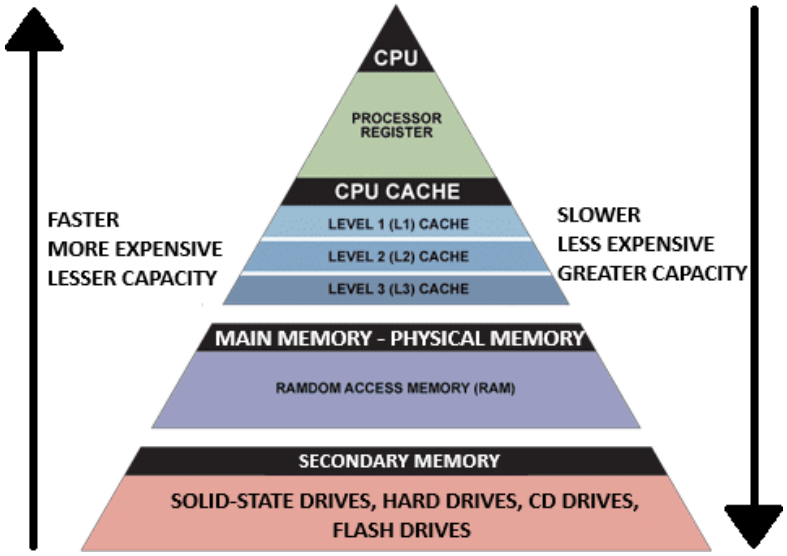


Figure 2.3.: Illustration of the Memory Hierarchy: Starting from lower-level devices with higher capacity, lower cost, and higher latency, the hierarchy ascends to more expansive, quicker latency, and smaller storage memory levels. Adapted from [35].

Secondary Memory

Secondary memory is a key layer in the memory hierarchy of a computer by providing long-term data storage without the need for power, unlike main memory, which requires power to retain information. This characteristic allows secondary memory devices, including SSDs (Solid-State Drives), hard drives, CD drives, flash drives, etc. to preserve data even when the computer has no power. While these devices offer greater storage density at a lower cost, they inherently suffer from higher latency compared to main memory. During workloads data from the secondary memory must be loaded into the main memory, for example RAM (Random Access Memory), before the processor has access to it. The transition is a significant factor in the delay experienced during program startups. SSDs, leveraging flash memory without moving parts, offer a compromise between the larger capacity and lower cost of hard drives and the faster performance, albeit at a higher price and with average capacity.

Main Memory - Physical Memory

Physical Memory, for example, RAM, is a critical layer in the memory hierarchy and is used in the Stream-based Prefetcher implementation, more on that in Section 3.3. A computer loads and processes data and programs that are currently in use in the physical

memory. The speed of physical memory significantly exceeds that of the secondary memory. However, it is volatile, leading to loss of data when the power is turned off.

Virtual memory allows systems to manage larger applications beyond the constraints of physical memory, by efficiently swapping data between secondary memory and main memory. The Memory Management Unit (MMU) translates virtual addresses to physical addresses, thus ensuring that programs operate within a secure and isolated virtual space without direct manipulation of physical memory locations.

Physical addresses are actual locations where data and programs reside. In contrast to virtual addresses, physical addresses refer directly to these physical locations in the hardware. This distinction is vital for understanding how the computer manages and accesses memory [7, 35].

CPU Cache

The CPU cache is a small, high-speed memory resource located in proximity to the CPU, designed to store frequently accessed data and instructions, reducing the time the CPU spends waiting for this information from the main memory. Caches have their own hierarchy, often called L1 (level one), L2 (level two), and L3 (level three) cache. The Last Level Cache in a system is referred to as LLC, with a specialized designation as SLC (System Level Cache) when discussing individual segments of the LLC allocated per core. The L1 cache, directly integrated into the CPU chip, is the highest level cache, but the smallest and fastest, providing the quickest data access but with limited storage capacity. Moving down, the L2 cache, slightly larger and slower than L1. This is the cache level where the Prefetcher considered in this work, more in Chapter 3, try to improve the cache hit rate to achieve maximum performance. The L3 cache, often shared among cores, offers a balance between a larger storage capacity and speed. Figure 2.4 illustrates such a cache hierarchy.

A “hit” implies faster access to data compared to a “miss”, where data retrieval from the main memory is required. Therefore, workload performance can be improved by maximizing the hit rate, leading to lower latency.

To sum up, the cache hierarchy is designed to balance speed and storage capacity effectively and boosts workload performance by reducing the time spent on memory access, ensuring that the most frequently accessed data is available at the highest speed possible. In higher levels of the cache hierarchy, the capacity decreases, while the latency increases [7, 35].

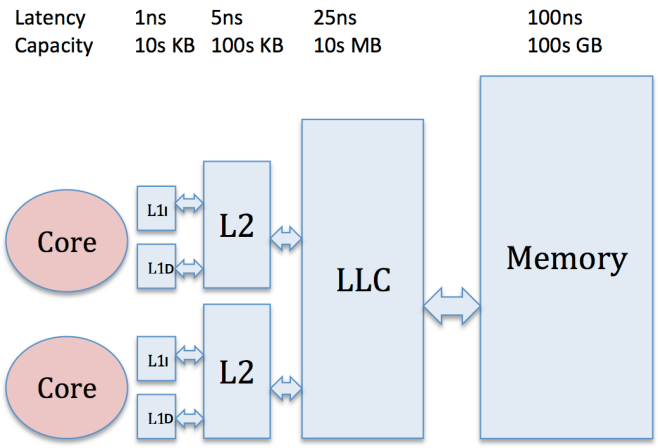


Figure 2.4.: Illustration of the cache hierarchy. Higher levels have lower latency but smaller storage capacity. The LLC is shared with other cores. L1I marks the level one instruction cache and L1D the level one data cache. Adapted from [16].

CPU

At the top of the memory hierarchy is the CPU and its closest memory, registers. These are small and fast storage units used to hold temporary data and instructions during computation and follow rules regulated by the ISA (Instruction Set Architecture) [7, 35].

2.2. Prefetching

Prefetching is a technique that improves performance by proactively fetching instructions and data from comparatively slower storage locations, such as main memory, or higher-level caches, to faster caches ahead of their actual need. This approach significantly enhances overall access time by hiding latency behind ongoing computations, thereby fostering superior system performance. It is important to note that prefetching typically involves fetching full memory blocks rather than partial or individual data elements, thereby ensuring the efficiency of data retrieval.

This technique can be implemented in hardware or even in software. Hardware prefetching is built right into the CPU. This form of prefetching exploits various access patterns including spatial and temporal locality. It predicts access patterns of memory by analyzing historical data requests at the microarchitecture level. Software prefetching, however, involves inserting special instructions into the program code by the compiler or the programmer to guide the prefetching process. Such techniques aim to reduce the cache miss rate, leading to fast data access and therefore to enhanced performance.

Prefetching can be utilized for data or instruction requests. Data prefetching deals with fetching the data needed for computations, while instruction prefetching fetches the commands that the CPU will execute next [16].

2.2.1. Prefetching Techniques

There are various hardware prefetching techniques, each offering unique benefits and challenges depending on data access types. This work primarily focuses on Stride and Stream-based hardware prefetching. However, notable prefetching techniques are:

Next-Line Prefetching: This technique preloads the next sequential memory block by exploiting spatial locality. It is especially effective in scenarios where data or instructions are stored sequentially, such as when processing elements of an array in a loop. This approach minimizes the time needed to fetch subsequent elements, based on the assumption that accessing a memory location is likely to lead to the next one. Let P_{next} be the predicted next prefetch address, $A_{current}$ the current accessed read address, and S the size of the memory region. Then the address prediction can be calculated as $P_{next} = A_{current} + S$ [16, 29]. Next-Line Prefetching is simple and cost-effective but assumes sequential data access, limiting its use in non-sequential scenarios where it may waste resources. It is best suited for environments with strong spatial locality, highlighting the need for selective strategy application based on access patterns.

Stride Prefetcher: Stride prefetching is a technique that enhances memory access by predicting future data requests based on identified access patterns, known as "strides". Consider a sequence of read requests like $R[i], R[i+Q], R[i+2Q], R[i+3Q]$, where R denotes a read request, i is the initial read address, and Q represents the stride [25].

To determine the stride value, one can calculate the difference between the current read address and the previous one. This process may seem straightforward but requires precise prediction. Irregular memory accesses that do not follow the identified stride pattern are challenging this process. Done incorrectly, it can lead to performance loss due to useless prefetches. Therefore, accurately training the pattern detection algorithm is significant for effective prefetching.

Stride prefetching is beneficial in regular data access scenarios, such as matrix operations in scientific computing, where data is accessed at predictable intervals. Thus, allowing the prefetching mechanism to precisely predict future requests. For example, during matrix multiplication tasks, stride prefetching can efficiently predict and prefetch the next set of matrix elements based on the observed stride, thereby reducing memory access latency and improving overall computational performance [16, 25].

Markov Prefetcher: Utilizing a history table that maps previously accessed locations to their subsequent accesses, this technique predicts future accesses based on observed patterns. These Prefetcher excel in complex but repetitive patterns, like linked data structures or multidimensional arrays. In database applications with repetitive query patterns, Markov Prefetcher can effectively predict future data accesses [16, 21].

Stream Prefetcher: Stream prefetching, like stride prefetching, is a method to optimize memory access by prefetching future data requests. It focuses on recognizing sequences of memory accesses, termed "streams", which typically involve consecutive memory locations. Once a stream is detected, stream prefetching predicts forthcoming memory accesses, focusing on contiguous data access unlike the fixed intervals in stride prefetching.

The challenge for stream prefetching is similar to stride prefetching and lies in the correct prediction of its access patterns. Thus, Stream Prefetcher are designed to dynamically adjust their behavior based on the effectiveness of their predictions, balancing between prefetching aggressiveness and system efficiency [16, 33].

Delta Correlation Prefetcher: Focusing on the differences, or "deltas", between successive memory addresses, these Prefetcher are adept at handling workloads with irregular but delta-correlated memory accesses. This pattern is common in graph traversal algorithms, where the access pattern is not linear but shows a predictable relationship between successive accesses. By identifying these delta correlations, Delta Correlation Prefetchers can effectively prefetch data in scenarios where traditional linear methods are less effective [16, 17].

2.2.2. Prefetching Balance

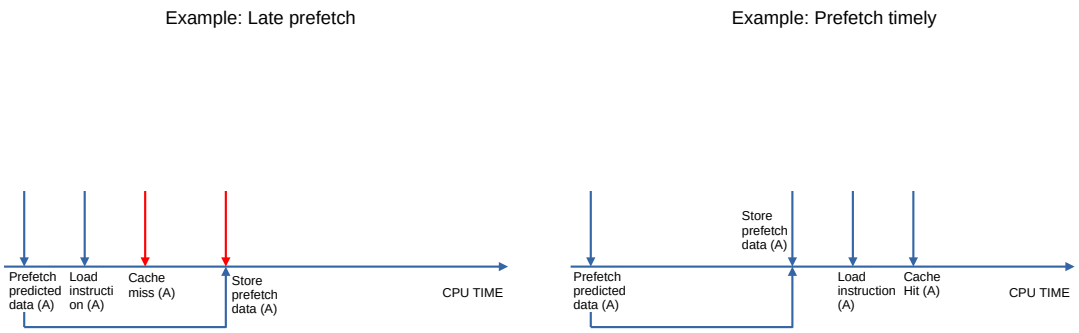
In the context of prefetching, the term "aggressiveness" refers to the number of prefetches sent by the Prefetcher w.r.t. the prefetch degree and distance. Aggressiveness is determined by two major factors: prefetch degree and prefetch distance.

Degree: This metric defines the number of data blocks or memory locations the Prefetcher fetches in the current prefetching process. For instance, a prefetch degree of four would request the next four memory blocks subsequent to the current prefetching request.

Distance: The metric of distance describes how far ahead the Prefetcher requests a memory block. It is essentially the gap between the current memory access and the furthest memory location from which the Prefetcher may request a data block.

Consider the following access stream of memory requests $R[i+D]$, $R[i+D+1]$, ..., $R[i+K]$. As before, R denotes a read request, and i is the initial read address. The distance can be observed by the address inside the brackets to read from. In this example, D describes the distance; therefore, the Prefetcher will start prefetching the data block beginning from $R[i + D]$. The number of data blocks that will be prefetched is described by the degree K [25].

Overly aggressive prefetching can result in cache evictions and memory contentions, adversely affecting the overall available bandwidth. Furthermore, heightened energy consumption emerges as a notable side effect. However, adopting a less aggressive approach may sacrifice potential performance gains [16, 18, 22, 33]. Consider the following prefetching scenarios: At low to medium system bandwidth consumption, when coupled with high prediction accuracy, increasing the prefetch degree can significantly enhance performance. This method efficiently improves cache utilization without exceeding the available system bandwidth. Conversely, at high bandwidth utilization, increasing the prefetch degree often results in performance degradation. This is attributed to cache eviction and heightened memory contention, as the system is challenged in managing the high demand requests to the memory. Moreover, increasing the prefetch degree under conditions of poor prediction accuracy proves to be counterproductive due to cache pollution and thus replacing important data in the cache.



(a) Late Prefetch (b) Timely Prefetch

Figure 2.5.: Illustration of Prefetching Timeliness: The left Figure shows a late prefetch, resulting in a cache miss due to the prefetch request occurring too close to the data load instruction. The right Figure demonstrates a timely prefetch where the data arrives in the cache just before the workload requires it, effectively preventing a cache miss and enhancing system performance.

Understanding the concept of "aggressiveness" in prefetching necessitates the understanding of what "timeliness" means in this context. To illustrate this, refer to Figure 2.5. On the left side of this figure, a prefetch request for block A is depicted. The arrow at the bottom represents the time required for the requested memory block to arrive. It is noticeable that the load instruction for the data closely follows the prefetch request, leading to a cache miss for data A because the data block has not yet been written into the cache. Consequently, the CPU may stall the application while waiting for the data to arrive, resulting in a performance loss. This scenario underscores the importance of timely prefetching. On the right side of this figure, a timely prefetch request is depicted, where the requested data block arrives into the cache before the load instruction. Consequently, the cache experiences a hit, enabling the CPU to continue operations without stalls, thereby enhancing performance by hiding the latency associated with data arrival. However, caution is required. Prefetching too early may result in the eviction of the necessary cache block, as other data blocks might need to be written into the cache, and the cache replacement policy may choose to evict the prefetched memory block.

The success of prefetching is largely dependent on its precise implementation. Factors such as untimely data arrival, incorrect predictions, or suboptimal settings of prefetch aggressiveness can adversely affect system performance [16, 18, 20, 33]. Therefore, maintaining a careful balance among these key aspects of prefetching is crucial for optimal functionality and efficiency.

2.3. Evolution and Significance of Arm Architecture

The ARM (Advanced RISC Machine) architecture has its origin in the 1980s [2] and has developed significantly in recent years, establishing itself in the realm of HPC (High Performance Computing). This architecture is characterized by its RISC (Reduced Instruction Set Computing) methodology, which is designed for efficiency and simplicity, leading to reduced power usage and cost-effectiveness [6]. Such benefits have made Arm a favored option for embedded systems [6]. Arm architecture has evolved from a simple device to a fundamental part of mobile technology. Being energy-efficient and still holding strong processing capabilities, it enables Arm to be adopted in modern HPC systems [6, 8, 9, 31].

Moreover, the Arm architectural design is scalable and adaptable, enabling the development of specialized Arm-based chips for specific HPC applications [8, 31], like the implemented Arm-based chip (Section 3.2). The impact of Arm in HPC is quite notable, offering a strong alternative to traditional x86 architectures, due to its balance of energy efficiency and high performance. The use of Arm-based supercomputers, like Japan's Fugaku, illustrates the ability of Arm CPUs in dealing with complex, large-scale computational tasks [9, 31].

2.4. Gem5 Simulator

Gem5 is a flexible simulation tool widely used in the field of computer architecture research. Its design focuses on being highly adaptable and open to experimentation, making it a valuable resource for both academic and industrial research. Managed by a community of users and developers, Gem5 is regularly updated and improved, ensuring it stays relevant and useful for ongoing research [24].

Using Gem5 in this work brings the following key benefits [24]:

1. **Flexibility:** Gem5 is a modular environment, allowing researchers to adapt it to their specific needs for exploring various types of computer architectures.
2. **Community Support:** Gem5 is supported by a worldwide community. Therefore, a lot of updates and contributions in this area of research are taking place, making the tool more reliable. Moreover, there are Prefetcher available that have been previously implemented by researchers.
3. **Wide Range of Uses:** Although Gem5 was first made for academic research, it has grown to be useful in industry research and teaching too.

The simulation tool offers two primary modes of simulation, each designed to support distinct research needs. Both of them are used in this work. In the subsequent subsections, a brief explanation of the modes is given. More details on their impact on this work are explained in Chapters 3 and 4.

2.4.1. SE (System Emulation) Mode

The SE mode of Gem5 is specifically designed for fast and efficient simulation of individual workloads for a simple computer architecture. This mode simplifies the simulation process by focusing on the CPU and memory system, reducing the complexity of a simulation by bypassing an OS (Operating System). It is beneficial to gain in-depth insight into the simulated microarchitecture. This mode should be used to evaluate specific aspects of architectural design, such as pipeline structure, cache behavior, or ISA (Instruction Set Architecture) [24].

2.4.2. FS (Full System) Mode

In contrast to the SE Mode, the FS mode of Gem5 offers a complex simulation environment that includes the simulation of a complete computer system, including the OS, drivers, peripherals, software, and other system-level components. It is beneficial for gaining in-depth understanding of the interactions between software and hardware in a fully integrated system, like system-level power management, hardware-software co-design, or the impact of system-level architecture decisions on application performance [24]. The latter is of utmost importance for this work.

3. Methodology and Implementation

In this chapter, the methodological framework central to this thesis is meticulously outlined. The primary focus lies in the development of timely prefetching strategies, specifically designed for heterogeneous memory in high-performance Arm processors. At the heart of this methodology is a simulation-based evaluation, executed using the Gem5 Simulator. This approach is instrumental in testing the effectiveness of both Stream-based and PC-based Stride Prefetcher. Moreover, the chapter also delves into the implementation and the optimization of these Prefetcher.

Distinctive contributions of this work include the novel implementation of a Stream-based Prefetcher, an addition that is unavailable among the collection of Gem5 Prefetcher. Additionally, the thesis introduces an enhanced mechanism for the timely and dynamic adjustment of prefetching parameters. This mechanism is optimizing Prefetcher performance in real-time, responding adaptively to varying workload conditions and memory system states.

Moreover, this chapter offers an in-depth analysis of the implementation aspects related to the Prefetcher and the optimization techniques formulated. This meticulous approach is supported by simulation-based assessments, serving as a robust basis for empirical investigation and precise enhancements. The techniques and practical applications presented within this chapter make a substantial contribution to the field of study.

3.1. Methodology

The primary goal of this thesis is the development and in-depth evaluation of advanced, timely prefetching strategies, designed for heterogeneous memory systems within high-performance Arm processors. A simulation-based approach, utilizing the Gem5 Simulator, is at the core of this methodology. The Gem5 environment, known for its highly adaptable design and openness to experimentation, is thereby selected for emulating the processor architectures.

During the developmental phase, the SE mode of the Gem5 Simulator plays a meaningful role. This mode is utilized for verifying the functionality of the implemented Prefetcher, providing detailed insights into its behavior through extensive debugging messages. As

the SE mode operates without an operating system, it restricts testing to hardware-specific functionalities under a predefined workload. Consequently, this mode is limited to simple test architectures with a singular type of memory device, but it suffices for initial functional evaluations.

In contrast, the FS mode of the Gem5 Simulator offers a simulation of a complete computer system. A critical component of Gem5 hereby is the Ruby subsystem, offering a comprehensive model for the memory subsystem. Ruby facilitates the exploration of alternative cache organizations, interconnection networks, and cache coherency protocols. Notably, it includes the implementation of the Arm AMBA CHI protocol, essential in defining cache and memory controllers as state machines. These controllers manage CHI transactions and block allocation and replacement policies effectively. In parallel, the Garnet subsystem of Gem5 provides an on-chip network interconnect model, supporting detailed simulation of network traffic and timing effects. This is achieved by modeling network routers at the micro-architectural level [39]. The full system simulation scripts are generated using the benchmarking platform provided by [15]. These exploration scripts, developed in Python, necessitate two primary inputs: a `.yaml` file for defining the architectural setup and a `.py` file detailing the benchmark configurations. These configurations might include specific architectural elements, such as the type of Prefetcher employed and their linkage to various cache levels. It is noteworthy that prefetching parameters are established within the benchmark configuration and passed to the architectural configuration. The final stage of this research employs full system simulations to evaluate the prefetcher performance. The architecture to be simulated, described in the `.yaml` file, will receive an in-depth discussion in Section 3.2. This structured simulation framework is pivotal for conducting controlled experiments and gathering essential performance metrics. Such metrics are significant for assessing the impact and efficacy of various prefetching techniques on Arm architectures equipped with heterogeneous memory devices.

3.1.1. Gem5: Basic Prefetcher

In the Gem5 simulator, the execution of the `calculatePrefetch(const PrefetchInfo &pfi, std::vector<AddrPriority> &addresses)` function is a key component in the prefetching process. This function is invoked to calculate prefetching addresses based on the `PrefetchInfo` parameter, which has information about the prefetch trigger. In the `calculatePrefetch()` function, the prefetch logic undergoes detailed parsing. After processing this logic, the function adds the resulting prefetch address to the prefetch queue. This queue is referred to by the second parameter, `std::vector<AddrPriority>`. There are two kinds of memory systems which invoke the `calculatePrefetch()`function.

The Classic Memory System and the Ruby Memory System.

Gem5: Classic Memory System with BaseCache

In the classic memory system of Gem5, prefetching is managed by the BaseCache class. This class includes a Prefetcher object, which is an instance of `prefetch::Base`. This object is important as it determines the prefetching strategy.

During cache access events, such as a cache hit or miss, the cache system activates the Prefetcher. This activation leads to the invocation of the `calculatePrefetch` function within the Prefetcher, which is tasked with computing the addresses of memory blocks that should be prefetched. The specific conditions and logic under which `calculatePrefetch` is called are determined by the logic of the cache and the chosen prefetching strategy.

Gem5: Ruby Memory System

The Ruby memory system within the Gem5 simulator represents a highly configurable framework, adept at modeling more complex memory systems. It provides a comprehensive platform for simulating various cache coherence protocols and memory hierarchies. Prefetching in this system is integrated into the Ruby architecture. Typically, cache controllers and sequencers handle prefetching, issuing requests based on memory access patterns. The logic for prefetching, which may include functions like `calculatePrefetch`, is embedded in specific controller classes. These details are distributed across multiple files and classes in the `src/mem/ruby directory` of the Gem5 source code [5]. In the simulation of the described architecture, the Ruby system was utilized, implementing the AMBA CHI protocol [10]. The implementation of this protocol was not part of the thesis work and was provided by [15].

Gem5: Class Hierarchy

To gain a deeper understanding of the Prefetcher implementation presented in this thesis, it is beneficial to delve into the class hierarchy of the Prefetcher within Gem5.

Base Prefetcher Class (Base): This class forms the foundation of the prefetching framework. It defines the fundamental structures, such as `PrefetchInfo`. The Base class also introduces virtual methods, which are critical for handling cache access notifications and for retrieving prefetch requests. These methods are the core mechanism for all subsequent Prefetcher implementations.

Queued Prefetcher Class (Queued): Advancing from the Base class, the Queued class introduces a queue-based mechanism for managing prefetch requests. This approach enables storing, organizing, and processing prefetch requests that are delayed.

In this thesis, the implemented Prefetcher are an extension of the Queued Prefetcher Class. Furthermore, the optimization strategies for the Prefetcher (Section 3.6) are linked to the statistical calculations performed in the Queued Prefetcher Class, demonstrating the interconnectedness and the criticality of the class hierarchy in the Prefetcher implementation.

3.2. Simulated high performance Arm processor with Gem5

The Arm Neoverse V1 Reference Design (RDV1) plays a significant role in the high-performance and exascale computing sectors. At its core, the RDV1 utilizes high-performance Armv8.4-A Neoverse V1 cores, which marks a notable advancement in micro-architectural design compared to previous iterations of Arm architectures [39]. This thesis primarily focuses on heterogeneous memory aware prefetching, rather than delving into the intricate details of CPU core design. Consequently, an exhaustive discussion of every facet is beyond its scope. Therefore, a small overview is given and illustrated by Figure 3.1. For more detailed exploration of the Neoverse V1 architecture, reference [1] is recommended as a comprehensive resource.

3.2.1. Arm Neoverse V1 Core

Front End: The front-end of a CPU is responsible for preparing instructions for execution. Its primary tasks include fetching instructions from the memory, decoding them into the ISA format understandable by the CPU, denoted as MOP (Macro-Operation) in Figure 3.1, and performing branch prediction. Notably, Arm has significantly upgraded the BPU (Branch Prediction Unit), increasing the size for the nano BTB (Branch Target Buffer) from 16 to 96 entries and for the main BTB from 6000 to 8000 entries. This advancement leads to fewer pipeline stalls, due to higher accuracy in predictions, and thereby boosting overall performance. Additionally, the front end benefits from the integration of a novel MOP cache and a wider 5-Way Decoder, which leads to a higher throughput of instructions. [1].

Execution Engine: The execution engine is where the actual processing of instructions occurs. It consists of various components such as the ALU (Arithmetic Logic Unit), FPU (Floating-Point Unit), and SVE (Scalable Vector Extension) units. Additionally, the

execution engine inlcudes a ROB (ReOrder Buffer). In the Neoverse V1, the ROB takes MOPs (Macro-Operations) as input and converts them into μOPs (Micro-Operations). These μOPs delineate the finer, more detailed steps that the CPU must execute to process the macro-operations. Micro-operations, specific to a particular CPU design, are not visible at the ISA level. During the transformation of MOPs, the ROB reorders them and optimize the execution of complex instructions, thus enabling out-of-order executions. Herein lies a significant enhancement by Arm. By doubling the size of the ROB, it promotes advanced handling of micro-operations, fostering efficient out-of-order execution and complex instruction dependencies. Moreover, Arm has also improved the size of the SVE units to 256 bits. This enables an impressive capability of 16 double-precision Flop/cycle [1, 39].

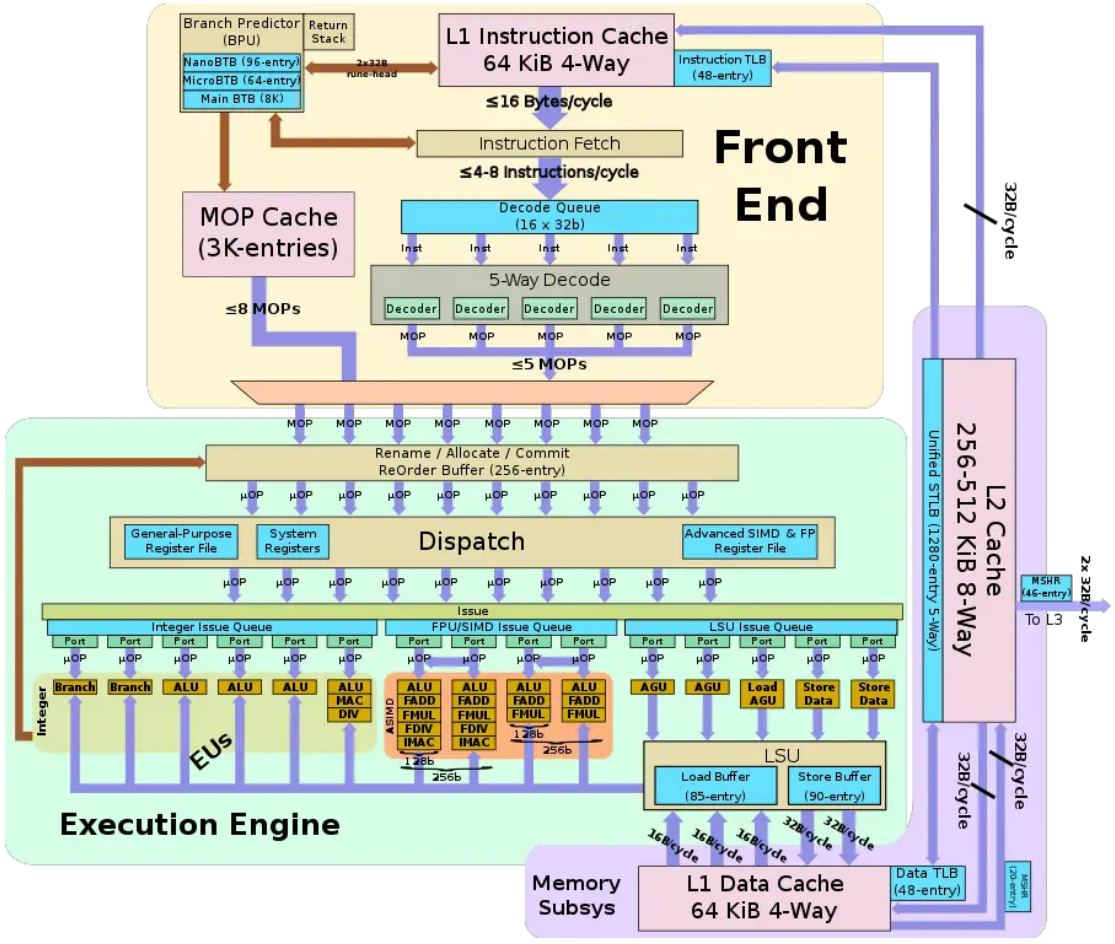


Figure 3.1.: Detailed architecture of the Arm Neoverse V1 Core [37], showcasing the enhanced front-end with improved branch prediction, the execution engine with expanded ReOrder Buffer and SVE units, and the advanced memory subsystem with optimized cache sizes for high-performance computing.

Memory Subsystem: The memory subsystem includes all components involved in storing and retrieving data, such as caches (L1, L2, L3), memory controllers, and the connections to the main memory devices. The Neoverse V1 architecture has undergone substantial enhancements in its memory subsystem, particularly in cache size and latency, which are essential for the efficiency of HPC applications. Each core within this architecture is configured with a private 64 KiB L1I and L1D cache, both 4-way set associative, and an L2 cache that ranges from 512 KiB to 1 MiB. These caches, equipped with SECDED (Single-bit Error-Correction, Double-bit Error Detection) ECC (Error Checking and Correction) and write-back features, are designed to enhance data retrieval and processing speeds. The SLC, which varies from 2 MiB to 4 MiB depending on the specific chip implementation, plays an influential role in enhancing data management efficiency. These advancements in the memory hierarchy, including improved bandwidth and latency, directly impact the performance of memory-intensive HPC applications [1, 36, 39]. In the configuration for the simulation, the size of the l2D is set to 1 MiB and the SLC is set to 2 MiB. Moreover, all Prefetcher are attached to the L2D.

3.2.2. Hybrid memory configuration

The interconnect component of the architecture, the Arm CoreLink Coherent-Mesh-Network 650 (CMN-650), stands out for its high-bandwidth and low-latency characteristics. It is specifically optimized for Armv8-A processors, such as the Neoverse V1, and plays an essential role in integrating various memory elements like HBM memory stacks and DDR5/4 memory into the system [39]. Figure 3.2 illustrates the mesh configuration of one quadrant of the simulated CPU.

In the described architecture, each quadrant is connected to eight HBM2 memory channels and one DDR5 memory channel. The blue routers, linked to the HBM2 channels, incorporate a pair of Neoverse V1 cores each. Orange routers, connected to the DDR5 channel, also house two Neoverse V1 cores. Serving a critical backup role, the red router acts as a fail-safe for the other cores in cases of malfunction for various reasons, and it too contains two Neoverse V1 cores. Depicted as the conduit for inter-quadrant connections, the purple router completes this setup.

The KNL (Knight's Landing) processors, showcase notable enhancements in their memory subsystem, as introduced in Section 5.6. However, a comparable implementation for Arm-based devices is currently lacking. It is essential to note that the architecture being discussed also utilizes the benefits of KNL's SNC (Sub-NUMA Clustering) modes. In Figure 3.2, the NUMA domains are distinctly illustrated: the domain for HBM2 is depicted in blue, consisting of 8 routers and thereby 16 cores, while the domain for DDR5 is represented in orange, comprising 2 routers which equate to 4 cores.

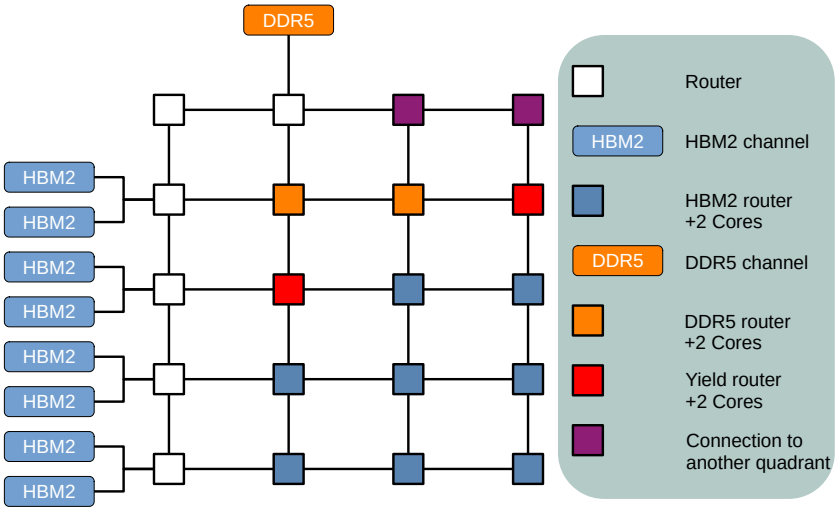


Figure 3.2.: One Quadrant of the Simulated CPU with Arm CoreLink CMN-650 Configuration: Illustrates the allocation of 16 Neoverse V1 cores to 8 HBM2 channels (blue) and 4 cores to 1 DDR5 channels (orange routers). The red router serves as a fail-safe, and the purple router enables inter-quadrant connections.

Collectively, these configurations account for a total of 20 cores within the displayed quadrant. Enhancing the Prefetcher with memory device awareness could further decrease latency, as this addition enables the Prefetcher to recognize the type of memory device and adjust its aggressiveness accordingly. Comprehensive configuration details are provided in `exploration/architectures/numa-2n.yaml` [4], referenced in Appendix A.1. Additionally, Appendix A.2 outlines the specifications for the implemented memory devices.

3.3. Implementation: Stream-based Prefetcher

The Stream-Based Stride Prefetcher integrates the Markov Prefetcher, utilizing a history table, with the Stride Prefetcher, which operates exclusively on physical addresses without consideration of a PC. In L2 cache design, indexing with physical addresses increases efficiency by bypassing the need for virtual address translation, thereby further reducing latency. This approach prevents incorrect fetching of physical pages in the TLB (Translation Lookaside Buffer), as it does not rely on virtual addresses, which are associated with virtual pages. Normally, virtual addresses require a lookup process to find

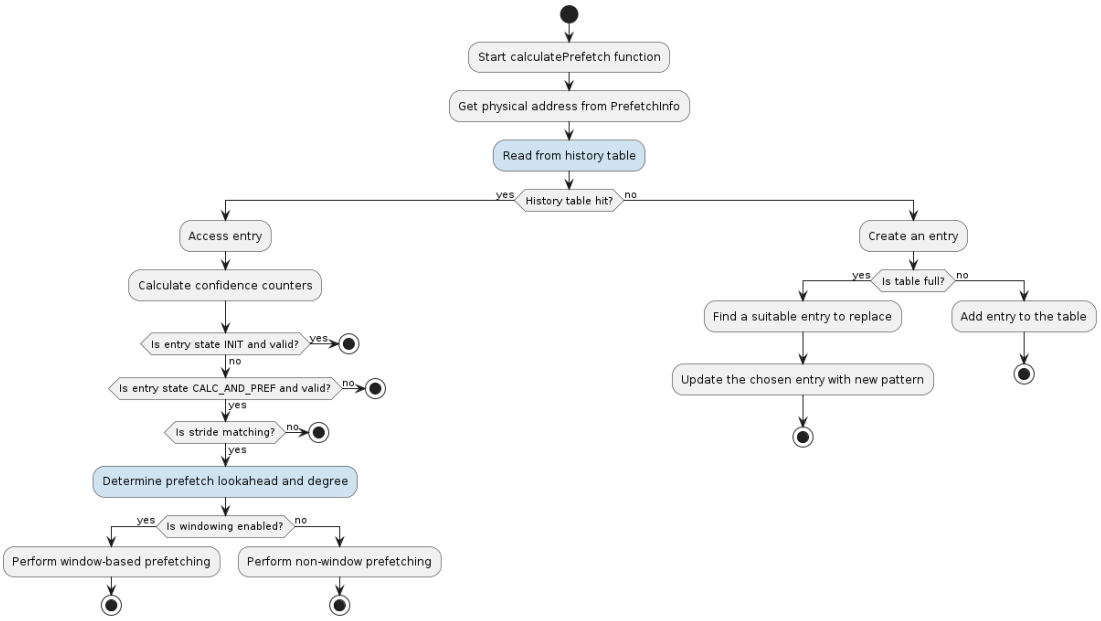


Figure 3.3.: Areas highlighted in blue represent the stages where optimizations are applied, illustrating the key points of enhancement in the prefetching mechanism.

the corresponding physical page. Incorrect fetching leads to additional latency due to the address translation process [28]. Consequently, the Prefetcher attached to the L2 cache operates with physical addresses. The Stream-Based Stride Prefetcher aims to examine stream accesses, extracting patterns in a strided approach, as elaborated in Section 2.2.1. For a visual representation, see Figure 3.3, which presents an activity diagram highlighting the steps of the prefetching process triggered by the `calculatePrefetch` function in this Prefetcher.

The prefetching process begins with the invocation of the `calculatePrefetch` function. Within this process, only the physical address is extracted from the `PrefetchInfo` parameter. Other details available in `src/mem/cache/prefetch/base.hh` [5] are not considered. The subsequent step involves calling the readCache function, which reads data corresponding to the extracted physical address from the history table. This table is often referred to as a "cache" due to its resemblance to a Fully Associative Cache in terms of behavior.

To identify the relevant data, the tag of the physical address is determined using $\lfloor \log_2(pageBytes) \rfloor$ and stored as a function variable. The `pageBytes` parameter, predefined at the architectural configuration, is passed to Gem5, making it accessible for operations. In the current configuration, `pageBytes` is set to 4096 Bytes.

Each parameter stored in a table `entry` is detailed in Table 3.1. Every parameter inside the entry is notated as `entry.x`, where `x` is the accessed parameter. The `readCache`

function is responsible for verifying the presence of a valid entry (`entry.valid`) in the table and matching the tag (`entry.tag == tag`). If the comparison yields false, the function returns -1, indicating the absence of a corresponding entry and the necessity to create a new one.

The addition of a new `entry` initiates with the `writeCache` function. This function creates an entry populated with default values. Similar to previous operations, the `tag` is extracted and stored in `entry.tag`, and the complete physical address is stored in `entry.lastAddr`. The `entry.lastAddr` plays a crucial role in subsequent stride calculations and pattern recognition. A `entry.valid` flag is set to true, indicating the legitimacy of the new `entry` within the table. Depending on their respective types, other parameters within the `entry` are initialized to zero or set to false.

To locate a suitable position for this new `entry`, the function iterates through the table. In scenarios where the table is at full capacity, it becomes necessary to employ the LRU (Least Recently Used) algorithm. LRU, a prevalent cache replacement policy, functions by removing the item that was least recently accessed when the cache is saturated. Within this implementation, the LRU entry is determined by identifying the entry with the highest `entry.lru` value. Alternatively, if an invalid `entry` exists, it is replaced instead.

This decision-making process ensures that frequently accessed patterns are kept in the table while less critical patterns are evicted. Upon identifying the appropriate position for the new `entry`, the `entry.lru` counter is assigned the value of the function's initial static counter, which is incremented by one for the next replacement. However, in both cases where an `entry` needs to be created, the `calculatePrefetch` function terminates.

In the case that an `entry` is found in the history table, then the index of the `entry` is passed and accessed. Now the function `calcConfCounters` is called. The `calcConfCounters` function, a critical component of the Stream-based Prefetcher, is adept at dynamically managing the state of the Prefetcher, using the `entry.state` parameter, based on observed memory access patterns, focusing on stride calculation and confidence level adjustments. The description of the `entry.state` values can be found at the Table 3.2.

Stride calculation is a key process in the Stream-based Prefetcher, involving the computation of the difference between the `current address` accessed and `entry.lastAddr`. This difference is assigned to the variable `new_stride`. The stride, `new_stride`, is then compared with the existing stride value in `entry.stride`, ensuring it is non-zero. A zero stride, indicating repeated access to the same memory location, is not useful for pattern detection. Therefore, a true value in `entry.stride_match` signifies that a consistent access pattern was previously detected and matches the current stride pattern.

Subsequently, `entry.lastAddr` is updated to reflect the current memory access. The parameter `entry.conf` represents the confidence in the detected pattern. If `entry.conf` is zero and `entry.stride` is also zero, indicating initial entry setup, the pattern is observed

for the first time. In this case, `entry.conf` is incremented by one, and `entry.stride` is updated to `new_stride`.

If `entry.stride_match` is true and `entry.conf` is non-zero, it implies a previously detected pattern. Here, `entry.stride` is updated to `new_stride`, even though `entry.stride_match` already suggests this value. This redundancy ensures accuracy. Additionally, `entry.conf` is incremented.

When `entry.conf` reaches or exceeds a threshold `threshConf`, set based on configuration in `benchmark.py`, the state of the Prefetcher transitions to `CALC_AND_PREF`. The threshold, typically set to two based on extensive testing in SE mode, optimizes the number of prefetches for improved coverage. In the `CALC_AND_PREF` state, the Prefetcher actively prefetches data based on the calculated stride, enhancing memory access efficiency. Additionally, upon entering this state, `entry.failCnt` is reset to zero. This counter tracks the number of times a detected pattern fails to match the stride, facilitating pattern updates.

However, if the step mentioned before results in false, indicating a valid entry (`entry.valid == true`) with a mismatch in stride (`entry.stride != new_stride`), it leads to `entry.stride_match` holding the value false. This mismatch suggests that the detected pattern may be outdated or incorrectly computed, possibly due to out-of-order executions or requests exceeding physical page boundaries. In such cases, the same stride pattern might continue under a different tag, leaving an incorrect entry in the table.

To address this, `entry.conf` is decreased, and `entry.failCnt` is incremented by one. If `entry.conf` falls to 1 or below, or `entry.failCnt` reaches or exceeds `threshConf`, an update is necessary to maintain the accuracy and coverage of prefetches. This update involves setting `entry.stride` to `new_stride`, `entry.conf` to one, `entry.failCnt` to zero, `entry.state` to INIT, and `entry.windowInit` to false. This approach, updating the pattern after two failures, is an optimization that yields higher accuracy and coverage in prefetches compared to conventional Prefetcher.

Name	Type	Function
tag	Addr	Tag for identifying the cache line.
lastAddr	Addr	Address of the last access.
windowStart	Addr	Start of the prefetch window.
windowEnd	Addr	End of the prefetch window.
windowInit	bool	Flag to indicate if the window is initialized.
stride	int	Stride length for prefetching.
stride_match	bool	Flag to indicate if the current stride matches the historical stride.
state	STATE	Current state of the Prefetcher.
conf	short	Confidence level in the prefetch decision.
valid	bool	Validity of the entry.
failCnt	int	Count of prefetch failures.
lru	uint64_t	Least Recently Used counter.

Table 3.1.: Description of Parameters in the Prefetcher Entry

State	Description
INIT (Initialization)	The starting state, where the Prefetcher is in a learning phase, gathering data about strides to identify consistent patterns. Transition out of this state occurs when a consistent stride is established.
CALC_AND_PREF (Calculate and Prefetch)	Entered when a high confidence level is reached, indicating a stable stride pattern. In this state, the Prefetcher actively prefetches data based on the calculated stride.

Table 3.2.: States of the Stream-based Prefetcher

Upon completion of all internal updates relevant to the entry, a comparison is executed between `entry.state` and the `INIT` state, concurrently assessing if `entry.valid` is set. A positive result in this evaluation signifies the termination of the prefetching process, indicating insufficient confidence in the established stride pattern. Conversely, should the `entry.state` equate to `CALC_AND_PREF` and `entry.valid` be set, and if this evaluation yields a negative outcome, the prefetching process is similarly concluded. This mechanism serves to ascertain that prefetching is initiated only when sufficient confidence, as delineated in the `calcConfCounters` function, is established.

In the event of a favorable comparison, the prefetching process is initiated. The Prefetcher retrieves the values for "`lookahead`", where `lookahead` refers to distance and will be used interchangeably in the following text, and `degree`, parameters defined and incorporated through the benchmark or architectural configuration. Note that the prefetch addresses are calculated in byte, therefore the `lookahead` is multiplied by the cache line size. An optimization is implemented by employing a monitoring window approach for

prefetch trigger addresses. This feature is enabled but can be disabled in the benchmark configuration; if enabled, the corresponding logic for prefetching is executed. In contrast, a conventional state-of-the-art prefetching algorithm is invoked. Upon the completion of these algorithms, the prefetching process is terminated. The subsequent section delineates two distinct prefetching methodologies:

Window-based Prefetching: The concept of the window-based Prefetching approach in stream prefetching, as inspired by the paper [33], involves dynamically tracking memory accesses within a specific range, like explained in Section 5.4.

Within the Stream-based Prefetcher, the function `doPrefetch` is implementing the window-based prefetching algorithm. This function manages a monitoring window for each entry, utilizing the `entry.windowInit` field to indicate whether the window has been initialized. The fields `entry.windowStart` and `entry.windowEnd` define the start and end of the monitoring region, respectively. If the window is not initialized, `entry.windowStart` and `entry.windowEnd` are set to the current trigger address (`pf_addr`), and `entry.windowInit` is switched to true. The function handles data streaming in both forward and reverse directions, as indicated by the sign of `entry.stride`. The elucidation below focuses on the forward direction. The reverse direction is trivial and follows the same principle.

Figure 3.4 depicts the windowing approach utilized for prefetching and illustrates three distinct zones within the monitoring window. The first zone, highlighted in `blue`, represents the `monitoring window` itself, where the span between `entry.windowStart` and `entry.windowEnd` matches the `lookahead`. The `second` and `third` zones, shown in `gray` and `red` respectively, correspond to `addresses beyond` `entry.windowEnd` and addresses `below` `entry.windowStart`.

The algorithm begins by evaluating the current prefetch trigger address (`pf_addr`). If `pf_addr` falls below `entry.windowStart`, it is considered outside the monitoring region and thus disregarded, leading to the termination of the `calculatePrefetch` function. This scenario is exemplified as Prefetch Trigger 1 in Figure 3.4. Conversely, if `pf_addr` is within or beyond the monitoring window, the algorithm proceeds to determine whether `pf_addr` is either less than `entry.lastAddr`, as demonstrated by Prefetch Trigger 2, or exceeds `entry.lastAddr` and resides within the gray zone, illustrated by Prefetch Trigger 3. Should either condition be met, `pf_addr` is tied up to `entry.lastAddr`. Consequently, `pf_start_addr` is updated to `pf_addr`. If not, the logic continues from `entry.lastAddr` and thus `pf_start_addr` remains as `pf_addr`. This strategy ensures the maintenance of sequential prefetching integrity, pivotal in out-of-order execution contexts, and mitigates the occurrence of unnecessary prefetches.

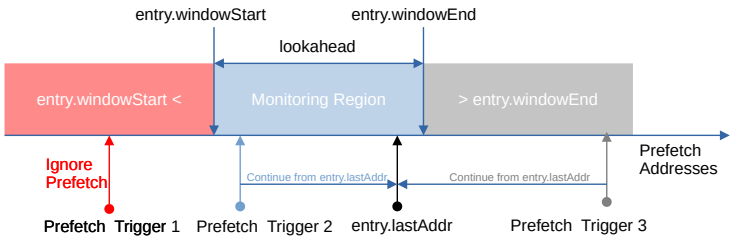
To prevent prefetching across page boundaries, the algorithm checks if the `pf_addr + lookahead` is within the same page as `pf_addr`. If the page boundary is exceeded, the algorithm calculates the remaining `pageBytes` to prefetch, enhancing efficiency. The

next phase involves "warming up" the window, where the "degree of run-up" (`degRunUp`) is calculated, dividing the window into memory block segments. The initial degree is superseded by (`adj_degree`), and once the window is "warmed up", the original degree is reinstated. The "warming up" process gradually builds certainty about the observed access pattern. As the window "warms up" and gains more certainty about the pattern, the algorithm switches to using the prefetch distance as determined by the benchmark parameters. This gradual adaptation allows for a more accurate prediction and reduces unnecessary prefetches.

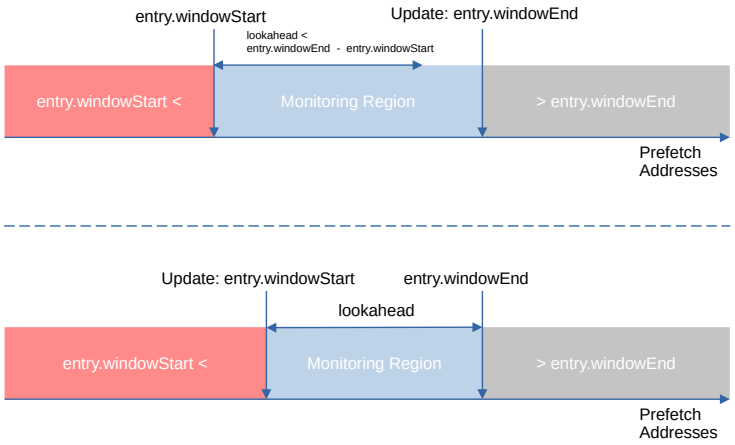
The address to prefetch (`new_addr`) is determined by equation 3.1, where d ranges from one to `adj_degree`, and `prefetch_stride` is the adjusted entry.stride.

$$new\_addr = pf\_start\_addr + d \times prefetch\_stride \tag{3.1}$$

Finally, if `new_addr` does not exceed the page boundary, it is enqueued for prefetching. Otherwise, the calculation ceases. The `entry.windowEnd` pointer is now updated to `new_addr`. If the distance between `entry.windowEnd` and `entry.windowStart` exceeds the `lookahead`, `entry.windowStart` is updated, aligning with the methodology in [33] and is illustrated at Figure 3.4b.



(a)



(b)

Figure 3.4.: Visualization of the dynamic window-based prefetching algorithm with three distinct zones. The first zone, highlighted in blue, represents the monitoring window itself, where the span between `entry.windowStart` and `entry.windowEnd` matches the `lookahead`. The second and third zones, shown in gray and red respectively, correspond to addresses beyond `entry.windowEnd` and addresses below `entry.windowStart`. Figure 3.4 illustrates the three prefetch trigger scenarios: Trigger 1 identifies addresses below the Monitoring Region and ending the process; Trigger 2 and Trigger 3 involve addresses within or just beyond the window, leading to adjustments in the monitoring window based on whether the address is before or after `entry.lastAddr`, respectively. This ensures prefetch efficiency by continuing from `entry.lastAddr`, thereby maintaining prefetch sequence integrity and minimizing redundant prefetches. Figure 3.4b shows the updating process of the Monitoring Region for valid prefetches within page boundaries, and adjusting `entry.windowStart` when `lookahead` < `entry.windowEnd`−`entry.windowStart`.

State-of-the-art Prefetching: The `doPrefetchNoWindow` method within the Stream-based Prefetcher calculates prefetch addresses without employing a window-based approach. This method bases its calculations on predetermined parameters, like `degree` and `lookahead`, defined in the Prefetcher configuration.

The function iterates up to the specified degree. For each iteration, the prefetch address (`new_addr`) is calculated by equation 3.2, where d ranges from 1 to the fixed degree, and `prefetch_stride` is defined exactly like in the windowed case.

$$new\_addr = \begin{cases} pf\_addr + lookahead + d \times prefetch\_stride, & if entry.stride > 0 \\ pf\_addr + lookahead - d \times prefetch\_stride, & if entry.stride < 0 \end{cases} \tag{3.2}$$

Prefetch addresses are calculated differently depending on the stream direction, as shown in the equation 3.2 above, which is determined by the sign of `entry.stride`. Furthermore, the logic ensures that prefetch addresses do not cross page boundaries, which is a critical key point. If the new address crosses a page boundary, the prefetching loop breaks, and no further addresses are calculated or prefetched for this iteration. If the new prefetch address remains within the same page, it is enqueued for prefetching.

3.4. Implementation: PC-based Stride Prefetcher

The PC-based Stride Prefetcher is utilizing `PC` (Program Counter) values to predict and enhance memory access patterns. This mechanism, distinct from the Stream-Based Stride Prefetcher, associates memory accesses with the initiating PCs, thereby offering a context-sensitive approach to prefetching. Adapted from its original version in the Gem5 repository, this implementation now includes enhancements for statistical data collection (see Section 3.5) and has been optimized as detailed in Section 3.6. The process flow of this Prefetcher is depicted in Figure 3.5.

In this Prefetcher, the `calculatePrefetch` function initiates the operation by extracting parameters from `PrefetchInfo`, including `pf_addr` (the prefetch trigger address), `PC`, and `requestor_id`. The `requestor_id`, which is always set to true in this work, is differentiating PC Tables based on the identity of the requesting component. The system also employs a security check on `pf_addr` through the `is_secure` flag to balance security considerations with performance.

The PC-based Stride Prefetcher leverages the `PC` and `is_secure` flag to accurately locate the corresponding `entry` in the PC Table. If a valid `PC` is not present, the `calculatePrefetch`

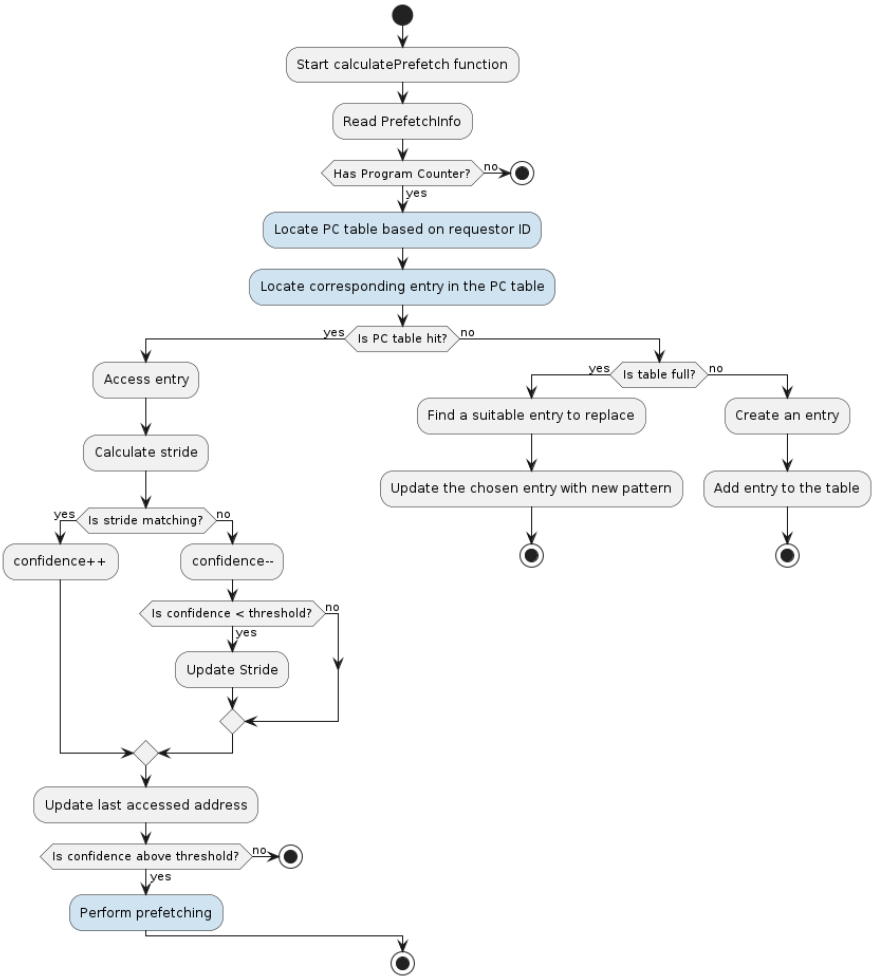


Figure 3.5.: Activity Diagram of the PC-based Stride Prefetching Process: Areas highlighted in blue represent the stages where optimizations are applied, illustrating the key points of enhancement in the prefetching mechanism.

function is immediately terminated, as a valid `PC` is influential for the prefetching process. Detailed parameters of an `entry` are outlined in Table 3.3.

In scenarios where an existing `entry` is not found in the PC Table, the next steps are governed by the current state of the table and the replacement strategy, which is RandomRP in this implementation. If the table has space, the Prefetcher initiates the creation of a new `entry`. This new `entry` is then initialized with `entry.lastAddr` set to the prefetch trigger address `pf_addr`, and other parameters such as `entry.stride` to zero and `entry.confidence` to the initial confidence level.

Alternatively, the table is full, meaning an update of an existing `entry`. The Prefetcher modifies `entry.lastAddr` to reflect the new `pf_addr`. The other values are set like in the case before to their initial values.

Through these processes, the Prefetcher ensures the PC Table is consistently up-to-date, aligning with the ongoing memory access patterns. This dynamic management of the PC Table is key to the Prefetcher efficiency, enabling it to adaptively predict and prefetch data.

Next, the Prefetcher calculates the `stride` and adjusts the confidence level based on the match with the existing stride value in the `entry`. This is governed by the `threshConf` parameter, a configuration setting that defines the minimum confidence threshold for pattern recognition. The Prefetcher terminates if the confidence level falls below this threshold, thus minimizing ineffective prefetching. Successful matches lead to prefetching as outlined in the state-of-the-art Prefetching Section for the Stream-Based Stride Prefetcher.

Parameter	Type	Initial Value
lastAddr	Addr	Address of the last memory access
stride	int	Calculated stride between accesses
confidence	short	Confidence level of the stride prediction

Table 3.3.: Key Parameters and Initial Values in PC-based Stride Prefetcher

3.5. Prefetcher Statistics

The prefetching mechanism in Gem5, fundamentally encapsulated within the Base class of the prefetch module, meticulously calculates and maintains a variety of prefetch-related statistics. However, the global scope of these statistical values in the standard Gem5 implementation does not suffice for the specific requirements of this work. To address this, the project significantly extends the basic prefetching statistics of Gem5, tailoring them to meet distinct experimental scenarios and research objectives. This customization is particularly evident in the handling of prefetch statistics, demonstrating a more detailed and scenario-specific approach.

These modifications have been implemented into the Queued class and are made available to all Prefetcher inheriting from it. A notable enhancement is the calculation of prefetch statistics for each NUMA node separately, on an epoch-by-epoch basis. An epoch is the number of cycles where statistics are accumulated. The optimization strategies, as detailed in Section 3.6, leverage these statistical values to fine-tune the Prefetcher at the end of each epoch, followed by a reset of the statistical counters. This approach enables

epoch-based tuning of the Prefetcher, offering a dynamic and responsive prefetching strategy that adapts to changing access patterns and workload characteristics over time. Such epoch-based tuning is advantageous as it allows the Prefetcher to continuously evolve and optimize its behavior, leading to potentially improved cache performance and reduced memory access latencies.

Furthermore, event functions are integrated into the AMBA CHI protocol and the Ruby cache interface to calculate fundamental statistics, including `demand_accesses`, `demand_misses`, `sent_prefetches`, `late_prefetches`, `timely_prefetches` and `bandwidth measurements`. It allows for the tracking and analysis of cache events and state changes, thereby enriching the overall understanding of cache dynamics.

Demand Misses and Accesses: In the context of cache management, two key statistics are tracked: demand_misses and demand_accesses. Both are updated within the `Queued::probeNotify` function, which is invoked upon each cache access.

The demand_misses statistic increments each time a cache access results in a miss. A cache miss occurs when the requested data is not found in the cache, necessitating retrieval from a slower memory source. This statistic is important for understanding the frequency of demand misses encountered by the Prefetcher.

Regarding the demand_accesses statistic, it is incremented for every cache access, irrespective of whether it results in a hit or a miss. This reflects the total number of demand access requests handled by the Prefetcher.

Given these statistics, the demand miss rate is calculated as follows:

$$demand\_miss\_rate = \frac{demand\_misses}{demand\_accesses}$$

This rate measures the proportion of cache accesses that result in misses.

Furthermore, the demand hit rate can be derived from the demand miss rate:

$$demand\_hit\_rate = 1 - demand\_miss\_rate$$

Thus, demand_hit_rate represents the proportion of cache accesses that successfully retrieve the desired data, indicating the effectiveness of the cache in fulfilling demand access requests.

Sent Prefetches: Updated within the `Queued::notifyCounterPfDispatch` function, sent_prefetches undergoes an increment each time a prefetch request is dispatched. It effectively quantifies the total number of prefetch requests sent by the Prefetcher.

Useful Prefetches: This metric indicates the effectiveness of the prefetching strategy. It is calculated based on whether the prefetched data is accessed before being evicted from the cache and is defined by the following equation:

$$useful\_prefetches = timely\_prefetches + late\_prefetches$$

It is imperative to underscore the significance of late_prefetches within this context, due to their potential for subsequent access (temporal locality) before being evicted from the cache.

Late Prefetches: Late prefetches are identified when data is fetched into the cache after the CPU has already requested it. Being fetched too late means that the prefetched data is not beneficial for the current CPU operation, but could still be useful for further accesses. The late_prefetches statistic is updated in the `Queued::notifyCounterPfLate` function. This function is called when a prefetch is identified as late, meaning the data was prefetched after it was needed. This statistic helps in assessing the timeliness of the prefetching mechanism. In the formula below, the late_rate is set to zero when there are no useful prefetches to avoid division by zero. Otherwise, it is the ratio of late_prefetches to useful_prefetches, providing a measure of the prefetching mechanism's timeliness and effectiveness.

$$late\_rate = \begin{cases} 0, & if\, useful\_prefetches = 0 \\ \frac{late\_prefetches}{useful\_prefetches}, & otherwise \end{cases}$$

Timely Prefetches: The timely_prefetches statistic is incremented by the `Queued::notifyCounterPfTimely` function. This function is invoked when a prefetch operation is deemed timely, which means that the prefetched data was available in the cache precisely when it was required by the CPU. A high count in this statistic implies that the Prefetcher is effectively predicting and meeting the CPU's data requirements in a timely manner, thereby enhancing the overall performance of the system.

Useless Prefetches: While not explicitly updated in the provided code snippets, the useless_prefetches statistic is typically incremented in scenarios where prefetched data is not utilized before eviction from the cache. This metric plays a significant role in evaluating the proportion of prefetch requests that fail to contribute to cache hits, thereby offering insight into the overall efficiency of the prefetching strategy. The equation below defines the calculation of the metric.

$$useless\_prefetches = \begin{cases} sent\_prefetches - useful\_prefetches, & if\, sent\_prefetches \geq \\ & useful\_prefetches \\ 0, & otherwise \end{cases}$$

Accuracy Rate: The statistic (accuracy_rate) measures how effective the Prefetcher is predicting data to improve cache hits. It is calculated as the ratio and is defined by the equation below.

$$accuracy\_rate = \begin{cases} 0, & if\, sent\_prefetches = 0 \\ \frac{useful\_prefetches}{sent\_prefetches}, & otherwise \end{cases}$$

Coverage Rate: In which extend the prefetches are covering demand_misses is defined by the statistic coverage_rate. It considers both *useful_prefetches* and *demand_misses*. The formula below calculates the ratio of *useful_prefetches* to the total opportunities for prefetching, which is the sum of *useful_prefetches* and *demand_misses*.

$$coverage\_rate = \begin{cases} 0, & if\, useful\_prefetches + demand\_misses = 0 \\ \frac{useful\_prefetches}{useful\_prefetches + demand\_misses}, & otherwise \end{cases}$$

Prefetch Rate: Measuring the efficiency of the Prefetcher in initiating prefetch request, the prefetch_rate statistic calculates the ratio of sent_prefetches and demand_accesses. This metric provides insights into how often prefetches are sent in relation to the total demand_accesses.

$$prefetch\_rate = \begin{cases} 0, & if\, demand\_accesses = 0 \\ \frac{sent\_prefetches}{demand\_accesses}, & otherwise \end{cases}$$

Memory Device Awareness: As discussed in Section 3.2, the architecture comprises 20 cores divided into two memory nodes. Node zero corresponds to the HBM2 memory with 16 cores, while node one is associated with DDR5 with four cores. This distinction is represented in the metric devType, which informs the Prefetcher about the specific memory device to target for prefetching operations. The node ID can be retrieved using the function `Base::mapAddressToNumaID(Addr addr, bool is_secure)`. By evaluating the address and the security flag, the Prefetcher can determine the node IDs, thereby gaining awareness of the memory devices involved.

Memory Device Bandwidth Measurement: In optimizing Prefetcher decisions, understanding memory device characteristics is essential, particularly in bandwidth measurement. For Gem5 simulations, measuring a bandwidth is achievable through the statistics defined in Gem5. However, this method is not transferable to real architectures, which lack the statistical capabilities of Gem5.

The role of TBE (Translation Buffer Entries) in cache controllers is critical, especially during cache misses and subsequent memory requests. When a cache miss occurs, the system retrieves data from lower levels of the memory hierarchy, with TBEs tracking these requests. The Prefetcher receives updates on memory request status through TBE_ALLOC and TBE_DEALLOC events. A TBE_ALLOC event triggers an increment, and a TBE_DEALLOC event a decrement in a statistical counter. This mechanism, implemented in `Queued::notifyCounter(const gem5::memory::PacketPfCounterPtr &pkt)`, informs Prefetcher about in-flight memory requests. The goal is to stress test the architecture, determining its capacity for handling in-flight memory requests. Based on the memory device type, thresholds for low, medium, and high bandwidth are established, aiding the Prefetcher in bandwidth utilization estimation.

Initially, the `tbe_usage_pf` metric was developed to monitor these parameters. However, it was found to track TBE information on a per-core basis, not system-wide, due to per core connection to their private L2 cache. To overcome this, a revised method measures memory device latency from TBE_ALLOC to TBE_DEALLOC, offering a more comprehensive view of system-wide memory interactions.

The statistic `tbe_latency_total` accumulates the total latency for all TBE_DEALLOC events, representing the sum of time intervals from TBE_ALLOC to TBE_DEALLOC. This metric provides an aggregate measure of time efficiency for the prefetching mechanism. In parallel, `tbe_dealloc_num` counts the occurrences of TBE_DEALLOC, reflecting the frequency of memory request completions.

The average latency, `tbe_latency_average`, is calculated as follows:

$$tbe\_latency\_average = \begin{cases} 0, & if tbe\_dealloc\_num = 0 \\ \frac{tbe\_latency\_total}{tbe\_dealloc\_num}, & otherwise \end{cases}$$

This formula ensures zero average latency when no TBE_DEALLOC events occur, avoiding division by zero. Otherwise, it computes the average latency by dividing `tbe_latency_total` by `tbe_dealloc_num`. A higher `tbe_latency_average` during simulation suggests increased memory contention.

Applying this methodology in a high-stress FS-mode simulation across diverse memory devices results in an intricate latency curve. This curve is categorized into low, medium, and high utilization brackets, significantly enhancing the comprehension of bandwidth

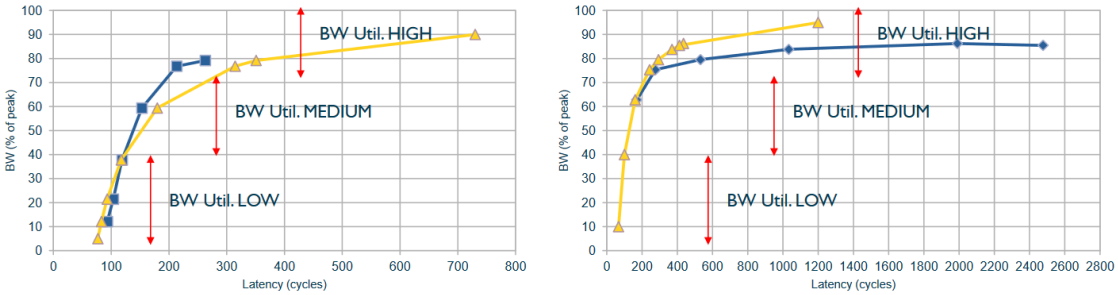


Figure 3.6.: Comparative latency measurement model for HBM and DDR Memory Devices. The left side of the figure illustrates the latency curve for the HBM memory device, while the right side depicts the DDR memory device. In both plots, the blue curves represent the actual TBE latency observed during the simulation. The yellow curves indicate the values calculated by the average latency model. The red range arrows are used to mark the thresholds for low, medium, and high bandwidth utilization, based on the identified saturation points in the latency curves.

utilization for each memory device. Figure 3.6 illustrates the latency measurement model as described.

Each plot in Figure 3.6 delineates the relationship between latency and peak bandwidth during the simulation. The left side of the figure presents data for the HBM memory device, while the right side focuses on the DDR memory device. The blue curves represent the actual TBE latency, and the yellow curve depicts the values from the introduced average latency model. Identifying the saturation points on these curves allows for categorization into three thresholds, defining low, medium, and high bandwidth utilization, as indicated by the red range arrows.

For HBM2, the low bandwidth threshold is identified at less than 120 cycles, corresponding to 40% peak bandwidth utilization. The medium bandwidth range is between 120 and 200 cycles, equating to 40% – 70% peak bandwidth utilization. High bandwidth is categorized as greater than 200 cycles, corresponding to more than 70% peak bandwidth utilization.

Similarly, for DDR5, the low bandwidth threshold is set at less than 100 cycles, aligning with 40% peak bandwidth utilization. The medium bandwidth range spans from 100 to 270 cycles, representing 40% – 70% peak bandwidth utilization. High bandwidth is defined as exceeding 270 cycles, which corresponds to more than 70% peak bandwidth utilization.

Tuning Statistics: The optimization process described in Section 3.6 heavily relies on certain statistics. Detailed discussion is provided in the corresponding section. The tuning of the Prefetcher primarily involves two parameters: degree and lookahead.

Adjusting the behavior of the Prefetcher involves the strategic use of `adjustDegree` and `adjustDistance` to store the values of degree and lookahead. At the conclusion of each epoch, these stored values are applied to Prefetcher in the Queued class, assuming these Prefetcher are set up to employ these optimization strategies.

Additionally, the parameters `distanceVote` and `avgDistance` play a pivotal role in the voting strategy, which will be elaborated upon in the subsequent section. In brief, these parameters are employed to aggregate votes for an optimal lookahead value. The best or average of these votes is then assigned to `adjustDistance`, which in turn influences the lookahead parameter during prefetch calculations.

3.6. Prefetcher optimization

As illustrated in Figure 3.3, the blue boxes within the activity diagram signify key processes where those optimizations are applied. These optimizations involve critical adjustments in the table entries, as well as in the degree and lookahead values. The primary objective of these changes is to achieve an optimal balance between timeliness and aggressiveness in the prefetching process. All configuration parameters for the Prefetcher, including those related to the base and the optimized strategies, are detailed in Appendix A.2.

Degree Optimization

Degree optimization in prefetching is governed by a dynamic model that adapts to varying bandwidth utilization levels, as depicted in Figure 3.6. This model categorizes bandwidth utilization into three distinct levels: L (Low), M (Medium), and H (High). The degree of operation is adjusted based on these levels, in conjunction with accuracy statistics.

In scenarios of L or M bandwidth utilization, the system experiences reduced pressure, enabling the Prefetcher to operate more aggressively. This involves increasing its degree to prefetch a larger number of memory blocks. Conversely, at H bandwidth utilization, it is prudent for the Prefetcher to reset its degree, to avoid worsening bandwidth constraints.

However, bandwidth utilization is not the sole determinant in this optimization process. The accuracy, categorized into L (Low), M (Medium), and H (High) levels, plays a crucial role. This dual-parameter approach ensures a balanced and efficient prefetching strategy. For instance, in a situation with High bandwidth, the Prefetcher should relearn its degree, thereby becoming less aggressive. When bandwidth utilization is at M or L levels, the decision to increase the degree is further influenced by both the accuracy of prefetching and the level of bandwidth utilization. It is important to note that the degree

of prefetching can only decrease to a minimum degree of 1, as a degree of zero would imply no prefetching. Conversely, the maximum degree is capped at a value defined by the benchmark configuration, ensuring that the Prefetcher operates within optimal and predefined limits.

Table 3.4 presents actions and degree adjustments based on the combined assessment of bandwidth utilization and Prefetcher accuracy, while adhering to these minimum and maximum degree constraints.

Accuracy Rate	Bandwidth Utilization	Action	Added Degree
H	H	set to min	reset (1)
H	M	Increase	inc_slow (+1)
H	L	Increase	inc_fast (+2)
M	H	set to min	reset (1)
M	M	Increase	inc_slow (+1)
M	L	Increase	inc_fast (+2)
L	-	set to min	reset (1)

Table 3.4.: Degree Adjustment w.r.t. Accuracy Rate and Bandwidth Utilization Level

Lookahead Optimization

This optimization strategy emphasizes the timeliness of prefetches, as detailed in Section 2.2.2). The lookahead parameter plays a critical role in determining the foresight in requesting memory blocks. An optimal lookahead size is essential for effective prefetching. In the configuration of this work, the lookahead is constrained to an interval from zero to 63. This limitation arises because of the product of $64 * 64 = 4096$, aligning with the physical page size.

A too small lookahead can lead to prefetches arriving too late, causing cache misses at the time of load instructions. On the other hand, an overly large lookahead may result in prefetched blocks being present in the cache but at risk of eviction before use, again leading to cache misses. This scenario necessitates re-requesting the same memory block, thereby reducing performance.

To address the challenge of optimizing prefetch timing, a queuing algorithm for prefetch table entries has been implemented. This algorithm utilizes data from previous prefetches when a cache event occurs, enabling dynamic adjustment of the prefetch trigger based on a flexible lookahead. The methodology of this approach is depicted in Figure 3.7.

The Access stream at the top of Figure 3.7 demonstrates the learning mechanism for optimizing the prefetch trigger. Upon the occurrence of a cache event, relevant information is queued. The foundational requirements for this implementation are sourced from the

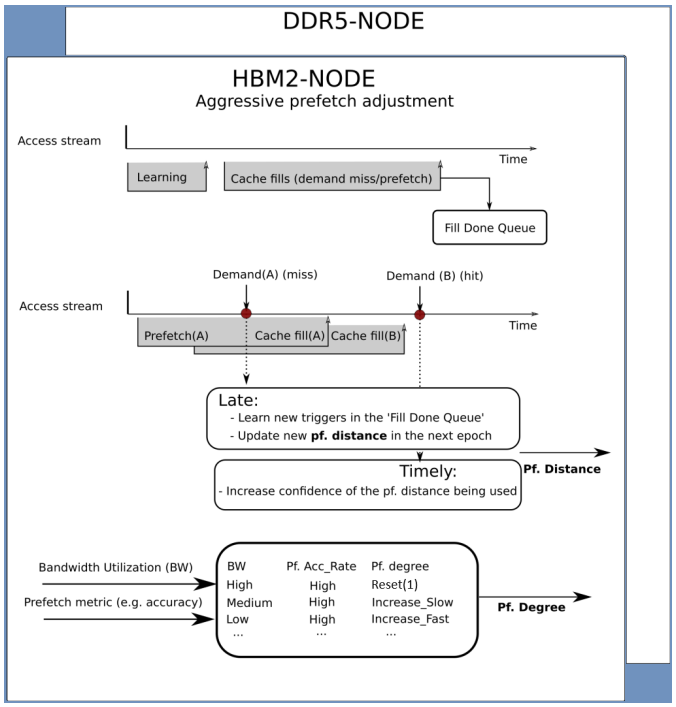


Figure 3.7.: Illustrative overview of dual prefetching optimization strategies: The upper portion of this figure illustrates the learning phase, capturing the trigger data collection. The lower segment demonstrates the lookahead optimization methodology, elucidating the adaptive modulation of prefetch triggers in reaction to cache event patterns, as explicated in Section 3.6. The illustration also comprehensively portrays the degree optimization strategy at its base, correlating it with fluctuating bandwidth utilization and accuracy metrics, as detailed in Section 3.6.

official Gem5 repository, integrated into the Base prefetcher class. A key modification was the selection of appropriate notification functions for different events. For instance, in the depicted scenario, the `notifyFill(const PacketPtr &pkt)` function is invoked, passing the packet that triggered the event. This pattern of pass-through is consistent across all three notification functions.

In the second Access stream of Figure 3.7, two critical points are marked in red, labeled Demand(A) (miss) and Demand(B) (hit). These represent a late prefetch resulting in a cache miss and a timely prefetch leading to a cache hit, respectively. When a late prefetch event occurs, it necessitates learning a new trigger for prefetching, thereby requiring the application of a new lookahead. This adjustment is executed in the `notifyLate(const PacketPtr &pkt)` function. Conversely, for a timely prefetch, the `notifyTimely(const PacketPtr &pkt)` function is used to increase the confidence in the prefetch trigger. This is achieved through a voting mechanism, if enabled. If the feature is not enabled, the

calculated lookahead value is applied to the next prefetch.

At the bottom of Figure 3.7, the degree optimization strategy, as detailed in Section 3.6, is effectively illustrated. This section of the diagram visually represents the intricate process of adjusting the degree based on current system conditions and requirements. This adaptive approach ensures that the Prefetcher operates at an optimal level, balancing aggressiveness and timeliness. The sections below delve into the specifics of each notification function.

notifyFill: This function is activated upon receiving a packet from the cache controller, which has been forwarded to the Prefetcher due to an event in the cache.

There are two primary reasons for this event. Firstly, the packet could be a hardware prefetch triggered by a cache miss. Alternatively, it might be due to a cache fill, indicating that prefetches are arriving at the cache and are being written into it.

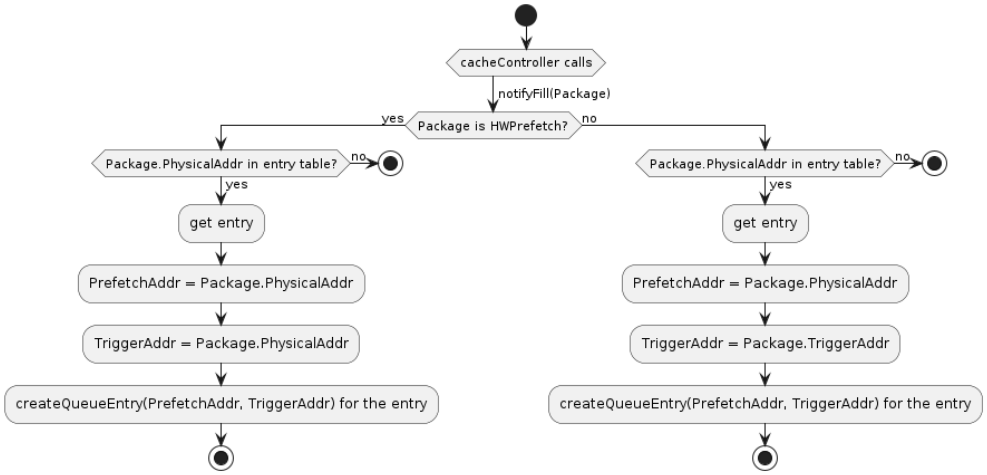


Figure 3.8.: Activity diagram of the `notifyFill` function, illustrating its role in handling prefetch-related events triggered by the cache controller. The diagram showcases the process of address verification, queue entry creation, and updating the fill-done queue based on incoming prefetch requests or cache fills. This function efficiently manages prefetch-related events within the cache system.

In the case of hardware prefetching, the function initially checks whether the physical address within the packet is present in the entry table of the Prefetcher. If the address is located, the corresponding entry is retrieved. In this scenario, the prefetch address and the trigger address are both identified as the physical address from the packet. This characteristic for hardware prefetching, where there is no distinction between the prefetch trigger and the prefetch address, as prefetching occurs without an external trigger. Subsequently, a new queue entry for this address is created and added to the fill-done

queue of the entry.

Conversely, in the event of a cache fill triggered by an incoming prefetch request to the cache, the function follows a similar procedure to check for the physical address in the entry table. Upon finding it, the corresponding entry is retrieved. In this case, the prefetch address coincides with the physical address from the packet, while the trigger address is derived from the trigger address of the packet. The function then proceeds to create a queue entry for this address, using both the identified prefetch and trigger addresses. This ensures that the prefetching queue is updated accurately in response to the demand request. This process is illustrated in Figure 3.8, which depicts an activity diagram for the `notifyFill` function.

notifyLate: This function is invoked when the cache controller signals a late prefetch event, identified by the `notifyLate` call with a packet as its argument. The primary purpose of this function is to handle scenarios where prefetch requests arrive later than expected, causing a cache miss.

Upon activation, the function first retrieves the physical address from the packet, referred to as the late prefetch address. It then checks if this address exists in the Prefetcher entry table. If the address is not found, the function terminates its execution.

If the address is present in the table, the corresponding entry is accessed. The function then examines the fdQueue (fill done queue) associated with this entry. If the queue is empty, the function concludes its operation as there are no prefetch requests to process.

However, if the queue already has entries, the function proceeds to iterate through the queue entries. It assesses each entry to determine if the trigger address aligns with the same memory page as the late prefetch address. When a match is identified, the function calculates a lookahead value. This value is derived from the distance between the late prefetch address and the trigger address, divided by the cache block size. The division by the cache block size is essential because prefetches operate at the level of cache blocks. By dividing through the cache block size, the lookahead value is effectively extracted. Subsequently, the function casts a vote for this lookahead value. The voting is critical in finding the optimal prefetching trigger for future prefetches, but can be disabled by the Prefetcher configuration of the architecture. Once a matching entry is found, the function ceases its iteration and concludes its operation.

This process is illustrated in Figure 3.9, which depicts an activity diagram for the `notifyLate` function.

notifyTimely: The `notifyTimely` function is triggered when the cache controller signals a timely prefetch event. Managing prefetch requests arriving within an expected time frame defines the primary role of this function, thereby contributing to efficient prefetches. Upon invocation, the function first retrieves the physical address associated with the

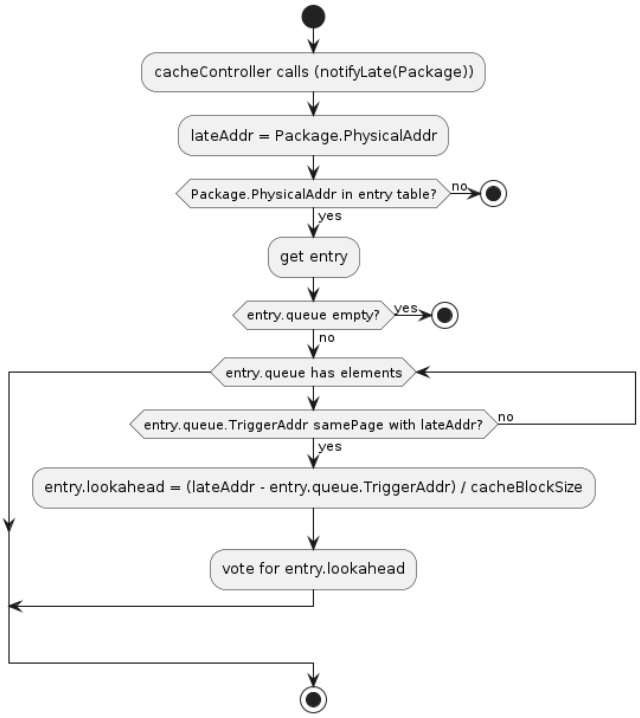


Figure 3.9.: Activity diagram of the `notifyLate` function, illustrating the handling of late prefetch requests. The diagram emphasizes the steps of address retrieval, table entry verification, fdQueue examination, and the calculation of lookahead values for voting. This process is critical for optimizing prefetch triggers in scenarios where prefetch requests arrive later than expected, potentially causing cache misses.

timely prefetch, termed as the timely prefetch address. It then checks for the presence of this address in the Prefetcher entry table. If the address is not found in the table, the function terminates immediately. If the address is found, the function proceeds to access the corresponding entry in the table. It then examines the fdQueue of this entry.

In cases where the fdQueue is not empty, the function iterates through the queue entries. The goal is to find a queue entry where the prefetch address matches the timely prefetch address. Upon finding such an entry, the function calculates a lookahead value. This lookahead is again computed as the absolute difference between the timely prefetch address and the trigger address of the queue entry, divided by the block size of the cache. Finally, the lookahead value is then used in the voting mechanisms, adjusting prefetch distances based on the timing of prefetch requests.

This operation is visually represented in Figure 3.10, which provides an activity diagram for the `notifyTimely` function.

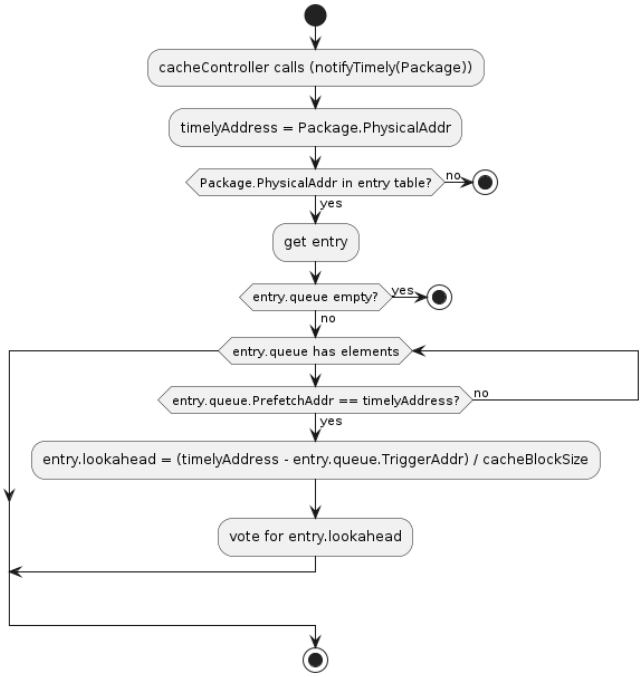


Figure 3.10.: Activity diagram of the `notifyTimely` function, illustrating the process of handling timely prefetch requests. The diagram highlights key steps such as address retrieval, table entry verification, fill-done queue examination, and the calculation of lookahead values for the voting mechanism. This mechanism is critical for adjusting prefetch distances based on prefetch request timing.

Voting Mechanism: In this work, two types of `lookahead voting` mechanisms are implemented. These approaches involve setting the `adjustDistance` metric either to the `most frequently voted` value or to the `mean of all votes`. After the completion of the epoch, the `adjustDistance` and `adjustDegree` parameters are applied to the prefetch address calculation. The choice of which voting mechanism to employ can be specified in the benchmark configuration.

4. Experimental Analysis and Evaluation

This chapter focuses on an in-depth analysis and assessment of FS-mode simulations conducted with the Gem5 simulator. The initial section details the experimental setup, specifically the hardware and software configurations utilized. This provides a clear understanding of the simulation environment.

Next, the chapter then introduces the benchmarks used in the simulations. Following this, the simulation results for these benchmarks are methodically evaluated. This evaluation includes an analysis of the data, interpretation of findings, and discussion of their significance in the context of the study.

Lastly, the chapter concludes with an estimation of the hardware resources required for the practical implementation of the introduced Prefetcher. This section bridges the theoretical aspects of the simulations with practical considerations, offering insights into the feasibility of applying these Prefetcher in real-world hardware scenarios.

4.1. Experimental Environment

During the thesis work, a transition in the experimental setup occurred due to the Juawei system being returned to its owner, leading to the move to the HAICGU cluster. The table in the Appendix A.3 details the hardware specifications of both systems.

The benchmarks “Simple Triad” and “MINIFE SpMV” (Sparse Matrix-Vector Multiplication) underwent simulation on the Juawei Haswell Node. Those two benchmarks are used to explore the benefits of different memory types, in this work HBM2 and DDR5, w.r.t. prefetching and the implemented optimization strategies. Knowing the outcome of those benchmarks, the next step is to explore NUMA-node aware benchmarks, where memory is assigned to NUMA-nodes and may be used together in various calculation. To facilitate these simulations, the ARM ISA was required, prompting the compilation of the benchmarks “Simple Triad” and “MINIFE SpMV” on the Juawei Hi1616 Node. Alternatively, for subsequent simulations conducted on the HAICGU cluster, an alterna-

tive approach was adopted, wherein the benchmarks were compiled within the operating system image and executed on the standard compute nodes.

The transition to the new architecture introduced specific dependencies and required adjustments to the existing scripts. These changes were effectively addressed and subsequently incorporated into the repository [5].

As elucidated in Chapter 3, this work made use of the benchmarking platform provided by [15]. The architectural description employed in these simulations is comprehensively detailed in Table A.1. In the following, a comprehensive analysis of the structure of the benchmarking `Python` file is given. This file, in conjunction with the architectural description provided in the `YAML` file, plays an essential role in generating the full system simulation.

The benchmarking `Python` file starts by defining `_params`, where the sizes of each cache level are specified, along with the attached Prefetcher. It also allows for the definition of kernel parameters.

In the `read_params` function, the values from `_params` are read and assigned to a parameters list. These values are generating distinct simulations for various configurations. For example, specifying `threads = [1, 4, 8, 16]` results in separate scripts for one, four, eight, and sixteen threads. Memory regions are also configured here at `mem_id = [0, 1]`, using "0" for HBM2 and "1" for DDR5. This section further enables the customization of exploration parameters for the Prefetcher, allowing for unique simulations. Parameters like degree and distance values or feature toggles, such as degree and distance adjustment explained in Section 3.6, can be chosen.

Subsequently, the parameters undergo validation in the `check_parameters` function to filter out any incorrect flags or settings.

The `update_conf` function is responsible for updating the configuration of various parameters, including those defined in the architecture. This step primarily focuses on adjusting prefetching parameters, initially set as defaults, necessitating this modification. Additionally, the epoch size is set and remains constant.

In addition, the `get_command` function selects the binary and its associated parameters for execution. Here, the path to the kernels located within the QEMU image is defined. Furthermore, `numactrl` parameters are configured to implement NUMA node-aware programming.

Finally, the `get_env_dict` function configures parameters for OpenMP, including CPU affinity based on the number of threads.

```
$SRCDIR/epi-gem5-pf/exploration/exploration.py \
--fforward-en \
--config numa-2n \
--benchmark numa_pf_bsc_SpMV_test \
--gem5-debug-flags="DBGStats1"
```

Figure 4.1.: Command used to generate FS-Mode scripts with the optimization –fforward-en and debugging flags.

Figure 4.1 shows the command used for the generation of the FS-Mode scripts. This command incorporates the –fforward-en argument, instructing an optimization for the simulation to operate with an atomic CPU until data from the kernels are initialized. Subsequently, the simulator switches to the predefined NeoverseV1 model. This optimization accelerates the simulation process by disregarding timing considerations until the CPU switch occurs. A more detailed explanation can be found in the next Section (4.2). The –config numa-2n argument chooses the numa-2n architectural design (Table A.2). The –benchmark numa_pf_bsc_SpMV_test selects the benchmarking script, as defined above. In this example, it is the Sparse Matrix-Vector Multiplication benchmark. Lastly, –gem5-debug-flags="DBGStats1" sets the debugging prints for the Gem5 simulator. DBGStats1 is used to generate statistical output on an epoch-by-epoch basis.

4.2. HPC Benchmarks

This chapter describes the benchmarks that are not developed during this work but have been adapted for compatibility with the Gem5 simulator, as shown in the Figure 4.2 and Figure 4.1 from the previous section. The adaptation involves `recompiling the benchmark code with specific Gem5 instruction` sets and `integrating it into a QEMU image for Full System (FS) mode simulation`. Figure 4.2 illustrates a pseudocode representation of the modifications required. Initially, the Gem5 instruction set is incorporated by defining `USEM5OPS` with the path to the Gem5 instructions. Concurrently, the `--fforward-en` feature is activated alongside `USEM5OPS` by compiling the benchmark with `USEM5OPS_ETRACE=1` or `USEM5OPS_FFINST=1`. This configuration enables the simulation to operate with an atomic CPU model, disregarding timing until the execution of the first `m5_exit(0)` command. Subsequently, the simulation resets the statistics with `m5_reset_stats(0,0)` after the initialization of benchmark-specific variables `initializeVariables()`. The computational phase is signaled by `startCalculation()`. This phase is pivotal for evaluating the impact of prefetcher. Upon completion of the benchmark, Gem5 generates statistical files, and the simulation concludes with another

`m5_exit(0)` command.

```
#ifdef USEM5OPS
#include <gem5/m5ops.h>
#endif
initializeVariables();
#ifdef USEM5OPS
    #if defined(USEM5OPS_ETRACE) || defined(USEM5OPS_FFINST)
        m5_exit(0);
        m5_reset_stats(0,0);
    #endif
#endif
startCalculation();
#ifdef USEM5OPS
    m5_reset_stats(0,0);
    #if defined(USEM5OPS_ETRACE) || defined(USEM5OPS_FFINST)
        m5_exit(0);
    #endif
#endif
```

Figure 4.2.: Pseudocode illustrating the modifications required to adapt benchmarks for Gem5 simulation.

4.2.1. Simple Triad

The Simple Triad benchmark is a critical tool for assessing memory bandwidth in HPC environments. As a component of the comprehensive STREAM suite, this benchmark is quantifying sustainable memory bandwidth and the associated computational throughput across four core vector operations: Copy, Scale, Add, and Triad. The Triad operation is utilizing memory-intensive tasks and computational demands, as depicted in Figure 4.3.

```
    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        a[i] = b[i] + scalar * c[i];
    }
```

Figure 4.3.: STREAM Triad Kernel Operation

This kernel succinctly models a scenario wherein each element in the `c` array is scaled by a fixed constant (`scalar`), then added to the corresponding element in the `b` array, with the result stored in the array `a`. This sequence of operations exemplifies a harmonious blend

of memory read and write activities, establishing the Triad benchmark as an essential measure for evaluating memory bandwidth within a multifaceted workload context.

Furthermore, the strategic alignment of OpenMP threads with the total number of available physical cores, along with precise OpenMP thread affinity settings, impacts the performance. These optimizations ensure the comprehensive utilization of hardware resources, providing a more accurate representation of the system's memory bandwidth capabilities. However, a higher thread count escalates overall memory bandwidth consumption due to increased traffic on the NoC (Network on Chip) and memory requests, thereby inducing latency. Given its sequential memory load and write operations, the benchmark significantly benefits from spatial locality, as no data is reloaded and operations proceed sequentially. Consequently, a Prefetcher can substantially enhance performance owing to the predictable access pattern of the benchmark and inherent spatial locality [34].

Let N_{core} represent the number of cores, SLC_{size} denote the size of the SLC per core in Mebibytes (MiB), D_{Bytes} signify the byte size of the data type utilized, and A the number of arrays involved. According to the recommendation by [34], the total cache memory should be multiplied by a factor of four. Consequently, the problem size, N, can be determined using the following formula:

$$N = \frac{N_{core} \times SLC_{size} \times 4}{D_{Bytes} \times A} \tag{4.1}$$

Given the configuration where $N_{core} = 20$ cores, $SLC_{size} = 2$ MiB (where 1 MiB $= 1024 * 1024$ Bytes), $D_{Bytes} = 8$ Bytes (since the "double" data types is used), and $A = 3$ (for the three arrays a, b and c), the problem size N is calculated as follows:

$$N = \frac{20 \times 2 \times 1024 \times 1024 \text{ Bytes} \times 4}{8 \text{ Bytes} \times 3} = \frac{20 \times 2 \times 1024 \times 1024 \text{ Bytes} \times 4}{24 \text{ Bytes}} \approx 13,981,867 \tag{4.2}$$

Thus, the problem size N is approximately 13,981,867 elements. In the simulation the problem size is defined as 15,000,000 elements per array to have a little more stress on the network devices.

4.2.2. MINIFE SpMV

MiniFE is designed to emulate the finite element generation, assembly, and solution processes for an unstructured grid problem, with SpMV being a core component in its iterative solvers. A sparse matrix, composed of zero values, necessitates an efficient storage and computational strategy. This efficiency is attained by concentrating on the non-zero elements, thus reducing memory demands and computational overhead.

The SpMV operation is delineated as $y = Ax$, where A represents a sparse matrix, x a dense input vector, and y the output dense vector. This operation forms the core computational kernel in the MiniFE benchmark [11].

The implementation of SpMV in MiniFE2 the SELL (Sliced ELLPACK) format is employed to optimize the sparse matrices, enhancing vectorization and memory access patterns. The SELL format divides the matrix into slices that align with vector registers, enabling simultaneous operations on multiple matrix elements and significantly boosting performance on modern hardware architectures. Each slice contains a fixed number of rows determined by the chunk size C. Within each slice, rows are padded as necessary to ensure that all have the same number of elements, facilitating efficient vectorized operations. This structure optimizes memory access patterns and reduces the overhead associated with sparse matrix storage and operations.

```
#pragma omp parallel for
for(int slice = 0; slice < num_slices; slice++) {
    for(int i = Asliceoffsets[slice_start]; i <
       Asliceoffsets[slice_end); i++) {
        y[i] = 0;
        for(int j = row_start[i]; j < row_end[i]; j++) {
            y[i] += nnz_values[j] * x[column_indices[j]];
        }
    }
}
```

Figure 4.4.: Simplified SpMV Kernel Operation in MiniFE2

This pseudo-code, Figure 4.4, provides a simplified view of the SpMV operation within the MiniFE2 benchmark. It shows the iteration over slices of the sparse matrix, where each slice is processed in parallel, utilizing OpenMP for multithreading. The non-zero elements of the matrix (nnz_values) are multiplied by the corresponding elements in the dense input vector x, indexed by $column\_indices$, and the results are accumulated to form the output vector y.

Given the SELL format, the memory requirement for storing the matrix can be estimated based on the number of non-zero elements and the slice configuration. Let N_{nz} represent the number of non-zero elements within the matrix, N_{rows} the total rows in the matrix, and C the selected chunk size for slices. The memory requirement for storing both the sparse matrix and its corresponding dense vectors can be succinctly represented as:

$$M_{\text{total}} = N_{nz} \times 8\text{Bytes} + (N_{rows} \times C + 2 \times N_{rows}) \times 4\text{Bytes} \tag{4.3}$$

where M_{total} represents the total memory requirement. Here, N_{nz} is represented by an 8-Byte floating-point number. The remaining components, including N_{rows} and C, are accounted for as integers, each occupying 4 Bytes.

Utilizing a three-dimensional (3D) box model within MiniFE, characterized by dimensions nx, ny, and nz, it is neccenecessary to understand the assocation of the box dimensions to the SpMV computational attributes, particularly N_{rows} and N_{nz}. The computation of grid points within the 3D domain, equating to N_{rows}, is given by:

$$N_{rows} = nx \times ny \times nz \tag{4.4}$$

establishing a direct linkage between grid points and matrix rows, premised on a one-to-one correspondence.

The computation of non-zero elements (N_{nz}) is a function of the mesh connectivity and the discretization scheme adopted. For a conventional 3D finite element model employing a 27-point stencil, N_{nz} is estimated as:

$$N_{nz} \approx N_{rows} \times 27 \tag{4.5}$$

This estimation assumes each interior node connects to 26 adjacent neighbors and itself, primarily focusing on interior nodes while excluding minor boundary condition adjustments at the domain edges. The derived value closely approximates that reported by the workload, thereby facilitating the determination of an appropriate problem size for the simulation. However, due to the striding pattern, the actual N_{nz} may be marginally lower.

To optimize computational efficiency, the chunk size C is dependent on the width of SVE units. Given two 256 Bit SVE units, C is delineated as:

$$C = \frac{256 \times 2\,\text{Bit}}{D_{\text{Bit}}} \tag{4.6}$$

where $D_{\text{Bits}} = D_{\text{Bytes}} \times 8$, ensuring alignment with hardware capabilities for memory operations and vectorized computations.

By analyzing the input parameters of MINIFE and applying equations 4.4, 4.5, and 4.6 to formula 4.3, the following equation for calculating memory usage is derived:

$$\begin{aligned} M_{\text{total}} &= nx \times ny \times nz \times 27 \times 8 \text{ Bytes} + (nx \times ny \times nz \times \frac{512 \text{ Bit}}{64 \text{ Bit}} + 2 \times nx \times ny \times nz) \times 4 \text{ Bytes} \\ &= nx \times ny \times nz \times 216 \text{ Bytes} + (nx \times ny \times nz \times 8 + 2 \times nx \times ny \times nz) \times 4 \text{ Bytes} \\ &= nx \times ny \times nz \times 216 \text{ Bytes} + nx \times ny \times nz \times 10 \times 4 \text{ Bytes} \\ &= nx \times ny \times nz \times 216 \text{ Bytes} + nx \times ny \times nz \times 40 \text{ Bytes} \\ &= 256 \times nx \times ny \times nz \text{ Bytes} \end{aligned}$$

For a system with $N_{\text{core}} = 20$ cores and a $SLC_{\text{size}} = 2$ MiB per core the equation to ascertain the feasible problem size is presented as:

$$\begin{aligned} 20 \times 2 \times 1024 \times 1024 \text{ Bytes} &< 256 \text{ Bytes} \times nx \times ny \times nz \\ 40 \times 1024 \times 1024 \text{ Bytes} &< 256 \text{ Bytes} \times nx \times ny \times nz \\ \frac{40 \times 1024 \times 1024 \text{ Bytes}}{256 \text{ Bytes}} &< nx \times ny \times nz \\ 163840 &< nx \times ny \times nz \end{aligned}$$

Our simulation adopts parameters nx, ny, and nz equal to 63, yielding a total of 250,047 elements. This exceeds the maximum of 163,840 elements determined to fit within the cache and makes possible to measure the influence of prefetching while saving time on simulatiuon time. Consequently, the implementation diverges from the conventionally recommended factor of four, as outlined in [34].

4.2.3. Simple Triad - NUMA Version

The Simple Triad Numa benchmark extends the conventional Simple Triad framework by incorporating support for NUMA (Non-Uniform Memory Access) nodes. This integration facilitates targeted memory operations across an array of NUMA nodes. The mechanism permits data to be transferred between arrays that may be allocated on disparate or identical NUMA nodes. This adaptability enables a comprehensive assessment of memory bandwidth and latency across various configurations, such as transfers from HBM2 to DDR5, DDR5 to HBM2, and within the same memory type. Contrary to the original Simple Triad benchmark, which was limited to simulating operations on HBM2 or DDR5, the current iteration explores the assignment of arrays to

different nodes and the capacity of the mesh of the architecture to manage traffic within the NoC. Additionally, the investigation extends to whether prefetching techniques and adaptive optimization strategies yield benefits under these conditions. Despite these enhancements, the benchmark retains the problem size of the original Simple Triad, owing to its utilization of the identical kernel.

Inspired by the study presented in [30], which focused on the allocation of vectors to either DRAM or NVDIMMs and their impact on memory bandwidth and latency, this work delves into the performance outcomes of assigning vectors within the Simple Triad NUMA benchmark to DDR5 and HBM2.

4.3. Evaluation

This section restructures the evaluation of prefetching techniques for clearer understanding. It begins by establishing the context and goals of the evaluation, focusing on the comparison between different prefetching configurations and their impact on execution times across various benchmarks. The evaluation aims to assess how Prefetcher, specifically the `Stream-based Prefetcher`, the `Agg Prefetcher` (Aggressive Prefetcher) and the `TiA Prefetcher` (Timely Aware Stride Prefetcher), contribute to performance improvements by minimizing latency during computation phases. This assessment compares the effectiveness of prefetching techniques against scenarios without prefetching, as discussed in Section 3.6.

The evaluation methodology is outlined, detailing the primary metric for analysis: the speedup provided by each Prefetcher, which indicates their effectiveness in reducing latency. Speedup is calculated by subtracting the execution time of the workload with prefetching from that without prefetching, and dividing the result by the execution time without prefetching. To illustrate the evaluation findings, block charts are employed, showcasing the speedup across different thread counts for each benchmark. These charts serve to visually represent the performance improvements facilitated by various Prefetcher configurations. An example of such a chart is Figure 4.5, which features the execution times with and without prefetching. In these charts, the x-axis denotes the execution times, while the y-axis lists the Prefetcher configurations. The configuration without prefetching is depicted in `orange`, labeled as `NoP`, while the `various prefetching strategies` are shown in `blue`, offering a clear and direct comparison of how different prefetching strategies affect the execution time of the same workloads. Additionally, the speedup values are shown as percentages on the bars, indicating the improvement in execution time against the scenario with no prefetching. This color coding and percentage annotation on the bars provide a comprehensive overview of the impact of prefetching on

benchmark performance, emphasizing the contrast and quantifying the speedup achieved by prefetching configurations compared to the baseline without prefetching.

Ten configurations of employed Prefetcher are selected for visual representation in the barchart, highlighting the variations in feature enablement and static settings. For the `Stream-based Prefetcher`, the best static configuration and the worst static configuration are elected for display, indicated as **Str(Dg, Dist)**, where Dg is the degree and Dist is the distance. Additionally, the top 7 configurations of the TiA and Agg Prefetcher are also depicted in blue, providing a detailed comparison. The Agg Prefetcher configurations may be labeled as **Agg(adjDg, adjDist, votEn, vot_avg)**, where adjDg enables degree adjustment, adjDist enables distance adjustment, votEn activates voting, and vot_avg indicates if the average (1) or most voted (0) distance is used. Similarly, for the TiA Prefetcher, configurations are labeled as **TiA(adjDg, votEN, vot_avg)**, sharing the same feature descriptions as those for the Agg Prefetcher. This detailed labeling and representation of Prefetcher configurations offer insights into the specific features and adjustments that contribute to their performance, allowing for an in-depth analysis of their impact on execution time improvement.

The evaluation omits the PC-based Stride Prefetcher for several reasons. Foremost, the evaluation already incorporates the static configuration of the Stream-based Prefetcher. Additionally, all evaluated Prefetcher employ a windowing technique, which is incompatible with the PC-based Stride Prefetcher. Lastly, including this Prefetcher would entail conducting hundreds of additional simulations for various configurations, similar to those executed for the Stream-based Prefetcher.

Following this, the study presents an in-depth analysis of significant or unusual configurations through `epoch-based plots`. These plots provide a granular view of performance metrics such as accuracy, coverage, prefetch degree, and bandwidth utilization. An exemplary demonstration of this analysis is provided in Figure 4.6a, which organizes the key metrics into `four quadrants` for a clearer understanding. This approach is first utilized in Section 4.3.1 under the context of the HBM2 Node with 1 Thread, serving as an initial application to elucidate how these metrics are interconnected. It aims to make the complex relationships between these performance metrics understandable.

The `top left quadrant` of Figure 4.6a showcases **demand accesses** (depicted in **blue**), **total prefetches** (in **orange**), **useful prefetches** (in **green**), and **late prefetches** (in **red**), offering insights into the operational dynamics of the Prefetcher by highlighting the balance between initiated prefetches and their utility in computation. In the `top right quadrant`, focus is placed on **bandwidth utilization levels**. The `bottom left quadrant` provides details on the **used prefetch degree**, shown in **blue**, and the **prefetch distance**, displayed in **orange**. Lastly, the `bottom right quadrant` illustrates **accuracy** (in **blue**) and **coverage** (in **orange**). Employing this quadrant-based

analysis allows for a comprehensive examination of Prefetcher configurations, facilitating a deeper understanding of their effectiveness and behavior.

4.3.1. Simple Triad

HBM2 Node with 1 Thread

As elucidated in Section 4.3, Figure 4.5 displays the simulation results for a single-thread execution of the Simple Triad benchmark on an HBM2 device. The least effective static configuration of the Stream-based Prefetcher (`Str(degree,distance)` with both degree and distance set to two) demonstrated a speedup of 5.30%. Conversely, the most effective configuration of the Stream-based Prefetcher, with degree four and distance 32, resulted in a speedup of 27.25%. Additionally, the Figure illustrates performance variations attributable to the activation or deactivation of various features in the Agg Prefetcher and TiA Prefetcher.

Notably, the Agg Prefetcher, with the distance adjustment and voting feature enabled, and selecting the mean of distance votes, closely approached the performance of the optimal static configuration, showing a speedup of 26.66%. Using the most voted distance over the mean of votes, for the Agg Prefetcher, yielded a speedup of 25.71%. The marginal difference between both voting strategies suggests negligible impact, indicating their near-optimal efficacy. These findings indicate that different workloads may require distinct degree and distance settings, and bandwidth enhancements could modify these parameters. Thus, the adaptive nature of distance and degree adjustments, which respond to varying environmental conditions, offer a significant advantage. The TiA Prefetcher configurations display similar results followed by the Agg Prefetcher configurations within the same configuration. Further details on this adaptive behavior are presented in Figure 4.6.

On the left, Figure 4.6a presents epoch-by-epoch accumulated statistics and metrics for the Aggressive Prefetcher with both degree and distance optimization activated, applying the mean of votes to determine distance. Conversely, the right-hand Figure 4.6b displays the static configuration of the Stream-based Prefetcher, with distance and degree set to two. Analysis of the upper left quadrants in both Figures reveal a correspondence between total prefetches (in orange) and useful prefetches (in green). However, a notable difference exists in the proximity of useful prefetches to demand accesses (in blue) in Figure 4.6a, indicating a higher efficacy of the Agg Prefetcher in exploiting prefetching opportunities compared to the static configuration of the Stream-based Prefetcher.

The variation in the degree used, observable in the bottom left quadrants of both Figures 4.6a and 4.6b, accounts for this discrepancy. The Agg Prefetcher, operating under low

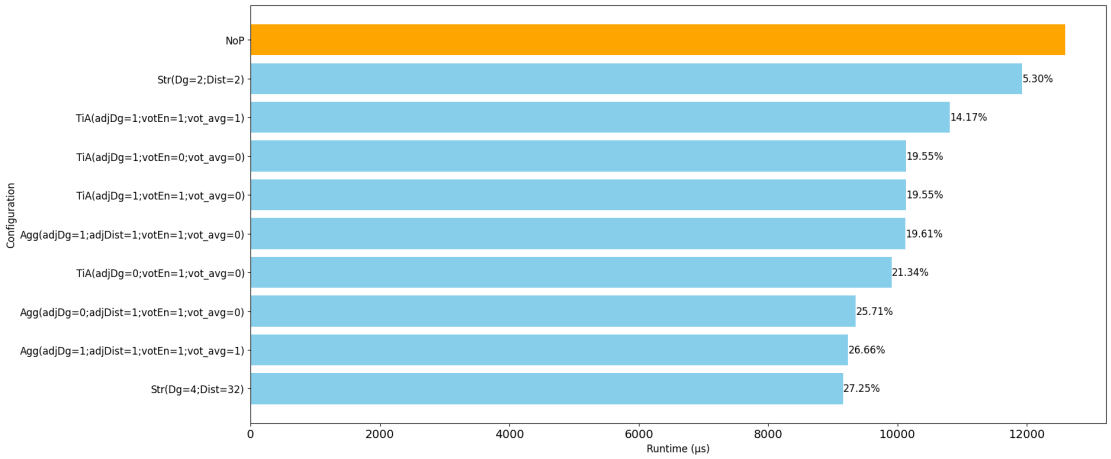
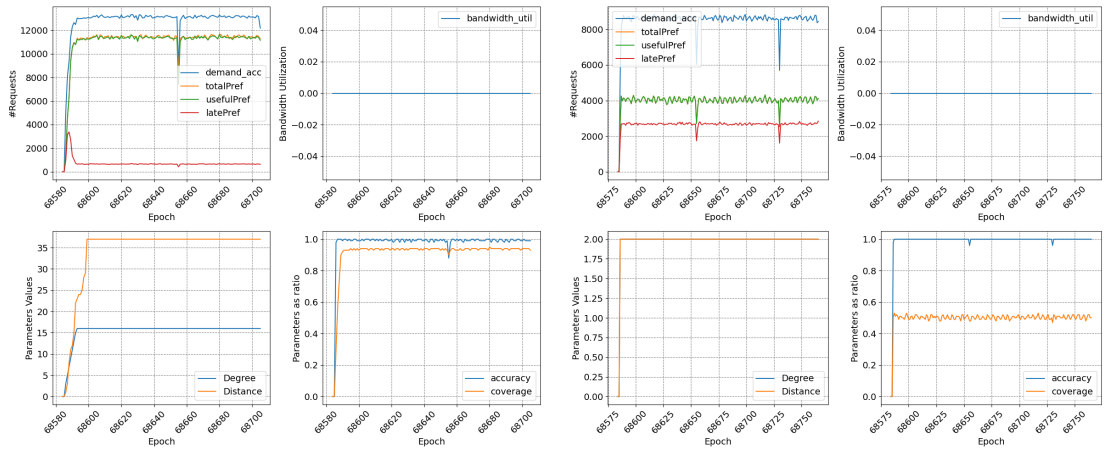


Figure 4.5.: Performance comparison of prefetching strategies for the Simple Triad benchmark on an HBM2 node with 1 thread. Execution times for the Stream-based (Str), Aggressive (Agg), and Timely Aware Stride (TiA) Prefetchers are shown alongside the no-prefetching baseline (NoP). Notable findings include a 27.25% speedup with the best Stream-based configuration and a closely matched 26.66% improvement by the Agg Prefetcher with all advanced features enabled. These results highlight the effectiveness of adaptive prefetching in reducing computation latency, while being close to the optimal solution.

bandwidth utilization, (upper right quadrant of Figure 4.6a) and demonstrating high accuracy in pattern prediction (bottom right quadrant of the same Figure), adjusts its degree up to 16 as per the guidelines in Table 3.4. This flexibility is not available in the static configuration, leading to a higher coverage of 95% for the Agg Prefetcher compared to 50% for the static configuration, as evidenced in the bottom right quadrants of both Figures. The accuracy levels, however, are similar for both configurations.

The investigation into late prefetches, depicted in the top left quadrants of each Figure in red, initially reveals a greater occurrence in the Aggressive Prefetcher, rapidly diminishing in the following epochs as a result of the implemented optimization strategies, in contrast to the static configuration. This difference is influenced by the greater volume of prefetches issued by the Agg Prefetcher. Despite this, 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 enhancing timely prefetches and improving overall workload performance. The implemented degree adjustment values are visible in the bottom left quadrant of each Figure, with the Agg Prefetcher setting a larger distance to optimize prefetches for patterns exhibiting high spatial locality.



(a) (b)

Figure 4.6.: Epoch-based performance metrics analysis for the Agg Prefetcher and the static configuration of the Stream-based Prefetcher during the Simple Triad benchmark on an HBM2 node with 1 thread. Left 4.6a: Agg Prefetcher with adjustable degree, distance, and voting enabled. Right 4.6b: Static configuration of the Stream-based Prefetcher. These Figures compare the operational dynamics, bandwidth utilization, prefetching accuracy, and coverage of dynamic versus static prefetching configurations. The analysis underscores the adaptive advantage of the Agg Prefetcher in optimizing prefetching parameters for enhanced benchmark performance, demonstrating its superior ability to adjust to workload demands and improve execution times compared to static prefetching approaches.

HBM2 Node with 4 Threads

Building on the foundational insights garnered from the execution of the Simple Triad benchmark on an HBM2 node with a single thread, the transition to a multi-threaded environment introduces a new dimension of complexity and performance dynamics. The execution of the same benchmark, now leveraging four threads, slightly amplifies the demand on the HBM2 device, leading to an intricate interplay of memory requests and data traffic. This escalation not only tests the scalability of prefetching strategies under heightened workload conditions but also offers a revealing look into the efficacy of adaptive prefetching mechanisms when confronted with increased memory access concurrency.

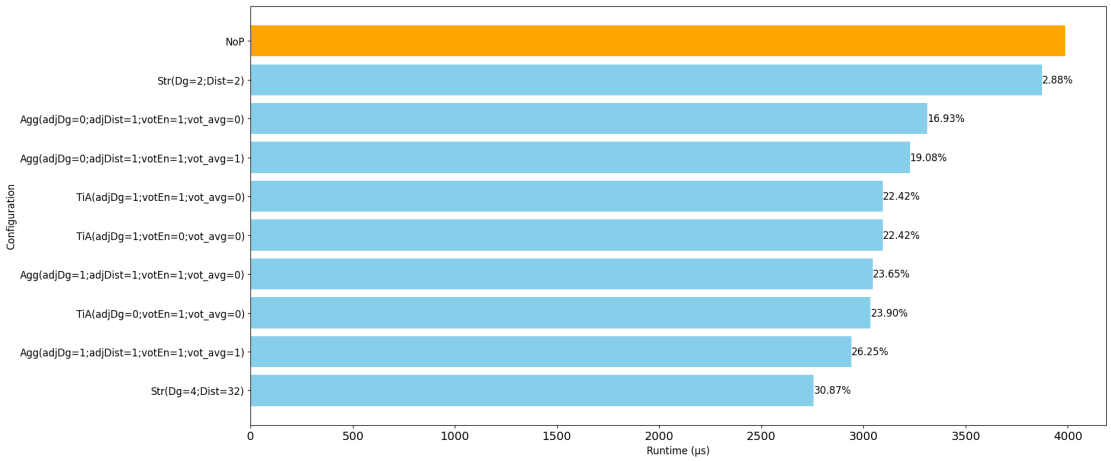


Figure 4.7.: Performance comparison of prefetching strategies for the Simple Triad benchmark on an HBM2 node with 4 threads. Execution times for the Stream-based (Str), Aggressive (Agg), and Timely Aware Stride (TiA) Prefetchers are shown alongside the no-prefetching baseline (NoP). Notable findings include the Stream-based Prefetcher achieving the highest speedup of 30.87% with a configuration of degree 4 and distance 32. the Aggressive Prefetcher, enabled with adjustable degree, distance, and voting features, applying the mean of votes to the distance, surpasses other configurations with a speedup of 26.25%, underscoring the effectiveness of distance adjustment in multi-threaded environments. However, similar configurations of the Agg Prefetcher, when applying the most voted distance, closely match, with a speedup of 23.65%.

Figure 4.7 illustrates the performance impact of prefetching configurations in a multi-threaded environment. The best static configuration of the Stream-based Prefetcher with a degree of four and distance of 32 significantly improves performance, achieving a 30.87% speedup.

The Agg Prefetcher and TiA Prefetcher exhibit a range of performances based on the adjustment of their parameters. A particular setup of the Aggressive Prefetcher, enabled with adjustable degree, distance, and voting features, applying the mean of votes to the distance, surpasses alternative configurations in performance, resulting in a speedup of 26.25%. Closly followed by the same configuration with applying the most voted distance with a speedup of 23.65%. The TiA Prefetcher configurations display similar speedup values around 22.42% to 23.90%. However, as in the single-threaded environment, the Agg Prefetcher with degree adjustment and distance adjustment, while applying the most voted distance, is again close to the optimal static configuration.

HBM2 Node with 8 Threads

The execution of the Simple Triad benchmark on an HBM2 node, previously assessed with single and four-threaded configurations, has now been extended to eight threads. This expansion significantly heightens the operational demands of the HBM2 memory, offering an in-depth evaluation of prefetching strategies under escalated workloads and the effectiveness of adaptive prefetching mechanisms amidst substantial memory access concurrency.

In an eight-threaded environment, as depicted in Figure 4.8, the performance impact of distinct prefetching configurations emerges. The Stream-based Prefetcher, when set to a static configuration with a degree of two and a distance of 32, achieves the highest speedup at 20.28%. This marks a notable shift from the four-threaded setup, where a higher degree was advantageous, suggesting an optimal trade-off between prefetching aggressiveness and the volume of memory requests within the system.

The Agg Prefetcher, with distance and degree adjustments enabled and voting mechanism activated, employing the most voted distance, achieves the second-highest speedup of 16.77%. Conversely, applying the mean of votes to the distance in the same prefetcher configuration yields a speedup of 13.61%. This performance, while slightly below the optimal configuration, still underscores the adaptability of the Agg Prefetcher to environmental stress. These dynamic adjustments, however, require a span of epochs to effectively adapt, suggesting that simulations with variable epoch durations could yield additional insights into prefetching efficiency.

The TiA Prefetcher configurations, especially when utilizing degree and distance adjustments along with the mean of votes for determining the distance, demonstrate limited performance improvements, achieving a speedup of 5.91%. The adaptive behavior, implemented in both prefetchers, prompts an investigation into the reasons behind the significant performance disparity between them.

In the left side of Figure 4.9, the Agg Prefetcher (Figure 4.9a) is depicted with adjustable

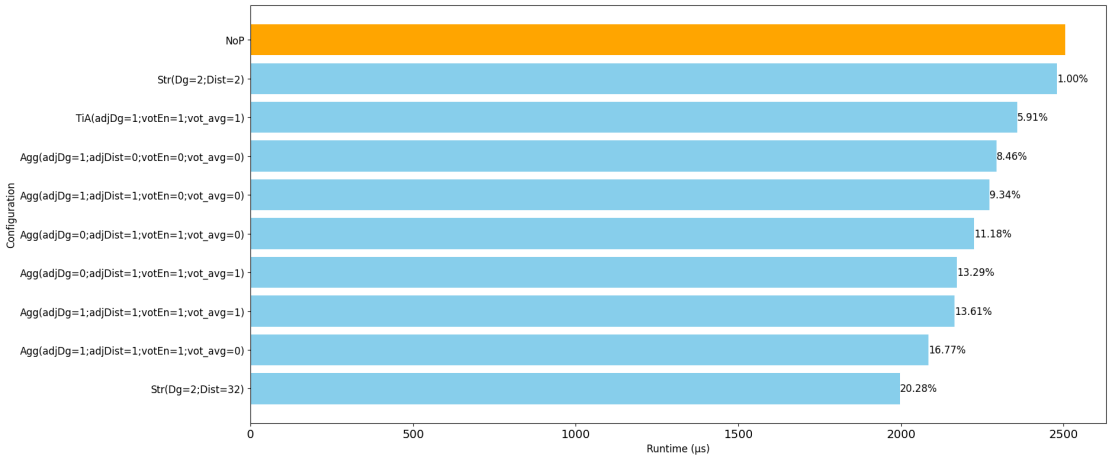
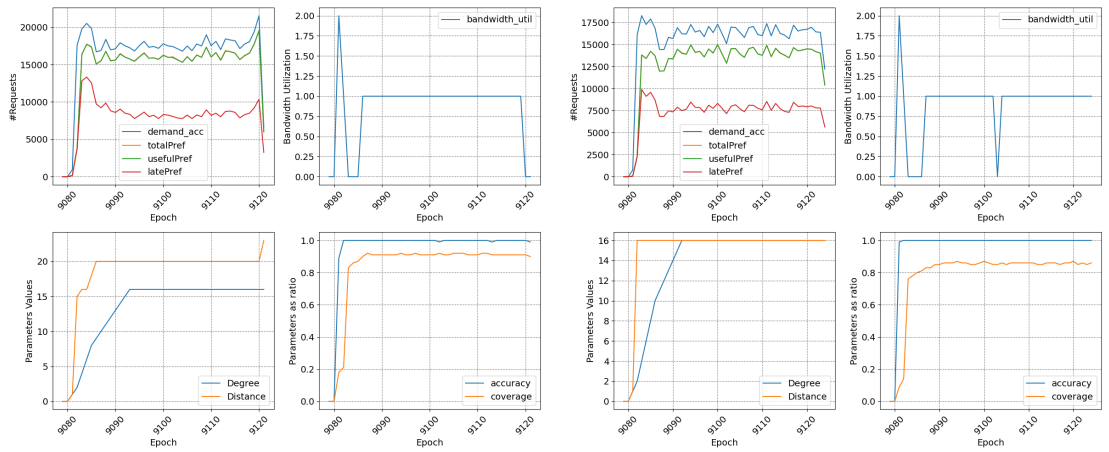


Figure 4.8.: Performance comparison of prefetching strategies for the Simple Triad benchmark on an HBM2 node with 8 threads. Execution times for the Stream-based (Str), Aggressive (Agg), and Timely Aware Stride (TiA) Prefetchers are shown alongside the no-prefetching baseline (NoP). The Stream-based Prefetcher leads in optimality with a 19.6% speedup in its static configuration with a degree of two and a distance of 32. The Agg Prefetcher, with its distance adjustment disabled and voting mechanism engaged, attains the second-best speedup of 18.68%, while the TiA Prefetcher configurations, especially when employing degree and distance adjustments with the most voted distance, exhibit performance on par with the Agg Prefetcher under analogous conditions. Results suggest that the mechanics of prefetch trigger learning and queue management could benefit from a re-examination in conjunction with experimentation across varied epoch lengths, which may unravel the nuances of these adaptive strategies.

degree, distance, and mean voting. On the right (Figure 4.9b), the TiA Prefetcher is presented with the same configuration. Both demonstrate similar bandwidth utilization patterns in the top right quadrant. However, a noticeable drop in bandwidth utilization for the TiA Prefetcher is observed post epoch 9100 (Figure 4.9b), attributed to two main reasons. Firstly, the confidence algorithm halts prefetching when the confidence for the observed pattern is low. Secondly, prefetching ceases if the observed addresses lack a corresponding PC value. However, a distinction is observed in the coverage, denoted in orange at the bottom right quadrant of each Figure. The TiA Prefetcher exhibits a coverage of approximately 85%, as shown in the Figure 4.9a, where the Agg Prefetcher demonstrates a coverage of around 95% in Figure 4.9b. Analysis of the top left quadrant in each Figure reveals that the useful prefetches, indicated in green, are closer to the demand accesses, shown in blue, for the Agg Prefetcher compared to the TiA Prefetcher, elucidating the observed coverage disparities. Nonetheless, the decline in the count of



(a) (b)

Figure 4.9.: Epoch-based performance metrics analyses for the Agg Prefetcher and TiA Prefetcher during the Simple Triad benchmark on an HBM2 node with 8 thread. Left 4.9a: Agg Prefetcher with adjustable degree, distance, and mean voting enabled. Right 4.9b: TiA Prefetcher with same features enabled. These Figures compare the operational dynamics, bandwidth utilization, prefetching accuracy, and coverage of dynamic versus static prefetching configurations. The analysis underscores the impact of adaptive strategies on prefetching efficiency, revealing the significant performance disparity driven by the reliance to PC.

late prefetches, depicted in red, is not as pronounced for the TiA Prefetcher as it is for the Agg Prefetcher, leading to a higher number of late prefetches and consequently, extended execution time. This explains the lower speedup value for the TiA Prefetcher. The reason behind this is evident in the bottom left quadrant of each Figure. The graph, highlighting the learned distance in orange, swiftly stabilizes at a distance of 16, indicating that the TiA Prefetcher exclusively relies on the PC for trigger learning. In contrast, the Agg Prefetcher considers all addresses for trigger learning and more adeptly adjusts its distance, leading to prefetches that are more timely. Another observed distinction is the slower degree adjustment for the Agg Prefetcher compared to the TiA Prefetcher, despite both employing the same algorithm for this purpose, pointing to potential areas for further exploration. It is suggested that the mechanics of prefetch trigger learning and queue management could benefit from a re-examination in conjunction with experimentation across varied epoch lengths, which may unravel the nuances of these adaptive strategies.

DDR5 Node with 1 Thread

Transitioning from HBM2 to DDR5 memory introduces a trade-off between increased capacity and reduced memory bandwidth. This trade-off is evident in the performance results illustrated in Figure 4.10 for the Simple Triad benchmark.

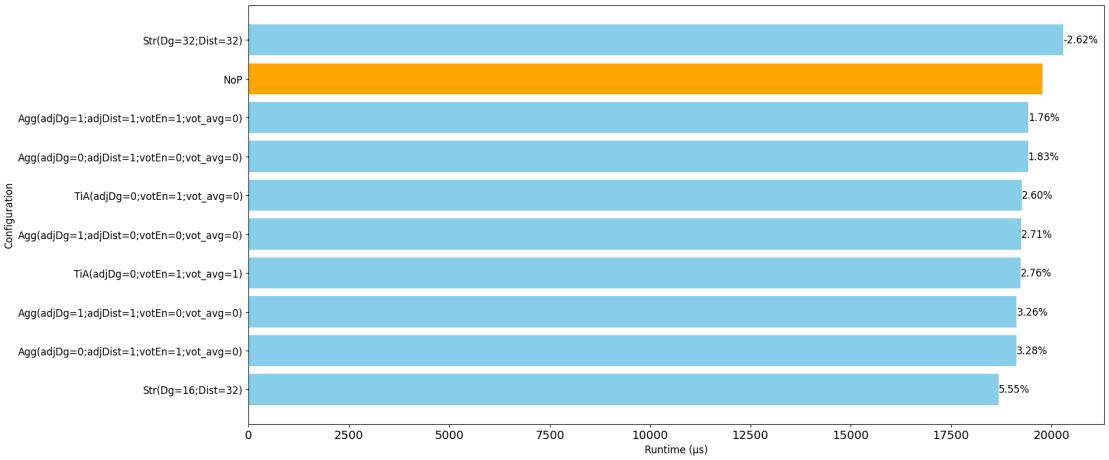


Figure 4.10.: Performance comparison of prefetching strategies for the Simple Triad benchmark on a DDR5 node with 1 thread. Execution times for the Stream- based (Str), Aggressive (Agg), and Timely Aware Stride (TiA) Prefetchers are shown alongside the no-prefetching baseline (NoP). A performance degradation is observed when transitioning from HBM2 to DDR5 memory with a single thread in the Simple Triad benchmark. This Figure demonstrates the impact of different Stream-based Prefetcher configurations on DDR5 memory performance, highlighting a decrease of -2.62% with the least effective static configuration and an improvement of 5.55% with the most efficient configuration. The results underscore the importance of optimal Prefetcher settings in mitigating the inherent trade-offs between capacity and bandwidth in DDR5 memory systems.

In comparison to the single-threaded HBM2 simulation, the DDR5 memory shows a noticeable performance degradation. Employing the least effective static configuration of the Stream-based Prefetcher, with both degree and distance parameters set to 32, results in a performance decrease of -2.62%. This decrease is attributed to the excessive prefetch degree, which generates more memory requests and consequently increases latency, leading to delayed data delivery. Conversely, the most efficient static configuration for the Stream-based Prefetcher, maintaining the distance at 32 but reducing the degree, exhibits a performance improvement of 5.55%. This suggests that a balanced degree is crucial for optimal prefetching performance.

The adaptive configurations of both the TiA and Agg Prefetchers demonstrate comparable

performance enhancements, with speedups ranging from 1.76% to 3.28%. These configurations, despite minor differences in performance, effectively illustrate that tunable parameters could yield substantial performance benefits. The results also indicate that enabling all features and applying the most commonly selected distance parameter consistently yield favorable outcomes.

DDR5 Node with 4 Threads

Elevating the thread count to four exerts additional stress on the DDR5 memory system, as delineated in Figure 4.11.

Both the optimal and suboptimal static configurations of the Stream-based Prefetcher exhibit notable performance degradations. The least effective setup, with a high prefetch degree as previously discussed, incurs a substantial performance decrease of −33.23%. Conversely, other configurations lead to marginal performance reductions around −2.39% or slight improvements of up to 0.28%, essentially aligning with the performance observed with no prefetching. This phenomenon suggests an in-depth examination depicted in Figure 4.12.

Figure 4.12a highlights the TiA Prefetcher as the most effective performing configuration, achieving a speedup of 0.28% with all optimization strategies activated. Conversely, Figure 4.12b illustrates the least effective static configuration for the Stream-based Prefetcher, resulting in a performance reduction of −33.23%. Although not depicted in Figure 4.12c, due to the representation constraints of only 10 configurations, the performance of the Agg Prefetcher with the same configuration as the best performing TiA Prefetcher is also examined. This analysis is conducted to elucidate the factors contributing to the superior performance of the TiA configuration in this experiment.

The analysis of the depicted Figures reveal a high bandwidth utilization, visible at the top right quadrants, indicating substantial stress on the memory device. A comparative examination of the adjusting Prefetcher, TiA and Agg, is presented in Figures 4.12a and 4.12c, respectively. In the bottom right quadrant of each Figure, it is observed that the accuracy of both Prefetcher, represented in blue, achieves 100%. Because of the high bandwidth utilization, the intial logic of the optimization strategy does not increase the prefetch degree. However, a significant distinction is observed in the coverage, denoted in orange. The TiA Prefetcher exhibits a coverage of approximately 15%, as shown in the top left of Figure 4.12a, whereas the Agg Prefetcher demonstrates a substantially higher coverage of around 85% to 90% in Figure 4.12c.

This discrepancy suggests that the TiA Prefetcher does not encompass all potential prefetch opportunities, a conclusion further supported by the observation of low late prefetches in the top right quadrant of Figure 4.12a.

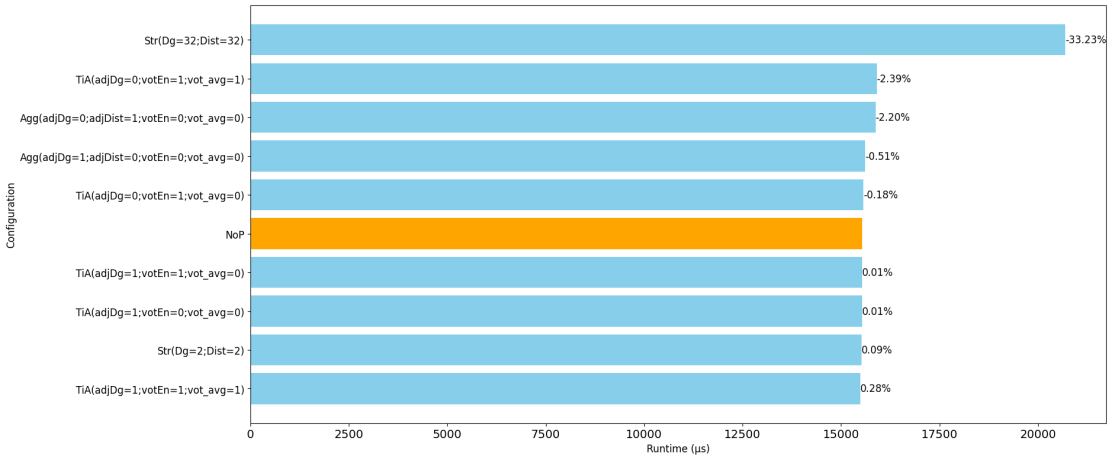


Figure 4.11.: Performance comparison of prefetching strategies for the Simple Triad benchmark on a DDR5 node with 4 threads. Execution times for the Stream-based (Str), Aggressive (Agg), and Timely Aware Stride (TiA) Prefetchers are shown alongside the no-prefetching baseline (NoP). Increasing the thread count to four for DDR5 memory impacts the performance. The Figure illustrates significant performance degradation up to −33.23% with the least effective setup and minor performance variations with other configurations. This analysis highlights the critical role of prefetching strategies in managing additional stress on memory systems induced by higher thread counts.

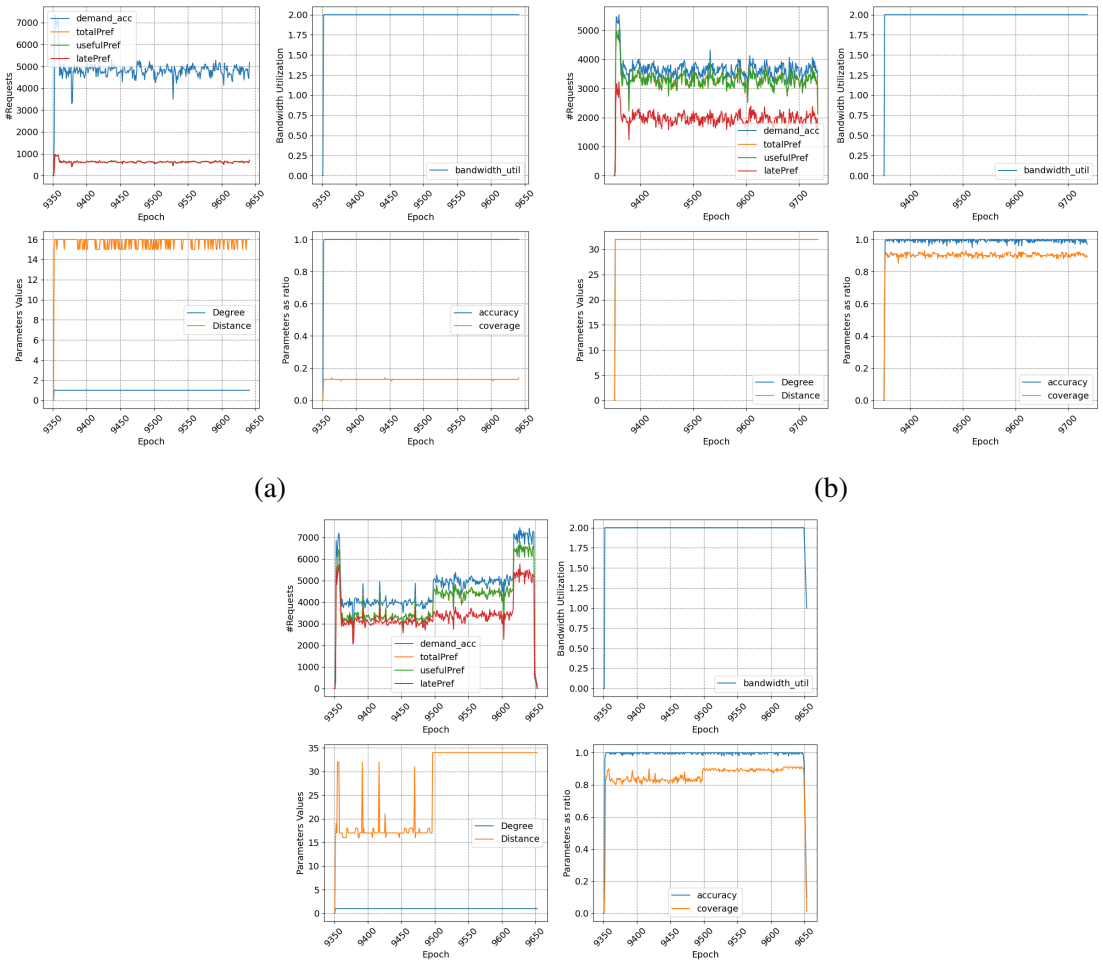
This phenomenon may be attributable to two factors. Firstly, if the total prefetches (in orange) and useful prefetches (in green) were to align with demand accesses (in blue), it would imply an ideal prefetch scenario with perfect prediction and timely data delivery, thereby indicating high coverage, which is contradicted by the observed data. Consequently, the more plausible explanation is the alignment of total and useful prefetches with the late prefetches (in red), suggesting that the Prefetcher issues a limited number of prefetches. This approach, whereby fewer prefetches are issued, even at the risk of them being late, provides performance benefits in environments utilizing DDR5 Memory with high bandwidth utilization. With this understanding, the focus shifts towards comparing the behavior of Agg Prefetcher with the least effective static configurations.

Both the Agg Prefetcher, as shown in Figure 4.12c, and the Stream-based Prefetcher with a static configuration, as depicted in Figure 4.12b, exhibit similar accuracy. However, the coverage is costantly arround 90% for the static configuration for the Stream-based Prefetcher, visable in the bottom right quadrant of their respective Figures.

As illustrated in Figures 4.12c and 4.12b, both the Agg Prefetcher and the Stream-based

Prefetcher with a static configuration demonstrate comparable accuracy levels. Notably, the Stream-based Prefetcher maintains a consistent coverage of approximately 90%, as observed in the bottom right quadrant of the corresponding Figures. The Agg Prefetcher is observed to have a lower degree of prefetching, a point previously elucidated, and displays fluctuating prefetch distances before stabilizing at a value of 34. This suggests an attempt by the Prefetcher to fetch data as close to the page boundary as possible, aiming for timely prefetches akin to the fixed configuration of 32 observed in the Stream-based Prefetcher. Analysis of the top right quadrants for both Prefetcher reveals that total and useful prefetches closely align with demand accesses. Nonetheless, the data arrives late for more than half of the prefetches, as indicated by the late prefetches marked in red, culminating in a decline in performance for benchmarks with high spatial locality, as Simple Triad is.

The findings necessitate a deeper examination of the strategy for adjusting distances in situations characterized by high bandwidth usage. Yet, this task may present significant challenges, given that employing the most effective static strategy has not resulted in any performance improvements.



(a) (b)

(c)

Figure 4.12.: Comparative epoch-based analysis of prefetching in a 4-thread DDR5 setting underscores the TiA Prefetcher's efficiency with limited prefetches, positioned at the top left 4.12a. In contrast, the Agg and Stream-based Prefetchers, shown at the bottom 4.12c and top right 4.12b respectively, demonstrate similar accuracies yet differ in strategy. A lower prefetch degree and greater distance for the Agg Prefetcher aim to approach page boundaries closely, enhancing timeliness but not necessarily efficiency. Both strategies exhibit alignment between total, useful, and demand accesses but suffer from late deliveries in high spatial locality benchmarks, impacting performance.

DDR5 Node with 8 Threads

The transition to an 8-thread configuration in the DDR5 memory system necessitates a reevaluation of prefetching strategies previously examined under a 4-thread scenario. Initial observations suggested minimal to no benefits from prefetching with four threads, as shown in Figure 4.11. Advancing to eight threads, one might anticipate an intensified challenge for prefetching effectiveness due to increased demands on the memory system.

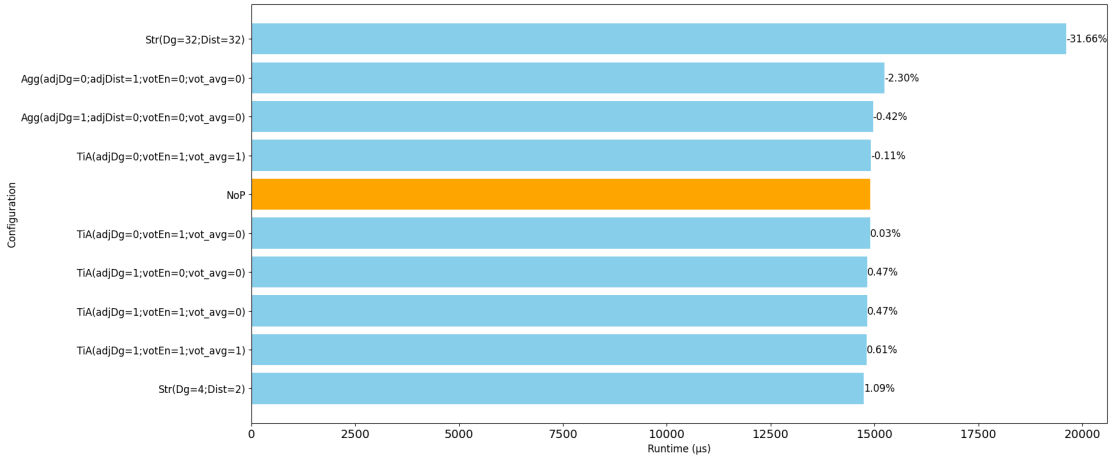


Figure 4.13.: Performance comparison of prefetching strategies for the Simple Triad benchmark on a DDR5 node with 8 threads. Execution times for the Stream- based (Str), Aggressive (Agg), and Timely Aware Stride (TiA) Prefetchers are shown alongside the no-prefetching baseline (NoP). The evaluation of prefetching strategies in an 8-thread DDR5 memory system configuration, indicating minimal performance benefits across different prefetcher configurations. The TiA Prefetcher, maintaining its configuration from the 4-thread scenario, shows a marginal speedup, underscoring the challenges of prefetching effectiveness with increased memory system demands. This Figure reflects the diminishing returns of prefetching at higher thread counts.

Figure 4.13 reveals that the TiA Prefetcher, maintaining the same configuration as in the 4-thread case, emerges as the most effective self adjusting Prefetcher, with a speedup of 0.61%. The speedup of 0.61% achieved by the TiA Prefetcher, is negligible due to delivering basically no performance gaing for the benchmark. Consequently, configurations yielding speedup variations within the range of -2.0% to 2.0% are deemed negligible, echoing the findings at high bandwidth utilization for DDR memory, the benefits of prefetching diminish. Furthermore, the performance degradation observed with the least effective static configuration of the Stream-based Prefetcher, experiencing a -31.66% decline, mirrors the trend noted in the 4-thread scenario. The best performing

static configuration has a speedup of 1.09% and is negligible, underscoring that the adaptive configurations that demonstrate a notable capacity for self-optimization by modulating the prefetch degree in response to the changing system conditions.

Conclusion

Analysis spanning single, four, and eight-threaded executions of the Simple Triad benchmark on an HBM2 node have yielded clear insights into the performance of prefetching strategies. Throughout the simulations, the most voted distance and mean distance showed similar performance. At all levels of threading, adaptive prefetching strategies proved their efficacy, closely rivaling the top results achieved by static configurations. The distinction between static and adaptive approaches became apparent. While static configurations vary within benchmarks, adaptive configurations maintained performance independence from specific workloads. Notably, under conditions without memory contention, both the TiA and Agg Prefetcher showed comparable performance enhancements. However, the performance of the Agg Prefetcher was distinctly better. This divergence suggests underlying differences in how each Prefetcher utilizes adaptive behavior, warranting further exploration of prefetch trigger mechanisms and queue management, especially in light of varying epoch lengths.

The transition from HBM2 to DDR5 in the context of the Simple Triad benchmark shows a performance bottleneck, associated to the inferior bandwidth of DDR5, notwithstanding its superior capacity. Comparative analysis across single-threaded, four-threaded, and eight-threaded simulations illustrate that the inferior bandwidth of DDR5 notably decrease performance at high bandwidth utilization, thereby underscoring the significance of memory type for memory-intensive operations. While HBM2 facilitates greater performance in such benchmarks, due to higher bandwidth, the shift to DDR5, despite its advantages in capacity, introduces constraints that negatively impact performance. Nonetheless, even marginal performance improvements, peaking at 0.61%, are achievable. In particular, static configurations such as the Stream-based Prefetcher, with its distance and degree parameters set at a constant 32, exhibit a significant performance decline, quantified at −31.66%, as illustrated in Figure 4.13. This observation underscores the effectiveness of self-adjustment strategies, which are capable of adapting to diverse memory environments and securing benefits, even under adverse conditions.

Drawing insights from the analysis, it is clear that architectures benefiting from the integration of heterogeneous memory devices, each distinct in their capabilities, greatly enhance their performance through NUMA node-aware programming. By choosing the right memory device, particularly HBM2 in the given architecture, adaptive and balanced prefetching techniques significantly improve workload efficiency. However, selecting

an unsuitable memory device could negate the benefits of prefetching and even lead to decreased performance due to memory contention.

4.3.2. MINIFE SpMV

HBM2 Node with 1 Thread

The superior performance within a single-threaded simulation on HBM2 memory is analyzed in Figure 4.14, where the Stream-based Prefetcher, statically configured with both prefetch degree and distance set to 32, achieves the highest speedup of 45.42%. Closely following this, the Agg Prefetcher, optimized with all available strategies and the most favored distance setting, attains a speedup of 44.73%. A slightly lower speedup of 44.70% is observed when employing a mean voting strategy for the same configuration of the Agg Prefetcher, which is nearly equivalent to the highest performance level. Moreover, various configurations of the Agg Prefetcher demonstrate significant speedups starting from 36.06%, indicating the effectiveness of each implemented balancing strategy. The maximum speedup is realized when adjustments to degree and distance are made simultaneously.

A notable performance gap of 7.69% between the optimal configurations of the TiA Prefetcher and the Agg Prefetcher underscores the advantage of stream-based pattern detection over PC-based detection in scenarios with a minor proportion of temporal locality. Furthermore, the least speedup of 17.21% is recorded with the static configuration of the Stream-based Prefetcher at a degree and distance of two, reaffirming the benefit of adaptive balancing.

This analysis aims to elucidate the performance gap observed between the TiA Prefetcher and the Agg Prefetcher. Figure 4.15 serves as comparison, presenting side-by-side epoch-based analysis plots of the Agg Prefetcher (left) and the TiA Prefetcher (right), under identical configurations. The analysis particularly focuses on the approach of deactivating the degree adjustment feature whilst activating the distance adjustment feature, with a preference for the most frequently selected distance. This specific configuration adjustment, notable in the speedup of the TiA Prefetcher, is visible in Figure 4.14. Given that the degree adjustment is deactivated, it is feasible to omit consideration of the upper right quadrant in each Figure presented in 4.15a and 4.15b. Both prefetcher, Agg and TiA, exhibit a commendable alignment of useful prefetches (indicated in green) with the total number of issued prefetches (shown in orange) in the top left quadrant of each Figure. This alignment is similarly reflected in the coverage metric (also in orange) within the bottom right quadrant of each Figure. A closer inspection reveals that the Agg Prefetcher (4.15a) secures a marginally superior coverage rate compared to the TiA

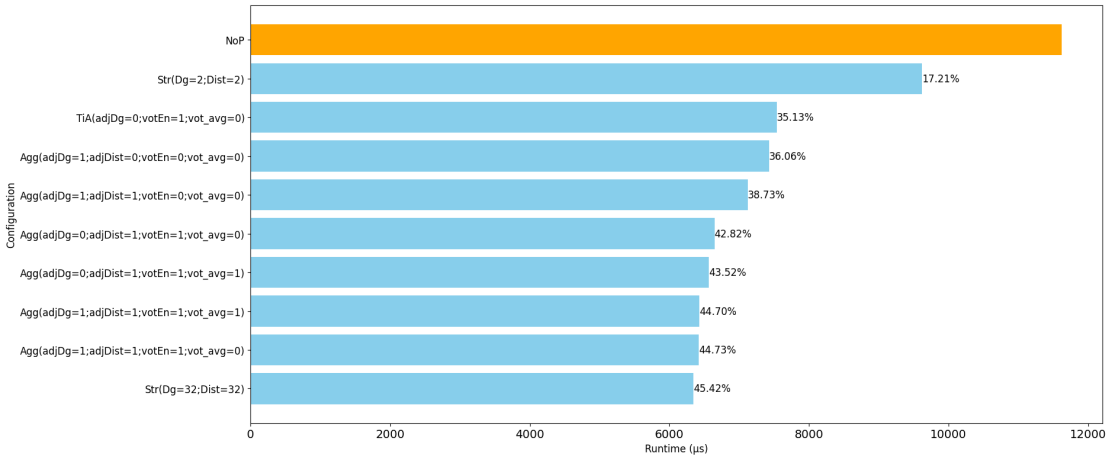
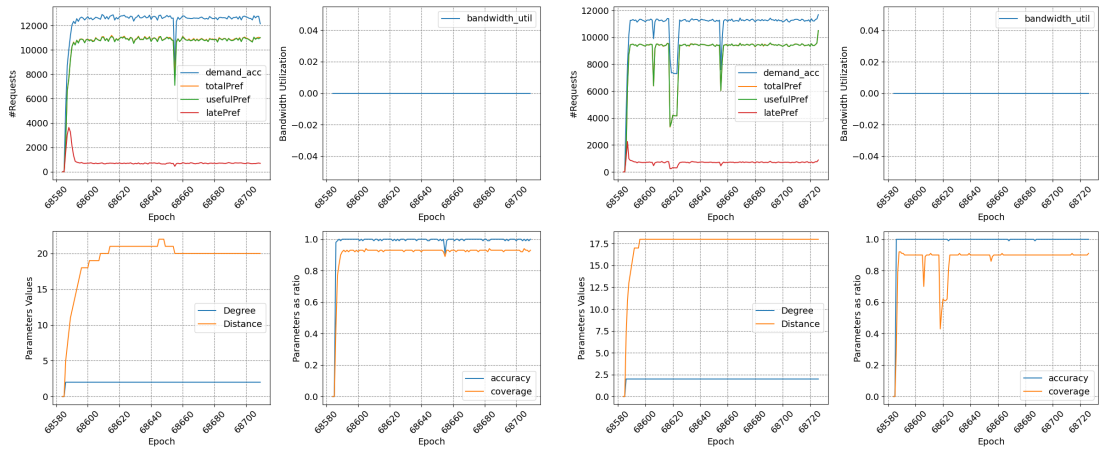


Figure 4.14.: Performance comparison of prefetching strategies for the SpMV benchmark on an HBM2 node with 1 thread. Execution times for the Stream-based (Str), Aggressive (Agg), and Timely Aware Stride (TiA) Prefetchers are shown alongside the no-prefetching baseline (NoP). The Stream-based Prefetcher, with both prefetch degree and distance set to 32, achieves the highest speedup of 45.42%, closely followed by the Agg Prefetcher with a speedup of 44.73%. Employing a mean voting strategy for the Agg Prefetcher results in a similar speedup of 44.70%. These results highlight the effectiveness of various configurations for the Agg Prefetcher and the advantage of Stream-based pattern detection, particularly in scenarios with limited temporal locality. The least speedup of 17.21% with a static configuration underscores the importance of adaptive strategy adjustments.

Prefetcher (4.15b). This differentiation is also noticeable in the upper left quadrant of each Figure, where the Agg Prefetcher records a higher number of useful prefetches, thus yielding closer proximity to demand accesses than the TiA Prefetcher. The basis for this variance is straightforward. The TiA Prefetcher dedicates additional effort to building confidence. Given the presence of temporal locality, this results in the TiA Prefetcher spending more time in this phase. This is also evident in the downward spikes observed in the bottom-right and top-left quadrants of Figure 4.14. In contrast, the update mechanism employed by the Agg Prefetcher is more adeptly suited to these patterns, thereby reducing its processing time. Regarding the applied distance (in orange) at the bottom left quadrant of each Figure, both Prefetcher exhibit similar behavior. The distance managed by the TiA Prefetcher initially increases to 17 before stabilizing at a consistent value of 18. This fluctuation is attributed to the history table possessing less data due to the additional time allocated for confidence building. Conversely, the Agg Prefetcher undergoes more frequent adjustments due to the trigger learning queue discarding the PC, resulting in a higher frequency of updates in terms of votes. This mechanism explains the observed

variance in behavior until the distance remains constant at 20.



(a) (b)

Figure 4.15.: Epoch-based analysis comparing the Agg and TiA Prefetchers under identical configurations with the degree adjustment deactivated and the distance adjustment activated. The analysis showcases the side-by-side performance of the Agg Prefetcher (left) and the TiA Prefetcher (right), emphasizing their approaches to prefetching in a single-threaded HBM2 memory simulation. Notably, the Agg Prefetcher demonstrates a marginally superior coverage rate and a higher number of useful prefetches, reflecting its efficiency in adapting to demand accesses with a more variable distance setting. In contrast, the TiA Prefetcher exhibits a fixed distance, attributed to the additional time spent on building confidence due to temporal locality. This comparison reveals strategic differences in prefetching behavior and their impact on performance, underscoring the importance of optimal distance settings in achieving higher speedups.

HBM2 Node with 4 Threads

Increasing the number of threads to four does not exert sufficient pressure on the HBM2 device to significantly alter the performance observed in single-threaded simulations, as observable in Figure 4.16. Despite this, an increase in speedup is noted, attributable to the utilization of additional cores and, consequently, more Prefetcher. The Agg Prefetcher, when optimized with all available strategies and employing the most favored distance, achieves a significant speedup of 59.84%. This performance is close to the static Stream-based Prefetcher, configured with a prefetch degree and distance of 32, which leads to a speedup of 61.64%.

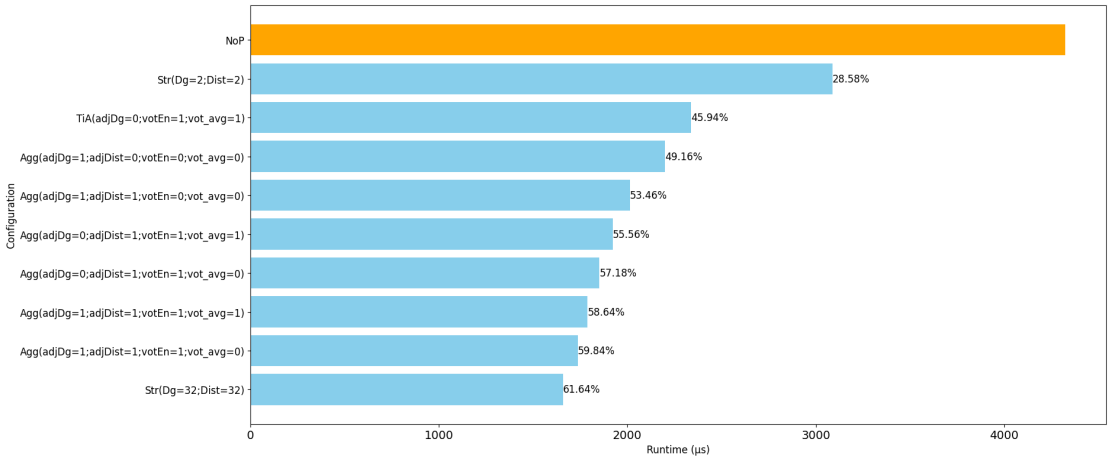


Figure 4.16.: Performance comparison of prefetching strategies for the SpMV benchmark on an HBM2 node with 4 threads. Execution times for the Stream-based (Str), Aggressive (Agg), and Timely Aware Stride (TiA) Prefetchers are shown alongside the no-prefetching baseline (NoP). The Agg Prefetcher, optimized with all available strategies and employing the most favored distance, achieves a notable speedup of 59.84%, closely rivaling the Str Prefetcher configured with a degree and distance of 32, which leads to a speedup of 61.64%. This marginal difference of only 1.80% between the two demonstrates the effectiveness of optimization strategies.

The marginal difference of only 1.80% between these configurations highlights the efficacy of the optimization strategies employed. Furthermore, the TiA Prefetcher, under the same configuration as Agg Prefetcher with degree optimization disabled and distance voting enabled, exhibits a reduced speedup of approximately 9.62%. This discrepancy, as elucidated in the single-threaded evaluation for HBM2, underscores the performance differences attributable to specific prefetcher configurations. Notably, all adjustable Prefetcher outperform the least effective static configuration, with the performance of the Agg Prefetcher nearly matching that of the best static configuration.

HBM2 Node with 8 Threads

Doubling the thread count to eight on the HBM2 device results in a shift in performance dynamics, yet it does not impose a significantly higher computational load as might be expected. This observation is detailed in the forthcoming Figure 4.17, where the performance metrics of various prefetching strategies under an eight-thread configuration are analyzed.

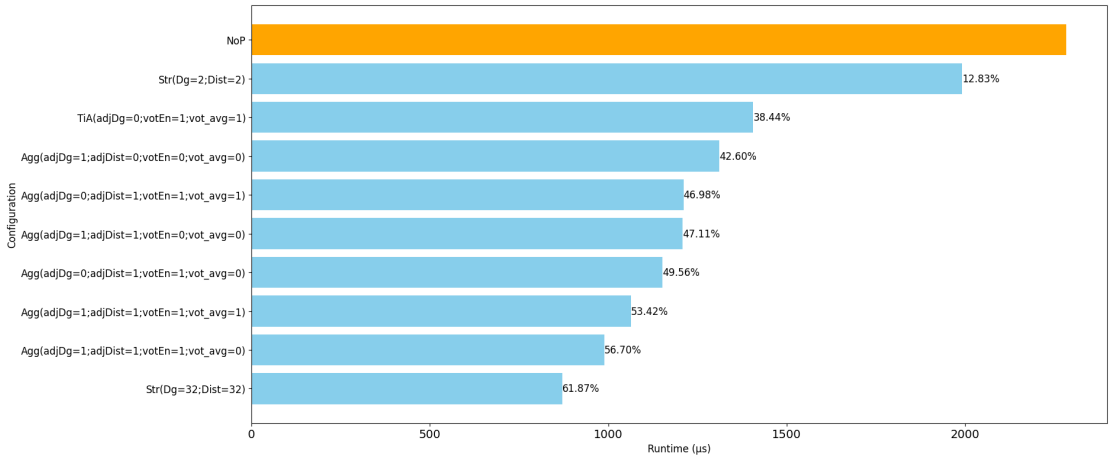


Figure 4.17.: Performance comparison of prefetching strategies for the SpMV benchmark on an HBM2 node with 8 threads. Execution times for the Stream-based (Str), Aggressive (Agg), and Timely Aware Stride (TiA) Prefetchers are shown alongside the no-prefetching baseline (NoP). The Stream-based Prefetcher, with a configuration of degree and distance set to 32, achieves the highest speedup of 61.87%. In contrast, with a configuration of degree and distance set to 2 yields only a 12.83% speedup, indicating the critical role of Prefetcher configuration. The Agg Prefetcher, fully optimized, secures the second-best performance with a 56.70% speedup, demonstrating the efficacy of optimization strategies.

The static Stream-based Prefetcher, with both degree and distance set to 32, retains its

position as the top performer, delivering a remarkable speedup of 61.87%. Conversely, the same Prefetcher, when configured with a degree and distance of 2, markedly underperforms, achieving only a 12.83% speedup. This performance is more than two times slower compared to a four-thread scenario, subtly indicating an elevated pressure on the HBM2 device. However, this pressure is still insufficient to dramatically alter the performance. The contrasting results between the best and worst static configurations underscore the critical importance of configuration adaptability in both static and dynamic contexts. The Agg Prefetcher, fully optimized with all available strategies and employing the most voted distance, secures the second-best performance with a 56.70% speedup. This finding once again highlights the efficacy of the optimization strategies, demonstrating the capability of the Agg Prefetcher to nearly match the performance of the best static configuration through epoch based adjustments. The TiA Prefetcher continues to exhibit commendable performance, albeit slightly trailing behind the Agg Prefetcher. This trend, consistent with observations from single-threaded simulations on HBM2 memory, points to limitations of the TiA Prefetcher.

DDR5 Node with 1 Thread

Switching to DDR5 from HBM for a single-threaded simulation, as depicted in Figure 4.18, yields results that are approximately equivalent to those obtained with the HBM2 device. This similarity in outcomes suggests that the SpMV benchmark requires less memory bandwidth than the Simple Triad benchmark, where the transition from HBM2 to DDR5 exhibited a significant decrease in performance, as evidenced by the reduction in speedup values. Specifically, the least effective configuration of the Stream-based Prefetcher, with both distance and degree set to two, results in a speedup of 12.84%. In contrast, the optimal static configuration, with prefetch degree and distance set to 32, achieves a higher speedup of 41.38%. This is closely followed by the Agg Prefetcher, with all optimization strategies enabled and employing the most favored distance, leading to a speedup of 40.16%. These findings underscore again the memory device awareness and effectiveness of the optimization strategies under conditions of low memory pressure. Previously, only HBM2 was observed to benefit from prefetching and optimization strategies. However, this data indicates that DDR5 can also gain from these strategies, depending on the specific workload.

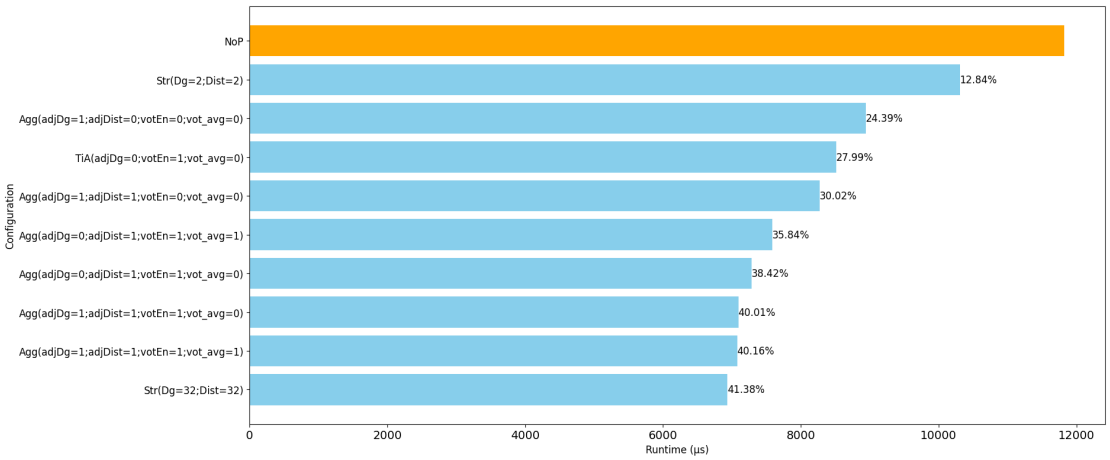


Figure 4.18.: Performance comparison of prefetching strategies for the SpMV benchmark on a DDR5 node with 1 thread. Execution times for the Stream-based (Str), Aggressive (Agg), and Timely Aware Stride (TiA) Prefetchers are shown alongside the no-prefetching baseline (NoP). Results show the performance of DDR5 is comparable to HBM2. Notably, the least effective Stream-based Prefetcher configuration (distance and degree set to two) yields a 12.84% speedup, while the optimal static configuration (prefetch degree and distance set to 32) achieves a 41.38% speedup, closely followed by the Agg Prefetcher at 40.16%. These results underline the effectiveness of optimization strategies under conditions of low memory pressure, highlighting that DDR5, similar to HBM2, benefits from prefetching and optimization strategies depending on the workload.

DDR5 Node with 4 Threads

Increasing the number of threads to four on DDR5 memory naturally leads to an expected rise in demand requests, placing more stress on the DDR5 device. Despite this increased load, results depicted in Figure 4.19 demonstrate that a maximum speedup of 25.77% is achievable.

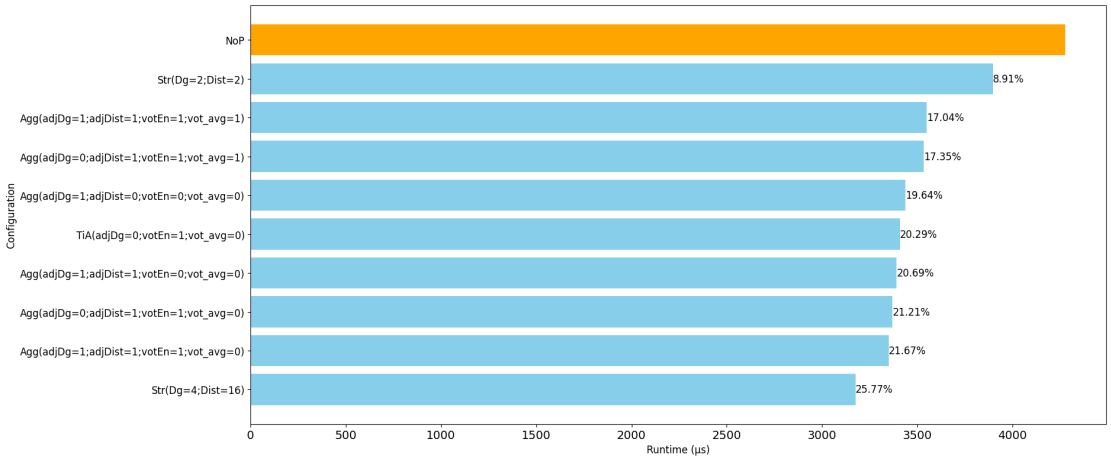


Figure 4.19.: Performance comparison of prefetching strategies for the SpMV benchmark on a DDR5 node with 4 threads. Execution times for the Stream-based (Str), Aggressive (Agg), and Timely Aware Stride (TiA) Prefetchers are shown alongside the no-prefetching baseline (NoP). Raising the thread count to four on DDR5 systems amplifies demand requests, placing greater stress on the memory system. However, this adjustment yields a significant speedup, achieving an increase of up to 25.77%. This performance boost is provided by the static configuring the Stream-based Prefetcher with a degree of four and a distance of 16. The lowest observed speedup is 8.91%, with degree and distance set to two, highlighting the importance of optimal prefetch settings. The Aggressive (Agg) Prefetcher closely follows with a speedup of 21.67%, underlining the effectiveness of tailored optimization strategies for DDR5 under increased workloads.

This is in contrast to the Simple Triad benchmark, where performance gains were not observed and even lead to performance drawbacks. The highest speedup is attained with the Stream-based Prefetcher, configured with a degree of four and a distance of 16. In comparison, the lowest speedup of 8.91% is achieved with the same Prefetcher, but with both degree and distance set to two. This indicates that for the SpMV benchmark, increasing the prefetch degree to utilize available bandwidth and extending the prefetch distance to minimize late prefetches effectively optimizes performance. Furthermore, the Agg Prefetcher, with all optimization strategies enabled and employing the most favored

distance, achieves a closely matched speedup of 21.67%. Similar levels of performance improvement are noted with other configurations of the TiA or Agg Prefetcher, closely aligning with the setup of the Agg Prefetcher when all optimization strategies are enabled and the optimal distance is utilized. This consistency underscores the benefits of fully enabling optimization strategies, yielding the effectiveness of memory device-aware optimizations.

DDR5 Node with 8 Threads

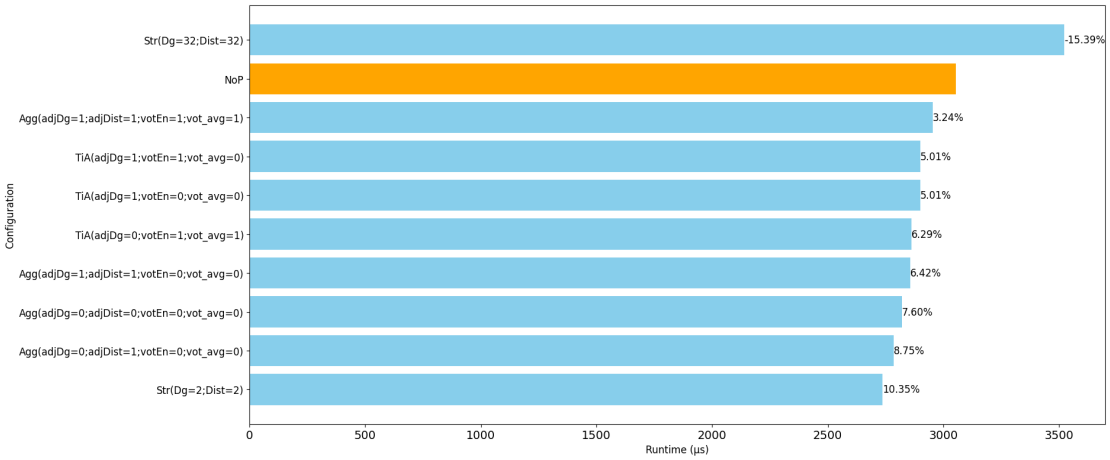


Figure 4.20.: Performance comparison of prefetching strategies for the SpMV benchmark on a DDR5 node with 8 threads. Execution times for the Stream-based (Str), Aggressive (Agg), and Timely Aware Stride (TiA) Prefetchers are shown alongside the no-prefetching baseline (NoP). Increasing the thread count to eight on DDR5 systems results in a performance degradation of up to −15.39%, underscoring the challenges of handling increased demand accesses and the consequential memory latencies. This condition is exacerbated when the Stream-based Prefetcher is configured with a high degree and distance of 32, leading to further memory contention. Conversely, optimal configurations, such as a degree and distance of two for the Stream-based Prefetcher, demonstrate a speedup of 10.35%, suggesting the critical role of prefetcher parameter tuning in high contention scenarios. The performance of the Agg and TiA Prefetchers, under various settings, illustrates the direct impact of prefetch strategies on controlling memory bandwidth and latency, with the adaptability of the TiA Prefetcher pointing towards the need for further refinement in prefetching mechanisms.

Scaling the number of threads to eight introduces the first significant performance degradation of −15.39% for DDR5 memory during the SpMV benchmark, as illustrated in Figure 4.20. This reduction in performance is primarily due to the increased number of demand accesses on the DDR5 memory, which leads to longer latencies until data becomes available. This issue becomes more pronounced when the Stream-based Prefetcher is configured with both degree and distance set to 32, exacerbating memory contention and leading to the observed performance drop. This situation highlights the importance of fine-tuning the prefetch degree in high bandwidth utilization scenarios. The best performance is recorded with the Stream-based Prefetcher configured to a degree and distance of two, achieving a speedup of 10.35%. Similar observations were made in the Simple Triad benchmark, indicating that minimizing the degree and distance in high memory contention scenarios lead to optimal performance. For example, the Agg Prefetcher with all optimization strategies disabled, effectively operating like a Stream-based Prefetcher with degree set to two and distance set to zero, secures the third-best performance with a speedup of 7.60%. However, activating this configuration of the Agg Prefetcher with distance optimization, while employing the most voted distance, yields an improvement of 1.15% and results in a speedup of 8.75%. This outcome demonstrates that distance adjustment remains beneficial in this scenario, utilizing 8 threads to achieve timely data retrieval through prefetching.

Furtheremore, it is observed that more configurations of the TiA Prefetcher are presented in the Figure 4.20 compared to those of the Agg Prefetcher, diverging from simulations involving both HBM2 and DDR5 for the SpMV benchmark, where Agg Prefetcher configurations typically lead the top performance metrics. This variation is due to the TiA Prefetcher issuing fewer prefetches as a result of its confidence-building process, which is advantageous in situations of high memory utilization. Nevertheless, the aim is to improve the adaptability of Prefetcher to allow for as many prefetches as possible to enhance performance gains, suggesting the need for refinements in the adaptive behavior of prefetching mechanisms, particularly under conditions of high bandwidth utilization.

Conclusion

Evaluating MINIFE SpMV across HBM2 and DDR5 under varied thread scenarios shows that MINIFE SpMV utilizes fewer demand requests to the memory device than Simple Triad, benefiting from prefetching. The move from single to multi-threaded executions on HBM2 demonstrates the ability of the Agg and TiA Prefetcher to effectively utilize high-bandwidth memory with optimization strategies, achieving significant speedups. Results indicate the Agg Prefetcher, optimized with all strategies, consistently approaches the performance of the best static configurations across threading levels and memory types. Moreover, the transition to DDR5, despite its lower bandwidth compared to

HBM2, highlights the importance of device-aware optimization strategies. MINIFE SpMV results on DDR5 show that strategic prefetching and optimization can mitigate performance declines, achieving comparable results to HBM2 in certain scenarios.

The shift to eight threads on DDR5 reveals challenges due to increased demand accesses and subsequent memory contention, highlighting the need for improvements in adaptable prefetching strategies in high bandwidth utilization situations. This context underscores the importance of finely tuned degree and distance adjustments. The best performance is recorded with the Stream-based Prefetcher configured to a degree and distance of two. Similar observations were noted in the Simple Triad benchmark, indicating that minimizing both degree and distance in scenarios characterized by high memory contention indeed results in better performance. However, further refinement of optimization strategies for scenarios affected by memory contention would yield significant performance enhancements.

4.3.3. Simple Triad - NUMA Version

In the analysis presented in Section 4.3.1, the performance of the Simple Triad kernel is examined under two NUMA nodes: DDR5 and HBM2, alongside the impact of Prefetcher settings when memory is allocated across those NUMA nodes. This investigation probes the performance impact of distributing vectors a, b, and c, denoted as (a, b, c) where 0 indicates the allocation on the HBM2 node, 1 on the DDR5 node, and x either 0 or 1, across different NUMA nodes. It is essential to highlight that the core affinity is determined based on vector a, which serves as the storage location for the computational result. Furthermore, each vector necessitates a load operation. Additionaly, vector a necessitates an additional write-back instruction due to its role in preserving the result. The findings, delineated in table 4.1, reveal significant gaps in execution times and speedup percentages. These results offer comprehensive insights into the influence of memory type allocations on the kernel performance.

Focusing on the DDR5 (NoP) column within the table 4.1 provides a detailed insight into the performance dynamics under different vector allocation configurations in a NUMA setting. The performance changes, denoted as percentages, reflect the impact of using the DDR5 node without prefetching (NoP) compared to a baseline where all vectors are allocated at the HBM2 node. The data reveals notable performance drawbacks when one or more vectors are assigned to DDR5 without prefetching. Specifically, on the allocation $(0, 0, 1)$ or $(0, 1, 0)$ results in performance drawback of -8.52% and -8.38%. Such reductions occur because vectors allocated on HBM2 must await data from DDR5, elucidating the observable performance drawback. This pattern persists on the allocation $(0, 1, 1)$, which amplifies the demand requests on the DDR5 device and

Vector a	Vector b	Vector c	DDR5 (NoP)	Agg (All)	TiA (All)	Agg is (DDR5)	TiA (DDR5)	Agg (HBM2)	TiA (HBM2)
0	0	0	0.00%	19.73%	19.40%	0.06%	-0.02%	19.79%	19.50%
0	0	1	-8.52%	7.45%	4.21%	-0.35%	-0.11%	9.58%	5.69%
0	1	0	-8.38%	4.21%	2.01%	1.09%	1.55%	6.66%	1.95%
0	1	1	-16.22%	0.14%	-4.36%	-1.83%	-4.31%	2.72%	-0.36%
1	0	0	-27.11%	6.64%	4.13%	4.56%	0.09%	3.82%	4.14%
1	0	1	-53.23%	3.25%	0.86%	5.20%	2.68%	0.90%	0.68%
1	1	0	-54.37%	0.24%	-1.64%	5.80%	3.41%	-0.16%	-0.39%
1	1	1	-57.98%	1.88%	-0.49%	1.83%	-0.32%	0.06%	0.06%

Table 4.1.: This table presents the simulation outcomes for the Simple Triad benchmark, comparing the efficiency of various NUMA memory allocation strategies. Here, HBM2 and DDR5 are encoded as 0 and 1, respectively. The column "DDR5 (NoP)" illustrates the speedup introduced by allocating vectors on DDR5 against HBM2 $(0, 0, 0)$. Furthermore, the columns labeled "Agg" and "TiA", associated with the corresponding Prefetcher, provide detailed insights into the performance variations, both improvements and declines, relative to the NoP baseline across different NUMA memory allocations. Here, "all" signifies the activation of prefetching on both DDR and HBM memories, while the remaining categories specify the devices on which prefetching is enabled.

leads to memory contention. Under these conditions, vector a is waiting for data from both vectors for its writeback operation. This is observable by the performance drawback of -16.22%. A significant performance drawback is noted when vector a is allocated at DDR5. In the initial scenario $(1, 0, 0)$, contrary to expectations which might suggest a mitigation of performance impact relative to previously discussed configurations, this specific arrangement results in a performance drawback of -27.11%. This outcome is largely attributed to the necessity for the CPU to complete the load operation for vector a in DDR5. Moreover, the shift in core affinity closer to DDR5 nodes within the mesh topology, as shown in Figure 3.2, makes the situation worse by introducing additional latency to the data transfer from the HBM2 node to the CPU core. Insights from prior analysis reveal significant performance drawbacks for configurations $(1, 0, 1)$ and $(1, 1, 0)$, with performance degradations of -53.23% and -54.37%, respectively, underscoring the substantial effect of DDR5 node allocation on system efficiency.

The performance outcomes highlight the limitations associated with the NUMA node allocation, specifically HBM2 and DDR5, for this benchmark. The Subsequent analysis will focus on the potential effects of prefetching with the introduced optimization strategies. Hereby, distance and degree optimization strategies are enabled, employing the most voted distance for both the TiA and Agg Prefetcher.

In scenarios where prefetching is activated for both HBM2 and DDR5 memory, with the allocation $(0, 0, 0)$, notable performance improvements are observed, with speedup values reaching 19.40% for the TiA Prefetcher and 19.73% for the Agg Prefetcher. Conversely,

allocating $(0, 0, 1)$ or $(0, 1, 0)$ results in a decrease in speedups, highlighting the memory contention issues for DDR5. This is further elucidated by examining the configurations where prefetching is individually enabled for DDR5 and HBM2. Specifically, prefetching solely on DDR5 demonstrates negligible impact on performance, with speedup values ranging between -4.31% and 1.55%. In contrast, when prefetching is exclusively applied to HBM2, a substantial increase in speedup is noted, with values spanning from -0.36% to 9.58%. Given the observed impacts of individual prefetching strategies, it becomes evident that DDR5 memory acts as the primary bottleneck, accounting for the observed decline in performance when prefetching is employed across both memory types (in the "all" case).

When allocating $(1, 0, 0)$, a noticeable improvement in performance is observed. This enhancement can be attributed to the adjustment in core affinity, which shifts closer to the DDR5 channel, optimizing the latency for data retrieval from vector a. This improvement is consistent across all prefetching scenarios. Specifically, when prefetching is applied to both HBM2 and DDR5, the observed speedup is 6.64% for the Agg Prefetcher and 4.13% for the TiA Prefetcher. Prefetching exclusively on DDR5 yields a speedup of 4.56% for Agg and a marginal speedup of 0.09% for TiA. Conversely, when prefetching is limited to HBM2, the speedup recorded is 3.82% for Agg and 4.14% for TiA. These findings indicate that aligning core affinity closer to the DDR5 channel mitigates data retrieval latency for vector a to some extent. Nevertheless, the overall impact of prefetching on performance enhancement remains modest. Importantly, prefetching solely on HBM2 proves beneficial for the data retrieval from vectors b and c, which now incur increased latency due to the proximity of the core to the DDR5 channel. This elucidates the performance speedups observed when employing prefetching across both HBM2 and DDR5 memory types. The pattern holds true when allocating $(1, 0, 1)$ or $(1, 1, 0)$. However, as the number of vectors allocated on DDR5 increases, the positive impact observed on HBM2 diminishes, and performance becomes predominantly reliant on DDR5. Concurrently, an escalation in the number of demand requests on DDR5 memory exacerbates stress, leading to increased memory contention.

To prove these insights, the configuration is modified to increase the number of DDR5 channels from one to two for the same simulation. Results, as illustrated in Table 4.2, with prefetching enabled for both HBM2 and DDR5, reveal a significant improvement in speedup values, particularly evident in the "DDR5 (NoP)" column. This adjustment showcases a considerable enhancement in performance compared to the scenario with a single DDR5 channel. Notably, the performance speedup for the allocation $(0, 0, 1)$ or $(0, 1, 0)$ improved from -8.52% to -2.96% and from -8.38% to -3.19%, respectively. Allocating $(0, 1, 1)$ exhibited an even more significant improvement, from -16.22% to -1.83%, suggesting the elimination of memory contention. However, due to the

proximity of the core to HBM2, some latency in data retrieval for vectors allocated on DDR5 persists. Adjusting the core affinity closer to the DDR5 channels resulted in a significant performance improvement, with speedup values shifting from -27.11% to 5.58%. This enhancement can be attributed to the differential bandwidth constraints associated with specific memory allocations. For instance, the allocation $(1, 0, 0)$ is primarily limited by the DDR read bandwidth, which stands at $8.76GB/s$. Conversely, the baseline allocation $(0, 0, 0)$ is constrained by the HBM write bandwidth, which is $8.06GB/s$. However, allocating additional vectors on DDR5 still lead to performance drawbacks. Specifically, in the context of a single DDR5 channel scenario, performance degradation was observed at -54.37% and -57.98%. When the configuration is adjusted to include two DDR5 channels, the performance degradation improves but remains significant, recorded at -23.70% and -23.27%, respectively. This shift underscores a significant performance bottleneck, further exacerbated by the increased demand on DDR5 memory. Nonetheless, across all scenarios, the implementation of prefetching strategies, along with their respective optimization techniques, exerts a substantial positive impact on system performance. This observation attests to the efficacy of these strategies, indicating their potential to significantly enhance performance in environments that are not facing memory contention.

Vector a	Vector b	Vector c	DDR5 (NoP)	Agg (All)	TiA (All)
0	0	0	0.00%	19.44%	19.43%
0	0	1	-2.96%	19.87%	20.00%
0	1	0	-3.19%	21.65 %	11.94%
0	1	1	-1.83%	16.40%	9.43%
1	0	0	5.58%	13.09%	11.06%
1	0	1	-23.28%	14.73%	15.16%
1	1	0	-23.70%	7.49%	10.59%
1	1	1	-23.27%	6.92%	4.50%

Table 4.2.: This table presents the simulation outcomes for the Simple Triad benchmark, comparing the effectiveness of different NUMA memory allocation strategies while highlighting the impact of doubling the DDR5 channels from one to two. Here, HBM2 and DDR5 are encoded as 0 and 1, respectively. The column "DDR5 (NoP)" illustrates the Speedup introduced by allocating vectors on DDR5 memory against HBM2 $(0, 0, 0)$. Furthermore, the columns labeled "Agg" and "TiA", associated with the corresponding Prefetcher, provide detailed insights into the performance variations, both improvements and declines, relative to the NoP baseline across different NUMA memory allocations. Here, "all" signifies the activation of prefetching on both DDR and HBM memories.

Conclusion

The detailed examination of the performance from the Simple Triad benchmark in NUMA configurations, focusing on DDR5 and HBM2 memory nodes, reveals dynamics that are crucial for optimizing computational tasks. It outlines the impact of memory allocation across different NUMA nodes and the efficacy of prefetching strategies, leading to several insights.

Firstly, the analysis underscores that the choice of memory allocation significantly affects the performance, with the allocation of vectors on DDR5 without prefetching notably reducing efficiency. This finding show the importance of HBM2 for highly spartial locality benchmarks, maintaining high performance.

Secondly, the impact of prefetching on HBM2 and DDR5 memory demonstrates that performance enhancements depend significantly on the NUMA node allocation. When all vectors are allocated on the HBM node, speedups of up to 19.73% are achievable. Prefetching on HBM2 typically outperforms prefetching on DDR5, signaling DDR5 as a bottleneck in the system. Enhancing DDR5 by adding more channels significantly improves outcomes, mitigating the adverse effects of contention. This improvement underscores the effectiveness of the implemented Prefetcher together with the introduced optimization strategies. Overall the Agg Prefetcher demonstrates superior performance than the TiA Prefetcher under the same configuration. This advantage is attributed to learning patterns that must not have a PC due to its pattern learning algorithm. In the evaluation, a single thread was sufficient to induce memory contention in the DDR5 device. Increasing the threads is not necessary and is going to mirror the behavior of the simple triad benchmark from section 4.3.1.

4.3.4. Resource Estimation

The resource estimation focuses on the optimization strategies discussed in Section 3.6. These strategies are enhancing the capabilities of existing state-of-the-art prefetchers, which, although not initially developed within the Gem5 simulation environment, are implemented in actual hardware.

Degree Optimization

For the implementation of degree optimization, it is essential to consider following parameters. Firstly, the number of NUMA nodes, with HBM2 and DDR5 this accounts to two. Secondly, the resource estimation of the epoch counter, bandwidth and accuracy

metrics. The optimization degree is assessed at the conclusion of each epoch, based on a detailed analysis of statistics, bandwidth, and accuracy metrics.

Epoch Counter: In the current configuration, an epoch length of 128,000 cycles is defined, necessitating a 17-bit counter for the representation. This counter resets to zero at the conclusion of each epoch and increments by one after every clock cycle. Consequently, the total bit requirement for the epoch counter is represented as $Size_{epochCount} = 17\text{-}bit$.

Bandwidth Measurement: Bandwidth is measured by initiating a counter that records the number of cycles required for data to arrive. This counter ceases upon deallocation in the TBE. As depicted in Figure 3.6 within Section 3.5, the maximum observed values were 750 cycles for HBM2 and 120 cycles for DDR5. To ascertain the average, all measurements are aggregated, and the sum is preserved until the end of an epoch. Additionally, the total number of measurements is counted. To prevent overflows and account for more latencies, a 32-bit design for each counter, one to compute and retain the average value and another to record the measurement count, are defined. Based on these averages, the logic subsequently classifies latency into high, medium, or low categories, for which two 10-bit registers suffice for setting medium and high thresholds. Consequently, considering that each TBE entry requires 84 bits, the storage calculation is performed for each NUMA node, denoted as N_{NUMA}. Hence, the total storage requirement, $Size_{BW}$, is determined by the formula $Size_{BW} = \text{TBE\_SIZE} \times N_{NUMA} \times 84\text{-}bit$.

Accuracy Measurement: As delineated in Section 3.5, accuracy is determined by the ratio of useful prefetches to total prefetches, with the highest value for sent prefetches approximated at 40,000, indicating that 16 bits are sufficient for representation.

Useful prefetches are defined by the sum of timely and late prefetches. To differentiate between data that have been requested by a prefetcher and by other means a bit is added to each entry in the cache and MSHR. Upon a cache hit involving marked data, the corresponding timely counter is incremented. Conversely, when a demand request encounters a hit within the MSHR, an entity that enhances cache miss handling by monitoring outstanding requests and facilitating access coalescence, as incorporated within the TBE of the provided simulation platform [15], the late prefetch counter is incremented if the prefetch marker is active.

Accuracy metrics are stored in a 32-bit float, necessitating three 16-bit counters for calculation. For categorization into high, medium, or low, two additional 32-bit floats, for medium and high thresholds, are required. Therefore, accounting into 144 bits. The calculation of the Accuracy metric is applied per NUMA node, denoted as N_{NUMA}. The bit encoding for both *L2D_SIZE* and *TBE_SIZE*, required to identify prefetches, is considered once for each parameter. This approach results in the total storage requirement, $Size_{Acc}$, being expressed by the equation $Size_{Acc} = TBE\_SIZE + L2D\_SIZE + N_{NUMA} \times 144\text{-}bit$.

Degree Adjustment: The maximal allowable degree is capped at 32, necessitating six bits for representation. Degree adjustment values, either to increment or decrement slowly or rapidly, influence the bit size of those registers, with two 4-bit registers deemed adequate for the actions defined in Table 3.4. Therefore, it is accounted for 14 bits. In this work, the adjustment of degree is conducted on a per NUMA node basis, represented as N_{NUMA}. Consequently, the total storage requirement, $Size_{degreeAdj}$, can be formulated as $Size_{degreeAdj} = N_{NUMA} \times 14\text{-}bit$.

After the end of an epoch, the decision making for the degree is executed. The calculations are small problems that do not affect the system efficiency and can be done in existing hardware. However, one bit per each entry in the MSHR and L2 data cache is needed.

Total Memory Estimation: The overall requirement bits for the memory can be expressed by the sum of $Size_{epochCount}$, $Size_{BW}$, $Size_{Acc}$ and $Size_{degreeAdj}$. And is expressed by $Size_{DegOpt}$ in the formula below:

$$\begin{aligned} Size_{DegOpt} &= Size_{epochCount} + Size_{BW} + Size_{Acc} + Size_{degreeAdj} \\ &= 17\text{-}bit + TBE\_SIZE \times N_{NUMA} \times 84\text{-}bit + TBE\_SIZE + L2D\_SIZE \\ &\quad + N_{NUMA} \times 144\text{-}bit + N_{NUMA} \times 14\text{-}bit \\ &= 17\text{-}bit + TBE\_SIZE \times N_{NUMA} \times 84\text{-}bit + TBE\_SIZE + L2D\_SIZE \\ &\quad + N_{NUMA} \times 158\text{-}bit \end{aligned}$$

Lookahead Optimization

The lookahead optimization technique is pivotal for enhancing the timeliness for Prefetcher, incorporating the following parameters. It employs a queue for learning new triggers, with each entry comprising two 64-bit address registers. Additionally, a lookahead component, typically implemented and addressed in state of the art Prefetcher as distance, is utilized. In scenarios where the lookahead register is absent, its maximum value is depending upon the page size. For a 4096 byte page, a maximum lookahead of 63 is calculateable, necessitating a 6-bit representation.

A key functionality is the employment of a voting mechanism. This mechanism determines the prefetch distance by either computing the average of all votes or selecting the most frequently occurring distance. This process requires only a single bit for decision-making. It relies on 63 arrays, each employing a 32-bit counter, to vote for corresponding lookahead values. Additionally, to facilitate the computation of averages, two 32-bit counters are employed to aggregate the values of all lookaheads and their respective counts.

The configuration of the queue size is adjustable, with the present setup specifying 32 entries per queue. Each entry within the history table, which accommodates distinct sizes

for any Prefetcher such as Agg and TiA, is allocated its own queue, culminating in a total memory requirement expressed as $N \times 32 \times 2 \times 64\text{-}bit = N \times 4096\text{-}bit$, where N represents the size of the history table. The memory footprint attributed to the voting mechanism, for each NUMA node, is determined as $N_{NUMA} \times (63 \times 32 + 64)\text{-}bit = N_{NUMA} \times 2080\text{-}bit$. Thus, the overarching memory footprint is defined by $N \times 4096\text{-}bit + N_{NUMA} \times 2080\text{-}bit$. Addressing the potential for escalated hardware costs with large N, it is proposed for future exploration to reduce this impact by not allocating queues to all table entries. Instead, leveraging a dynamic allocation strategy for active entries could significantly decrease the overall memory footprint, making the system more scalable and cost-effective.

For operational integration, the implementation of notification functionalities, specifically `notifyFill`, `notifyLate`, and `notifyTimely`, within the cache controller is necessary. These functions are instrumental in managing the queue effectively by signaling cache events, thereby enabling the computation of new triggers based on cache hits or misses. The process involves utilizing an adder to subtract two addresses to derive a new distance, followed by a right shift by the cache block size to divide through the cache block size, thus yielding the lookahead value. A 6-bit register is requisite for storing the lookahead value. Furthermore, a basic gate logic is required for managing the operations of the queue, including entry, read, and write functions. Additionally, a comparator is necessary for conducting searches within the queue. The principal implementation effort focuses on the modification of existing cache controllers to accommodate these enhancements together with the requiered memory.

5. Related Work

The quest to overcome the "Memory Wall" and enhance prefetching mechanisms has led to substantial research within the realm of high-performance computing. This chapter delves into seminal works that have shaped current understanding and practices in prefetching strategies, memory management, and the utilization of novel architectures to optimize data access and processing efficiency.

5.1. Berti: an Accurate Local-Delta Data Prefetcher

In the study [26] the authors present a timely local-delta Prefetcher, attached to the L1 data cache, with a focus on the difference between successive cache line addresses initiated by the same instruction. This methodology is similar to the history table used in the PC-based Stride Prefetcher and aims to refine prefetch decisions, thereby elevating system efficiency and minimizing energy consumption.

What sets Berti apart is the differentiation between deltas and strides. Deltas are characterized as the variances in cache line addresses accessed in sequence by the same IP (Instruction Pointer), providing an in-depth insight into memory access patterns. Conversely, strides are defined by the differences between addresses of consecutive load accesses, irrespective of their chronological order or the initiating instruction. Berti incorporates latency measurements to issue prefetch requests that are both timely and essential. This is achieved by meticulously monitoring access history to pinpoint deltas that result in successful prefetches, thereby honing the proficiency of the system in preloading data prior to its request, which in turn boosts overall system performance.

The optimization strategy deployed in this thesis, as discussed in Section 3.6, contrasts with other methodologies by adapting the prefetch degree to utilize system bandwidth more effectively during periods of low utilization. This approach aims to preload additional data into the cache, thereby enhancing accuracy and coverage. Berti, however, strives to load only those addresses deemed crucial for computation, a design choice reflecting a preference for maximizing bandwidth usage or prioritizing relevant requests to bolster energy efficiency. The methodology for learning timely prefetches in Berti parallels the approach taken in this work, with both employing a history table to discover

new prefetch triggers. While Berti utilizes the latency between a request and a write to identify suitable deltas, this work learns triggers by navigating through the queue to find a physical address within the same page as the prefetch trigger, thereby establishing the prefetch distance. Additionally, voting mechanisms are employed to determine the most frequently selected or mean distance for the Prefetcher to utilize.

Working with virtual memory over physical memory, Berti is mirroring the approach in this work. This strategy allows for prefetch requests irrespective of the memory hierarchy and therefore effectively implement cross-page prefetching and recognize larger deltas across page boundaries. However, this thesis predominantly focuses on physical addresses, noting that while virtual addresses facilitate the identification of streams due to the virtual address space, they also incur the overhead of address translation, potentially adding latency when prefetch predictions are incorrect.

It would be intriguing to implement Berti within the Gem5 simulator to assess its performance on the L2 data cache. Furthermore, exploring the potential of combining the methodologies of both Berti and this work for timely prefetches presents an interesting avenue for future investigation.

5.2. T-SKID: Predicting When to Prefetch Separately from Address Prediction

The research presented in [23] decouples the timing of prefetch operations from their address predictions. Unlike conventional Prefetcher that primarily predict the next cache line addresses based on observed strides, T-SKID incorporates a layer of temporal prediction. This innovative strategy aims to optimize the timing of prefetches, ensuring data is fetched not merely in anticipation of future requests but precisely at the most opportune moment to minimize cache evictions before the data's utilization.

T-SKID is utilizing the temporal correlation between load instruction Program Counters, supported by a sophisticated ensemble of four tables: the Address Prediction Table, the Target Table, the Insight Prefetch Table, and the Recent Request PC Queue. These components collectively enable T-SKID to pinpoint the optimal timing for prefetching through meticulous analysis of memory access sequences and timings.

- **Address Prediction Table:** Identifying which data to prefetch by recording stride patterns and the last accessed addresses, facilitating the prediction of future cache line requests based on historical access behaviors.
- **Target Table:** Links trigger PCs to their corresponding target PCs, effectively separating the timing of prefetches from their addresses. This enables T-SKID

to determine the most opportune moments for prefetching, associating specific instruction sequences with optimal prefetch timings.

- **Insight Prefetch Table:** Tracks issued prefetches that are pending cache fill, recording the initiating PC and the prefetch address. This table is integral in assessing the timing and effectiveness of prefetch operations.

- **Recent Request PC Queue:** Logs PCs that have recently triggered successful prefetches, providing a historical context that helps refine timing predictions for future prefetches based on recent outcomes.

Methodical observation and analysis of memory access patterns allow T-SKID to accurately predict when future accesses are likely to occur. By monitoring cache miss and fill events, T-SKID learns the optimal timing for prefetches, dynamically adjusting prefetch timings based on the immediacy of data usage post-cache fill. This results in a fine-tuned prefetching process that ensures data is fetched at neither too early to avoid unnecessary cache evictions nor too late to effectively conceal memory latency. The dynamic adjustment strategy, informed by learned intervals between prefetch requests and actual data usage. Exclusively operational at the L1 data cache level, T-SKID exerts a direct influence on data retrieval processes, significantly reducing cache misses. However, the study does not explicitly detail the use of virtual or physical addresses in its prefetching decisions.

The methodology introduced is akin to the Timely Aware Stride Prefetcher, an expansion of the PC-based Stride Prefetcher. This work differentiates by employing three tables for the Timely Aware Stride Prefetcher—PC Tables, PC Entry Tables, and the Fill-done Queue. Conversely, the Aggressive Prefetcher, an advancement of the Stream-based Prefetcher, utilizes two tables: the Stream Entry Table and the Fill-done Queue. Storing recent prefetch addresses in the Fill-done Queue enables the learning of new distances for enhanced timeliness in address prediction. Meanwhile, the Entry Tables are central to identifying accurate access patterns, thereby improving the precision of prefetching strategies. Unlike T-SKID, which incorporates a latency counter for timing prediction, this methodology opts for a more straightforward approach by examining the queue for addresses located within the same page. By evaluating the differences between these addresses, a new distance is calculated and then applied to the prefetching formula. This method simplifies the adjustment of prefetch distances, aiming to refine both the timing and accuracy of prefetch operations.

5.3. Classifying Memory Access Patterns for Prefetching

The work [3] introduces a software methodology for classifying memory access patterns to enhance prefetching techniques. By analyzing dataflow information, this methodology identifies diverse access patterns including reuse, strides, reference locality, and complex address generation across various workload kernels. This approach facilitates the computation of the next address for the majority of top-missing instructions, leveraging a dataflow-based analysis to classify significant cache misses. It enables precise reasoning about the necessary computations for address generation and the optimal timing for prefetching. This methodology contrasts sharply with traditional hardware Prefetcher, often constrained to simplistic next-line and stride designs, by showcasing a broader spectrum of access patterns. The application of this classification to memory-intensive applications illustrates its potential to significantly improve prefetching effectiveness, offering insights into reclaiming performance lost to memory access stalls.

Despite its comprehensive analysis framework, the methodology acknowledges the challenges in fully understanding complex memory access patterns, given the intricate data dependency analysis required for all load instructions causing cache misses. The integration of this software prefetching approach with the Prefetcher investigated in this work could provide further advancements in prefetching strategies, potentially unlocking new levels of performance optimization.

5.4. Feedback-Directed Prefetching

The work [33] introduces a feedback mechanism to dynamically enhance the functionality of hardware prefetching. This innovative method analyzes prefetch accuracy, timeliness, and cache pollution in execution time. This allows for immediate adjustments to the aggressiveness of Prefetcher and the cache insertion policies based on current performance.

Dynamically adjusting the placement of prefetched blocks in the cache, based on the behavior observed from the Prefetcher, is beyond the conventional MRU insertion. This approach selectively utilizes different cache positions, such as LRU or LRU-4, to diminish cache pollution and boost prefetching efficiency.

The Stream-based Prefetcher at the heart of this study observes multiple access streams to anticipate future data requests, ensuring data is preloaded into the cache efficiently. While the paper does not specify the memory addressing mode, it is implied that physical addresses are utilized, consistent with standard hardware prefetching practices, to avoid

the complications associated with address translation. The Stream-based Prefetcher in this thesis, introduced in Section 3.3 was inspired by this work and functions similarly. A significant aspect of this approach is the windowing technique, which dynamically tracks memory accesses within a specific range. This range is defined from a start address A to an end address P, with the maximum distance between A and P determined by the distance. When a demand L2 cache access is detected within this region, the Prefetcher initiates prefetch requests for subsequent blocks, ranging from $P + 1$ to $P + N$, where N represents the degree. The tracking window advances by N blocks to encompass the region between $A + N$ and $P + N$. However, the approach has been refined with several enhancements, including maintaining sequential integrity, ensuring a prefetch P does not exceed page boundaries, and implementing a window "warming-up" approach to gain deeper insights into the observed patterns. These improvements contribute to a more robust and effective prefetching strategy.

Crucial metrics influencing the feedback-directed adjustments include:

- **Prefetch Accuracy:** Indicates the success rate of prefetches in being utilized before cache eviction, reflecting the prediction accuracy.
- **Prefetch Lateness:** Measures the timeliness of prefetches to ensure data arrives neither too early nor too late.
- **Prefetcher-Generated Cache Pollution:** Assesses the negative impact of prefetches on cache health, especially the displacement of critical data by unnecessary prefetches.

The foundation of this approach is a Sampling-based feedback collection mechanism, which gathers prefetching behavior data over predetermined intervals to drive strategic modifications. The method outlined in [33] utilizes a tabular decision-making process to alter the aggressiveness of Prefetcher, prioritizing metrics such as prefetch accuracy, lateness, and cache pollution, while not accounting for bandwidth considerations. Moreover, [33] distinguishes itself by employing this decision table to specifically adjust prefetch distance and degree, marking a significant difference in the approach to optimizing prefetch efficiency. This is similar to the degree adjustment within this work, Section 3.6, but does not consider the bandwidth utilization. However, the tabular decision-making process contrasts with the strategy that leverages a queuing mechanism to improve the timeliness of prefetches through historical analysis and adjusts the prefetch distance, like explained in Section 3.6.

The implementation of feedback-directed prefetching demonstrates substantial performance enhancements and decreases in memory bandwidth consumption across various benchmarks, underscoring the effectiveness of feedback mechanisms in refining hardware prefetching strategies. Such advancements herald the development of more sophisticated and flexible prefetching techniques in upcoming processor architectures. Additionally,

exploring dynamic replacement strategies for caches, in conjunction with evaluating the cache pollution metric, presents a promising avenue for further refining the optimization strategies delineated in this work.

5.5. Access Map Pattern Matching

The paper [19] presents an interesting approach to hardware data prefetching, introducing AMPM (Access Map Pattern Matching). AMPM aims to overcome the limitations of existing Prefetcher by utilizing a memory access map and hardware pattern matching logic to accurately predict and prefetch data into cache memory in advance. This technique is designed to be robust against the modifications and alterations in memory address sequences caused by aggressive optimizations like out-of-order execution. The AMPM Prefetcher divides the memory address space into fixed-size regions, mapping each cache line within these regions to entries in a bitmap-like data structure. This allows for the detection of memory access patterns from the structure, independent of the memory access order or instruction addresses, thus achieving high performance even in aggressively optimized environments.

The research evaluates the AMPM Prefetcher through cycle-accurate simulations using memory-intensive benchmarks. It demonstrates that AMPM significantly improves prefetch coverage and IPC (Instructions Per Cycle) compared to state-of-the-art Prefetcher, showing an increase in IPC by 32.4%. AMPM contrasts with other prefetching methodologies by focusing on identifying all possible strides at a time through pattern matching, leading to high prefetch coverage and low latency in address prediction. The hardware cost for the prefetching mechanism is reasonable, using basic arithmetic units such as adders and shifters for the pattern matching logic.

Exploring how the optimization strategies, discussed in Section 3.6, of this work perform in conjunction with AMPM logic presents an intriguing research direction.

5.6. Clustering Modes in Knights Landing Processors

The study by [36] delivers an in-depth analysis of clustering modes in Intel's 2nd generation Xeon Phi processors Knights Landing (KNL). It explores the optimization of the on-die mesh interconnect via clustering modes such as all-to-all, quadrant, hemisphere, SNC-4 (Sub-NUMA Clustering-4), and SNC-2. The work emphasizes the importance of making applications NUMA-aware to fully exploit these modes for enhanced performance. It showcases notable enhancements in their memory subsystem. These advancements are primarily due to the integration of SNC modes, which optimize latency and bandwidth by dividing the processor into several NUMA domains. HBM is designed to provide high bandwidth but at a lower capacity, making it ideal for rapidly processing critical data. By contrast, DDR memory, offering a larger storage capacity suitable for less frequently accessed data, comes with the trade-off of lower bandwidth. This combination offers a balanced approach to memory management, crucial for high-performance computing applications.

This comprehensive guide is presented as an essential resource for developers seeking to optimize KNL processor applications. By following the recommended strategies for memory allocation, thread pinning, and selecting the appropriate clustering mode, developers can significantly enhance application performance, especially in scenarios demanding high parallelism and memory bandwidth. Efficient use of HBM and DDR, along with NUMA-aware programming, significantly enhances the performance of memory-intensive tasks in the KNL architecture [36]. However, a comparable implementation for ARM-based devices is currently lacking. It is essential to note that the architecture discussed in Section 3.2 also utilizes the benefits of SNC modes for an Arm-based chip.

6. Conclusions and Future Directions

In this thesis, the exploration and evaluation of optimization strategies for prefetching, focusing on Stream and PC-based stride Prefetcher, have been pivotal in addressing latency challenges in HPC environments. The core of these strategies lies in the dynamic adjustment of prefetching parameters, degree and distance, to match system demands, thus optimizing system performance by mitigating latency through strategic prefetching. The adaptability of these strategies ensures an efficient balance between aggressiveness in prefetching during high and low demand request periods, alongside the refinement of prefetch distance to reduce late prefetches and effectively conceal latency.

The findings demonstrate that adaptive prefetching significantly improves system performance by achieving a near-optimal balance in prefetching parameters, leading to substantial performance gains, particularly when leveraging the high bandwidth of HBM2 memory. However, the performance impact varies with memory type. While DDR5 introduces challenges for memory-intensive applications due to increased latency, less demanding applications still benefit from the adaptive approach. The research underscores the necessity for sophisticated prefetching strategies, particularly for DDR5 memory, to manage memory contention efficiently. The strategic approach, informed by insights from benchmarks, recommends maintaining a consistent minimum prefetching degree and distance in high bandwidth scenarios, while allowing for adaptability in other contexts, to optimize performance benefits. This approach merits further investigation to fully understand its impact on system efficiency and performance optimization.

This work lays the groundwork for future research in prefetching optimization, suggesting further exploration in adaptive strategies across heterogeneous memory architectures and computational workloads. The adaptability and performance gains observed highlight the potential for advanced prefetching techniques to significantly contribute to the efficiency and effectiveness of high-performance computing systems.

The implementation of prefetching data across different cache locations, coupled with the consideration of metrics such as cache pollution as discussed in Section 5.4, could significantly enhance the efficacy of heterogeneous memory-aware prefetching strategies. Incorporating the capability to prefetch across page boundaries, as demonstrated by [22], is anticipated to improve coverage, thereby leading to more precise prefetching than current mechanisms offer. Additionally, incorporating lookahead adjustment for

more timely data retrieval presents a promising avenue for research. The exploration of large page sizes may also contribute to improved performance and warrants further investigation.

Extending the optimization strategies introduced in this work to other prefetching algorithms, such as AMPM (Section 5.5), and other diverse techniques reviewed in Chapter 5, will provide a more comprehensive understanding of the presented work. It is noteworthy that the Prefetcher discussed herein primarily focus on Stream-based access patterns. However, applications reliant on machine learning, graph analytics, and sparse linear algebra often exhibit irregular memory access patterns due to operations such as traversing graph edges or accessing non-zero elements in sparse matrices. These patterns typically lack temporal or spatial locality, leading to prolonged memory stalls and increased bandwidth demands, as also observed with the input dense vector in MINIFE SpMV. The Indirect Memory Prefetcher (IMP) proposed by [38] efficiently addresses these access patterns. Evaluating the optimization strategies with IMP offers a promising direction for enhancing prefetching in scenarios characterized by indirect memory accesses.

Further research into applying these optimization strategies across a broader spectrum of benchmarks will enrich the understanding of their effectiveness. Optimizing memory requirements for the developed strategies could also reduce hardware costs, making the implementation of such Prefetcher more feasible in real-world hardware. Another potential improvement involves the adoption of variable epoch lengths, allowing the prefetching strategies to adjust the degree and lookahead parameters based on real-time statistics rather than adhering to a fixed epoch duration. This approach could lead to a more dynamic and efficient update mechanism for Prefetcher, optimizing their configuration more rapidly and enhancing performance across various workloads.

A. Appendix Chapter

A.1. Hardware Configuration Parameters

Parameter	Value	Description
Simulation Mode	Full System Mode	Indicates that the simulation is in Full System Mode.
Checkpoint CPU Model	AtomicSimple	Specifies the CPU model used for checkpointing.
Number of CPUs	20	Indicates there are 20 CPUs in the system.
SLCs per CPU	1	Represents the number of SLC per CPU.
CPU Clock Speed	2.4GHz	Specifies the clock speed of the CPUs.
CPU SVE Support	Enabled	Indicates that Scalable Vector Extension (SVE) is enabled for the CPUs.
SVE Vector Length	256	Specifies the vector length for SVE.
Branch Predictor	BiMode	Describes the branch predictor used.
Memory Size (HBM2)	1GB	Indicates the size of HBM2 memory.
HBM2 Channels	8	Represents the number of channels for HBM2 memory.
Memory Size (DDR5)	2GB	Specifies the size of DDR5 memory.
DDR5 Channels	1	Indicates the number of channels for DDR5 memory.
Cache Levels	3	Specifies the number of cache levels in the system.
Last Level Shared	Yes	Indicates that the last-level cache is shared among CPUs.
Cache Line Size	64 Bytes	Specifies the size of cache lines.
L1 Instruction Cache Size	64KB	Represents the size of the L1 Instruction Cache.

Parameter	Value	Description
L1 Instruction Cache Assoc.	4	Describes the associativity of the L1 Instruction Cache.
L1 Instruction Cache Latency	Tag: 1, Data: 1, Response: 1	Shows the latency for L1 Instruction Cache (tag/data/response).
L1 Data Cache Size	64KB	Specifies the size of the L1 Data Cache.
L1 Data Cache Assoc.	4	Describes the associativity of the L1 Data Cache.
L1 Data Cache Latency	Tag: 1, Data: 2, Response: 1	Shows the latency for L1 Data Cache (tag/data/response).
IPTW Cache Size	8KiB	Indicates the size of the Instruction Prefetch Table Walker (IPTW) cache.
IPTW Cache Assoc.	4	Describes the associativity of the IPTW cache.
IPTW Cache Latency	Tag: 1, Data: 1, Response: 1	Shows the latency for the IPTW cache (tag/data/response).
DPTW Cache Size	8KiB	Specifies the size of the Data Prefetch Table Walker (DPTW) cache.
DPTW Cache Assoc.	4	Describes the associativity of the DPTW cache.
DPTW Cache Latency	Tag: 1, Data: 1, Response: 1	Shows the latency for the DPTW cache (tag/data/response).
L2 Cache Size	1MB	Indicates the size of the L2 Cache.
L2 Cache Assoc.	8	Describes the associativity of the L2 Cache.
L2 Cache Clusivity	Mostly Inclusive	Specifies that the L2 Cache is mostly inclusive.
L2 Cache Latency	Tag: 2, Data: 4, Response: 4	Shows the latency for the L2 Cache (tag/data/response).
SLC Cache Size	2MB	Represents the size of the SLC.
SLC Cache Assoc.	16	Describes the associativity of the SLC.
SLC Cache Clusivity	Mostly Exclusive	Specifies that the SLC Cache is mostly exclusive.
SLC Cache Latency	Tag: 2, Data: 10, Response: 10	Shows the latency for the SLC Cache (tag/data/response).
NOC Active	Yes	Indicates that the Network-on-Chip (NOC) is active.
NOC Clock Speed	2GHz	Specifies the clock speed of the NOC.
NOC Data Width	64 bits	Indicates the data width of the NOC.

Parameter	Value	Description
NOC Model	Garnet	Describes the model used for the NOC.
Garnet VCS per VNET	4	Specifies the number of Virtual Channels per Virtual Network in Garnet.
Garnet Routing Algorithm	0	Describes the routing algorithm used in Garnet.
Garnet Deadlock Threshold	128	Specifies the deadlock threshold for Garnet.
Garnet Link Bridges	No	Indicates whether link bridges are used in Garnet.
Topology	Mesh	Specifies that the system topology is Mesh.
Router Latency	Tag: 2, Data: 4, Response: 4	Describes the latency for routers in the Mesh topology.
Mesh Link Latency	2	Shows the latency for links in the Mesh topology.
Node Link Latency	1	Indicates the latency for node links in the Mesh topology.
Cross-NUMA Link Latency	5	Specifies the latency for cross-NUMA links in the Mesh topology.
CHI Protocol Model	CHI	Describes the protocol model used.
RNF Routers	0-3	Lists the router numbers for the RNF.
HNF Routers	4-7	Lists the router numbers for the HNF.
SNF_Mem Routers	8-11	Lists the router numbers for SNF_Mem, including High BW memory (e.g., HBM) and Low BW memory (e.g., DDR).
SNF_IO Routers	12-15	Lists the router numbers for SNF_IO.
RNI_IO Routers	16-19	Lists the router numbers for RNI_IO.

Table A.1.: Parameters of the Arm CPU used in the Simulation

Parameter	HBM2	DDR5
Device Bus Width	128 bits	8 bits (4x8), 4 bits (16x4)
Write Buffer Size	128	128
Read Buffer Size	128	64
Burst Length	4	16
Device Size	128MB	2GB (4x8), 1GB (16x4)
Device Rowbuffer Size	2kB	1kB
Devices per Rank	1	4 (4x8), 16 (16x4)
Ranks per Channel	1	2
Banks per Rank	16	32 (4x8), 16 (16x4)
Bank Groups per Rank	4	8 (4x8), 4 (16x4)
Frequency	2400 MHz	4800 MHz
tCK	0.833ns	0.416ns
tRP	14ns	16ns
tRCD	14ns	15ns (4x8), 15ns (16x4)
tCL	14ns	16.64ns (4x8), 16.25ns (16x4)
tRAS	33ns	32ns
tBURST	1.666ns	1.664ns (4x8), 3.3ns (16x4)
tCCD_L	3.332ns	3.75ns (4x8), 5ns (16x4)
tRFC	160ns	195ns (4x8), 420ns (16x4)
tREFI	3.9us	3.9us
tWR	8ns	30ns
tRTP	3.5ns	7.5ns
tWTR	3ns	5ns
tRTW	1.666ns	0.832ns
tCS	0ns	0.832ns
tRRD	1.666ns	3.328ns (4x8), 3.3ns (16x4)
tRRD_L	1.666ns	5ns
tXAW	12.5ns	13.333ns (4x8), 20ns (16x4)
Activation Limit	4	4
tXP	3.332ns	7.5ns (4x8), 9.375ns (16x4)
tXS	160ns	205ns (4x8), 420ns (16x4)

Table A.2.: Specifications of HBM2 and DDR5 Memory Technologies [15]

A.2. Prefetcher Configuration Parameters

This section highlights the configuration parameters for all implemented Prefetchers, where some of those implement all optimization techniques discussed in Section 3.6. The focus is on two primary types of prefetcher: the Stream-based Prefetcher, which has been advanced to create the Aggressive Prefetcher (Agg Prefetcher), and the PC-based Stride Prefetcher, which has evolved into the Timely Aware Stride Prefetcher (TiA Prefetcher).

A key distinction to note is that the original PC-based Stride Prefetcher does not employ the windowing approach, a feature that has been incorporated in the TiA Prefetcher. The configuration parameters for each of these prefetchers are systematically presented in dedicated tables. The parameters for the Stream-based Prefetcher are outlined in Table A.3, while those for the PC-based Stride Prefetcher are detailed in Table A.5. Additionally, the parameters for the Agg Prefetcher and the TiA Prefetcher are respectively found in Table A.4 and Table A.6.

Furthermore, for the purpose of epoch tuning, specific parameters are defined within the Queued Class. These parameters are inherited by all aforementioned prefetchers. Detailed information regarding these inherited parameters is available in Table A.7.

Parameter	Description
threshConf	Set confidence threshold for prefetch generation.
enableWindow	Toggle windowing feature.
distance	Initialize static distance.
degree	Initialize static degree.
table_entries	Set size of the prefetcher's cache.

Table A.3.: StreamPrefetcher Parameters

Parameter	Description
threshConf	Set confidence threshold for prefetch generation.
fdQueueSize	Define the queue size for each tag to calculate the distance.
enableLookahead	Toggle lookahead feature.
enableVoting	Toggle voting mechanism for distance adjustment.
voting_avg_value	Select mean voting (true) or most-voted (false).
enableAdjDegree	Toggle degree adjustment feature.
enableWindow	Toggle windowing feature.
distance	Initialize static distance.
degree	Initialize static degree.
table_entries	Set size of the prefetcher's cache.

Table A.4.: AggressivePrefetcher Parameters

Parameter	Description
confidence_counter_bits	Initialize the confidence counter with specified bits.
initial_confidence	Initialize the confidence counter with initial value.
confidence_threshold	Set the confidence threshold as a fraction of 100.
use_requestor_id	Configure whether to use the requestor ID in prefetching decisions.
degree	Set the degree of prefetching as specified.
distance	Set the prefetch distance as specified.
table_assoc	Configure the Program Counter (PC) table's associativity.
table_entries	Configure the number of entries in the PC table.
table_indexing_policy	Configure the indexing policy for the PC table.
table_replacement_policy	Configure the replacement policy for the PC table.

Table A.5.: Stride Prefetcher Parameters

Parameter	Description
confidence_counter_bits	Initialize the confidence counter with specified bits.
initial_confidence	Initialize the confidence counter with initial value.
confidence_threshold	Set the confidence threshold as a ratio.
use_requestor_id	Configure the use of requestor ID in prefetch decisions.
degree	Set the prefetch degree.
fill_done_queue_size	Define the size of the fill-done queue.
adapt_lookahead_as_degree	Configure lookahead adaptation for degree or distance.
enable_voting	Enable or disable the voting mechanism for prefetch decisions.
voting_avg_value	Determine the voting method: average or most-voted.
enable_adj_degree	Enable or disable degree adjustment.
table_assoc	Set up the PC table's associativity.
table_entries	Set up the number of entries in the PC table.
table_indexing_policy	Set up the indexing policy for the PC table.
table_replacement_policy	Set up the replacement policy for the PC table.
window_monitoring	Enable or disable window-based prefetching.

Table A.6.: TiAStride Prefetcher Parameters

Parameter	Description
printStatsThreshold	Stats print at epoch number ≥ threshold for plot generation.
numa_epoch_cycles	Cycles in an epoch period.
numa_epoch_cycles_reset	Cycles in an epoch period for reset.
numa_aware_tuning_enable	Enable/disable global or numa-aware tuning.
numa_pf_distance_max_threshold	Pf distance threshold.
numa_pf_degree_max_threshold	Pf degree threshold.
numa_accuracy_rate_high_threshold	High pf accuracy rate threshold.
numa_accuracy_rate_low_threshold	Low pf accuracy rate threshold.
numa_hbm_lat_high_threshold	TBE latency threshold for HBM2 High BW Util.
numa_hbm_lat_medium_threshold	TBE latency threshold for HBM2 Medium BW Util.
numa_ddr_lat_high_threshold	TBE latency threshold for DDR5 High BW Util.
numa_ddr_lat_medium_threshold	TBE latency threshold for DDR5 Medium BW Util.

Table A.7.: Inherited Parameters from Queued Class for Prefetchers

A.3. HPC Machine Details

Table A.8.: Detailed hardware specifications of the Juawei and HAICGU systems.

System	Hardware Specifications
Juawei Hi1616 Node	• CPU: 2x HiSilicon Hi1616 (32x 2.4 GHz Cortex-A72 Cores) • Memory: 256GB (16x16GB) DDR4-2133 • OS: CentOS 7.6.1810 (AltArch)
Juawei Haswell Node	• CPU: 2x Intel Xeon E5-2660 v3 (10x 2.6 GHz Haswell Cores, 3.3 GHz Boost) • Memory: 128GB (8x16GB) DDR4-2133 • OS: CentOS 7.6.1810
HAICGU Cluster-Standard Compute Node	• Model: TaiShan 200, Model 2280 • CPU: 2x Kunpeng 920 processor (64 cores; 2.6GHz) • Memory: 224GB (16x 8GB DDR4) • OS: Rocky Linux 8 • Network: 1x 100Gbit/s EDR Infiniband HCA
HAICGU Cluster-Development Compute Node	• Model: TaiShan 200, Model 2280 • CPU: 2x Kunpeng 920 processor (64 cores; 2.6GHz) • Memory: 128GB (16x 8GB DDR4) • OS: Rocky Linux 8 • Network: 1x 100Gbit/s EDR Infiniband HCA • Storage: 2 x 960GB SSD SATA 6Gb/s
HAICGU Cluster-IO Node	• Metadata storage: 2x 960GB SSD RAID 1 • Object v: 32x 1.2 TB HDD RAID 10) • Mgmt. storage: 4x 1.2 TB HDD SAS 12Gb/s; 10.000rpm (RAID 10)

System	Hardware Specifications
HAICGU Cluster-AI Training Node	• Model: Atlas 800, Model 9000 • CPU: 4x Kunpeng 920 processor (ARMv8 AArch64) • Memory: 1024GB (32x 32GB DDR4 2933MHz RDIMM) • OS: Rocky Linux 8 • Neural Processing Unit (NPU): 8x Huawei Ascend 910 with 32 AI cores and 32GB HBM2 memory • Local storage: 2x 960 GB SSD SATA 6Gb/s • Data storage: 4x 3.2TB SSD NVMe
HAICGU Cluster-AI Inference Node	• Model: Atlas 800, Model 3000 • CPU: 2x Kunpeng 920 processor (ARMv8 AArch64) • Memory: 512GB (16x 32GB DDR4 2933 MHz RDIMM) • OS: Rocky Linux 8 • Local storage: 2x 960GB SSD SATA 6Gb/s • Data storage: 4 x 960GB SSD SATA 6Gb/s • GPU: 5x Atlas 300 AI Inference Card; 32GB; PCIe3.0 x16

A.4. Barcelona Supercomputing Center - Sparse Matrix-Vector Multiplication

Similar to MINIFE SpMV this operation multiplies a sparse matrix by a dense vector to produce another dense vector, using the CSR (Compressed Sparse Row) format. This version of SpMV optimized across NUMA nodes, and the impact of prefetching techniques, are highly dependent on the underlying matrix patterns [32]. However, we observed low bandwidth utilization across the numa domains, which is the reason why we further implemented the MINIFE SpMV to the NUMA version. Subsequently, the description of the SpMV from Barcelona Supercomputing Center is going to be explained.

The CSR format is engineered to optimize storage by solely retaining non-zero matrix elements and their corresponding column indices. It comprises three fundamental components:

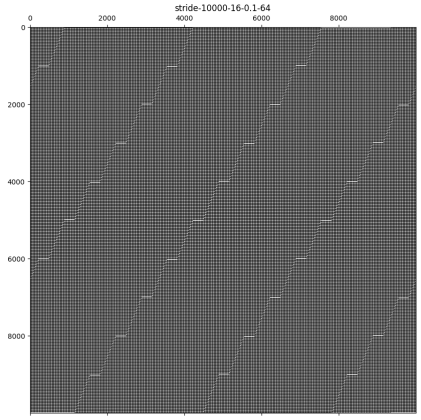
- *Array of Non-Zero Values*: This component stores all non-zero matrix elements in a row-major order.
- *Column Indices* (`col_idxs`): Working in conjunction with `nnz`, this array holds the column indices of the non-zero elements.
- *Row Pointers* (`row_ptrs`): It includes indices that mark the start of each row within the `nnz_vals` array, enabling swift access to rows.

The direct row access capability and compact storage of the CSR format significantly enhance the efficiency of SpMV operations. Moreover, its alignment with vectorized operations further boosts performance on contemporary hardware architectures.

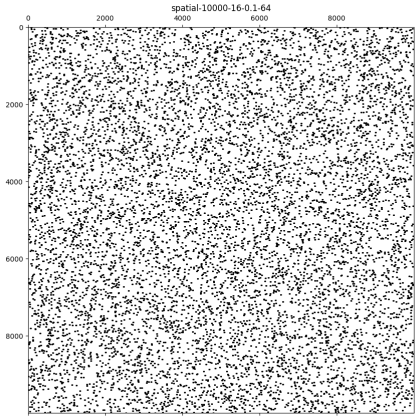
Optimizing SpMV for NUMA systems is vital to enhance performance in multi-processor environments, where memory access latency varies based on the memory location. This optimization involves strategically distributing the sparse matrix and vector data across different NUMA nodes to reduce cross-node memory accesses and evenly distribute computational load among processors. In the architecture under consideration, the heterogeneity of the implemented memory presents a particularly intriguing aspect. Utilizing the HBM2 node for matrix operations alongside a DDR5 memory node for dense vector processing could enhance system performance. This is primarily due to the significant number of read operations; the HBM's superior bandwidth is expected to minimize latency, thereby improving data throughput. Exploring the outcomes across all four possible configurations of HBM2 and DDR5 Node associations presents an intriguing aspect.

The efficacy of prefetching in SpMV computations is contingent upon the matrix's inherent patterns:

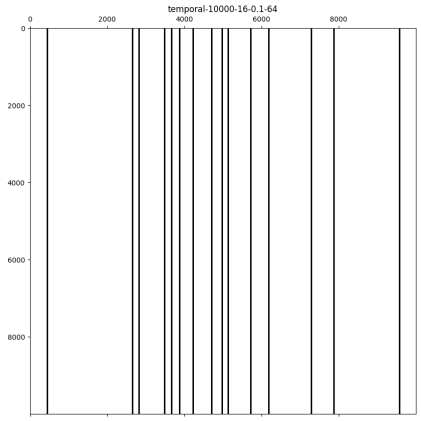
- **Stride Access Pattern:** Matrices exhibiting a stride access pattern, characterized by regularly spaced non-zero elements, can significantly leverage prefetching techniques. This optimization preemptively loads anticipated elements into the cache, thereby reducing memory access latency and enhancing computational throughput, as illustrated in Figure A.1a.
- **Spatial Locality Access Pattern:** For matrices demonstrating a spatial locality access pattern, where non-zero elements are closely clustered, prefetching is particularly beneficial. It allows for concurrent access to adjacent memory locations, effectively minimizing cache miss rates and optimizing data retrieval processes, as depicted in Figure A.1b.



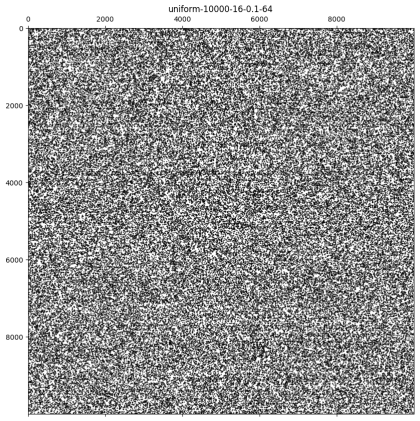
(a) Stride Access Pattern



(b) Spatial Locality Access Pattern



(c) Temporal Locality Access Pattern



(d) Uniform Distribution Pattern

Figure A.1.: Access Patterns in Sparse Matrix-Vector Multiplication (SpMV)

- **Temporal Locality Access Pattern:** In cases where matrices display a temporal locality access pattern, with frequent re-accesses to certain elements, the efficacy of prefetching may be limited due to the high probability of these elements being retained in the cache from prior accesses. This characteristic is highlighted in Figure A.1c.
- **Uniform Distribution Pattern:** Matrices with a uniform distribution of non-zero elements may experience modest improvements in performance through prefetching. This technique facilitates the anticipatory loading of data likely to be required in the near future, although the gains may not be as substantial as those observed in matrices with more defined access patterns, as shown in Figure A.1d.

To determine the required size for the given problem, it is essential to analyze the memory requirements. The CSR format is characterized by the use of three distinct arrays. An array containing the non-zero elements of the matrix, an array for the column indices of these non-zero elements, and an array for the row pointers. The size of the non-zero elements array is computed as the product of *nnz* (the number of non-zero elements per row) and N (the dimension of the square matrix), with the column indices array mirroring the size of the non-zero elements array. The row pointers array extends one element beyond the matrix dimension, totaling $N + 1$ elements. Thus, the memory footprint for a CSR matrix is formulated as $(nnz \times N \times 2 + N + 1) \times D_{\text{Bytes}}$, where D_{Bytes} signifies the size of each data element in Bytes. Contrarily, the memory requisites for both the input and output dense vectors are $N \times D_{\text{Bytes}}$ each.

Referencing the work by Sgherzi et al. [32], the overall memory requirement integrates the CSR matrix and both vectors, calculated as follows:

$$(\text{nnz} \times N \times 2 + N + 1 + 2N) \times D_{\text{Bytes}} = (\text{nnz} \times N \times 2 + 3N + 1) \times D_{\text{Bytes}}.$$

This equation indicates that the matrix size is influenced by the values of nnz and N. Setting nnz = 16 establishes the basis for the simulation parameters.

Given a system configuration with $N_{\text{core}} = 20$ cores and an SLC size of $SLC_{\text{size}} = 2$ MiB per core, with data elements sized at $D_{\text{Bytes}} = 4$ Bytes, the equation to ascertain the problem size can be delineated as:

$$\begin{aligned} 20 \times 2 \times 1024 \times 1024\,\text{Bytes} &< (16 \times N \times 2 + N \times 3 + 1) \times 4\,\text{Bytes}, \\ \Rightarrow 10 \times 1024 \times 1024 &< 35 \times N + 1, \\ \Rightarrow N &> \frac{10 \times 1024 \times 1024 - 1}{35} \\ \Rightarrow N &\approx 299593.1143. \end{aligned}$$

Following the calculation, with nnz set to 16, the matrix dimension N is approximated to 299594 for simulation purposes. During the execution of benchmarks, it was observed that the process stalled while reading matrices, rendering it impossible to evaluate within the Gem5 Simulation framework. To address compatibility issues, modifications were made to the codebase to ensure compliance with the compiler version supported by the QEMU image, which is utilized for FS mode simulation. While these adjustments are executed successfully on hardware, challenges persist in executing the modified code in FS mode simulation. This discrepancy warrants further investigation.

List of Figures

List of Tables

List of Abbreviations and Acronyms

Abbreviation	Meaning
Agg Prefetcher	Aggressive Prefetcher
ALU	Arithmetic Logic Unit
AMPM	Access Map Pattern Matching
Arm	Advanced RISC Machines
CPU	Central Processing Unit
DDR5	Double Data Rate 5
ECC	Error Checking and Correction
FS Mode	Full System Mode
fdQueue	fill done queue
FPU	Floating-Point Unit
H	High
HBM2	High-Bandwidth Memory 2
HPC	High Performance Computing
IMP	Indirect Memory Prefetcher
IP	Instruction Pointer
ISA	Instruction Set Architecture
I/O	Input/Output
IPC	Instructions Per Cycle
KNL	Knight's Landing
L	Low
L1 Cache	level one Cache
L2 Cache	level two Cache
L3 Cache	level three Cache
LLC	Last Level Cache
LRU	Least Recently Used
M	Medium
MOP	Macro-Operation
NoC	Network on Chip
NoP	No Prefetching
NUMA	Non-Uniform Memory Access

Abbreviation	Meaning
OS	Operating System
PC	Program Counter
RAM	Random Access Memory
RISC	Reduced Instruction Set Computing
ROB	ReOrder Buffer
SECDED	Single-bit Error-Correction, Double -bit Error Detection
SE Mode	System Emulation Mode
SELL	Sliced ELLPACK
SLC	System Level Cache
SNC	Sub-NUMA Clustering
SpMV	Sparse Matrix-Vector Multiplication
SSD	Solid-State Drive
SVE	Scalable Vector Extension
TBE	Translation Buffer Entries
TLB	Translation Lookaside Buffer
TiA Prefetcher	Timely Aware Stride Prefetcher
μOP	Micro-Operation
w.r.t.	with respect to

Bibliography

[1] Arm. Arm® Neoverse™ V1 Core - Technical Reference Manual. Accessed: 02.01.2024. Arm. 2021. URL: https://developer.arm.com/documentation/101427/0101?lang=en.

[2] M. N. Asghar. "A review of ARM processor architecture history, progress and applications". In: *Journal of Applied and Emerging Sciences* 10.2 (2020), pp–171.

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

[4] D. N. H. Berk Saglam and C. Falquez. EPI-GEM5-PF: NUMA-Aware Branch. Branch: epi-gem5-pf-numa-aware. 2024. URL: https://gitlab.jsc.fz-juelich.de/arm-jsc-projects/epi-gem5-pf.git.

[5] D. N. H. Berk Saglam and C. Falquez. gem5-arm-prefetcher. Branch: epi-numa-aware-pf. 2024. URL: https://gitlab.jsc.fz-juelich.de/arm-jsc-projects/gem5-arm-prefetcher/-/tree/epi-numa-aware-pf.

[6] J. L. Bez, E. Bernart, F. Santos, L. Schnorr, and P. Navaux. "Performance and energy efficiency analysis of HPC physics simulation applications in a cluster of ARM processors". In: *Concurrency and Computation: Practice and Experience* 29.21 (2017), e4014. URL: https://dx.doi.org/10.1002/cpe.4014.

[7] D. F. Boes. Side Channel Attacks. Accessed: 05.12.2023. University of Bonn. 2023. URL: https://ecampus.uni-bonn.de/ilias.php?ref_id=2282548&cmd=frameset&cmdClass=ilrepositorygui&cmdNode=yn&baseClass=ilRepositoryGUI.

[8] E. Calore, A. Gabbana, S. Schifano, and R. Tripiccione. "ThunderX2 Performance and Energy-Efficiency for HPC Workloads". In: *Computation* 8.1 (2020), p. 20. URL: https://dx.doi.org/10.3390/computation8010020.

[9] S. Chandrasekaran, M. Si, J. Zhai, and L. Oden. "Special issue on new trends in high-performance computing: Software systems and applications". In: *Software: Practice and Experience* 52.10 (2022), n/a. URL: https://dx.doi.org/10.1002/spe.3155.

[10] CHI Protocol Fundamentals. https://developer.arm.com/documentation/102407/0100/CHI-protocol-fundamentals. Accessed: 01.05.2024. 2023.

[11] P. S. Crozier, H. K. Thornquist, R. W. Numrich, A. B. Williams, H. C. Edwards, E. R. Keiter, M. Rajan, J. M. Willenbring, D. W. Doerfler, and M. A. Heroux. Improving performance via mini-applications. Tech. rep. Sandia National Laboratories (SNL), Albuquerque, NM, and Livermore, CA . . ., 2009.

[12] H. Devarajan, A. Kougkas, and X.-H. Sun. "Hfetch: Hierarchical data prefetching for scientific workflows in multi-tiered storage environments". In: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2020, pp. 62–72.

[13] D. Efnusheva, A. Cholakoska, and A. Tentov. "A survey of different approaches for overcoming the processor-memory bottleneck". In: *International Journal of Computer Science and Information Technology* 9.2 (2017), pp. 151–163.

[14] R. Eigenmann and D. J. Lilja. "Von neumann computers". In: *Wiley Encyclopedia of Electrical and Electronics Engineering* 23 (1998), pp. 387–400.

[15] C. Falquez. EPI-gem5. `https://gitlab.jsc.fz-juelich.de/arm-jsc-projects/epi-gem5-pf/-/tree/epi-gem5-pf-numa-aware`. Accessed: 29.12.2023. 2023.

[16] B. Falsafi and T. F. Wenisch. A primer on hardware prefetching. Springer Nature, 2022.

[17] M. Grannaes, M. Jahre, and L. Natvig. "Multi-level hardware prefetching using low complexity delta correlating prediction tables with partial matching". In: *International Conference on High-Performance Embedded Architectures and Compilers*. Springer. 2010, pp. 247–261.

[18] W. Heirman, I. Hur, U. Echeruo, S. Eyerman, and K. Du Bois. Apparatus, method, and system for enhanced data prefetching based on non-uniform memory access (NUMA) characteristics. US Patent 10,621,099. Apr. 2020.

[19] Y. Ishii, M. Inaba, and K. Hiraki. "Access map pattern matching for high performance data cache prefetch". In: *Journal of Instruction-Level Parallelism* 13.2011 (2011), pp. 1–24.

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

[21] D. Joseph and D. Grunwald. "Prefetching using Markov predictors-25th Anniversary Retrospective". In: *Symposium on Microarchitecture*. 1995, pp. 231–236.

[22] J. Kim, S. H. Pugsley, P. V. Gratz, A. N. Reddy, C. Wilkerson, and Z. Chishti. "Path confidence based lookahead prefetching". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2016, pp. 1–12.

[23] T. Koizumi, T. Nakamura, Y. Degawa, H. Irie, S. Sakai, and R. Shioya. "T-SKID: predicting when to prefetch separately from address prediction". In: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2022, pp. 1389–1394.

[24] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillón, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, M. Fariborz, A. F. Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. S. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and É. F. Zulian. "The gem5 Simulator: Version 20.0+". In: *CoRR* abs/2007.03152 (2020). URL: https://arxiv.org/abs/2007.03152.

[25] S. Mittal. "A survey of recent prefetching techniques for processor caches". In: *ACM Computing Surveys (CSUR)* 49.2 (2016), pp. 1–35.

[26] A. Navarro-Torres, B. Panda, J. Alastruey-Benedé, P. Ibánez, V. Viñals-Yúfera, and A. Ros. "Berti: an Accurate Local-Delta Data Prefetcher". In: *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2022, pp. 975–991.

[27] N. Neves, P. Tomás, and N. Roma. "Compiler-assisted data streaming for regular code structures". In: *IEEE Transactions on Computers* 70.3 (2020), pp. 483–494.

[28] D. A. Patterson and J. L. Hennessy. Computer organization and design ARM edition: the hardware software interface. Morgan kaufmann, 2016.

[29] D. K. Poulsen and P.-C. Y. P.-C. Yew. "Data prefetching and data forwarding in shared memory multiprocessors". In: *1994 Internatonal Conference on Parallel Processing Vol. 2*. Vol. 2. IEEE. 1994, pp. 280–280.

[30] A. R. Proaño. "Data Placement Strategies for Heterogeneous and Non-Volatile Memories in High Performance Computing". PhD thesis. Université de Bordeaux, 2021.

[31] N. Rajovic. "Enabling the use of embedded and mobile technologies for high-performance computing". PhD thesis. Universitat Politècnica de Catalunya, 2017. URL: https://dx.doi.org/10.5821/dissertation-2117-113680.

[32] F. Sgherzi, M. Siracusa, I. Fernandez, A. Armejach, and M. Moretó. "SpChar: Characterizing the Sparse Puzzle via Decision Trees". In: *arXiv preprint arXiv:2304.06944* (2023).

[33] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. “Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers”. In: *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE. 2007, pp. 63–74.

[34] V. K. Sripathi and K. Raman. Optimizing Memory Bandwidth on Stream Triad. Accessed: 2024-01-24. Intel. 2021. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/optimizing-memory-bandwidth-on-stream-triad.html.

[35] E. Suarez. HIGH PERFORMANCE COMPUTING (HPC). Accessed: 27.11.2023. University of Bonn / FZJ-JSC. 2022. URL: https://ecampus.uni-bonn.de/goto_ecampus_crs_2798331.html.

[36] A. Vladimirov and R. Asai. “Clustering modes in Knights Landing processors: Developer’s guide”. In: *Colfax International* (2016).

[37] WikiChip. Neoverse V1 - Microarchitectures - ARM. https://en.wikichip.org/wiki/arm_holdings/microarchitectures/neoverse_v1. Accessed: 02.01.2024. 2021.

[38] X. Yu, C. J. Hughes, N. Satish, and S. Devadas. “IMP: Indirect memory prefetcher”. In: *Proceedings of the 48th International Symposium on Microarchitecture*. 2015, pp. 178–190.

[39] L. Zaourar, M. Benazouz, A. Mouhagir, F. Jebali, T. Sassolas, J.-C. Weill, C. Falquez, N. Ho, D. Pleiter, A. Portero, et al. “Multilevel simulation-based co-design of next generation HPC microprocessors”. In: *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE. 2021, pp. 18–29.