



Characterizing Memory-Level Parallelism and Guiding Prefetch Control with ROB-Visible Feedback in Multicore Systems

— 24 pt research project (S1/S2 2026)

A report submitted for the course
COMP8800, Advanced Computing Research Project

By:
Jingyu Lu

Supervisor:
Dr. Shoaib Akram

October 2026

Declaration:

I declare that this work:

- upholds the principles of academic integrity, as defined in the [Academic Integrity Rule](#);
- is original, except where collaboration (for example group work) has been authorised in writing by the course convener in the class summary and/or LMS course site;
- is produced for the purposes of this assessment task and has not been submitted for assessment in any other context, except where authorised in writing by the course convener;
- gives appropriate acknowledgement of the ideas, scholarship and intellectual property of others insofar as these have been used;
- in no part involves copying, cheating, collusion, fabrication, plagiarism or recycling.

I acknowledge that I am expected to have undertaken Academic Integrity training through the Epigeum Academic Integrity modules prior to submitting an assessment, and so acknowledge that ignorance of the rules around academic integrity cannot be an excuse for any breach.

October (2026), Jingyu Lu

Statement of AI Usage

I used ChatGPT to help me understand relevant research, clarify technical concepts, organize research ideas, and structure this report. ChatGPT was also used to support translation and improve academic writing.

All research decisions, methodological designs, experimental evaluations, and interpretations of results were made and reviewed by me. AI-generated suggestions were examined and modified as necessary, and I take full responsibility for the accuracy and integrity of the final work.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr. Shoaib Akram. It was through his excellent teaching in COMP6300 that I first developed an interest in computer architecture and began to understand the fundamental principles of how computers work. As a junior researcher, I benefited greatly from his insight and guidance throughout the project. I am especially thankful for his patience and the time he devoted to discussing my research—often late into the night. His support has been invaluable to me as I take my first steps in my research career.

I would also like to express my heartfelt gratitude to my parents for supporting my decision to pursue my studies at the Australian National University. Throughout this academic journey, their understanding and unconditional support have always given me the confidence to keep moving forward.

Abstract

The performance of modern memory-intensive applications depends on memory access latency, the degree of overlap among outstanding requests, and whether memory delays are exposed to processor execution. Traditional memory-level parallelism (MLP) metrics characterize concurrent memory activity, but do not directly capture its performance impact.

This work first revises MLP Stacks by averaging each metric over the cycles when the corresponding memory activity occurs. Unlike prior work, we extend the framework to multicore systems with per-core demand and prefetch accounting. Our results show that higher measured memory parallelism may arise from additional memory accesses, such as compiler-generated loads and hardware prefetch requests, or from requests remaining outstanding longer under constrained memory service, rather than more effective latency hiding. Moreover, similar MLP profiles across cores may hide differences in how memory delays affect their performance.

In multicore systems, memory requests from different cores compete for shared resources, which can delay demand requests, leading to memory interference. We refer to the additional latency experienced by demand requests due to shared-resource contention or prefetch-induced cache pollution as memory-side harm. Previous studies do not fully capture how memory interference affects processor performance. We therefore develop a request-level, source-aware interference attribution framework for L3 miss status holding register (MSHR) contention, DRAM contention, and L3 prefetch pollution. We relate the attributed memory-side harm to reorder buffer (ROB) execution to determine the portion that affects application progress. The results show that memory-side harm is generally substantially larger than the corresponding ROB-visible harm, indicating that a considerable fraction of memory interference is hidden by out-of-order (OoO) execution and MLP.

We also develop a request-level prefetch benefit model that determines whether the latency reduction provided by useful prefetches becomes visible to processor execution. Both prefetch benefit and interference are influenced by prefetch aggressiveness, which describes how many requests a prefetcher generates and how far ahead it fetches data. Based on ROB-visible benefit and cross-core harm (harm caused by prefetch requests to other cores), we propose the ROB-Visible Prefetch Aggressiveness Controller (RVPAC), which dynamically adjusts the number of prefetch requests generated per prefetching event.

We implement the framework in the Sniper multicore simulator and evaluate it on 4-, 8-, and 16-core configurations. Across these configurations, RVPAC achieves maximum speedups of $1.11\times$, $1.17\times$, $1.30\times$, and $1.10\times$ for the Microbenchmark, BC, SSSP, and Heterogeneous workloads, respectively. Overall, our study integrates MLP analysis, memory interference attribution, and ROB-visible prefetch aggressiveness control.

Table of Contents

1. Introduction	1
2. Background	5
2.1. Memory-Level Parallelism	5
2.1.1. Out-of-Order Execution and Memory Latency Overlap	5
2.1.2. MLP Stacks	6
2.2. Shared Memory Hierarchy	8
2.2.1. Non-Blocking Caches and MSHRs	8
2.2.2. Shared Last-Level Cache	9
2.2.3. DRAM Organization and Contention	9
2.3. Hardware Data Prefetching	10
2.3.1. Prefetch Operation	10
2.3.2. Prefetch Effectiveness and Aggressiveness	11
3. Related Work	13
3.1. Memory-Level and Cache-Level Parallelism Analysis	13
3.2. Memory Interference and Performance Modeling	14
3.2.1. Memory-Side Interference Modeling	14
3.2.2. Criticality-Based Performance Analysis	15
3.3. Adaptive Prefetcher Management	16
4. MLP Analysis	19
4.1. Motivation and Overview	19
4.2. Metric Redefinition	19
4.3. Multicore Extension	21
4.4. Implementation	22
4.5. Experimental Methodology	23
4.5.1. Default Hardware	23
4.5.2. Benchmarks	24
4.6. Results and Analysis	24
4.6.1. Validation of MLP and CLP Measurement	24
4.6.2. Memory-System Sensitivity and Interpretation	26
4.6.3. Multicore and Prefetch Characterization	29
5. Memory Interference and ROB-Visible Performance Modeling	33
5.1. Motivation and Overview	33

Table of Contents

5.2. Memory-Side Interference Modeling	33
5.2.1. L3 MSHR Contention	34
5.2.2. DRAM Contention	35
5.2.3. L3 Prefetch Pollution	35
5.3. ROB-Visible Performance Modeling	36
5.4. Implementation	38
5.5. Results and Analysis	41
5.5.1. Microbenchmark Validation	41
5.5.2. GAPBS SSSP Case Study	46
6. Adaptive Prefetch Throttling	53
6.1. Motivation and Overview	53
6.2. Prefetch Benefit Modeling	54
6.3. RVPAC – Adaptive Throttling Policy	56
6.3.1. Utility Signal and Mode Selection	57
6.3.2. GAIN and GUARD Control	58
6.3.3. Probe-Up and Rollback	60
6.4. Implementation	61
6.5. Experimental Evaluation	62
6.5.1. Sensitivity to DDR4 Configuration	67
7. Conclusion and Future Work	73
7.1. Conclusion	73
7.2. Limitations and Future Work	74
A. Additional MLP Results	77
A.1. BFS	77
A.2. PageRank	78
A.3. SPMV	78
B. MLP Stacks Visualization Guide	81
Bibliography	83

Introduction

Memory access latency is an important factor limiting the performance of modern processors, particularly for memory-intensive applications with frequent irregular memory accesses. Modern out-of-order (OoO) processors can continue executing independent younger instructions while long-latency memory accesses remain outstanding. They can also allow multiple memory requests to proceed concurrently, thereby hiding part of the memory access latency. As a result, the latency observed in the memory system does not directly translate into processor stall time. Analyzing memory performance therefore requires considering not only memory access latency, but also the overlap among memory requests and whether these delays ultimately affect processor execution.

Memory-Level Parallelism (MLP) describes the number of outstanding memory accesses and is an important metric for analyzing memory access overlap. MLP Stacks ([Hasnat et al., 2025](#)) further extend this analysis across the cache hierarchy and DRAM by introducing cache-level parallelism (CLP) metrics, while distinguishing demand and prefetch activity. In multicore systems, different cores generate their own memory requests, making per-core MLP an important consideration when analyzing shared memory behaviors. However, MLP and CLP describe memory concurrency rather than its performance impact. Our analysis shows that higher measured MLP may result from additional memory traffic or from request accumulation under constrained memory service, while different cores may also exhibit similar MLP profiles. These MLP and CLP metrics are therefore insufficient for directly identifying shared-memory interference and its performance impact. This work first revises the measurement methodology of MLP Stacks and extends the framework to analyze multicore demand and prefetch activity.

In multicore processors, memory requests from different cores share resources such as the last-level cache (LLC) and DRAM. Because these resources have finite capacity and bandwidth, concurrent requests can contend for them, increasing access latency. This performance degradation caused by competition for shared memory-system resources is referred to as memory interference. Hardware prefetching can further increase such interference by generating additional memory requests. Effective prefetching can fetch data required by subsequent demand accesses in advance and hide memory access latency, but prefetch requests also consume cache capacity and memory bandwidth and may introduce additional contention and cache pollution. As workload behavior and system resources change, the balance between prefetching benefit and interference also

1. Introduction

changes. Under relatively low contention, the latency reduction from useful prefetches may outweigh the additional interference. However, under high contention, the additional memory traffic generated by prefetching may introduce sufficient interference to outweigh its latency-hiding benefit. A fixed level of prefetch aggressiveness therefore cannot consistently deliver suitable performance across different conditions.

Previous studies often analyze memory interference and control prefetching using aggregate memory system metrics, such as memory latency, bandwidth pressure, and prefetch accuracy. These metrics characterize memory-system behavior, but do not directly reveal how the benefit or interference associated with individual memory requests affects processor progress. OoO execution and MLP may hide additional memory delays. Similarly, a useful prefetch may reduce memory access latency without improving performance if the demand request would not otherwise delay processor progress. Therefore, decisions based solely on memory-side activity may not accurately reflect the actual impact of memory requests on execution performance.

We develop a request-level, source-aware memory interference model. It attributes the additional delay experienced by each demand request to the corresponding source cores, while distinguishing between demand and prefetch requests. The model considers major sources of interference, including shared-cache miss handling, DRAM contention, and prefetch-induced cache pollution. Compared with approaches that rely only on core-level or resource-level statistics, this analysis preserves both the source of interference and the affected demand request, providing a basis for further evaluating its performance impact.

Request-level attribution identifies the source and victim of memory interference, but the magnitude of memory-side delay does not directly represent its actual performance impact. This work combines the attributed delay with the ROB behavior of the affected instruction to determine whether the delay actually limits processor progress, retaining the corresponding portion as ROB-visible harm. Our results show that interference harm measured at the memory side is generally much larger than the corresponding ROB-visible harm. This indicates that a considerable fraction of shared-resource interference does not directly translate into processor-visible performance loss. The relative importance of different interference sources can also change between the memory-side and ROB-visible results, further showing that memory-side delay magnitude alone cannot accurately determine their actual performance impact.

Reducing interference alone, however, is insufficient for determining an appropriate level of prefetch aggressiveness, because useful prefetches may also provide substantial latency reduction. We associate each useful prefetch with the demand request that consumes the prefetched data. We then determine whether the resulting latency reduction affects processor progress to obtain ROB-visible prefetch benefit. We can therefore compare prefetch benefit and harm at the same processor-visible level rather than rely separately on aggregate usefulness and memory-side latency.

Based on these two feedback signals, we propose the ROB-Visible Prefetch Aggressiveness Controller (RVPAC). RVPAC uses ROB-visible prefetch benefit and cross-core prefetch harm to evaluate the utility of prefetching for each core. The utility compares the latency-hiding benefit of a core’s prefetches against the interference they cause to other cores. Based on this feedback, the controller determines whether to increase, decrease, or maintain each core’s prefetch aggressiveness. RVPAC adjusts the prefetch degree, which determines the number of prefetch requests generated per prefetching event, without modifying the underlying prediction algorithm.

RVPAC further coordinates these per-core decisions through two operating modes, GAIN and GUARD, based on the overall balance between ROB-visible prefetch benefit and cross-core harm. In GAIN mode, the controller primarily follows each core’s utility, while GUARD mode prioritizes throttling prefetchers that cause unusually high cross-core interference. To adapt to changes in workload behavior and memory contention, RVPAC also allows prefetchers to probe higher aggressiveness levels after throttling. If a probe produces negative utility or causes severe cross-core interference, the controller restores the previous level. This mechanism allows RVPAC to re-evaluate prefetching opportunities as execution conditions change.

We implement the proposed analysis and control mechanisms in the Sniper multicore simulator (Carlson et al., 2014; Kanellopoulos et al., 2025) and evaluate them using 4-, 8-, and 16-core configurations. The results show that the performance impact of a fixed prefetch aggressiveness varies substantially across workloads and levels of system concurrency. Under heavier shared-resource contention, additional prefetch requests can introduce greater interference, while RVPAC adapts the prefetch aggressiveness of individual cores according to their observed benefit and harm. Across the evaluated configurations, RVPAC achieves maximum speedups of $1.11\times$, $1.17\times$, $1.30\times$, and $1.10\times$ over the fixed-aggressiveness Sniper simple prefetcher for the Microbenchmark, BC, SSSP, and Heterogeneous workloads, respectively.

Overall, we first analyze MLP and its relationship with performance. We then investigate interference among demand and prefetch requests from different cores and relate memory-side benefit and harm to processor execution. The results show that MLP, memory-side interference, and prefetch usefulness describe different aspects of memory-system behavior, but none of these measures alone directly reflects their impact on execution performance. We use ROB-visible analysis to connect memory-system behavior with processor progress and guide adaptive prefetch control.

Background

This chapter introduces the background concepts relevant to this study. We first discuss MLP and OoO execution, followed by the shared memory hierarchy and resource contention in multicore systems. We then introduce hardware data prefetching, including its operation, effectiveness, and aggressiveness.

2.1. Memory-Level Parallelism

2.1.1. Out-of-Order Execution and Memory Latency Overlap

Long-latency memory accesses can significantly limit processor performance when their latency is exposed to execution. Modern out-of-order processors tolerate part of this latency by allowing younger instructions that do not depend on a pending long-latency access to continue executing (Mutlu et al., 2003). This execution model allows multiple memory operations to overlap in time rather than being serviced strictly serially. MLP quantifies this concurrency and is commonly characterized by the number of long-latency memory accesses that are outstanding simultaneously (Tuck et al., 2006; Qureshi et al., 2006), as illustrated in Figure 2.1.

Let $MLP(t)$ denote the number of useful long-latency off-chip accesses outstanding at cycle t (Chou et al., 2004). The conventional average DRAM-level MLP is computed only over cycles with at least one active DRAM access:

$$MLP = \frac{\sum_t MLP(t)}{T(\text{DRAM})}, \quad (2.1)$$

where $T(\text{DRAM})$ is the number of such cycles (Hasnat et al., 2025). The resulting MLP therefore represents the average concurrency of DRAM accesses during DRAM-active cycles.

MLP is important for understanding the performance impact of memory latency because long-latency memory accesses can overlap rather than contribute independently to execution time. When several such accesses are outstanding concurrently, part of their latency can overlap with other memory operations and independent instruction

2. Background

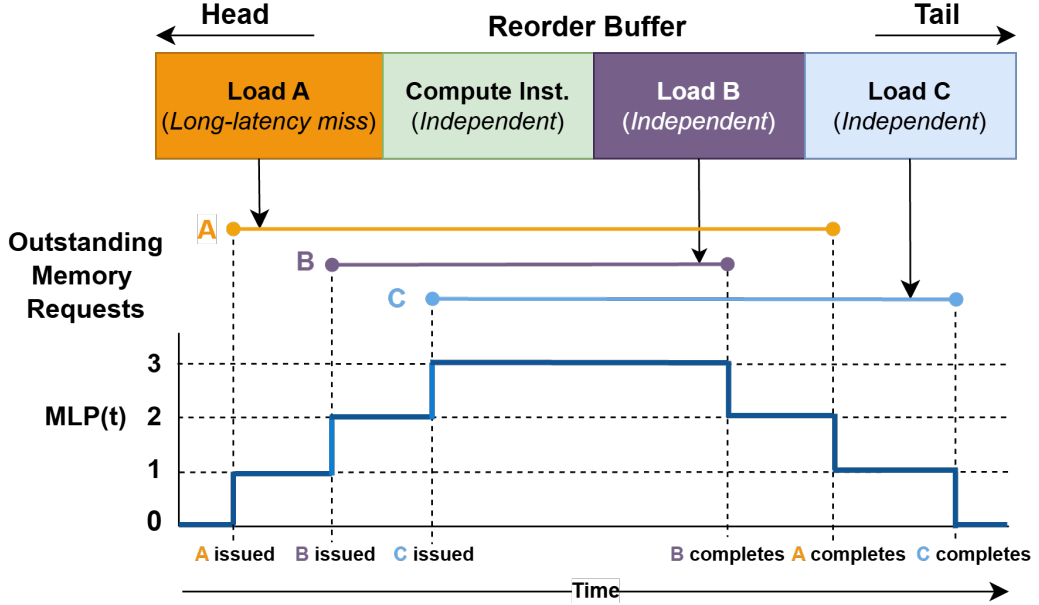


Figure 2.1.: Illustration of memory-level parallelism in an out-of-order processor.

execution. Consequently, an access that overlaps with other long-latency accesses may impose a smaller effective performance cost than one serviced in isolation (Qureshi et al., 2006). The exploitable MLP is jointly determined by program characteristics, such as data dependencies and memory-access patterns, and by the processor resources available to keep independent operations in flight (Van den Steen and Eeckhout, 2018).

Out-of-order processors commonly use a ROB to retain the state of in-flight instructions that may complete out of program order while preserving in-order retirement. The ROB is organized in program order, with new instructions allocated at the tail and the oldest instruction retiring from the head (Smith and Pleszkun, 1988). Its finite capacity therefore places a bound on the number of instructions that can remain in flight.

An instruction awaiting completion of a long-latency memory access can block retirement when it becomes the oldest incomplete instruction in the ROB. Younger instructions may continue to execute and accumulate behind it, but if the delay persists, the ROB eventually fills and dispatch stalls until the memory operation completes and retirement resumes (Karkhanis and Smith, 2004).

2.1.2. MLP Stacks

Conventional MLP characterizes concurrency at the off-chip memory level, although memory accesses may also overlap while being serviced within the cache hierarchy. MLP

2.1. Memory-Level Parallelism

Stacks extend this view by introducing cache-level parallelism (CLP) metrics for individual cache levels (Hasnat et al., 2025). For cache level L at cycle t , total cache-level parallelism $\text{TCLP}(L, t)$ denotes the number of ongoing accesses, while miss and hit cache-level parallelism, $\text{MCLP}(L, t)$ and $\text{HCLP}(L, t)$, denote the numbers of pending misses and hits, respectively, such that

$$\text{TCLP}(L, t) = \text{MCLP}(L, t) + \text{HCLP}(L, t). \quad (2.2)$$

MLP Stacks further separate memory activity into core- and prefetch-generated components, with prefetch activity additionally classified as useful or useless. This decomposition distinguishes the parallelism generated by processor requests from the additional memory activity introduced by hardware prefetching (Hasnat et al., 2025).

To retain consistent relationships among these metrics after averaging, MLP Stacks use a common normalization interval. Let $T(\text{Hier})$ denote the number of cycles during which at least one access is active anywhere in the memory hierarchy. For $X \in \{T, M, H\}$, the average form of a CLP metric is defined as

$$\text{XCLP}(L) = \frac{\sum_t \text{XCLP}(L, t)}{T(\text{Hier})}. \quad (2.3)$$

The same $T(\text{Hier})$ denominator is also applied to DRAM-level MLP, replacing the conventional $T(\text{DRAM})$ denominator:

$$\text{MLP}_{\text{Stacks}} = \frac{\sum_t \text{MLP}(t)}{T(\text{Hier})}. \quad (2.4)$$

Using a common normalization interval preserves relationships such as $\text{TCLP}(L) = \text{MCLP}(L) + \text{HCLP}(L)$ in the averaged metrics (Hasnat et al., 2025).

MLP Stacks additionally estimate speculative MLP from load instructions that have not yet been issued. Loads whose addresses are blocked by unresolved dependencies are classified as dependency-bound (DP-bound), while loads that are ready but cannot issue because the required processor resources are unavailable are classified as structural-hazard-bound (ST-bound) (Hasnat et al., 2025). The average DP-bound and ST-bound components are normalized over the same $T(\text{Hier})$ interval used by the other MLP Stacks metrics.

Finally, MLP Stacks combine the parallelism metrics with cycles-per-instruction (CPI) stack information to visualize both memory-hierarchy concurrency and the extent to which memory activity contributes to execution time (Hasnat et al., 2025). A detailed visual guide to the MLP Stacks representation is provided in Appendix B.

2. Background

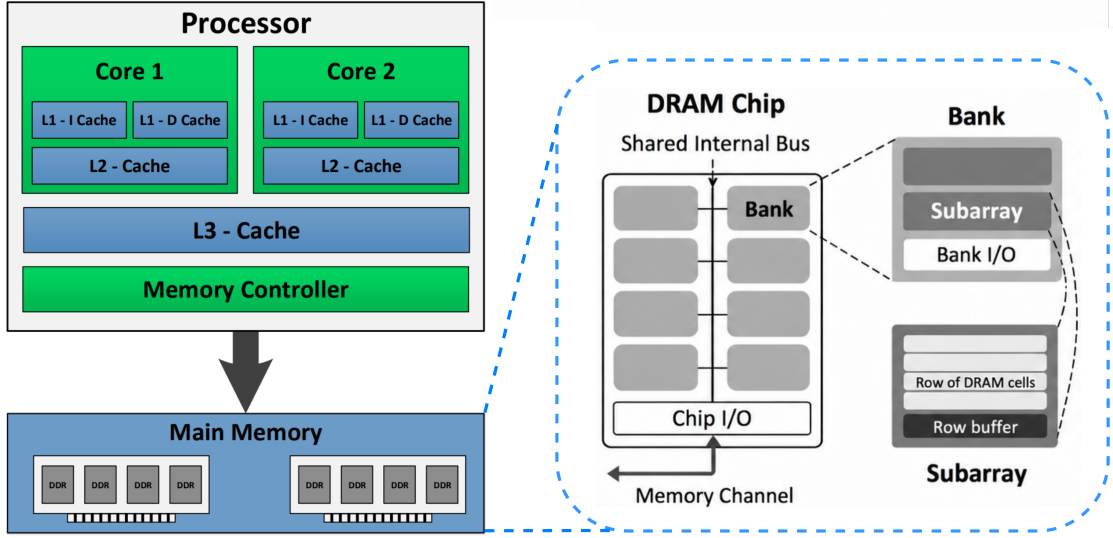


Figure 2.2.: Overview of a multicore memory hierarchy and DRAM organization, adapted from [Ghasempour \(2015\)](#); [Seshadri et al. \(2013\)](#).

2.2. Shared Memory Hierarchy

Modern multicore processors typically combine private upper-level caches with a shared LLC and a DRAM-based main-memory system connected through a memory controller, as illustrated in Figure 2.2.

2.2.1. Non-Blocking Caches and MSHRs

Supporting multiple outstanding memory accesses requires a cache to continue servicing requests while earlier misses are still pending. A blocking cache prevents subsequent accesses from proceeding until an outstanding miss is resolved, whereas a non-blocking cache retains the state of pending misses and allows other accesses to proceed concurrently. Miss status holding registers (MSHRs) provide this miss-tracking state and are a key mechanism for supporting multiple outstanding cache misses ([Kroft, 1981](#); [Tuck et al., 2006](#)).

An MSHR is associated with an outstanding cache line and records the information required to complete the corresponding miss. A miss to a line with no existing outstanding request allocates a new MSHR entry as a primary miss. If another request subsequently misses on the same line, it is treated as a secondary miss and merged into the existing entry rather than issuing a duplicate request to the next memory level ([Tuck et al., 2006](#)).

MSHR capacity is finite and therefore limits the number of independent cache misses that can be tracked concurrently. When a new miss requires an MSHR but no entry is available, the cache cannot admit that miss until an occupied entry is released. If this condition persists, subsequent requests may also be delayed and request processing can eventually stall (Tuck et al., 2006). This constraint is particularly relevant in multicore processors because MSHRs associated with a shared LLC may themselves be shared among cores, allowing concurrent request streams to contend for miss-handling capacity (Valsan et al., 2016). Prefetch-generated requests can further increase MSHR occupancy and the frequency with which the structure becomes full (Nath et al., 2024).

2.2.2. Shared Last-Level Cache

Modern multicore processors commonly employ private upper-level caches and an LLC that is shared across cores, often organized into multiple banks or slices (Chaudhuri, 2021). The LLC provides an additional on-chip opportunity to service requests that miss in the private caches before they access off-chip memory. In a set-associative LLC, each cache block maps to a particular set. When a new block is inserted into a set with no invalid way, the replacement policy selects one of the resident blocks in that set for eviction (Chaudhuri, 2021).

Because LLC capacity is finite and shared, requests from different cores compete for cache space. Cache blocks inserted on behalf of one core can displace blocks that would otherwise remain available to another core, effectively reducing the cache capacity available to the latter (Wang et al., 2017). If a displaced block is subsequently reused, the access incurs an additional LLC miss and must be serviced by a lower level of the memory hierarchy. Shared LLC capacity and replacement therefore provide a mechanism through which the memory activity of one core can affect that of another.

Hardware prefetch requests also participate in LLC insertion and replacement decisions and can consequently alter which blocks remain resident in the cache. Prefetch-induced cache pollution occurs when prefetched data displaces a block that would otherwise be useful, potentially converting a subsequent demand hit into a miss (Wu et al., 2011; Jain and Lin, 2018). In a shared LLC, such prefetch-induced interference can occur within the same core or across cores (Wu et al., 2011).

2.2.3. DRAM Organization and Contention

DRAM-based main memory is hierarchically organized to expose parallelism across memory requests. A memory channel connects one or more ranks, where each rank consists of multiple DRAM chips operating in lockstep and provides multiple banks that can be accessed in parallel. DDR4 further organizes banks into bank groups, while each bank is logically composed of rows and columns (Kim et al., 2012; Khadem et al., 2025). Requests to different banks can therefore overlap in service, whereas requests targeting

2. Background

the same bank contend for the bank and are generally serialized in conventional DRAM systems (Kim et al., 2012).

Each bank has a row buffer that can hold an active row, and the latency of a memory access depends on the state of this buffer. A row-buffer hit occurs when the requested row is already active and requires only a read or write command. If no row is active, the requested row must first be activated. If a different row is active, it must first be precharged before the new row can be activated and accessed. These cases correspond to a row hit, a closed-row access, and a row conflict, respectively (Mutlu and Moscibroda, 2008; Kim et al., 2012). DRAM commands are also constrained by timing requirements that determine when subsequent commands to a bank, rank, or channel may be issued (Kim et al., 2012).

The memory controller buffers outstanding requests and schedules their DRAM commands subject to bank availability, row-buffer state, shared command/address/data buses, and DRAM timing constraints (Mutlu and Moscibroda, 2008). In a multicore system, requests from different cores share these resources and can delay one another through bank, row-buffer, and bus conflicts. Such interference can also serialize requests whose latencies would otherwise overlap across different banks, reducing effective bank-level parallelism (Mutlu and Moscibroda, 2008). Consequently, the latency observed by a memory request includes not only the intrinsic DRAM access latency but also waiting introduced by competing requests in the memory controller and DRAM system.

2.3. Hardware Data Prefetching

2.3.1. Prefetch Operation

Hardware data prefetching is a technique for reducing memory-access latency by predicting future memory accesses and fetching the corresponding data into on-chip caches before the corresponding demand access occurs (Bera et al., 2021). Hardware prefetchers infer recurring access patterns from previously observed memory requests, using contextual information such as the instruction program counter (PC), cache-line address, or page offset to generate predictions for subsequent accesses. When the prediction is correct and the data arrives sufficiently early, a subsequent demand access can be served from a closer level of the memory hierarchy.

Stride-based prefetching targets access streams in which successive addresses exhibit a stable difference or *stride*. A PC-based stride prefetcher associates memory accesses with the instruction that generated them, tracks this stride across consecutive accesses, and extrapolates it to predict future addresses (Fu et al., 1992; Bera et al., 2021), as shown in Figure 2.3(b). In contrast, access patterns depicted in Figure 2.3(c), involving pointer traversal, indirect indexing, or other data-dependent address generation, are generally less regular and more difficult to predict.

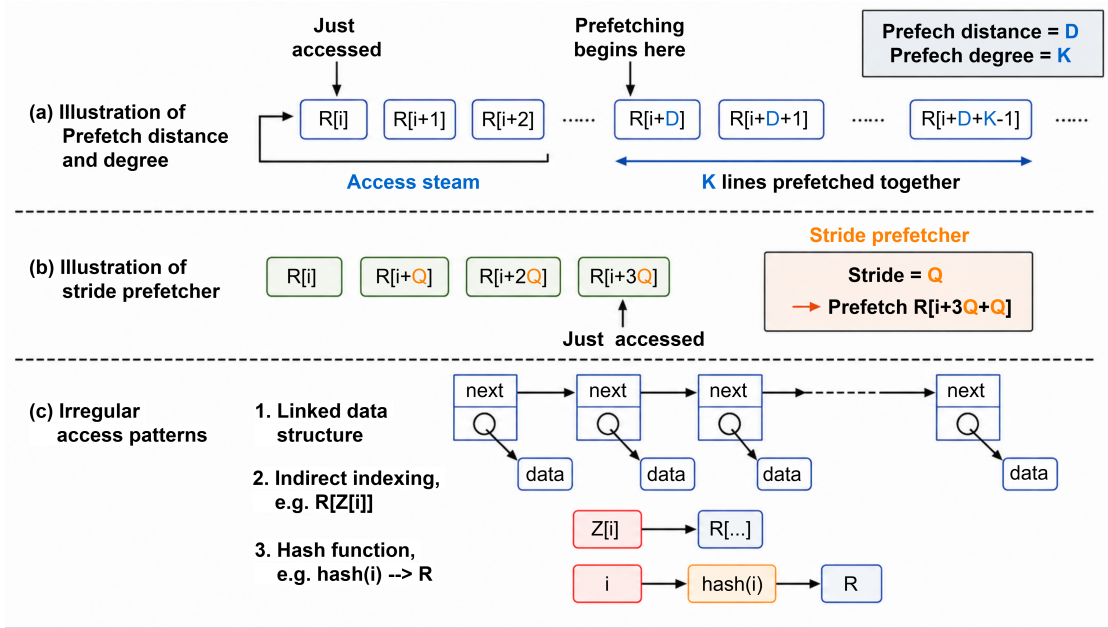


Figure 2.3.: Prefetch distance and degree, stride-based prediction, and representative irregular access patterns.

Unlike demand requests, which are generated directly by program execution, prefetch requests are issued speculatively and consume memory-hierarchy resources while attempting to make predicted cache blocks available in advance. Their effectiveness therefore depends not only on prediction correctness, but also on whether the data arrives in time and whether the additional requests impose unnecessary pressure on the memory hierarchy.

2.3.2. Prefetch Effectiveness and Aggressiveness

Prefetch effectiveness depends on both prediction correctness and timeliness. A prefetch is *useful* if the prefetched block is subsequently accessed by a demand request before eviction, and *useless* otherwise. Prefetch accuracy is the fraction of issued prefetches that are useful, while coverage measures the reduction in cache misses relative to a no-prefetch baseline (Srinath et al., 2007; Ai et al., 2025). A late prefetch may still hide part of the access latency, but the demand request must wait for the remaining service time if the block has not yet arrived.

Prefetch aggressiveness describes how actively a prefetcher generates speculative memory requests. As displayed in Figure 2.3(a), two common parameters for controlling aggressiveness are prefetch *distance* and *degree*. Prefetch distance determines how far ahead of the demand access stream requests are generated, while prefetch degree determines

2. Background

how many blocks are requested per trigger ([Srinath et al., 2007](#)). Increasing aggressiveness can improve timeliness and latency hiding, but can also increase speculative traffic, which consumes cache capacity and memory bandwidth and may delay demand requests through additional contention ([Srinath et al., 2007](#); [Pugsley et al., 2014](#)).

Related Work

In this chapter, we discuss three lines of work related to this study. First, we introduce methods for characterizing MLP and CLP, and discuss MLP Stacks and their limitations in metric normalization and support for multi-core scenarios. Second, we review shared-cache and DRAM interference modeling, along with criticality-based approaches that map memory-side interference delay to its impact on processor performance. Third, we discuss adaptive hardware prefetch management based on accuracy, resource pressure, criticality and utility. Existing approaches typically evaluate memory behavior at the granularity of resources, cores, or threads. Although some approaches consider processor criticality, they do not directly combine request-level ROB-visible benefit and interference harm to guide prefetch throttling.

3.1. Memory-Level and Cache-Level Parallelism Analysis

MLP describes the degree to which a processor overlaps multiple outstanding memory accesses. [Chou et al. \(2004\)](#) proposed the epoch model to analyze how factors such as the instruction window size, branch mispredictions, and serializing instructions limit achievable MLP, and to relate it to off-chip memory stall time. [Mehta \(2022\)](#) applied Little’s Law to estimate MLP on real hardware from memory bandwidth and average access latency, while emphasizing MSHR capacity as a key constraint on the number of concurrent outstanding misses. In contrast to prior work that focuses solely on DRAM requests, [Tang et al. \(2019\)](#) extended this analysis to CLP, showing that the latency of accesses to a shared LLC can likewise be hidden through parallel execution. [Carlson et al. \(2015\)](#) instead proposed epoch profiles, using epochs per kilo-instruction (EPKI) to quantify non-overlapped long-latency accesses, and showed that MLP must be interpreted jointly with cache behavior, miss count and instruction window size in order to explain performance.

[Hasnat et al. \(2025\)](#) extended traditional DRAM-level MLP to the entire memory hierarchy, decomposing MLP into core and prefetch components and separately measuring hit and miss cache-level parallelism. This framework further combines these metrics with a CPI stack, showing the CPI contribution of each memory level alongside the speculative parallelism bounded by dependencies and structural hazards.

3. Related Work

To preserve relationships across cache levels and the additivity of hit and miss components, MLP Stacks normalize against the active time of the entire memory hierarchy as a common basis. While this design yields a unified stacked representation, the resulting metric reflects an amortized contribution over global memory active time rather than the average degree of parallelism during the periods when the corresponding activity actually occurs, and is therefore susceptible to latency variations at other memory levels. Moreover, the approach targets single-core systems and leaves multi-core support to future work. This study redefines the measurement semantics of the MLP Stacks metric using active time specific to each memory level and access type, and extends the framework to multi-core scenarios. Although the individual components no longer remain directly additive, the resulting values more accurately reflect the average degree of parallelism during the periods when the corresponding memory activity actually occurs.

3.2. Memory Interference and Performance Modeling

3.2.1. Memory-Side Interference Modeling

In multi-core processors, demand and prefetch requests issued by different cores compete for shared cache and main memory resources. Cache capacity contention can introduce additional misses, while contention at the memory controller, banks and bus increases the latency of acquiring resources through request queuing. Existing work primarily models these effects from three dimensions, identifying interference sources, estimating performance impact and decomposing memory-access latency.

Fairness via Source Throttling (FST) decomposes shared-memory interference into cache pollution, DRAM bus and bank conflicts and row-buffer interference, estimating the excess cycles introduced by each core through cache pollution tracking and per-core DRAM state estimation (Ebrahimi et al., 2012). Per-Thread Cycle Accounting (PTCA) instead uses auxiliary tag structures to identify inter-thread misses in the shared cache and cycle-accounts for the increased access latency caused by contention for memory resources and bandwidth (Du Bois et al., 2013). Both ultimately summarize interference at the core or thread level, which obscures differences among individual memory requests. FST assumes demand-only traffic, while PTCA considers prefetch-induced delay without separately attributing demand and prefetch sources.

The Application Slowdown Model (ASM) estimates performance slowdown from the change in shared-cache access rate between isolated and shared execution, while accounting for the overlapping latency among multiple concurrent requests in an aggregate manner (Subramanian et al., 2015). DRAM Bandwidth and Latency Stacks instead decompose bandwidth loss and access latency into components such as refresh, precharge and activation, row conflicts and queuing, in order to explain the hardware sources of DRAM bottlenecks (Eyerman et al., 2022). Helm and Taura (2020) further detect DRAM contention from the gap between observed access latency and contention-free

3.2. Memory Interference and Performance Modeling

latency, attributing high-latency accesses to their corresponding functions and data objects.

These approaches primarily characterize interference at the granularity of applications, threads, cores, hardware resources, or software locations. However, individual memory requests within the same thread can differ substantially in their interference sources, delay magnitudes, and visibility to processor execution. Aggregating interference at a coarser granularity can therefore obscure which request is affected by which source and make it difficult to relate the memory-side impact of an individual request to its corresponding ROB behavior. This study instead constructs a request-level memory-interference modeling framework. For each affected demand request, the framework records the benefit it receives or the additional latency it incurs and attributes the corresponding impact to its source. The framework separately models L3 MSHR admission blocking, L3 prefetch pollution, and DRAM contention while distinguishing self- and cross-core effects as well as demand- and prefetch-generated interference. This request-level attribution provides the basis for subsequently evaluating whether these memory-side benefits and interference effects translate into ROB-visible performance impact.

3.2.2. Criticality-Based Performance Analysis

Memory access latency does not necessarily translate into performance loss, since OoO and MLP can overlap a substantial portion of the latency. Criticality analysis therefore considers not only the magnitude of memory latency, but also whether the affected thread or memory access constrains processor progress.

Thread-level approaches typically estimate criticality from aggregate execution characteristics. TCPAC computes thread criticality from misses at different cache levels weighted by their respective miss penalties, and identifies progress-limited threads from the criticality gap between threads (Panda and Balachandran, 2014). This notion is most meaningful within a parallel application, where slower threads can delay overall progress through synchronization, and is less directly applicable to multiprogrammed workloads whose programs execute independently. PTCA instead performs thread-level cycle accounting for the extra cycles caused by shared-resource interference. Although latency information is maintained per individual miss request, the corresponding interference is charged to a thread’s execution cycles only once that request reaches the head of the ROB and causes a stall (Du Bois et al., 2013). Both methods therefore ultimately target thread progress rather than the performance contribution of individual requests.

Load-level approaches further identify memory instructions more likely to limit execution. CLIP tags load PCs that frequently block the ROB head during L1 misses as critical loads and prioritizes prefetch generation for these accesses (Panda, 2023). This

3. Related Work

approach uses historical criticality to predict future prefetch value, but does not quantify how much execution time a completed request’s benefit or interference actually changed.

These approaches improve the connection between memory behavior and processor performance, but they primarily characterize criticality at the thread or load level. They do not directly quantify how much of the benefit or interference associated with an individual completed memory request is exposed to processor execution.

3.3. Adaptive Prefetcher Management

Hardware prefetch management must balance the benefit of latency hiding against cache and bandwidth overhead. FDP dynamically adjusts prefetch distance and degree based on prefetch accuracy, lateness and cache pollution (Srinath et al., 2007). HPAC extends this feedback mechanism to multi-core systems, coordinating prefetch aggressiveness across cores by combining per-core local control with prefetch accuracy, cross-core cache pollution, bandwidth utilization, and other cores’ bandwidth demand (Ebrahimi et al., 2009).

Subsequent research improves prefetch control from the perspectives of thread importance, safe run-time evaluation, cross-core coordination and low-overhead feedback. TCPAC adjusts prefetch distance and degree based on thread criticality, execution progress, and prefetch accuracy, and manages prefetch requests in the LLC and DRAM accordingly (Panda and Balachandran, 2014). Sandbox Prefetching stores candidate prefetch addresses in a Bloom filter without actually issuing requests, estimates the accuracy of the candidate prefetchers based on subsequent accesses, and enables only those configurations that meet the threshold, thereby safely evaluating aggressive prefetching strategies (Pugsley et al., 2014). SPAC groups prefetchers into aggressive and meek categories and searches a reduced configuration space for joint throttling schemes that improve fair-speedup (Panda, 2016). Band-pass Prefetching bounds prefetch traffic using local and global prefetch-fraction metrics (Sridharan et al., 2017), while NST adjusts prefetch aggressiveness based on the late-prefetch fraction, keeping prefetches timely while avoiding bandwidth saturation (Heirman et al., 2018).

Utility-driven approaches further merge the positive and negative effects of prefetching into a unified feedback signal. CAFFEINE estimates the processor cycles saved and added by a prefetcher along the LLC-to-DRAM path and coordinates prefetch aggressiveness across cores using the resulting net utility (Panda and Balachandran, 2015). I-POP instead aggregates the benefit of useful prefetches, cache pollution, and shared-resource contention caused by each prefetcher into a single metric, Prefetch Effectiveness, and uses it to control the enable/disable state and aggressiveness of multiple prefetchers (Lin et al., 2026).

3.3. Adaptive Prefetcher Management

This study likewise adopts utility-driven control, but the benefit and harm used to compute utility are first passed through ROB-visibility analysis rather than taken directly from raw memory latency or resource pressure. The controller adjusts prefetch aggressiveness based on the resulting net utility, and reassesses the potential value of higher prefetch aggressiveness through a tentative step-up and fallback mechanism.

MLP Analysis

4.1. Motivation and Overview

MLP Stacks extend conventional DRAM-level MLP to the cache hierarchy by characterizing concurrent hits and misses at individual cache levels. To preserve relationships among the averaged metrics, the original framework normalizes these quantities using a common hierarchy-wide active interval $T(\text{Hier})$. Therefore, the reported parallelism at a given memory level is averaged over the activity of the entire memory hierarchy rather than the activity associated with that level. This makes the resulting metrics more difficult to interpret when comparing memory-system configurations with different access latencies or service times.

A second limitation is that the original MLP Stacks implementation was primarily designed for single-core analysis, making it unable to provide a consistent description of per-core memory parallelism in multi-core systems, particularly when demand and prefetch requests from different cores coexist within a shared memory hierarchy.

This chapter addresses these limitations by revising the normalization of MLP Stacks to use the corresponding active time at each memory level and by extending the framework to support per-core MLP and CLP analysis in multicore systems. The following sections describe the revised metric definitions and the multicore extension.

4.2. Metric Redefinition

The original MLP Stacks framework uses a common hierarchy-wide active time, $T(\text{Hier})$, to normalize hit, miss, and total cache-level parallelism, as well as DRAM-level MLP. For a cache level L , the original metrics are defined as

4. MLP Analysis

$$\text{HCLP}_{\text{orig}}(L) = \frac{\sum_t \text{HCLP}(L, t)}{T(\text{Hier})}, \quad (4.1)$$

$$\text{MCLP}_{\text{orig}}(L) = \frac{\sum_t \text{MCLP}(L, t)}{T(\text{Hier})}, \quad (4.2)$$

$$\text{TCLP}_{\text{orig}}(L) = \frac{\sum_t \text{TCLP}(L, t)}{T(\text{Hier})}. \quad (4.3)$$

DRAM-level MLP is normalized over the same interval:

$$\text{MLP}_{\text{orig}} = \frac{\sum_t \text{MLP}(t)}{T(\text{Hier})}. \quad (4.4)$$

MLP Stacks also report speculative MLP from loads that are present in the issue queue but cannot yet be issued. DP-bound represents loads whose issue is delayed by unresolved address dependencies, while ST-bound represents loads that are ready but cannot issue because of structural constraints, such as insufficient load-queue resources. In the original framework, both components are also averaged over the hierarchy-wide active interval:

$$\text{DPBound}_{\text{orig}} = \frac{\sum_t \text{DPBound}(t)}{T(\text{Hier})}, \quad (4.5)$$

$$\text{STBound}_{\text{orig}} = \frac{\sum_t \text{STBound}(t)}{T(\text{Hier})}. \quad (4.6)$$

Using a common denominator preserves relationships among the averaged cache-level metrics, including the decomposition of total parallelism into hit and miss components. However, it also averages each metric over cycles in which the corresponding activity may not be present, making the resulting values amortized measures over the hierarchy-active interval rather than the average concurrency observed when that activity is active. We therefore normalize each metric using the active interval of the activity it represents.

For cache level L , let $T_{\text{hit}}(L)$ denote the number of cycles with at least one active hit, $T_{\text{miss}}(L)$ the number of cycles with at least one active miss, and $T_{\text{access}}(L)$ the number of cycles with at least one active access. The revised cache-level metrics are

$$\text{HCLP}(L) = \frac{\sum_t \text{HCLP}(L, t)}{T_{\text{hit}}(L)}, \quad (4.7)$$

$$\text{MCLP}(L) = \frac{\sum_t \text{MCLP}(L, t)}{T_{\text{miss}}(L)}, \quad (4.8)$$

$$\text{TCLP}(L) = \frac{\sum_t \text{TCLP}(L, t)}{T_{\text{access}}(L)}. \quad (4.9)$$

4.3. Multicore Extension

Similarly, let T_{DRAM} denote the number of cycles with at least one active DRAM request. DRAM-level MLP is redefined as

$$\text{MLP} = \frac{\sum_t \text{MLP}(t)}{T_{\text{DRAM}}}. \quad (4.10)$$

For speculative MLP, let T_{DP} and T_{ST} denote the number of cycles with at least one DP-bound or ST-bound load, respectively. The revised speculative metrics are

$$\text{DPBound} = \frac{\sum_t \text{DPBound}(t)}{T_{\text{DP}}}, \quad (4.11)$$

$$\text{STBound} = \frac{\sum_t \text{STBound}(t)}{T_{\text{ST}}}. \quad (4.12)$$

Under the revised definitions, each metric represents the average concurrency observed while the corresponding activity is present. HCLP, MCLP, and TCLP characterize concurrent hits, misses, and total accesses at each cache level, while DRAM MLP characterizes concurrency during DRAM-active periods. DP-bound and ST-bound similarly represent the average number of dependency-bound and structurally bound loads while the corresponding blocking condition is active.

Using separate active intervals changes the relationship among the averaged cache-level metrics. Although $\text{TCLP}(L, t) = \text{HCLP}(L, t) + \text{MCLP}(L, t)$ holds at each cycle, the corresponding averages need not satisfy $\text{TCLP}(L) = \text{HCLP}(L) + \text{MCLP}(L)$ because they are normalized over different intervals. Likewise, DP-bound and ST-bound no longer represent speculative parallelism amortized over the entire hierarchy-active interval. Instead, they characterize the degree of blocking when the corresponding limitation is present. This change prioritizes the direct interpretation of each metric over preserving relationships imposed by a common denominator.

4.3. Multicore Extension

The original MLP Stacks implementation was developed primarily for single-core execution. Although demand-side statistics are maintained by a per-core ROB model, other parts of the framework rely on global or single-core assumptions. In particular, prefetch statistics do not preserve the core that generated each request, and the original post-processing assumes a single-core organization. These limitations prevent demand and prefetch activity from being consistently separated across cores in multicore workloads.

We extend the framework by preserving the originating core throughout the parallelism accounting. Demand requests retain their existing association with the core that issued

4. MLP Analysis

them, while prefetch requests are augmented with source-core information and are accounted for only by the corresponding core. This association is preserved when requests enter shared levels of the memory hierarchy. Thus, an access issued by core c contributes to the MLP and CLP statistics of core c even when it is serviced by the shared LLC or DRAM.

The active-time counters introduced in the previous section are likewise maintained independently for each core. Consequently, the hit, miss, and total CLP of a core are normalized by the corresponding active intervals generated by that core, and its DRAM MLP is normalized by its own DRAM-active interval. The resulting analysis therefore reports per-core demand and prefetch parallelism rather than combining activity from different cores into a system-wide metric.

4.4. Implementation

We implement the revised MLP Stacks framework in the Sniper multicore simulator by extending its ROB-side parallelism accounting and prefetch tracking. Demand accesses remain associated with the per-core ROB timer that tracks their execution, while prefetches, which are not represented by ROB entries, are explicitly annotated with their issuing-core identity. These changes affect only the measurement framework and do not modify cache replacement, request scheduling, or memory-system behavior.

For each core, the existing level-wise outstanding-request histograms provide the total parallelism observed at each cache level. Hit and miss parallelism are derived from the same per-core hierarchy information using the relationship between adjacent memory levels. For instance, requests that proceed beyond L1 to L2 or lower levels contribute to the L1 miss component, while the remaining L1 activity contributes to the hit component. In parallel, separate active-time counters record the cycles in which at least one hit, miss, or total access from that core is outstanding. These counters provide the revised normalization intervals defined in Section 4.2.

Multicore support additionally requires prefetch activity to remain associated with its source core. The original prefetch records are shared across cores and do not preserve this information. We therefore propagate the issuing-core identity with each prefetch record and include it only in the accounting of the corresponding core. Demand accesses require no additional source metadata because their ownership is already implicit in the per-core ROB timer.

The same ownership is retained at shared levels of the hierarchy. A request originating from core c contributes to core c 's LLC and DRAM parallelism even when it is serviced concurrently with requests from other cores. The active intervals are likewise maintained per core, including the DRAM-active interval, rather than using system-wide LLC or DRAM activity. The statistics and post-processing path are correspondingly extended to produce separate demand and prefetch MLP Stacks for each core.

4.5. Experimental Methodology

4.5.1. Default Hardware

We evaluate the revised MLP Stacks framework using a modified version of Sniper 8.0. The simulated system is based on the Intel Xeon X5550 (Gainestown), which uses the Nehalem microarchitecture. We retain the baseline processor and memory-hierarchy configuration used by the original MLP Stacks implementation, while extending the framework to support multicore execution. The relevant hardware parameters are summarized in Table 4.1. Unless otherwise stated, this configuration is used throughout the evaluation.

The L1 data cache supports up to 16 outstanding misses. The L2 cache does not impose an explicit MSHR limit, following the original MLP Stacks configuration, where the L2 miss-handling capacity was assumed to be larger than that of L1 and unlikely to be reached in the evaluated workloads. Hardware prefetching is provided by Sniper’s simple linear stride prefetcher at L2 and is enabled or disabled depending on the experiment.

Table 4.1.: Default hardware configuration used for the MLP Stacks evaluation.

Processor	
Processor	Intel Xeon X5550 Gainestown
Core frequency	2.66 GHz
Core model	Out-of-order (ROB)
Issue width	4-wide
ROB size	128 entries
Reservation station	36 entries
Max. outstanding	48 loads, 32 stores, 16 L1-D misses
Cache Hierarchy	
L1-I	32 KB, 4-way, 4-cycle access time
L1-D	32 KB, 8-way, 4-cycle access time
L2 cache	256 KB, 8-way, 8-cycle access time
L3 cache	8 MB, 16-way, 30-cycle access time
Cache block size	64 bytes
Prefetcher	
L2 prefetcher	Simple linear stride
Enabled	Varies by experiment
DRAM	
Model	Constant latency
Queue model	<code>history_list</code>
Bandwidth	7.6 GB/s per controller
Latency	45 ns

4. MLP Analysis

4.5.2. Benchmarks

The evaluation combines a controlled pointer-chasing microbenchmark with memory-intensive applications from the GAP Benchmark Suite (GAPBS) (Beamer et al., 2015) and a sparse matrix-vector multiplication (SPMV) kernel (Hasnat et al., 2025). Pointer chasing provides a basic validation case for the measured MLP, while GAPBS represents graph workloads and SPMV represents sparse numerical computation. For the multicore experiments in this chapter, we run GAPBS as a multithreaded OpenMP application across multiple simulated cores. SPMV uses the Compressed Sparse Row (CSR) format, with sequential accesses to matrix values and column indices and indirect accesses to the input vector.

The GAPBS kernels used across the evaluations in this report are summarized below. Individual experiments use subsets of these kernels, as specified in their corresponding methodology.

- **BC (Betweenness Centrality)** measures vertex centrality based on the number of shortest paths that pass through each vertex.
- **CC (Connected Components)** identifies mutually connected vertices and groups them into connected components.
- **PR (PageRank)** repeatedly propagates and updates vertex scores to estimate vertex importance from graph connectivity.
- **BFS (Breadth-First Search)** traverses the graph level by level from a given source vertex until all reachable vertices are visited.
- **SSSP (Single-Source Shortest Paths)** repeatedly updates path distances to compute the shortest paths from a given source vertex to all reachable vertices.

Table 4.2 summarizes the workload configurations. Selected memory-system parameters are varied to examine their effect on measured parallelism. L3 LLC capacity is varied between 8 and 1 MB, while DRAM bandwidth is varied between 7.6 and 0.95 GB/s per controller.

The original hierarchy-wide normalization and the revised active-time normalization are evaluated under otherwise identical configurations. We report HCLP, MCLP, and TCLP at each cache level together with DRAM MLP.

4.6. Results and Analysis

4.6.1. Validation of MLP and CLP Measurement

We first evaluate the MLP and CLP measurement using the pointer-chasing microbenchmark. Because each pointer load depends on the result of the preceding load, only one

Table 4.2.: Workload configurations used for MLP Stacks evaluation.

Workload	Parameter	Setting
Pointer chasing	Number of nodes	32,000
	Pointer chains	1
	Working set	256 KB
	Compilation	-00, -03
GAPBS	Kernels	BC, BFS, PR
	Input graph	Kronecker graph, scale 16 (2^{16} vertices)
	Average degree	16
	Trials	1
	Multicore execution	Multithreaded OpenMP
	Compilation	-03
SPMV	Matrix dimension	$33,554,432 \times 33,554,432$
	Maximum nonzeros	33,554,432
	Iterations	2
	Software prefetch distance	{0, 1, 2, 4, 8, 16, 32, 64, 128}
	Compilation	-03

pointer-chasing access is expected to be outstanding at a time. Figure 4.1 shows the resulting total MLP Stacks profiles. Under -03, the measured L1-D total parallelism is approximately 1, consistent with the expected behavior of the dependent pointer chain. Under -00, however, it increases to approximately 4. To investigate the source of this difference, we disassembled the binaries generated under the two optimization levels and examined the instructions in the pointer-chasing loop.

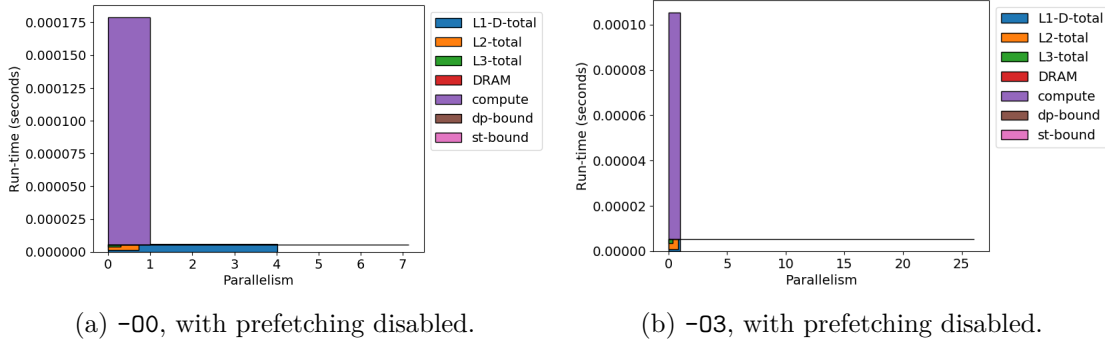


Figure 4.1.: Comparison of total MLP Stacks for the microbenchmark.

Listings 4.1 and 4.2 show the relevant instructions generated under the two optimization levels. Under -03, loop-control state is kept largely in registers, leaving the dependent

4. MLP Analysis

pointer accesses as the main loads in the loop. In contrast, `-O0` repeatedly reloads values such as the array base and loop index from the stack. These additional loads are independent of the pointer-chasing load and can therefore overlap with it under out-of-order execution. Consequently, the measured L1-D total parallelism increases to approximately four under `-O0`. The increase therefore reflects compiler-generated memory traffic rather than additional independent pointer chains.

Listing 4.1: Relevant disassembly of the pointer-chasing loop under `-O3`.

14e0:	mov	(%rax), %rdx	# p = array[0]
14e3:	add	\$0x8,%rax	
14e7:	mov	(%rdx), %rdx	# p = p->next
14ea:	mov	(%rdx), %rdx	# p = p->next
14ed:	mov	%rdx, -0x8(%rax)	# array[0] = p
14fa:	add	\$0x2, %r9d	# update loop counter
14fe:	cmp	%r10d, %r9d	# compare loop bound

Listing 4.2: Relevant disassembly of the pointer-chasing loop under `-O0`.

1525:	mov	-0x48(%rbp), %rax	# load array base
1529:	mov	-0x5c(%rbp), %edx	# load loop index
152f:	mov	(%rax,%rdx,8), %rax	# load array[j]
1533:	mov	(%rax), %rcx	# load p->next
1536:	mov	-0x48(%rbp), %rax	# reload array base
153a:	mov	-0x5c(%rbp), %edx	# reload loop index
1540:	mov	%rcx, (%rax,%rdx,8)	# store array[j]
1548:	mov	-0x5c(%rbp), %eax	# reload loop index

This experiment shows that the measured parallelism reflects the executed memory-access stream rather than the source-level dependence structure alone. More generally, higher observed parallelism does not necessarily imply better performance, since it may result from additional memory traffic rather than more effective latency overlap.

4.6.2. Memory-System Sensitivity and Interpretation

We next use GAPBS to examine how the measured MLP responds to changes in the memory system. BC is used as the representative workload for the detailed analysis. Additional results for BFS and PR, together with the SPMV software-prefetch-distance experiments, are provided in Appendix A.

Figure 4.2 first compares the original hierarchy-wide normalization with the revised activity-specific normalization under the same memory-system configuration. Under the original normalization, each parallelism metric is divided by $T(\text{Hier})$, which includes cycles when other parts of the memory hierarchy are active but the corresponding activity is absent. As shown in the left column of Figure 4.2, this produces not only a DRAM MLP below 1, but also HCLP values below 1 at L2 and L3. These values do not represent

the average concurrency during the corresponding active periods, because every such period necessarily contains at least one corresponding request. The average concurrency over the activity-specific interval should therefore be at least 1.

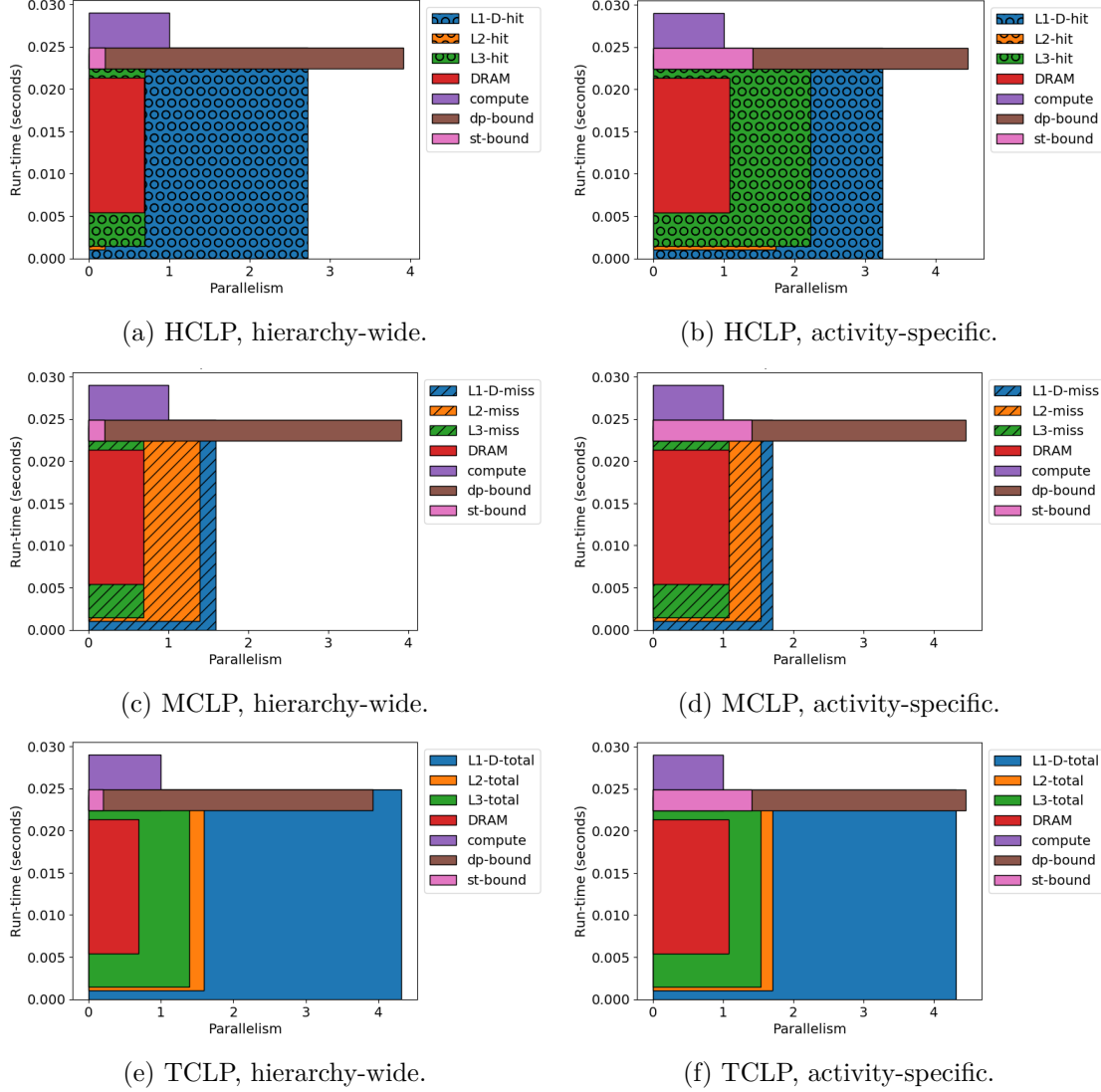


Figure 4.2.: Normalization comparison for BC with an 8 MB LLC and 0.95 GB/s DRAM bandwidth per controller. Columns show hierarchy-wide and activity-specific normalization; rows show HCLP, MCLP, and TCLP.

Using T_{DRAM} for DRAM MLP and the corresponding hit-, miss-, and access-active intervals for HCLP, MCLP, and TCLP restricts each denominator to the periods in which the measured activity is present. The revised metrics are therefore bounded below by 1 whenever the corresponding activity occurs and directly represent the average concurrency during those active periods, rather than hierarchy-wide amortized quantities.

4. MLP Analysis

Having revised the normalization, we next examine how MLP responds to cache capacity and memory bandwidth. Figure 4.3 compares BC across two LLC capacities and two DRAM bandwidths.

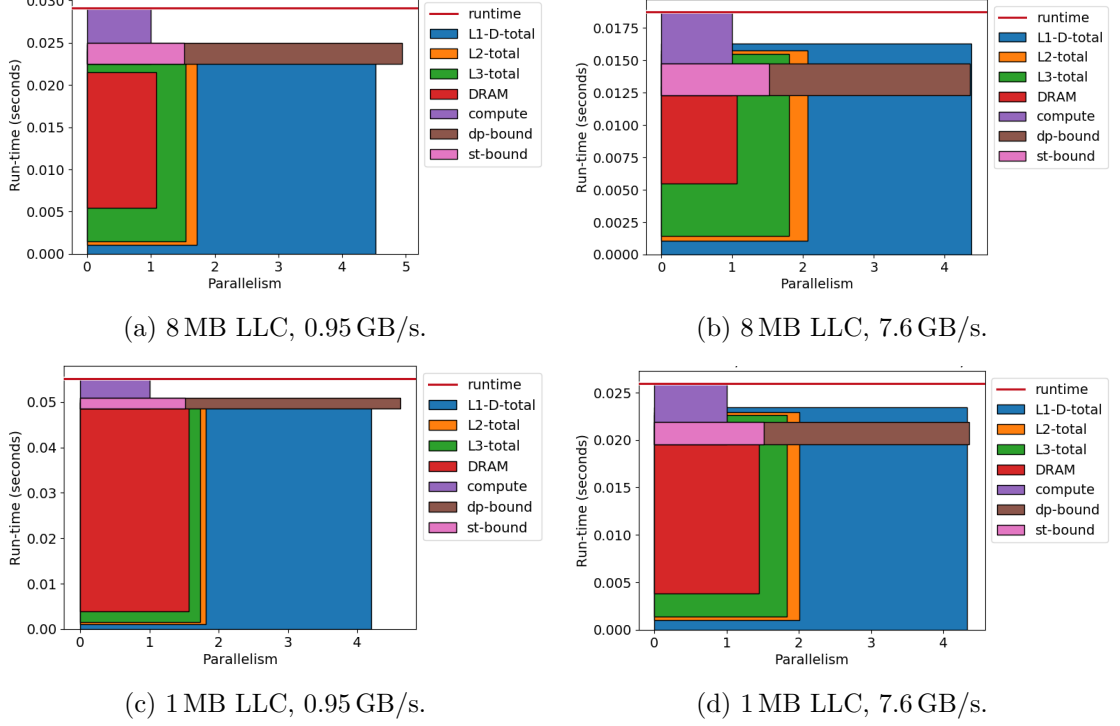


Figure 4.3.: MLP sensitivity to LLC capacity and DRAM bandwidth for BC. DRAM bandwidth is reported per controller. All configurations use activity-specific normalization.

At both LLC capacities, reducing DRAM bandwidth has little effect on the measured DRAM MLP. Although the lower-bandwidth configuration increases execution time and DRAM-active time, the average number of outstanding DRAM requests does not increase correspondingly. This suggests that reducing the memory service rate does not necessarily increase concurrency at the DRAM level. Instead, slower memory service primarily extends the time required to complete requests, while the average number of outstanding DRAM requests remains roughly unchanged.

Compared with DRAM MLP, CLP changes more noticeably with DRAM bandwidth. At both LLC capacities, reducing DRAM bandwidth decreases the CLP observed at L2 and L3. This indicates that the service capability of the lower memory system affects not only the completion time of DRAM requests, but also the number of accesses that can remain active concurrently at upper cache levels. Slower DRAM service may create backpressure toward the upper levels of the memory hierarchy and alter how requests progress across cache levels.

LLC capacity has a different effect. Reducing the LLC capacity from 8 MB to 1 MB produces a clear increase in DRAM MLP under both DRAM bandwidth configurations. This trend is consistent with a smaller LLC causing more memory accesses to reach DRAM, thereby increasing off-chip request concurrency. Overall, DRAM bandwidth and LLC capacity affect measured parallelism through different mechanisms. DRAM bandwidth primarily changes the rate at which requests are serviced and completed, while also affecting CLP through backpressure. LLC capacity more directly influences how many accesses proceed to DRAM. Notably, DRAM MLP remains broadly similar across different DRAM bandwidths even though execution time changes substantially. This further illustrates that MLP characterizes memory request concurrency, but does not directly capture memory-system service efficiency or overall execution performance.

4.6.3. Multicore and Prefetch Characterization

We next evaluate the multicore extension using BC executed as a multithreaded OpenMP workload across four simulated cores. Although the LLC and DRAM are shared, the extended framework attributes memory activity to the core that originates each request and reports a separate MLP Stack for each core.

Figure 4.4 shows the per-core total MLP Stacks without prefetching. The four cores exhibit broadly similar parallelism profiles, while their cache- and DRAM-level activity remains separately accounted for. This enables per-core characterization even when requests from different cores share the lower levels of the memory hierarchy.

Enabling prefetching changes both the amount and composition of outstanding memory activity, as shown in Figure 4.5. Prefetch-generated traffic appears alongside demand activity at multiple levels, while the framework preserves the corresponding per-core decomposition.

Table 4.3 summarizes the corresponding performance. Enabling prefetching reduces execution time from 8.012 to 6.497 ms (18.9%) and increases the mean per-core IPC from 0.98 to 1.17.

Table 4.3.: Performance of four-core BC with and without prefetching.

Configuration	Metric	Core 0	Core 1	Core 2	Core 3	Mean
Prefetch off	Runtime (ms)	8.012				
	IPC	0.95	0.99	1.04	0.94	0.98
Prefetch on	Runtime (ms)	6.497				
	IPC	1.22	1.14	1.12	1.19	1.17

When demand and prefetch activity are decomposed over the same level-active interval, an individual component can be below one because that component need not be present

4. MLP Analysis

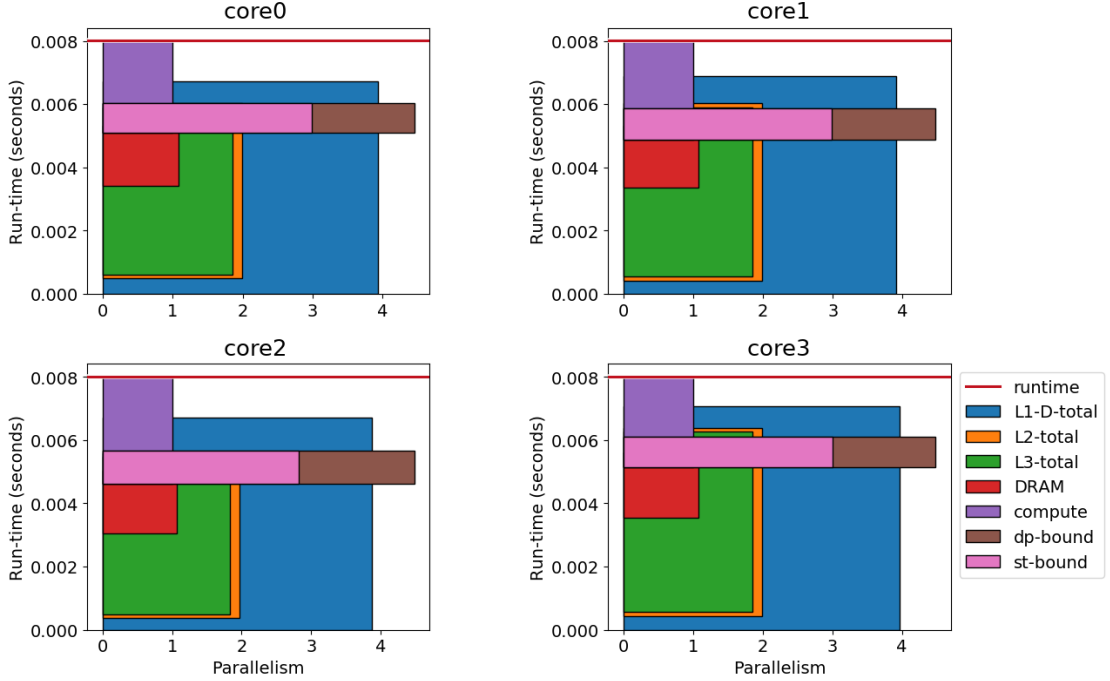


Figure 4.4.: Per-core total MLP Stacks for four-core BC without prefetching. All results use activity-specific normalization.

throughout the entire interval. In particular, the DRAM demand component may fall below one when some DRAM-active cycles contain only prefetch requests. This differs from the total DRAM MLP below one under the original hierarchy-wide normalization. Here, the value represents only the demand contribution to the total parallelism, while the total DRAM MLP still includes all outstanding DRAM requests and therefore remains at least one.

MLP characterizes the concurrency of outstanding memory activity, but not its direct performance impact. A higher measured MLP may arise from additional memory traffic or from request accumulation under constrained memory service, rather than from more effective latency hiding. In multicore execution, similar MLP profiles across cores may also hide differences in how memory delays affect each core’s performance. More fundamentally, MLP does not identify which requests cause delay to particular demand accesses or determine whether the resulting latency ultimately limits processor progress. These limitations motivate the request-level, source-aware performance attribution developed in the following chapter, which attributes memory-side interference to its responsible requests and evaluates whether its effects become visible at the ROB.

4.6. Results and Analysis

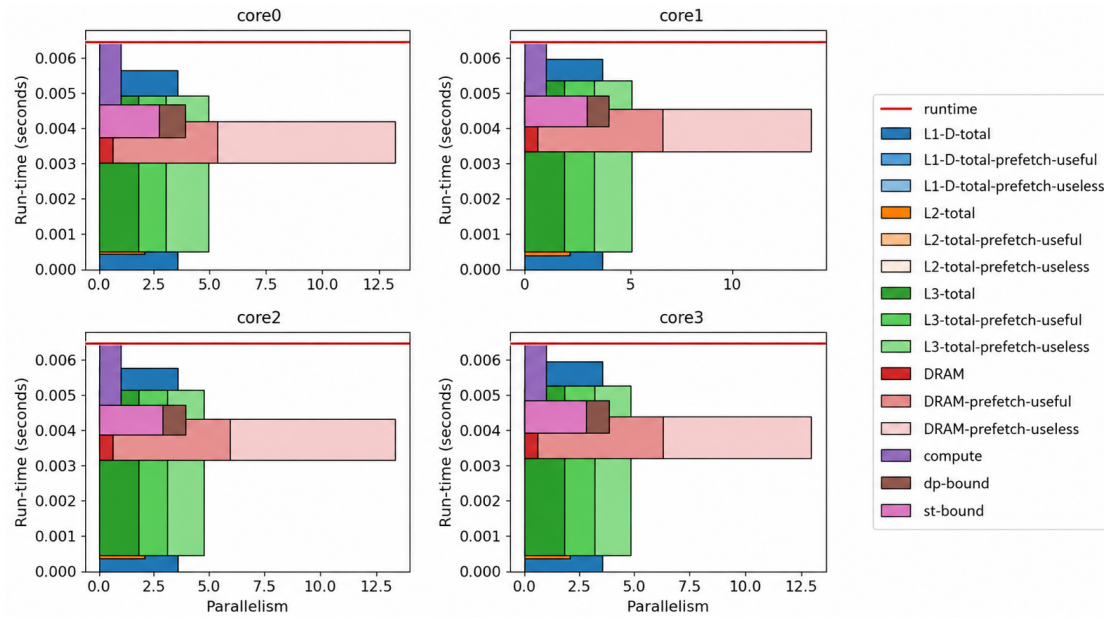


Figure 4.5.: Per-core total MLP Stacks for four-core BC with prefetching, including the decomposition of demand and prefetch activity.

Memory Interference and ROB-Visible Performance Modeling

5.1. Motivation and Overview

In multicore systems, demand and prefetch requests from the same or different cores compete for shared memory-system resources, potentially delaying individual demand requests. Such interference can increase demand-request latency through contention for shared resources or additional cache misses caused by prefetch-induced evictions. Characterizing this interference requires identifying where the additional delay arises, which requests contribute to it, and whether the resulting latency affects application (CPU) progress.

However, additional memory latency does not necessarily lead to a corresponding loss in application performance, as OoO execution can hide part of the delay by executing independent instructions. The ROB maintains in-flight instructions and ensures that they retire in program order. When an incomplete memory instruction reaches the ROB head, it can block instruction retirement. If the ROB becomes full, this blockage also prevents further instruction dispatch, limiting CPU execution progress. Therefore, we consider the ROB-visible impact of memory interference rather than relying solely on memory-side latency. Specifically, we model three sources of interference: L3 MSHR contention, DRAM contention, and L3 prefetch pollution. We then map the attributed memory-side delays to ROB-visible harm to estimate the extent to which this interference limits application progress.

5.2. Memory-Side Interference Modeling

We first quantify the additional latency experienced by each demand request due to shared-resource contention or prefetch-induced cache pollution. We refer to this additional latency as raw memory-side harm, without considering whether it limits application progress. The affected demand request is referred to as the *victim request*, while a request contributing to its interference is referred to as a *source request*. A source is classified as *self* when it originates from the same core as the victim and as *cross-core*

5. Memory Interference and ROB-Visible Performance Modeling

when it originates from another core. Each source is further classified as either demand or prefetch according to its request type. The model preserves these source categories together with the corresponding interference component for each attributed delay.

5.2.1. L3 MSHR Contention

In our model, an L3 miss requires an available MSHR entry before it can enter the miss-handling path. When all L3 MSHRs are occupied, an arriving demand miss must wait until an entry becomes available. For a victim request r , the raw MSHR blocking delay is

$$W_{\text{MSHR}}(r) = t_{\text{admit}}(r) - t_{\text{block}}(r), \quad (5.1)$$

where $t_{\text{block}}(r)$ is the time at which the request begins waiting for an MSHR and $t_{\text{admit}}(r)$ is the time at which it is admitted. As illustrated in Figure 5.1, the model tracks the MSHR occupants during this waiting period.

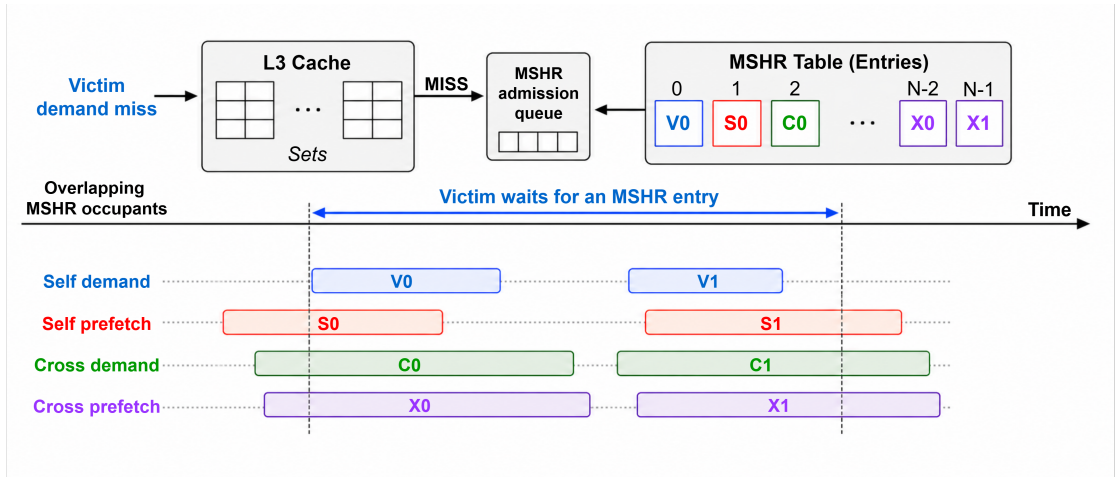


Figure 5.1.: L3 MSHR contention and occupancy-based attribution.

Let $O_{\text{MSHR}}(r, s)$ denote the occupancy overlap of source request s with the victim's wait. When multiple sources are present, the blocking delay is distributed in proportion to their overlap:

$$H_{\text{MSHR}}(r, s) = W_{\text{MSHR}}(r) \frac{O_{\text{MSHR}}(r, s)}{\sum_j O_{\text{MSHR}}(r, j)}. \quad (5.2)$$

The attribution retains the source core and whether the occupying request is a demand or prefetch request. This component captures interference before L3 MSHR admission. Delays encountered after admission are modeled separately.

5.2.2. DRAM Contention

We model DRAM interference at both the memory-controller and detailed DDR4 levels. A demand request may experience controller-level queueing before service and may subsequently be delayed by internal DDR4 resource and timing constraints. The model distinguishes five raw delay components:

$$\mathcal{C}_{\text{DRAM}} = \{\text{queue, bank, cmd, row, refresh}\}. \quad (5.3)$$

The queue component captures waiting at the memory-controller request queue. The remaining components capture delays associated with bank availability, DDR4 command-timing constraints, row-state conflicts, and refresh operations, respectively. Controller queueing and internal DDR4 timing delays are therefore treated as distinct components of DRAM interference.

For each victim demand request r , the model records its raw waiting time $W_c(r)$ for component c . For queue, bank, command, and row-state contention, it also identifies source requests whose relevant activity overlaps the victim's waiting period. Let $O_c(r, s)$ denote the overlap associated with source request s . When one or more valid sources are identified, the raw harm attributed to s is

$$H_c(r, s) = W_c(r) \frac{O_c(r, s)}{\sum_j O_c(r, j)}. \quad (5.4)$$

The equation distributes the observed waiting time according to each source's share of the total overlap. Source core and request type are retained, allowing self-demand, self-prefetch, cross-demand, and cross-prefetch interference to be distinguished. Any waiting time for which no valid source is identified remains unattributed.

Refresh-induced delay is recorded differently. A refresh operation can prevent a memory request from proceeding, but the refresh event is generated by the DRAM subsystem rather than by a demand or prefetch request from a particular core. It therefore has no natural request-level source and is generally retained as unattributed raw delay.

The resulting DRAM harm is accumulated by victim core, source core, source request type, and DRAM component. This preserves both where the interference is experienced and which core and request type are responsible for it.

5.2.3. L3 Prefetch Pollution

Prefetching can increase demand-access cost by evicting an L3 cache line that would otherwise have remained available. As shown in Figure 5.2, a pollution candidate is identified when a demand request misses in the actual L3 cache while the corresponding

5. Memory Interference and ROB-Visible Performance Modeling

line remains a permissible hit in the no-prefetch shadow cache and a live pollution record identifies a preceding prefetch eviction. The shadow state therefore provides evidence that, without the prefetch-induced eviction, the demand would have hit in L3.

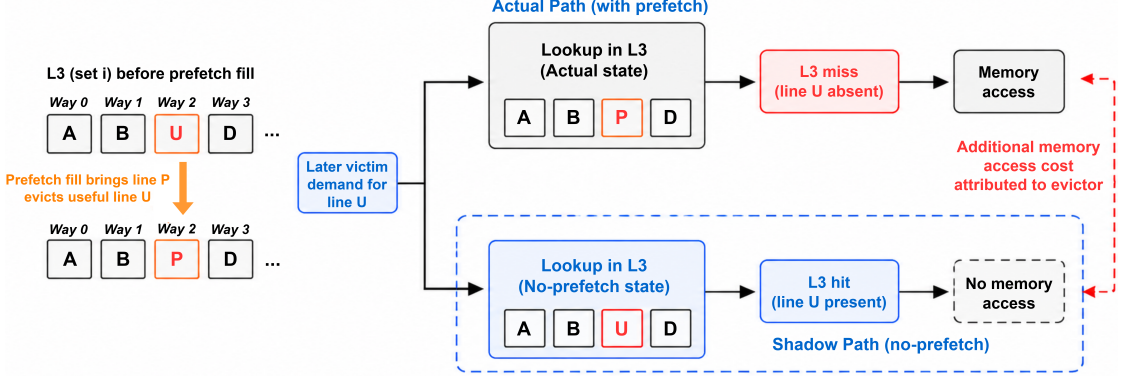


Figure 5.2.: L3 prefetch pollution and shadow-cache-based attribution.

For each pollution mark, the model records the expected cost of this counterfactual L3 hit using the modeled L3 tag-and-data access cost. After the demand miss completes, its miss-side base cost $C_{\text{miss}}(r)$ is reconstructed from the observed memory-service timing, excluding DRAM queueing delay because it is accounted for separately. These timing terms are used to estimate pollution harm rather than as independent pollution-attribution components.

For victim request r and prefetch evictor core b , let $N_{r,b}$ denote the number of pollution marks attributed to b , and let $C_{\text{L3,exp}}(r, b)$ denote their accumulated expected L3-hit cost. The raw pollution harm is

$$H_{\text{pollution}}(r, b) = \max(0, N_{r,b} C_{\text{miss}}(r) - C_{\text{L3,exp}}(r, b)). \quad (5.5)$$

For a single pollution mark, this corresponds directly to the additional cost of an observed miss relative to the L3 hit expected without pollution. The resulting harm is attributed to the recorded evictor core, preserving both same-core and cross-core prefetch-pollution effects.

5.3. ROB-Visible Performance Modeling

Memory-side interference does not necessarily translate into performance loss, since OoO execution and MLP can hide part or all of the additional latency. The raw harm measured in the memory system therefore needs to be mapped to its impact on processor execution.

5.3. ROB-Visible Performance Modeling

A demand request may accumulate interference across multiple memory-system components before the corresponding instruction reaches the ROB head. For example, it may experience L3 MSHR contention, DRAM contention, or L3 prefetch pollution earlier in its execution. Observing only the memory-system state when the request eventually blocks the ROB head would therefore lose information about the sources of the delay.

To preserve this information, our proposed model maintains the attributed harm separately for each victim request throughout its execution. Each request retains the interference component, source core, and source request type associated with its memory-side delay. These request-level harm components are subsequently mapped to the ROB behavior of the same instruction.

The mapping is performed separately for each victim request using its own ROB-visible scaling factor. The factor reflects the fraction of the instruction’s lifetime during which it stalls at the ROB head while the ROB is full, thereby capturing how strongly delays affecting that request are exposed to processor execution. Different requests therefore receive different scaling factors according to their individual ROB behavior.

We scale the raw harm from different memory-system components of the same request by the same factor because these components ultimately delay the same victim instruction. The memory-side attribution already captures their source and component distinctions, while the scaling factor characterizes the visibility of the affected request as a whole. This design preserves fine-grained request-level attribution without requiring a separate ROB-stall attribution mechanism for each memory-system component.

For a memory instruction i , its lifetime is defined as

$$L(i) = t_{\text{commit}}(i) - t_{\text{issue}}(i), \quad (5.6)$$

where $t_{\text{issue}}(i)$ is the time at which instruction i is issued, and $t_{\text{commit}}(i)$ is the time at which it commits from the ROB.

Within this lifetime, let $A(i)$ denote the total ROB-head stall time associated with instruction i while the ROB is full, as shown in Figure 5.3. The ROB-full condition ensures that only head stalls that block further instruction dispatch are considered visible. The ROB-visible scaling factor is then

$$\alpha(i) = \frac{A(i)}{L(i)}. \quad (5.7)$$

The factor $\alpha(i)$ captures the visibility of the affected instruction, while the magnitude of each interference component is already represented by its raw harm. A component that introduces a larger delay therefore retains a larger contribution under the same scaling factor, without requiring a component-specific adjustment.

5. Memory Interference and ROB-Visible Performance Modeling

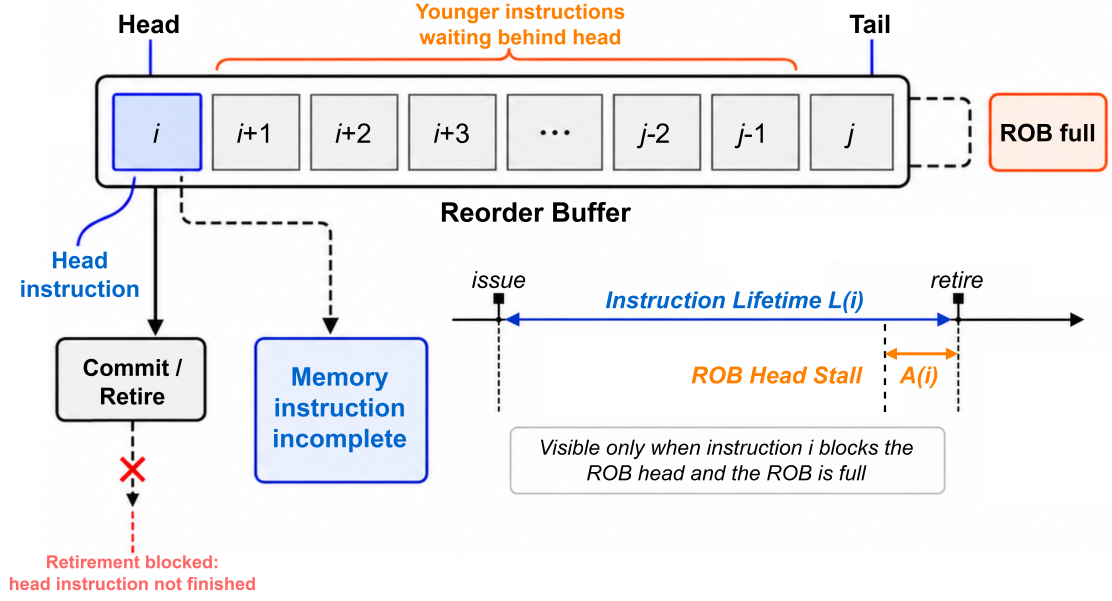


Figure 5.3.: ROB-head full stall and instruction-level visibility.

For raw harm $H_k(i, c)$ attributed to source core c through interference component k , the corresponding ROB-visible harm is

$$H_k^{\text{visible}}(i, c) = \alpha(i) H_k(i, c). \quad (5.8)$$

If $\alpha(i) = 0$, none of the memory-side harm is retained as ROB-visible harm. The delay is either hidden by OoO execution and MLP or does not cause the ROB head to stall while the ROB is full. As $\alpha(i)$ increases, a larger fraction of the raw harm is retained. The source core, request type, and interference component identified by the memory-side model are preserved after scaling.

This scaling provides a low-complexity mapping from memory-side harm to ROB-visible impact. It does not assign individual ROB-stall cycles to specific interference components. Instead, each request-level harm component is weighted by the visibility of the corresponding instruction. The sum of the scaled harm components therefore does not necessarily equal the measured ROB-head stall time.

5.4. Implementation

We implement the proposed memory interference and ROB-visible performance models in Sniper by extending its cache, DRAM, and ROB timing models. The MLP analysis in the previous chapter uses Sniper's normal DRAM model with the `history_list` queue

model. This configuration represents DRAM accesses using an aggregate access latency and bandwidth-related queueing delay. It is sufficient for characterizing the concurrency of outstanding memory requests, but does not explicitly model DDR4 behaviors such as bank availability, command timing, row-buffer state, and refresh.

In this chapter, we use Sniper’s detailed DDR4 model and configure the DRAM queue model as **contention**. The detailed model explicitly represents the DDR4 organization and timing behavior required by the component-level interference attribution described earlier. The main DDR4 parameters used in the experiments are summarized in Table 5.1. We further extend the model to record per-request delays across different DRAM components together with the source information required for source-aware interference attribution.

Table 5.1.: Detailed DDR4 configuration used for interference analysis.

Parameter	Setting
DRAM speed	1600 MT/s
Data bus width	64 bits
Data-bus bandwidth	12.8 GB/s
Controllers / channels	1 / 1
Ranks / banks / bank groups	2 / 8 / 4
DRAM page size	128 cache lines
Address mapping	XOR
Row policy	Open row
Base latency	45 ns
Bank open / close delay	20 / 20 ns
Access cost	20 ns
Inter-command delay	7 ns
Long / short command delay	7 / 4 ns
Controller delay	20 ns
Bank keep-open interval	120 ns
Refresh interval	64 ms
Refresh length	15 ns

Memory-side events are associated with processor execution through request identifiers carried by cache-line memory transactions. Raw harm generated in the L3 cache and DRAM is accumulated in corresponding per-request records and processed at instruction commit by the ROB-side model. Each record preserves the victim core, source core, source request type, and interference component. For a memory instruction that spans multiple cache lines, the line-level attribution records are combined before ROB-side processing by retaining the larger contribution for each component. This avoids double counting memory delays that may overlap.

Sniper does not explicitly model a finite-capacity L3 MSHR structure, and its interval-based timing model does not directly provide the actual release time of each MSHR entry. We therefore implement a virtual L3 MSHR admission model in the shared L3

5. Memory Interference and ROB-Visible Performance Modeling

cache controller. Each L3 miss occupies one virtual entry. When all entries are occupied, a new request waits for the earliest available entry, and the resulting admission blocking delay is directly added to the simulated execution time.

Because the completion time of a request is not known at admission, the virtual L3 MSHR model predicts how long the request will occupy its entry. Each admitted request is classified into one of four service classes: directory demand, directory prefetch, direct-DRAM demand, and direct-DRAM prefetch. For the two directory classes, the initial service-time estimate is the configured directory cache access latency plus the configured DRAM latency. For the two direct-DRAM classes, it is initialized using the configured DRAM latency alone.

After a request completes, the service-time estimate of its class is updated using the memory-service time observed by Sniper. The update follows a piecewise exponential moving average (EMA),

$$\hat{T}_x^{(n)} = (1 - \beta_n)\hat{T}_x^{(n-1)} + \beta_n T_x^{(n)}, \quad (5.9)$$

where $T_x^{(n)}$ is the observed service time of the n th completed request in service class x , and

$$\beta_n = \begin{cases} 1, & n = 1, \\ \frac{1}{4}, & 2 \leq n \leq 8, \\ \frac{1}{16}, & n > 8. \end{cases} \quad (5.10)$$

The resulting class-specific estimate is used at admission to predict the release time of the virtual MSHR entry,

$$t_{\text{release}}^{\text{pred}} = t_{\text{admit}} + \hat{T}_x. \quad (5.11)$$

When a victim request waits because all virtual MSHR entries are occupied, these predicted release times determine which requests occupy the entries during the waiting interval. The blocking delay is then attributed using the occupancy-overlap model described above while preserving the source core and request type.

DRAM attribution is integrated directly into the detailed DDR4 timing model. For each demand request, the implementation separately records delays caused by controller queueing, bank availability, command timing, row-state conflicts, and refresh. It also maintains information about active requests and resource occupancy for source attribution. Delays that can be associated with other memory requests are further classified as same-core or cross-core interference and as demand- or prefetch-induced interference. Refresh delay is generated by the DRAM subsystem itself and therefore usually cannot be associated with a specific source request. It is consequently retained as unattributed

harm. DRAM bus and device service times are not treated as independent interference components. In the interference-harm model, they are used to reconstruct the pollution miss cost.

L3 prefetch pollution is identified using a no-prefetch shadow cache. The shadow cache has the same capacity, associativity, block size, and replacement policy as the real L3 cache, but its cache and replacement states are updated only by demand accesses. Prefetch fills therefore do not change the shadow state. When a prefetch fill evicts a demand-owned line from the real L3 cache, the implementation records the source information for that eviction. Generation validation and expiration are used to prevent stale eviction records from being incorrectly associated with later accesses. A later demand miss is identified as prefetch pollution only if the corresponding line would still hit in the no-prefetch shadow cache and a valid prefetch eviction record exists. The raw pollution harm is then computed from the expected L3-hit cost and the actual miss-side cost and attributed to the source core responsible for the eviction.

ROB-visible harm is computed at instruction commit. For each relevant memory instruction, the model records its issue time, first arrival at the ROB head, completion time, and the accumulated time for which it stalls the ROB head while the ROB is full. Memory-side raw harm is scaled by the fraction of the issue-to-commit lifetime spent in ROB-head stall while the ROB is full. The same visibility factor is applied to the different harm components of the instruction, while their raw magnitudes preserve their relative contributions.

Finally, the raw and ROB-visible harm measurements are aggregated into window-level statistics. The aggregated results preserve the totals for each interference component together with victim–source core pairwise information. These measurements provide the interference signals used by the adaptive prefetch management mechanism presented in the next chapter, while the specific throttling policy is kept separate from the attribution implementation described here.

5.5. Results and Analysis

We evaluate the proposed interference and ROB-visible performance models using two benchmarks. A controlled microbenchmark is used to validate L3 MSHR contention attribution, while GAPBS SSSP is used to evaluate the complete model under a multi-programmed multicore setting. Table 5.2 summarizes the benchmark-specific configurations.

5.5.1. Microbenchmark Validation

We first use a controlled two-core victim–aggressor microbenchmark to validate the interference attribution model. Core 0 acts as the victim and repeatedly scans a 4 MB

5. Memory Interference and ROB-Visible Performance Modeling

Table 5.2.: Benchmark configurations used for interference and ROB-visible analysis.

Benchmark	Parameter	Setting
Microbenchmark	Victim	Sequential scan, 4 MB working set, 128 iterations
	Aggressor	16 interleaved streams, 64 MB working set
	Shared L3 cache	4 MB, 16-way, LRU
	L3 MSHR model	16 entries
	Compilation	-O3
GAPBS SSSP	Input graph	Kronecker graph, scale 16 (2^{16} vertices)
	Average degree	16
	Trials	1
	Execution	Four independent processes, one per core
	Shared L3 cache	4 MB, 16-way, LRU
	L3 MSHR model	ON
	Compilation	-O3

working set for 128 iterations. Its regular access pattern provides stable data reuse and useful prefetching opportunities. Core 1 acts as the aggressor and accesses a 64 MB region using 16 independent streams. Each stream accesses eight consecutive cache lines within a page before moving to the next page, generating sustained memory and prefetch traffic. The aggressor continues running until the victim completes, exposing Core 0 to shared-resource contention throughout its execution. The experiment uses a 4 MB shared L3 cache and limits the modeled L3 MSHR capacity to 16 entries to increase L3 MSHR contention under sustained memory traffic.

This configuration provides a clear expected interference source. With prefetching disabled, DRAM and L3 MSHR interference is generated only by demand requests. With prefetching enabled, the streaming workload on Core 1 generates additional prefetch traffic and is expected to become the primary source of prefetch-induced interference on Core 0. Comparing the two configurations therefore allows us to validate both source-core and source-request-type attribution.

Figures 5.4 and 5.5 show the attribution of DRAM contention and L3 MSHR contention, respectively. The vertical axis denotes the victim core and the horizontal axis the source core. Diagonal entries represent self-interference, while off-diagonal entries represent cross-core interference. Demand-to-demand and prefetch-to-demand distinguish harm caused by demand and prefetch requests, respectively. For each victim core, the attributed harm is normalized across source cores. The top and bottom rows show memory-side and ROB-visible attribution, respectively. The same representation is used for the remaining attribution results.

Core 0 experiences substantial cross-core demand interference even without prefetching.

5.5. Results and Analysis

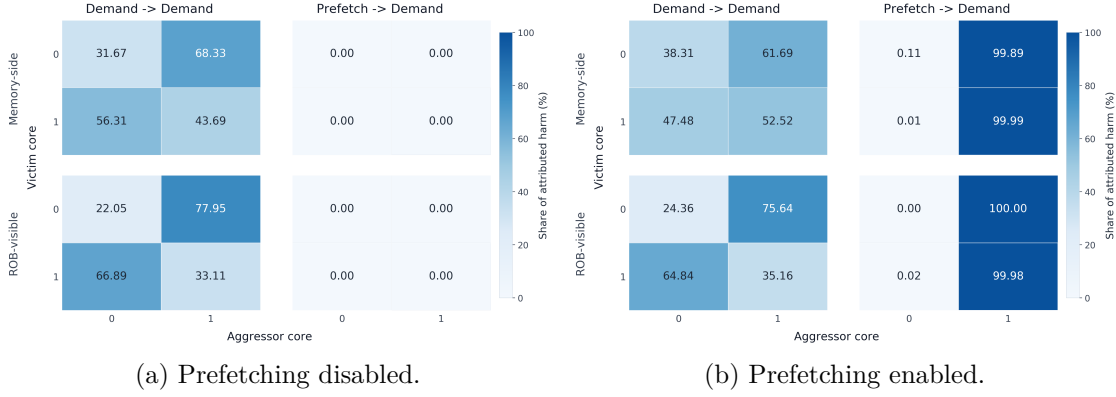


Figure 5.4.: DRAM interference attribution for the microbenchmark. The top and bottom rows show memory-side and ROB-visible attribution, respectively, while the left and right columns show demand-to-demand and prefetch-to-demand interference.

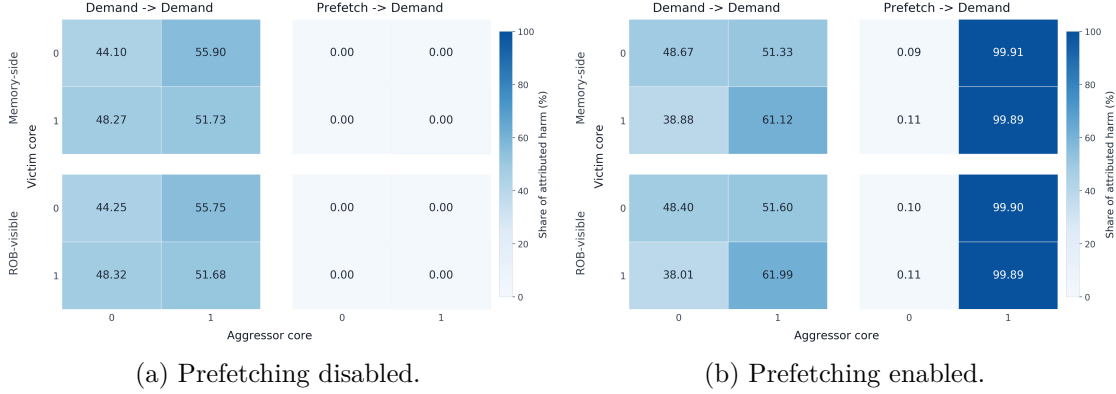


Figure 5.5.: L3 MSHR contention attribution for the microbenchmark. The top and bottom rows show memory-side and ROB-visible attribution, respectively, while the left and right columns show demand-to-demand and prefetch-to-demand interference.

For DRAM, 68.33% of its memory-side demand harm is attributed to Core 1, increasing to 77.95% in the ROB-visible attribution. L3 MSHR contention has a more balanced source distribution, with Core 1 contributing 55.90% and 55.75% of the memory-side and ROB-visible harm to Core 0, respectively. Demand requests from Core 1 therefore interfere with Core 0 through both DRAM and L3 MSHRs, although the two resources exhibit different source distributions.

With prefetching enabled, the prefetch-induced interference experienced by Core 0 is almost entirely attributed to Core 1. For DRAM, Core 1 contributes 99.89% of the memory-side prefetch-induced harm and 100% of the ROB-visible harm. L3 MSHR con-

5. Memory Interference and ROB-Visible Performance Modeling

tention shows the same pattern, with corresponding shares of 99.91% and 99.90%. The demand-to-demand attribution remains broadly similar to the no-prefetch configuration. Core 1 contributes 61.69% and 75.64% of the memory-side and ROB-visible DRAM demand harm to Core 0, while the corresponding L3 MSHR shares are 51.33% and 51.60%. The model therefore distinguishes prefetch-induced interference from demand interference and attributes it to the core generating the prefetch requests.

The prefetch-to-demand DRAM and L3 MSHR interference experienced by Core 1 is also almost entirely attributed to its own prefetch requests. The aggressor’s prefetch traffic therefore creates both cross-core interference on Core 0 and self-interference on its own demand requests.

Figure 5.6 shows the attribution of L3 prefetch pollution. For Core 0, Core 1 accounts for 99.55% of the memory-side pollution harm and 97.02% of the ROB-visible pollution harm. This behavior is consistent with the cache reuse characteristics of the two workloads. Core 0 repeatedly accesses the same 4 MB working set, while the shared L3 cache is also 4 MB, giving evicted data a high probability of being reused in later iterations. Core 1 instead continuously streams through a much larger 64 MB region with substantially less reuse of previously accessed data. Its prefetch fills can therefore evict victim data that will be reused later, while producing little self-prefetch pollution.

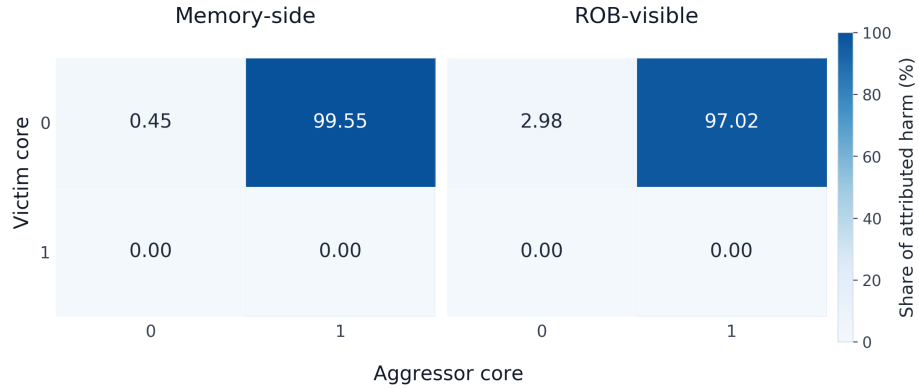


Figure 5.6.: L3 prefetch pollution attribution for the microbenchmark. The left and right panels show memory-side and ROB-visible attribution, respectively.

Figure 5.7 summarizes the overall self- and cross-core harm. Core 0 is dominated by cross-core harm in both configurations. Without prefetching, its memory-side and ROB-visible cross-core shares are 59.2% and 64.5%, respectively, compared with 57.4% and 57.2% with prefetching enabled. Core 1 shows a stronger self-interference contribution, reaching 61.8% at the memory side and 62.6% in the ROB-visible attribution when prefetching is enabled. These results are consistent with Core 0 acting as the victim and Core 1 generating sustained demand and prefetch traffic.

Figure 5.8 directly compares the memory-side and ROB-visible harm and their component breakdowns. In both configurations, the memory-side harm is much larger than the

5.5. Results and Analysis

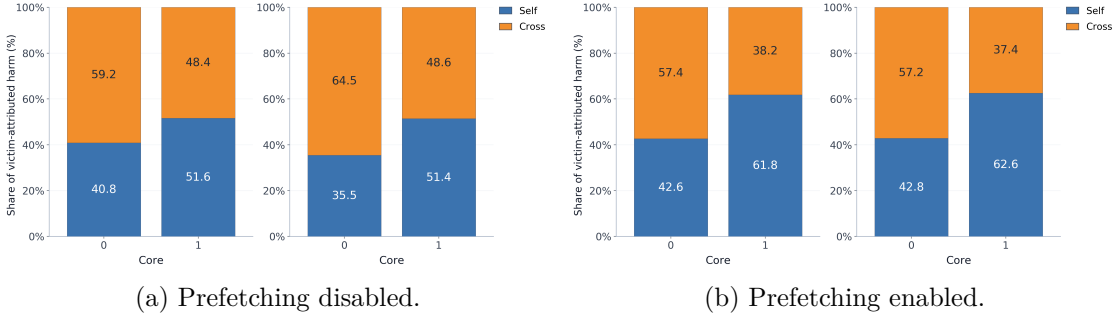


Figure 5.7.: Self- and cross-core shares of attributed harm for the microbenchmark. In each panel, the left and right subfigures show memory-side and ROB-visible attribution, respectively.

corresponding ROB-visible harm, indicating that most raw memory-side interference is hidden by OoO execution and MLP rather than reaching the ROB critical path. The reduction also differs across interference components. DRAM interference accounts for most of the memory-side harm, but a large fraction of it is hidden in the ROB-visible results. In contrast, L3 MSHR contention retains a larger ROB-visible contribution and becomes the dominant component of ROB-visible harm. This suggests that L3 MSHR availability is a more performance-critical resource for this workload under the evaluated configuration. The attribution therefore provides direct guidance for optimization, motivating mechanisms such as increasing L3 MSHR capacity or otherwise allowing more outstanding misses to be accommodated. Additionally, this difference shows that raw memory-side harm alone does not accurately reflect the performance impact of different interference mechanisms.

Figure 5.9 places the attributed ROB-visible harm in the context of the total ROB-head stall. The colored regions represent ROB-visible harm attributed to the modeled memory interference, while the gray region represents the remaining ROB-head stall not attributed to these interference mechanisms. In both configurations, the attributed memory interference accounts for only a small portion of the total ROB-head stall, indicating that the modeled interference explains only a limited portion of the observed ROB-head stall.

Overall, the microbenchmark results are consistent with the intended victim–aggressor behavior. With prefetching enabled, Core 1 is the dominant source of prefetch-induced DRAM and L3 MSHR interference on Core 0 and is responsible for almost all of its L3 prefetch pollution. The comparison between memory-side and ROB-visible harm further shows that a large fraction of memory-side interference is hidden by OoO execution and MLP. Together, these results validate the source-aware interference attribution and ROB-visible harm model.

5. Memory Interference and ROB-Visible Performance Modeling

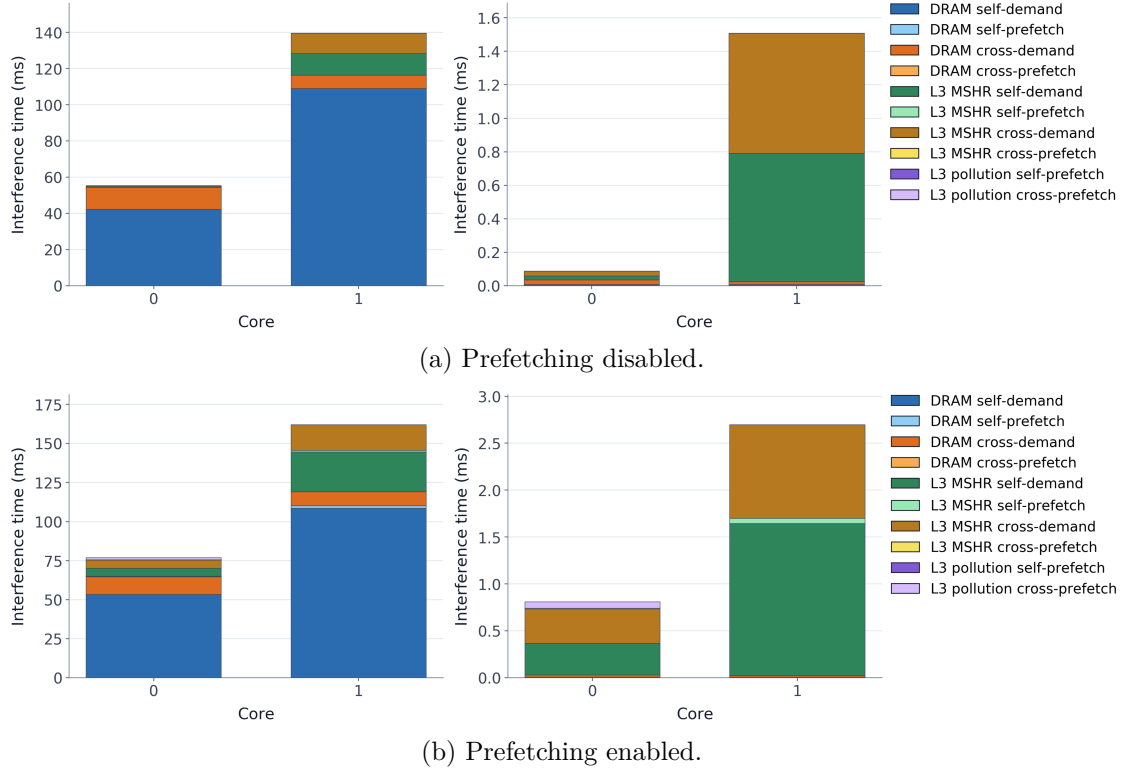


Figure 5.8.: Memory-side and ROB-visible interference breakdown for the microbenchmark. The left and right panels show memory-side and ROB-visible interference, while the upper and lower rows show results with prefetching disabled and enabled, respectively.

5.5.2. GAPBS SSSP Case Study

We next apply the attribution model to a homogeneous multiprogrammed workload consisting of four independent SSSP processes, each running on one core with the same benchmark and input. Figures 5.10 and 5.11 show the DRAM and L3 MSHR attribution using the same representation introduced in the previous subsection. L3 prefetch pollution with prefetching enabled is shown separately in Figure 5.12.

The source distributions are broadly similar across the four SSSP processes, with no single source core consistently dominating the interference in either prefetching configuration. Without prefetching, DRAM interference has a stronger cross-core contribution, whereas L3 MSHR contention shows a larger self-core contribution relative to DRAM. With prefetching enabled, both demand-to-demand and prefetch-to-demand interference remain reasonably balanced across the four cores, and L3 prefetch pollution shows a similarly balanced source distribution. Figure 5.13 further shows that cross-core harm accounts for the majority of the overall attributed interference. Unlike the asymmet-

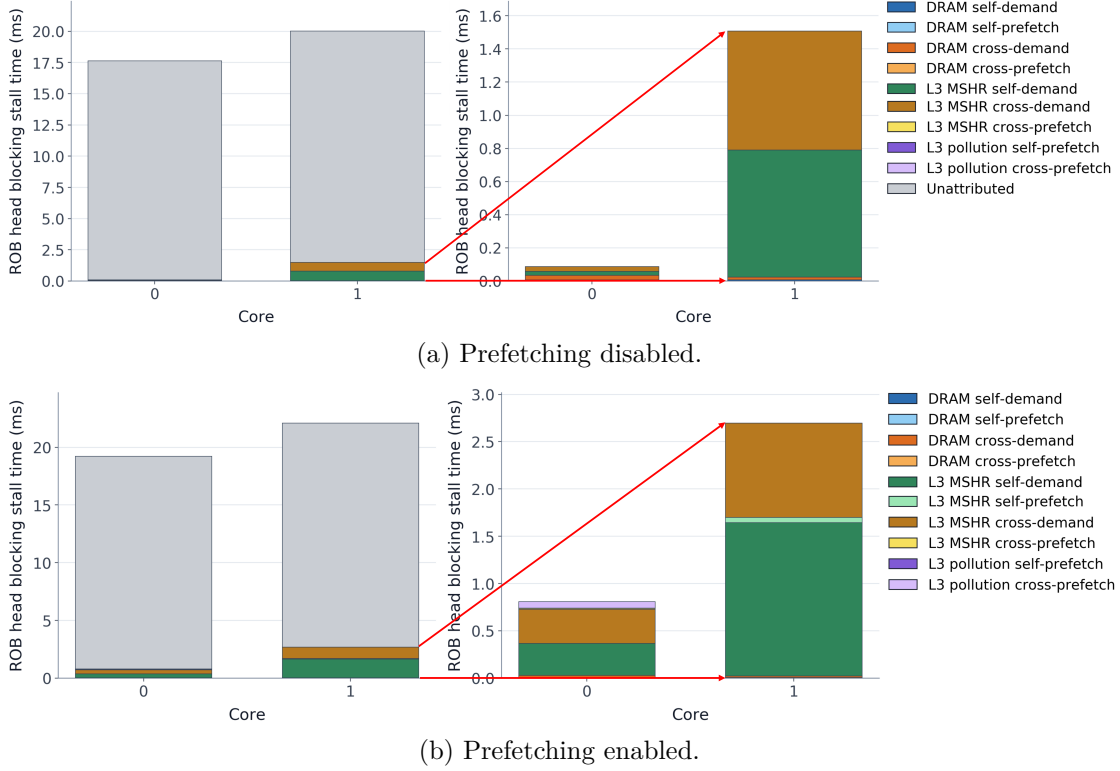


Figure 5.9.: ROB-head stall breakdown for the microbenchmark. The left panels show total ROB-head blocking stall time, while the right panels show the ROB-visible attributed harm in detail. Colored regions denote harm attributed to the modeled memory interference, and gray denotes the remaining unattributed stall.

ric victim–aggressor microbenchmark, SSSP therefore exhibits no dominant interference source, consistent with the four processes executing the same workload and input.

Figures 5.14 and 5.15 further show the difference between memory-side interference and its impact at the ROB. Consistent with the microbenchmark results, the memory-side harm is much larger than the corresponding ROB-visible harm, indicating that a large fraction of raw interference is hidden by OoO execution and MLP. At the same time, the attributed memory interference accounts for only a portion of the total ROB-head stall. These results further show that memory-side interference magnitude alone does not accurately reflect its performance impact.

These results motivate the adaptive prefetch throttling developed in the next chapter. ROB-visible harm captures the performance-relevant cost introduced by prefetching, but harm alone cannot determine whether prefetching should be throttled because useful prefetches may also reduce demand-access latency. The next chapter therefore introduces a complementary ROB-visible prefetch benefit model and combines the two signals to

5. Memory Interference and ROB-Visible Performance Modeling

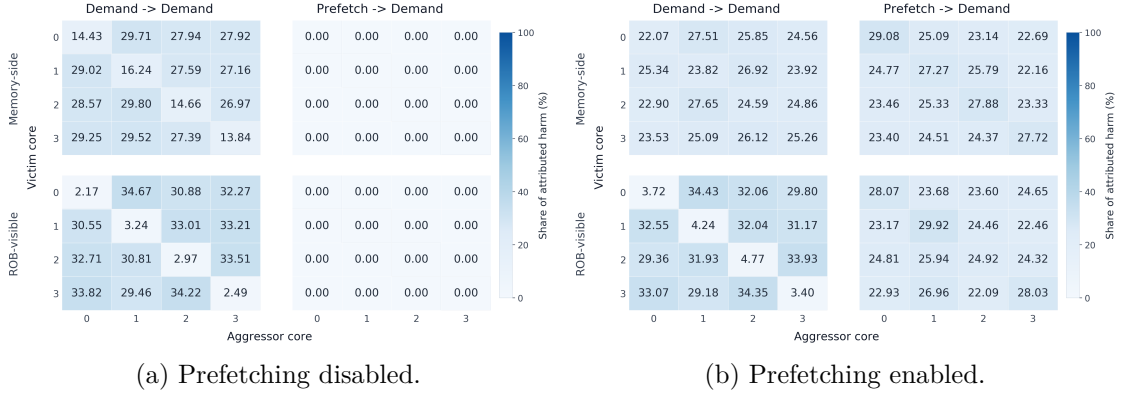


Figure 5.10.: DRAM interference attribution for GAPBS SSSP. The top and bottom rows show memory-side and ROB-visible attribution, respectively, while the left and right columns show demand-to-demand and prefetch-to-demand interference.

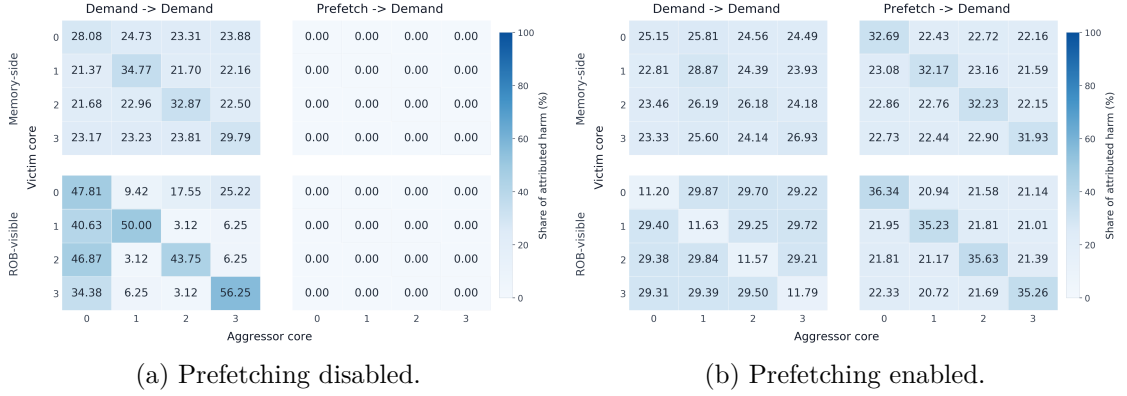


Figure 5.11.: L3 MSHR contention attribution for GAPBS SSSP. The top and bottom rows show memory-side and ROB-visible attribution, respectively, while the left and right columns show demand-to-demand and prefetch-to-demand interference.

control prefetch aggressiveness.

5.5. Results and Analysis

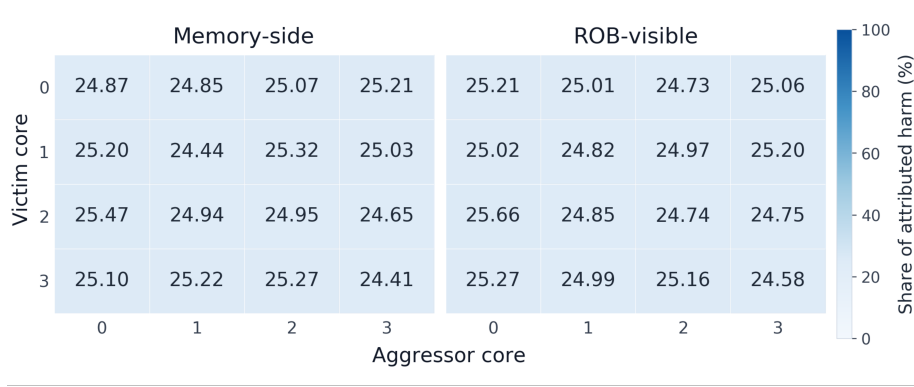


Figure 5.12.: L3 prefetch pollution attribution for GAPBS SSSP with prefetching enabled. The left and right panels show memory-side and ROB-visible attribution, respectively.

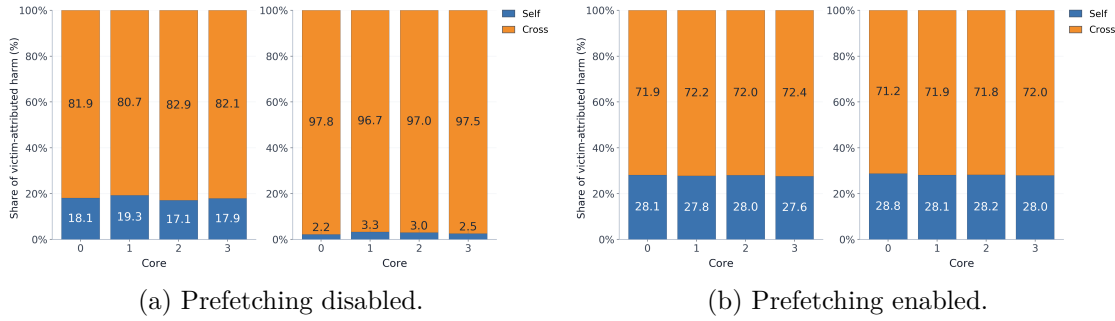


Figure 5.13.: Self- and cross-core shares of attributed harm for GAPBS SSSP. In each panel, the left and right subfigures show memory-side and ROB-visible attribution.

5. Memory Interference and ROB-Visible Performance Modeling

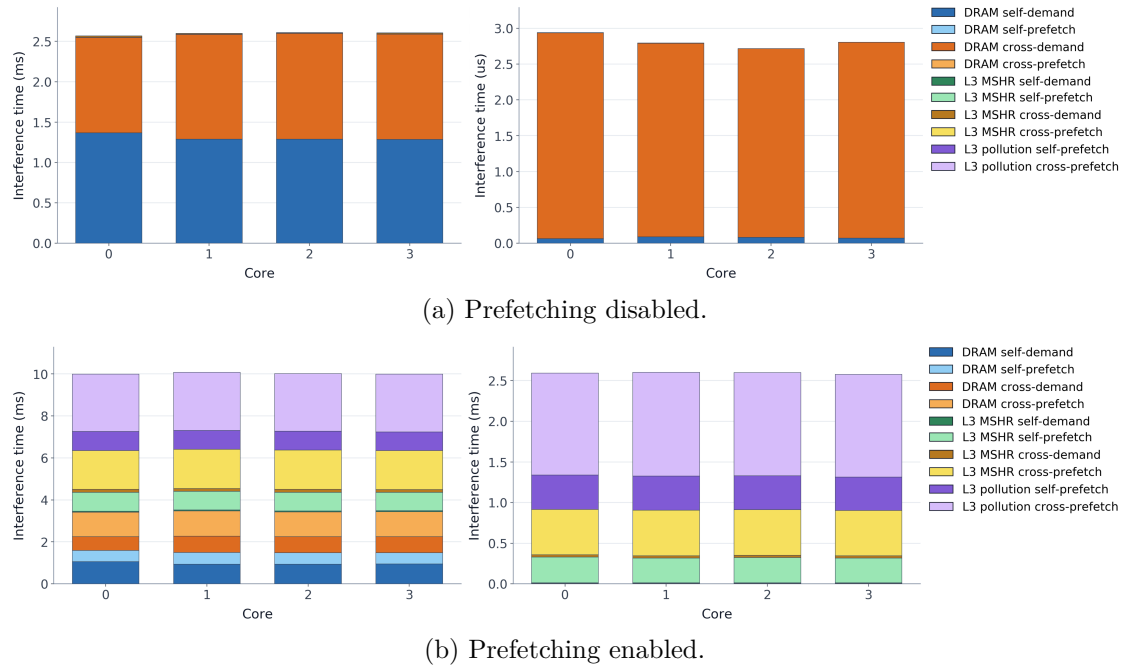
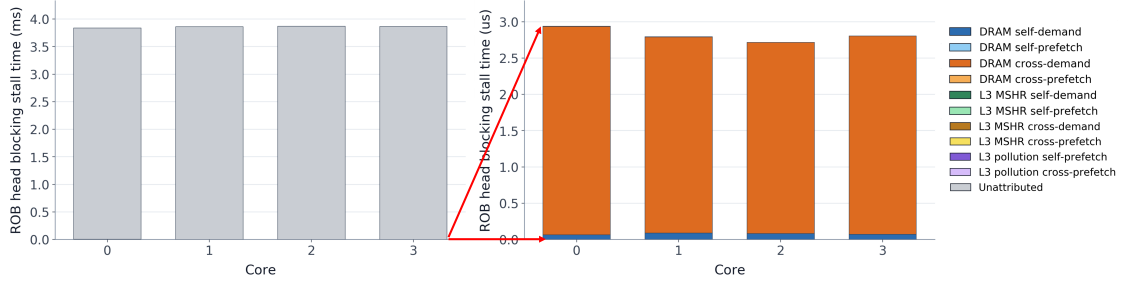
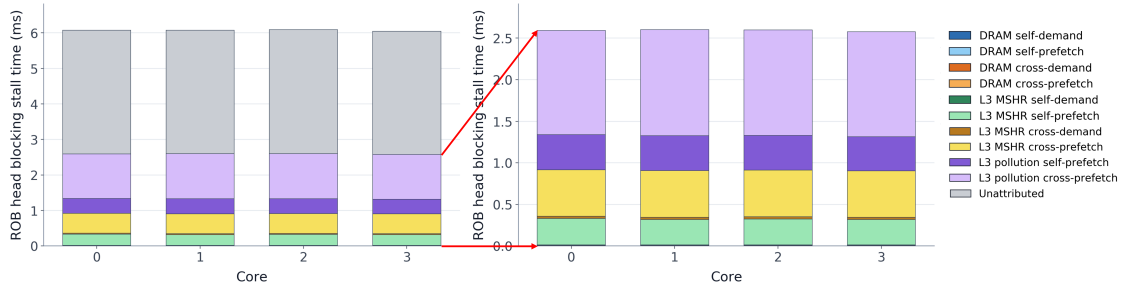


Figure 5.14.: Memory-side and ROB-visible interference breakdown for GAPBS SSSP. The left and right panels show memory-side and ROB-visible interference, while the upper and lower rows show results with prefetching disabled and enabled, respectively.



(a) Prefetching disabled.



(b) Prefetching enabled.

Figure 5.15.: ROB-head stall breakdown for GAPBS SSSP. The left panels show total ROB-head blocking stall time, while the right panels show the ROB-visible attributed harm in detail. Colored regions denote harm attributed to the modeled memory interference, and gray denotes the remaining unattributed stall.

Adaptive Prefetch Throttling

6.1. Motivation and Overview

Prefetching can reduce demand-access latency by bringing data into the cache before it is requested. However, aggressive prefetching also increases memory traffic and resource contention, and these costs become more significant when the prefetcher fails to identify the access pattern effectively or generates prefetches with limited usefulness. A fixed prefetch aggressiveness therefore cannot adapt to changes in workload behavior and may continue consuming shared memory resources even when the additional prefetch traffic provides little benefit.

The ROB-visible interference model developed in the previous chapter captures the performance-relevant cost of memory interference, including interference introduced by prefetch requests. Prefetch throttling should also account for the latency reduction provided by useful prefetches, since this benefit may offset the interference cost. We therefore use ROB-visible harm together with a complementary prefetch benefit signal to guide prefetch aggressiveness at runtime.

Existing utility-driven mechanisms estimate this benefit using useful prefetch statistics and average memory latency. CAFFEINE estimates the processor cycles saved by useful prefetches from the number of prefetch hits and the average LLC miss penalty ([Panda and Balachandran, 2015](#)). I-POP separately counts useful prefetches served by the LLC and DRAM and weights these counts by their corresponding average access latencies to estimate prefetch benefit ([Lin et al., 2026](#)). These approaches provide quantitative estimates of prefetch benefit, but the estimated latency reduction is derived from aggregate statistics and average latency rather than from the particular demand request that consumes a prefetched line.

Our model instead associates each useful prefetch with the demand instruction that consumes the prefetched line. A no-prefetch shadow cache determines whether that demand would have missed without prefetching, allowing the raw benefit to reflect the service cost avoided for the specific consuming demand rather than an estimate based on aggregate usefulness and average latency. The model further evaluates whether this latency reduction would become visible at the ROB. Latency reduction hidden by OoO execution and MLP does not contribute to ROB-visible benefit, while the portion that

6. Adaptive Prefetch Throttling

would otherwise be exposed on the ROB critical path is retained. The resulting benefit signal therefore captures both the request-level latency reduction provided by a useful prefetch and its ROB-visible impact.

The proposed adaptive throttling mechanism combines this ROB-visible benefit with ROB-visible harm. Prefetch aggressiveness is maintained when its benefit is sufficient relative to the interference cost and reduced when the benefit becomes limited or the interference cost increases. The remainder of this chapter first describes the prefetch benefit model and then presents the adaptive throttling policy and its implementation.

6.2. Prefetch Benefit Modeling

Prefetch benefit is more difficult to observe directly than interference harm. Memory-side harm corresponds to delays that occur in the observed execution and can therefore be measured as they are experienced. In contrast, prefetch benefit depends on how much later the demand would have completed without the prefetch, which is not directly observable in the actual execution. We therefore use counterfactual analysis to estimate the latency reduction provided by a useful prefetch and then determine how much of this reduction becomes ROB-visible.

When a prefetch request p fills a cache line, the model records the memory-service cost completed in fetching that line. For a prefetch supplied through the L3 path, the measured service cost is

$$C_{L3}(p) = T_{NoC}(p) + T_{directory}(p) + T_{remote}(p) + T_{NUCA}(p). \quad (6.1)$$

For a prefetch supplied from DRAM, the service cost additionally includes the DRAM-cache, bus, and device timing:

$$C_{DRAM}(p) = C_{L3}(p) + T_{DRAM\ cache}(p) + T_{DRAM\ bus}(p) + T_{DRAM\ device}(p). \quad (6.2)$$

DRAM queueing delay is excluded because it is attributed separately as memory-side interference.

The measured service cost represents the memory service completed in advance by the prefetch. When a later demand instruction i accesses the prefetched line, the corresponding no-prefetch shadow cache is used to determine whether the line would have been available without prefetching. If the line is absent from the shadow cache, the prefetch is considered to have avoided the lower-level service that the demand would otherwise have incurred. Let $C_{service}(p)$ denote the service cost selected according to whether prefetch p is supplied through the L3 or DRAM path. Its raw benefit is therefore estimated as

$$B_{\text{raw}}(i) = \begin{cases} C_{\text{service}}(p), & \text{if the line would be absent without prefetching,} \\ 0, & \text{otherwise.} \end{cases} \quad (6.3)$$

This associates the measured service cost of the prefetch with the demand request that later consumes the prefetched line, while the shadow cache avoids crediting a prefetch when the demand would have obtained the line without it.

Let $B_{L2}(i)$ and $B_{L3}(i)$ denote the raw benefits attributed through the L2 and L3 prefetch paths, respectively. These values are not added because they may represent the same avoided memory access. The model instead selects

$$B(i) = \max(B_{L2}(i), B_{L3}(i)). \quad (6.4)$$

The ROB-visible benefit is then evaluated using a no-prefetch counterfactual. Let $T_{\text{actual}}(i)$ denote the observed completion time and $T_{\text{head}}(i)$ the time at which instruction i reaches the ROB head. Without the selected prefetch benefit, the estimated completion time is

$$T_{\text{cf}}(i) = T_{\text{actual}}(i) + B(i). \quad (6.5)$$

For a completion time T , the corresponding ROB-head stall is

$$S(T) = \max(T - T_{\text{head}}(i), 0). \quad (6.6)$$

The candidate ROB-visible benefit is therefore

$$B_{\text{ROB}}(i) = \max(S(T_{\text{cf}}(i)) - S(T_{\text{actual}}(i)), 0). \quad (6.7)$$

This comparison covers three possible cases, as illustrated in Figure 6.1. If the demand completes after reaching the ROB head, the prefetch reduces an existing ROB-head stall. If the actual completion precedes the ROB-head arrival but the counterfactual completion does not, the prefetch prevents a stall that would otherwise occur. If both completion times precede the ROB-head arrival, the latency reduction is hidden by OoO execution and MLP and provides no ROB-visible benefit.

The candidate benefit is further evaluated according to ROB occupancy. If the ROB is already full when the instruction reaches the head, the candidate benefit is retained. Otherwise, the model uses the remaining ROB entries and recent dispatch activity to estimate whether the counterfactual stall would fill the ROB. If the ROB would remain non-full, the benefit is discarded. If the ROB would become full, only the portion of the counterfactual stall exposed after full occupancy is reached is retained. This restricts

6. Adaptive Prefetch Throttling

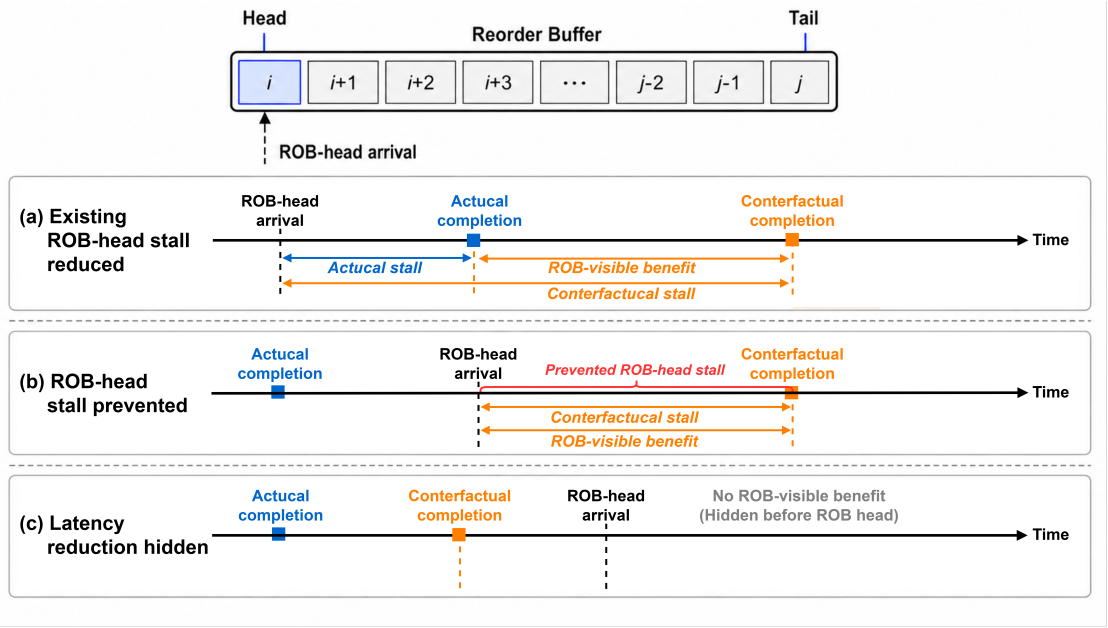


Figure 6.1.: Counterfactual analysis of ROB-visible prefetch benefit.

the final benefit to latency reduction that would become visible as ROB-head stall when the ROB is full.

The source core of the useful prefetch is preserved throughout the analysis, allowing the resulting ROB-visible benefit to be attributed to the core that generated the prefetch. Together with the ROB-visible harm developed in the previous chapter, this benefit provides the input to the adaptive throttling policy described next.

6.3. RVPAC – Adaptive Throttling Policy

RVPAC adjusts the aggressiveness of the L2 prefetcher associated with each core using the ROB-visible benefit introduced in the previous section and the ROB-visible interference attributed in the previous chapter. For core i , the corresponding prefetcher is controlled through a discrete aggressiveness level L_i . Each core first derives a local action from its observed utility. These local actions are then reconciled within a shared control domain before levels are committed. The policy selects between two system-level operating modes, *GAIN* and *GUARD*. *GAIN* preserves prefetch activity when the system remains benefit-favorable, whereas *GUARD* prioritizes reducing severe cross-core interference. An explicit probe-up mechanism allows throttled prefetchers to re-evaluate higher aggressiveness levels.

6.3.1. Utility Signal and Mode Selection

For core i , let B_i denote the ROB-visible benefit generated by its L2 prefetcher, H_i^{out} the ROB-visible harm caused by that prefetcher to other cores, and H_i^{in} the cross-core prefetch harm received by core i . The local throttling decision primarily evaluates B_i against H_i^{out} , while H_i^{in} is used to identify cores that experience unusually high cross-core interference.

Because different aggressiveness levels can issue different numbers of prefetch requests, directly comparing accumulated benefit and harm can bias the decision toward levels that simply generate more traffic. RVPAC therefore evaluates the average ROB-visible effect per issued prefetch. Over the accumulated observation windows,

$$\bar{B}_i = \frac{\sum_w B_{i,w}}{\sum_w I_{i,w}}, \quad (6.8)$$

$$\bar{H}_i^{\text{out}} = \frac{\sum_w H_{i,w}^{\text{out}}}{\sum_w I_{i,w}}, \quad (6.9)$$

where $I_{i,w}$ is the number of prefetches issued by core i in window w . The resulting local utility is

$$\bar{U}_i = \bar{B}_i - \bar{H}_i^{\text{out}}. \quad (6.10)$$

A positive $\bar{U}_i$ indicates that the ROB-visible benefit generated by the core's prefetches exceeds their cross-core harm on average, whereas a negative value indicates that the prefetch traffic causes more visible harm than benefit.

When a previous level adjustment provides a valid comparison baseline, the controller estimates the local utility slope with respect to aggressiveness:

$$\Delta U_i = \frac{\bar{U}_i^{\text{current}} - \bar{U}_i^{\text{previous}}}{L_i^{\text{after}} - L_i^{\text{before}}}. \quad (6.11)$$

Here, L_i^{before} and L_i^{after} are the prefetch levels before and after the previous adjustment that supplies the comparison baseline. A positive ΔU_i indicates that higher aggressiveness was associated with higher utility over the most recent level transition, whereas a negative value favors lower aggressiveness. Following a non-exploratory UP, RVPAC retains this previous baseline until the next decision-ready local observation; intermediate low-sample HOLD epochs do not replace it. If no previous level adjustment provides a valid comparison baseline, ΔU_i is not used.

6. Adaptive Prefetch Throttling

The system-level mode is selected from the aggregate ROB-visible benefit and outgoing cross-core harm,

$$B_{\text{sys}} = \sum_i B_i, \quad H_{\text{sys}} = \sum_i H_i^{\text{out}}. \quad (6.12)$$

RVPAC operates in

$$\begin{cases} \text{GUARD}, & H_{\text{sys}} > B_{\text{sys}}, \\ \text{GAIN}, & H_{\text{sys}} \leq B_{\text{sys}}. \end{cases} \quad (6.13)$$

GAIN therefore represents a state in which the overall ROB-visible benefit still covers the measured cross-core harm. GUARD is activated when cross-core harm becomes dominant and requires stronger interference control. If neither benefit nor harm is observed, the absence of a signal is not treated as negative utility: it does not trigger normal throttling, although an eligible lower-level prefetcher may still be considered for a controlled probe-up.

6.3.2. GAIN and GUARD Control

In *GAIN* mode, RVPAC primarily follows the local utility of each prefetcher. The policy is intentionally conservative when utility remains positive, avoiding unnecessary level reversals caused by short-term changes in the marginal signal. For a prefetcher with sufficient observations, the local action follows these rules:

Negative utility. If $\bar{U}_i < 0$, the controller selects DOWN. If the previous action was already DOWN and utility remains negative, throttling continues.

Successful increase. If the previous action was UP, utility $\bar{U}_i$ remains positive, and $\Delta U_i > 0$, the controller may select UP again.

Recovery after throttling. If the previous action was DOWN, utility $\bar{U}_i$ has recovered to a non-negative value, and $\Delta U_i > 0$, the controller may select UP.

Otherwise. The controller selects HOLD. In particular, a negative ΔU_i alone does not trigger DOWN while the current utility $\bar{U}_i$ remains non-negative.

This policy reduces aggressiveness when prefetching consistently produces negative utility while preserving prefetch activity that continues to provide a positive net effect.

In *GUARD* mode, RVPAC additionally identifies prefetchers that are responsible for unusually high cross-core interference. Let Q_1 and Q_3 denote the first and third quartiles of the outgoing-harm distribution across cores. The outlier threshold is

6.3. RVPAC – Adaptive Throttling Policy

$$T_{\text{out}} = Q_3 + 3(Q_3 - Q_1). \quad (6.14)$$

The L2 prefetcher of core i is classified as an *aggressor* when

$$H_i^{\text{out}} > T_{\text{out}} \quad \text{and} \quad H_i^{\text{out}} > 0. \quad (6.15)$$

The same outlier rule is applied to H_i^{in} to identify a *protected core*, which experiences unusually high cross-core prefetch harm. GUARD then applies asymmetric control:

- An *aggressor prefetcher* is forced to DOWN to reduce the cross-core harm caused by its prefetch traffic.
- A non-aggressor prefetcher first follows the local GAIN decision.
- If the prefetcher associated with a protected core would otherwise HOLD, it becomes an UP candidate, allowing the core to increase latency hiding while experiencing severe interference.

An aggressor classification takes precedence over protected-core promotion. Thus, GUARD avoids globally throttling all prefetchers when interference becomes high. Instead, it reduces aggressiveness at the major interference sources while retaining local utility-based control for the remaining cores.

The local UP, DOWN, and HOLD outcomes are candidates rather than immediate level changes. Within each shared decision epoch, multiple DOWN actions may be committed, whereas at most one UP action is committed. When multiple UP candidates are available, RVPAC ranks them using

$$S_i = \bar{U}_i (L_{\text{max}} - L_i), \quad (6.16)$$

which favors prefetchers with higher current utility and more remaining headroom. Ties favor the lower current level and then the prefetcher that has remained at its current level longer. This serializes increases in prefetch aggressiveness and allows the effect of an upward adjustment to be evaluated separately. It also avoids simultaneous level increases that could create sudden traffic surges and cause the recently reduced DRAM and L3 pressure to rise again.

6.3.3. Probe-Up and Rollback

After throttling to a lower aggressiveness level, fewer prefetches are issued and the resulting utility feedback becomes sparser. Moreover, the current level cannot reveal the benefit of additional prefetches that would only be generated at a higher level. A prefetcher may therefore remain at a reduced level even when higher aggressiveness would be beneficial. RVPAC performs explicit *probe-up* trials to periodically re-evaluate higher aggressiveness levels.

In GAIN mode, a prefetcher is eligible for an automatic probe-up when

- its current local action is HOLD,
- its current utility is non-negative,
- it has not reached the maximum prefetch level, and
- it is not prevented from probing by the current trial or cooldown state.

An eligible automatic probe-up enters the shared arbitration described above. If selected, it increases the prefetch level by one step and enters an exploratory trial. During this trial, new UP actions for other prefetchers are temporarily suppressed, while DOWN actions remain permitted.

The trial determines whether the higher level should be retained. Its outcome is handled as follows:

Negative trial utility. If the utility measured at the probed level becomes negative, the promotion is rejected and the prefetcher returns to its previous level.

Non-negative trial utility. The promoted level is retained. This includes the case in which the trial utility remains non-negative but is lower than the utility measured before the probe.

Insufficient evidence. If the trial reaches its bounded observation period without enough issued prefetches to assess utility reliably, the promoted level is retained rather than treated as harmful.

Interference escalation. If the system enters GUARD mode during the trial and the probed prefetcher is identified as an aggressor, the trial is immediately aborted and the prefetcher returns to the lower level.

Probe-up therefore provides an explicit recovery path after throttling. It allows the controller to explore whether additional prefetch aggressiveness has become beneficial again while limiting the interference caused by an unsuccessful promotion.

6.4. Implementation

We implement RVPAC in Sniper by extending its simple stride prefetcher and connecting it to the ROB-visible statistics generated by the performance model. The underlying address-prediction mechanism is unchanged. Each core maintains 16 independent stride-history streams, prefetches are restricted to the current 4KB page, and a hit on a prefetched line may trigger subsequent prefetches. The controller changes only the number of prefetch requests generated for each trigger.

Four effective aggressiveness levels are supported:

$$L_1 \rightarrow 1, \quad L_2 \rightarrow 2, \quad L_3 \rightarrow 3, \quad L_4 \rightarrow 4 \quad (6.17)$$

prefetches per trigger. The controller starts at L_4 , which is equivalent to the degree used by Sniper’s unmodified simple stride prefetcher. Although an L_0 configuration is retained, the current controller uses L_1 as its minimum effective level and therefore does not completely disable prefetching. An UP or DOWN action changes the current level by one step, while HOLD leaves it unchanged. This design keeps the address prediction mechanism fixed and isolates the effect of dynamic aggressiveness control.

RVPAC operates over shared control windows. A decision is triggered after 1024 prefetcher training events, at which point all cores are evaluated together using the window-level ROB-visible statistics. For each core, the controller obtains the prefetch benefit, outgoing cross-core prefetch harm, incoming cross-core prefetch harm, and the number of issued prefetches required by the utility calculations in the previous section.

For the current controller evaluation, the throttling signal includes ROB-visible prefetch-induced DRAM interference and L3 prefetch pollution. L3 MSHR harm is retained for performance analysis but is not used as a controller input, because Sniper does not natively model a finite physical L3 MSHR structure. The model introduced in the previous chapter therefore represents LLC miss-tracking capacity through virtual entries with predicted release times. The resulting L3 MSHR harm is useful for identifying miss-handling pressure, but depends more strongly on modeling assumptions than the DRAM and cache-pollution signals used by the controller. Because throttling decisions can be sensitive to small variations in feedback, this component is excluded from the current controller input. Moreover, in real processors, outstanding LLC misses may be managed by multiple implementation-specific tracking and buffering structures rather than a single monolithic MSHR structure.

At the end of each control window, RVPAC adds the benefit, outgoing harm, and issued-prefetch counts observed for each core to that core’s local accumulators. A core becomes eligible for a normal local evaluation once its accumulated observations contain at least 32 issued prefetches. Thus, a core may be evaluated after one control window when sufficient samples are already available; otherwise, its observations are carried across

6. Adaptive Prefetch Throttling

subsequent windows. To prevent a low-rate prefetcher from remaining indefinitely without an evaluation, the controller also evaluates a core after four control windows even if the issued-prefetch threshold has not been reached.

After a core is evaluated, its accumulated statistics are normally cleared, including when the resulting local action is **HOLD**, and collection begins again for the next local evaluation. A non-trial UP candidate deferred by an active probe trial retains its accumulated evidence until it can participate in arbitration.

The first four control windows form a warm-up period. Normal UP and DOWN actions are suppressed during this period to avoid decisions based on insufficient initial observations. A DOWN action is still allowed when the controller enters **GUARD** and identifies the core as an aggressor, allowing severe cross-core interference to be reduced immediately.

Probe-up uses a separate evaluation state. Only one core may be under a probe trial at a time. When a probe begins, the selected core is increased by one level and its normal local statistics are cleared. Probe-specific benefit, cross-core harm, issued-prefetch, and observation-window statistics are then accumulated separately so that the promoted level can be evaluated independently. New UP actions on other cores are suppressed during the trial.

A probe is evaluated after 32 issued prefetches or two control windows. A negative trial utility causes the controller to restore the previous level. The same rollback is performed immediately if the system enters **GUARD** and the trial core is identified as an aggressor. After a failed or aborted probe, the core enters a two-window cooldown before it can be selected again. Otherwise, the higher level is retained. The current implementation therefore uses a lenient acceptance rule: a promotion is revoked only when the trial produces negative utility or causes severe cross-core interference.

Table 6.1 summarizes the main implementation parameters.

6.5. Experimental Evaluation

We evaluate RVPAC using 4-, 8-, and 16-core configurations. Each core has private L1 and L2 caches, while every four cores share a 4 MB LLC domain. Table 6.2 summarizes the main system configuration used in the evaluation.

Figure 6.2 compares the execution time of three configurations across different core counts: prefetching disabled (*Off*), Sniper’s simple L2 prefetcher operating at a fixed aggressiveness level (*Baseline*), and the proposed RVPAC controller. Table 6.3 further reports the number of issued prefetches, the per-core useful-prefetch rate, the number of DRAM accesses, and the average DRAM request latency. These statistics provide additional insight into how RVPAC changes prefetch activity and the resulting pressure on the shared memory system.

Table 6.1.: Implementation parameters of the adaptive prefetch controller.

Parameter	Setting
Prefetcher	Simple stride
History streams	16 per core
Page-boundary checking	4 KB
Prefetch on prefetch hit	Enabled
Aggressiveness levels	L_1, L_2, L_3, L_4
Prefetches per trigger	1, 2, 3, 4
Initial level	L_4
Minimum effective level	L_1
Control window	1024 prefetcher training events
Local decision threshold	32 issued prefetches or 4 windows
Warm-up period	4 control windows
Concurrent probe trials	1
Probe evaluation	32 issued prefetches or 2 windows
Probe cooldown	2 control windows

Table 6.2.: Simulation configuration for the prefetch-control evaluation.

Parameter	Configuration
Core counts	4, 8, 16
Concurrent application instances	4, 8, 16
L1 data cache	32 KB/core, 8-way, 64 B line
L2 cache	256 KB/core, 8-way, private, 64 B line
LLC	4 MB per 4-core domain, 16-way, 64 B line
LLC domains (4/8/16 cores)	1 / 2 / 4
Effective total LLC (4/8/16 cores)	4 / 8 / 16 MB
DRAM model	Detailed DDR4 (Table 5.1)

Microbenchmark. We first evaluate RVPAC using a controlled victim–aggressor microbenchmark designed to create a prefetch-harmful scenario. Half of the cores execute victim threads that repeatedly scan private 2 MB working sets, while the remaining cores execute aggressor threads. Each aggressor uses eight independent streams and follows a train-and-abandon access pattern, accessing only two consecutive cache lines within each 4 KB page before moving to the next page. This pattern trains the stride prefetcher but rarely consumes the subsequently generated prefetches, intentionally creating low-value prefetch traffic and memory-system interference.

In this scenario, RVPAC achieves speedups of $1.10\times$, $1.11\times$, and $1.10\times$ over Baseline at 4, 8, and 16 cores, respectively. Execution time decreases from 6.46 ms to 5.88 ms at 4 cores, from 30.38 ms to 27.46 ms at 8 cores, and from 57.07 ms to 51.82 ms at 16 cores. Although disabling prefetching still gives the shortest execution time across all three core counts, this behavior is expected because the microbenchmark is deliberately

6. Adaptive Prefetch Throttling

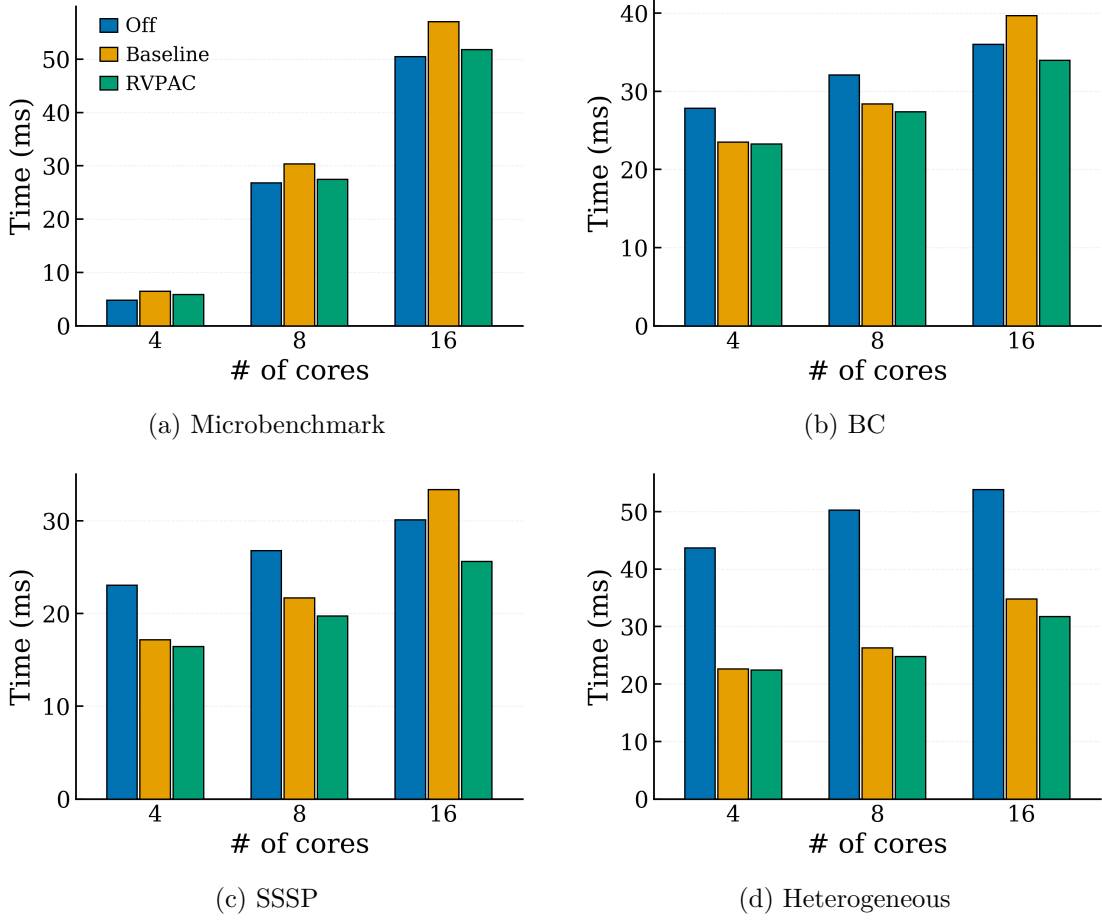


Figure 6.2.: Execution time of the evaluated workloads with prefetching disabled, the fixed baseline prefetcher, and RVPAC across different core counts.

constructed to amplify harmful prefetching. More importantly, RVPAC consistently reduces the performance loss introduced by fixed aggressive prefetching.

The internal statistics in Table 6.3 show a corresponding reduction in prefetch traffic and memory-system pressure. At 4 cores, the number of issued prefetches ranges from 34.9K–346.9K across cores under Baseline, but falls to 23.0K–47.3K under RVPAC. Total DRAM accesses decrease from 2.39M to 1.55M, while the average DRAM request latency decreases from 129.38 ns to 119.34 ns. The same trend continues at higher core counts. At 8 cores, DRAM accesses decrease from 14.73M to 13.11M and average DRAM latency from 231.29 ns to 181.24 ns. At 16 cores, they decrease from 27.80M to 25.32M and from 428.31 ns to 339.96 ns, respectively. These results indicate that RVPAC suppresses a substantial fraction of the low-value prefetch traffic generated by this stress test and reduces the additional pressure placed on the shared memory system.

BC. For BC, RVPAC achieves speedups of $1.01\times$, $1.04\times$, and $1.17\times$ over Baseline at 4, 8, and 16 cores, respectively. At 4 and 8 cores, fixed prefetching remains beneficial relative to disabling prefetching, indicating that the latency reduction provided by useful prefetches is sufficient to offset the additional memory traffic under relatively low contention. This balance changes as the number of cores increases. Baseline maintains a similar high prefetch aggressiveness on every core, issuing approximately 670K–690K prefetches per core, while all cores compete for shared LLC and DRAM resources. Consequently, aggregate prefetch traffic grows with concurrency and places increasing pressure on the shared memory system.

This trend is reflected in the memory-system statistics. Under Baseline, total DRAM accesses increase from 3.88M at 4 cores to 15.81M at 16 cores, while the average DRAM request latency increases from 126.01 ns to 263.96 ns. At 16 cores, Baseline requires 39.68 ms, compared with 36.02 ms when prefetching is disabled. Fixed high-aggressiveness prefetching therefore provides a net benefit at lower core counts, but the additional traffic becomes increasingly costly as shared-resource contention grows.

RVPAC adapts to this change by progressively reducing prefetch activity. The per-core prefetch count decreases from 526.2K–537.5K at 4 cores to 245.9K–288.3K at 16 cores. At 16 cores, the useful-prefetch rate increases from approximately 26% under Baseline to 35.96–37.80% under RVPAC. Total DRAM accesses are reduced from 15.81M to 10.52M, and the average DRAM request latency decreases from 263.96 ns to 160.80 ns. RVPAC consequently reduces execution time to 33.99 ms, corresponding to a $1.17\times$ speedup over Baseline and a $1.06\times$ speedup over Off.

The increasing advantage of RVPAC with core count is consistent with its intended behavior. When prefetch benefit remains sufficient to compensate for the associated interference, relatively aggressive prefetching can be retained. As concurrency increases and additional prefetch traffic becomes more costly, RVPAC applies stronger throttling rather than maintaining a fixed aggressiveness level.

SSSP. SSSP exhibits a similar overall trend, but the effect of increasing concurrency is more pronounced. RVPAC achieves speedups of $1.04\times$, $1.10\times$, and $1.30\times$ over Baseline at 4, 8, and 16 cores, respectively. At 4 and 8 cores, Baseline remains clearly faster than Off. At 16 cores, however, Baseline requires 33.37 ms, exceeding the 30.11 ms execution time with prefetching disabled. RVPAC reduces the execution time to 25.64 ms, which also corresponds to a $1.17\times$ speedup over Off.

The memory-system statistics show that SSSP experiences a particularly strong increase in memory pressure as concurrency grows. Baseline continues to issue approximately 700K prefetches per core across all three configurations, while total DRAM accesses increase from 3.56M at 4 cores to 14.45M at 16 cores. Over the same range, the average DRAM request latency increases from 155.30 ns to 362.61 ns. Because the fixed prefetcher does not adapt its per-core aggressiveness to the increasing pressure on shared

6. Adaptive Prefetch Throttling

resources, the cost of the additional traffic becomes progressively larger at higher core counts.

RVPAC responds by applying stronger throttling as concurrency increases. At 16 cores, the per-core prefetch count decreases from 702.3K–708.4K under Baseline to 314.2K–342.0K under RVPAC, while the useful-prefetch rate increases from 29.07–29.33% to 44.71–46.96%. Total DRAM accesses are reduced from 14.45M to 8.97M, and the average DRAM request latency decreases from 362.61 ns to 260.77 ns. Correspondingly, the speedup over Baseline increases from **1.04** $\times$ at 4 cores to **1.30** $\times$ at 16 cores.

This behavior is consistent with the design objective of RVPAC. Under lower contention, prefetch benefit can still compensate for much of the additional memory-system cost, and aggressive throttling is less necessary. As more cores execute concurrently and prefetch-induced interference increases, the runtime benefit and harm signals cause RVPAC to reduce aggressiveness more substantially. The stronger memory-system pressure observed for SSSP at high core counts is also consistent with its larger performance improvement relative to BC.

Heterogeneous workload. We further evaluate RVPAC under heterogeneous multiprogrammed workloads, where different cores can exhibit substantially different memory-access and prefetching behavior. For the 4-, 8-, and 16-core configurations, we randomly construct workload mixes containing 4, 8, and 16 GAPBS benchmark instances, respectively, with one instance running on each core. Unlike the homogeneous BC and SSSP experiments, where every core executes the same benchmark, these workload mixes expose the controller to different prefetch effectiveness and memory behavior across cores.

Baseline remains substantially faster than Off at all evaluated core counts, indicating that prefetching is broadly beneficial for these workload mixes. RVPAC nevertheless further improves upon Baseline, achieving speedups of **1.01** $\times$, **1.06** $\times$, and **1.10** $\times$ at 4, 8, and 16 cores, respectively. Unlike BC and SSSP at high core counts, the primary challenge here is not that prefetching becomes uniformly harmful across all cores, but that the value of prefetching varies substantially among the concurrently executing applications.

This heterogeneity is directly visible in the per-core useful-prefetch rates. At 16 cores, for example, the useful-prefetch rate under Baseline ranges from 25.65% to 98.16%, substantially wider than the ranges observed for the homogeneous BC and SSSP workloads. Some cores therefore make highly effective use of prefetching, whereas others exhibit considerably lower prefetch effectiveness. A single fixed aggressiveness level is unlikely to be equally appropriate for all cores in such a configuration.

RVPAC can instead adapt prefetch aggressiveness independently according to the runtime benefit and interference associated with each core. At 16 cores, the useful-prefetch rate ranges from 31.53% to 98.49% under RVPAC, while total DRAM accesses decrease

from 12.27M under Baseline to 10.09M. The average DRAM request latency also decreases from 289.12 ns to 239.96 ns. The corresponding performance advantage grows from $1.01\times$ at 4 cores to $1.10\times$ at 16 cores. These results show that RVPAC can preserve high-value prefetching while reducing lower-value traffic, rather than applying the same degree of throttling uniformly across all cores.

Overall, RVPAC improves upon the fixed Sniper simple prefetcher in all completed configurations, although the source of the improvement differs across workloads. In the deliberately prefetch-harmful microbenchmark, RVPAC substantially reduces the performance loss caused by fixed aggressive prefetching. For BC and SSSP, fixed prefetching remains beneficial at lower core counts, but increasing competition for the shared LLC and DRAM makes the additional prefetch traffic progressively more costly, causing RVPAC to apply stronger throttling. In the heterogeneous workloads, prefetch effectiveness varies substantially across cores, and RVPAC instead benefits from adapting prefetch aggressiveness to these per-core differences. Across the 4-, 8-, and 16-core configurations, RVPAC achieves maximum speedups of $1.11\times$, $1.17\times$, $1.30\times$, and $1.10\times$ for the Microbenchmark, BC, SSSP, and Heterogeneous workloads, respectively.

The DRAM statistics also illustrate why memory-side activity alone is insufficient for determining whether prefetching improves performance. Disabling prefetching typically produces fewer DRAM accesses and can therefore result in a lower average DRAM request latency, yet it does not necessarily provide the shortest execution time. A useful prefetch initiates the memory access before the corresponding demand requires the data, allowing part of the access latency to be hidden by out-of-order execution, memory-level parallelism, or overlapping computation. Conversely, fixed aggressive prefetching can generate substantial shared-resource interference even when a fraction of its requests are useful.

These results support the central motivation of RVPAC. Prefetch aggressiveness should not be determined solely from prefetch usefulness or memory-side latency, since these metrics do not directly capture whether the benefit or interference of a prefetch affects processor progress. By combining request-level ROB-visible prefetch benefit with ROB-visible interference, RVPAC adapts prefetch aggressiveness to both workload behavior and shared-resource contention. Across the evaluated workloads, this adaptive control consistently improves upon the fixed Sniper simple prefetcher, with larger gains as contention increases.

6.5.1. Sensitivity to DDR4 Configuration

The preceding evaluation uses the detailed DDR4 configuration summarized in Table 5.1. To evaluate the sensitivity of RVPAC to the DDR4 configuration, we repeat the 16-core BC, SSSP, and heterogeneous experiments using the higher-resource DDR4 configuration summarized in Table 6.4.

6. Adaptive Prefetch Throttling

Compared with the configuration used in the main evaluation, this setup provides a higher DRAM transfer rate, an additional memory channel, more ranks and banks, and shorter DRAM timing parameters. These changes collectively increase the memory system’s ability to service concurrent requests and reduce the pressure created by additional prefetch traffic.

Figure 6.3 compares the execution times of Off, Baseline, and RVPAC under this configuration. The performance difference between RVPAC and Baseline becomes substantially smaller than under the original 16-core configuration. Relative to Baseline, RVPAC achieves approximately $0.97\times$, $0.98\times$, and $1.01\times$ speedup for BC, SSSP, and the heterogeneous workload, respectively. Prefetching remains significantly beneficial across all three workloads, as the Off configuration has higher execution time.

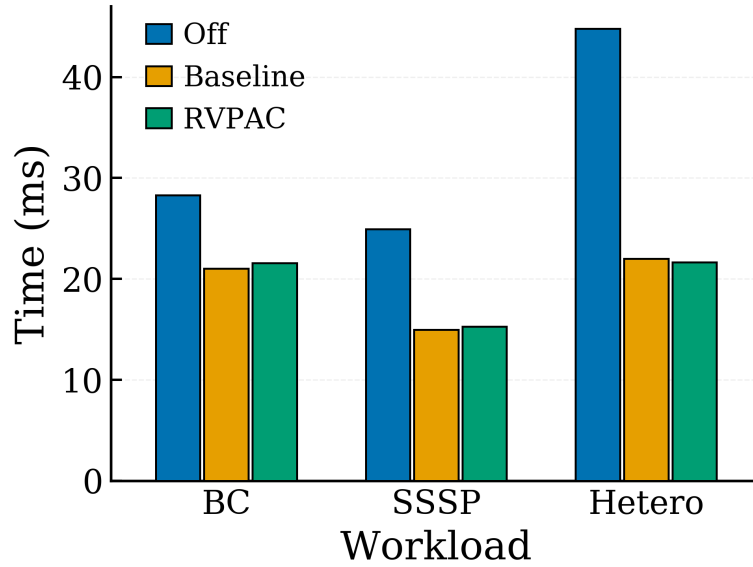


Figure 6.3.: Execution time under the higher-resource DDR4 configuration for the 16-core workloads.

The corresponding prefetch and memory-system statistics are summarized in Table 6.5. Under Baseline, the average DRAM latency decreases from 263.96 ns to 65.37 ns for BC and from 362.61 ns to 72.55 ns for SSSP relative to the original 16-core configuration. The heterogeneous workload shows the same trend, decreasing from 289.12 ns to 67.70 ns. In contrast, the number of DRAM accesses remains broadly similar between the two memory configurations. The higher-resource DDR4 system can therefore service a similar amount of traffic with substantially lower memory-access latency.

RVPAC nevertheless continues to reduce prefetch-generated traffic and improve the efficiency of the remaining prefetches. For BC, RVPAC reduces total DRAM accesses from 15.48M to 12.36M, while the useful-prefetch rate increases from 26.32–26.74% to 32.63–35.80%. For SSSP, DRAM accesses decrease from 14.13M to 10.53M, while the

useful-prefetch rate increases from 29.76–30.27% to 38.50–42.63%. However, the corresponding reductions in average DRAM latency are relatively small, from 65.37 ns to 62.81 ns for BC and from 72.55 ns to 69.55 ns for SSSP.

This behavior differs from the original configuration, where reducing prefetch traffic produces substantially larger reductions in DRAM latency and correspondingly larger performance improvements. Under the higher-resource configuration, the additional requests generated by aggressive prefetching can be accommodated at a lower interference cost. The latency-hiding benefit of the additional prefetches therefore becomes comparatively more important. Although RVPAC still removes a substantial amount of lower-value traffic in BC and SSSP, the resulting reduction in memory-system pressure is smaller and does not fully compensate for the useful prefetch opportunities removed by throttling.

This result also provides a useful sensitivity observation for the current control policy. RVPAC explicitly evaluates ROB-visible benefit and cross-core harm and tracks changes in utility following aggressiveness adjustments. Therefore, the controller can both reduce aggressiveness when prefetching becomes harmful and re-evaluate higher levels through probe-up. Under the higher-resource configuration, however, fewer memory delays become strongly exposed to processor execution and the performance differences between nearby aggressiveness levels become smaller. The resulting ROB-visible feedback therefore provides less separation between alternative levels. In this low-contention regime, occasional throttling can reduce useful latency-hiding opportunities while producing only a modest reduction in interference. The small performance differences observed for BC and SSSP identify an operating region in which finer-grained or longer-term feedback could further improve aggressiveness selection.

The heterogeneous workload exhibits a different behavior. As shown in Table 6.5, the useful-prefetch rate under Baseline ranges from 26.42% to 98.15%, demonstrating substantial variation in prefetch effectiveness across concurrently executing cores. RVPAC retains a similarly high upper-end prefetch count and useful rate while reducing total DRAM accesses from 12.03M to 10.85M. The average DRAM latency also decreases from 67.70 ns to 66.39 ns.

More importantly, RVPAC continues to outperform Baseline for the heterogeneous workload even when overall DRAM pressure is substantially reduced. This result highlights an advantage of per-core adaptive control that is distinct from simply reacting to high global memory contention. Different applications can benefit from different prefetch aggressiveness levels, and a single fixed setting cannot adapt to these per-core differences. By independently adjusting the L2 prefetcher associated with each core, RVPAC can preserve highly effective prefetch activity while selectively reducing lower-value traffic.

Overall, the sensitivity analysis shows that the performance improvements from prefetch throttling depend on the DDR4 configuration. RVPAC achieves its largest improvements

6. Adaptive Prefetch Throttling

when aggressive prefetching creates substantial shared-resource interference, while remaining close to the fixed aggressive Baseline under the higher-resource configuration. Importantly, RVPAC continues to improve performance for the heterogeneous workload, showing that per-core ROB-visible feedback remains useful even when overall memory contention is relatively low. These results demonstrate that RVPAC can adapt to both changes in shared-resource pressure and differences in prefetch effectiveness among concurrently executing cores.

6.5. Experimental Evaluation

Table 6.3.: Prefetch and memory-system statistics across different core counts. Issued prefetches and useful rate are reported as minimum–maximum ranges across cores.

Workload	Config.	Cores	Issued prefetches	Useful rate (%)	DRAM accesses	Avg DRAM latency (ns)
Microbenchmark	Off	4	0	0	1.13M	114.76
	Baseline		34.9K–346.9K	0.00–99.98	2.39M	129.38
	RVPAC		23.0K–47.3K	0.01–99.97	1.55M	119.34
	Off	8	0	0	12.71M	170.68
	Baseline		7–593.9K	0.00–30.00	14.73M	231.29
	RVPAC		4–126.5K	0.00–10.91	13.11M	181.24
	Off	16	0	0	24.62M	327.40
	Baseline		3–603.1K	0.00–33.33	27.80M	428.31
	RVPAC		1–93.9K	0.00–100.00	25.32M	339.96
BC	Off	4	0	0	2.04M	62.02
	Baseline		669.4K–672.2K	26.35–26.37	3.88M	126.01
	RVPAC		526.2K–537.5K	30.40–31.13	3.38M	119.72
	Off	8	0	0	4.08M	67.31
	Baseline		675.4K–682.3K	25.88–26.15	7.83M	160.46
	RVPAC		383.6K–411.0K	33.71–35.75	5.96M	137.27
	Off	16	0	0	8.20M	78.62
	Baseline		682.6K–692.3K	25.72–25.97	15.81M	263.96
	RVPAC		245.9K–288.3K	35.96–37.80	10.52M	160.80
SSSP	Off	4	0	0	1.49M	59.94
	Baseline		692.8K–694.3K	29.62–29.70	3.56M	155.30
	RVPAC		512.5K–532.8K	36.26–37.37	2.87M	149.88
	Off	8	0	0	2.99M	64.23
	Baseline		697.2K–704.7K	29.18–29.54	7.18M	206.31
	RVPAC		418.2K–435.8K	40.51–42.08	5.07M	179.83
	Off	16	0	0	5.98M	74.59
	Baseline		702.3K–708.4K	29.07–29.33	14.45M	362.61
	RVPAC		314.2K–342.0K	44.71–46.96	8.97M	260.77
Heterogeneous	Off	4	0	0	1.84M	55.63
	Baseline		140.8K–865.5K	26.15–98.03	2.82M	127.57
	RVPAC		140.0K–864.7K	28.06–98.07	2.73M	125.54
	Off	8	0	0	3.97M	60.28
	Baseline		140.8K–874.2K	26.03–97.66	6.52M	172.31
	RVPAC		140.2K–869.3K	31.09–98.08	5.75M	160.52
	Off	16	0	0	6.95M	68.14
	Baseline		139.2K–873.6K	25.65–98.16	12.27M	289.12
	RVPAC		132.0K–864.6K	31.53–98.49	10.09M	239.96

6. Adaptive Prefetch Throttling

Table 6.4.: Higher-resource DDR4 configuration used for sensitivity analysis.

Parameter	Setting
DRAM speed	2400 MT/s
Data bus width	64 bits
Data-bus bandwidth	38.4 GB/s
Controllers / channels	1 / 2
Ranks / banks / bank groups	4 / 32 / 4
DRAM page size	128 cache lines
Address mapping	XOR
Row policy	Open row
Base latency	45 ns
Bank open / close delay	15 / 15 ns
Access cost	15 ns
Inter-command delay	5 ns
Long / short command delay	5 / 2.5 ns
Controller delay	20 ns
Bank keep-open interval	120 ns
Refresh interval	64 ms
Refresh length	15 ns

Table 6.5.: Prefetch and memory-system statistics under the higher-resource DDR4 configuration. Issued prefetches and useful rate are reported as minimum–maximum ranges across cores.

Workload	Config.	Cores	Issued prefetches	Useful rate (%)	DRAM accesses	Avg DRAM latency (ns)
BC	Off	16	0	0	8.18M	52.12
	Baseline		660.6K–670.4K	26.32–26.74	15.48M	65.37
	RVPAC		384.2K–472.0K	32.63–35.80	12.36M	62.81
SSSP	Off	16	0	0	5.97M	52.71
	Baseline		677.9K–690.1K	29.76–30.27	14.13M	72.55
	RVPAC		417.0K–481.4K	38.50–42.63	10.53M	69.55
Heterogeneous	Off	16	0	0	6.98M	48.66
	Baseline		139.0K–875.1K	26.42–98.15	12.03M	67.70
	RVPAC		138.8K–869.6K	28.56–98.16	10.85M	66.39

Conclusion and Future Work

7.1. Conclusion

This work investigates the effects of memory-level parallelism, shared-memory interference, and hardware prefetching on the performance of memory-intensive applications, and further examines how processor-visible performance impact can be used to guide prefetch control. The results show that memory-side latency does not directly correspond to performance loss. Out-of-order execution and memory-level parallelism can hide part of the memory-access latency. Therefore, latency or request concurrency observed in the memory hierarchy alone is insufficient to determine the actual impact of a memory request on processor progress.

First, we analyze the effects of MLP on memory behavior and latency hiding. Aggregate MLP characterizes the concurrency of outstanding memory requests, but does not directly reflect their actual impact on execution performance. The experiments further show that, in multicore workloads, the MLP profiles of different cores can be similar even when the cores experience different levels of shared-resource interference and corresponding performance impact. MLP is therefore more suitable as a characterization metric for memory behavior, but is insufficient on its own to distinguish performance bottlenecks across cores or to guide fine-grained resource-management decisions.

Building on this analysis, we develop a source-aware memory interference model that attributes memory-side harm caused by L3 MSHR contention, DRAM contention, and L3 prefetch pollution while preserving the identity of the source core responsible for the interference. The model then combines this information with the execution state of the affected instruction in the ROB to map memory-side harm to ROB-visible harm. The results show that memory-side harm is generally substantially larger than the corresponding ROB-visible harm, indicating that a considerable fraction of memory-system interference can be hidden by out-of-order execution and MLP without directly limiting processor progress. Therefore, relying only on the magnitude of memory-side interference can overestimate the actual performance impact of some interference events.

We further apply this approach to hardware prefetch control. For prefetch benefit, the proposed request-level model associates each useful prefetch with the demand request that actually consumes the prefetched cache line and estimates the service latency

7. Conclusion and Future Work

avoided for that demand. Counterfactual ROB-stall analysis is then used to determine the portion of this latency reduction that becomes ROB-visible benefit. RVPAC combines this benefit signal with ROB-visible prefetch interference and dynamically adjusts prefetch aggressiveness on a per-core basis. This allows the controller to retain prefetching when its benefit is high, while reducing aggressiveness when the performance-visible interference introduced by additional prefetches increases.

Under the resource-constrained DDR4 configuration, RVPAC outperforms Sniper’s simple prefetcher with fixed aggressiveness across all workloads and core counts we evaluate. In the deliberately prefetch-harmful Microbenchmark, RVPAC reduces the performance loss caused by aggressive prefetching. For the homogeneous BC and SSSP workloads, the additional memory traffic and shared-resource contention introduced by fixed prefetching increase with core count, and the performance advantage of RVPAC over Baseline increases accordingly. For heterogeneous GAPBS workload mixes, different cores exhibit substantially different prefetch usefulness, and RVPAC adjusts prefetch aggressiveness according to these per-core differences. Across the 4-, 8-, and 16-core configurations, RVPAC achieves maximum speedups of $1.11\times$, $1.17\times$, $1.30\times$, and $1.10\times$ over Baseline for the Microbenchmark, BC, SSSP, and Heterogeneous workloads, respectively.

Overall, aggregate MLP, memory-side interference, and prefetch usefulness describe different aspects of memory-system behavior, but none of them alone directly captures their actual impact on processor performance. We use ROB-visible modeling to relate memory-side harm and request-level prefetch benefit to processor progress, and use these signals to drive adaptive prefetch throttling. The evaluation shows that this approach can adjust prefetch aggressiveness according to workload behavior and shared-resource contention, and improves performance over fixed prefetching policies in the evaluated multicore configurations.

7.2. Limitations and Future Work

This work is currently implemented and evaluated primarily using the Sniper simulator. Sniper uses interval simulation, which provides relatively high simulation speed while supporting large-scale multicore experiments. This makes it suitable for our 4-, 8-, and 16-core workloads and the large number of request-level interference events we analyze. However, interval-based modeling abstracts parts of the processor pipeline and some microarchitectural timing behavior. Future work can port the ROB-visible interference modeling framework and RVPAC to finer-grained cycle-level simulators such as gem5, enabling more detailed modeling and further validation of the performance trends and control behavior observed in this work.

RVPAC currently uses Sniper’s simple L2 stride prefetcher and controls aggressiveness by adjusting the number of prefetches generated on each trigger. This design allows the work to focus on the role of ROB-visible benefit and interference in prefetch throttling without

modifying the underlying prefetch prediction mechanism. Future work can extend the same feedback mechanism to more sophisticated hardware prefetchers, including designs based on historical information or reinforcement learning, and further investigate the applicability of ROB-visible feedback to multi-level prefetching and different prefetch algorithms.

The evaluation in this work mainly includes a controlled microbenchmark and GAPBS benchmarks under 4-, 8-, and 16-core configurations, but the workload and architecture coverage remains limited. Future work can extend the evaluation to benchmark suites such as SPEC CPU to cover a broader range of memory-intensive scenarios. The approach can also be evaluated with different cache capacities, LLC sharing topologies, memory bandwidths, and DRAM configurations to study the behavior of RVPAC across a wider range of workloads and architectural settings.

In addition, the current implementation maintains request-level source information, memory timing, and ROB-visible state in the simulation environment to support fine-grained performance analysis. Implementing the complete framework directly in a real processor may introduce substantial storage and logic overhead for maintaining per-request interference attribution. Future work can investigate lower-overhead approximations for practical hardware implementation, trading attribution granularity for hardware cost. For example, only a subset of demand and prefetch requests could be sampled, the number of performance-critical events being tracked could be limited, or periodically aggregated hardware statistics could be used to approximate the full request-level benefit and harm. By reducing the need for per-request tracking, such designs may preserve the information in ROB-visible feedback that is most relevant to control decisions while reducing hardware implementation overhead.

Additional MLP Results

This appendix presents additional MLP results for BFS and PR, together with the SPMV results across the evaluated software-prefetch distances. Unless otherwise stated, all results use the revised activity-specific normalization defined in Section 4.2. For the multicore workloads, the per-core MLP Stacks are highly similar across cores. We therefore present one representative core in each figure, with the total core count reported in the figure caption.

A.1. BFS

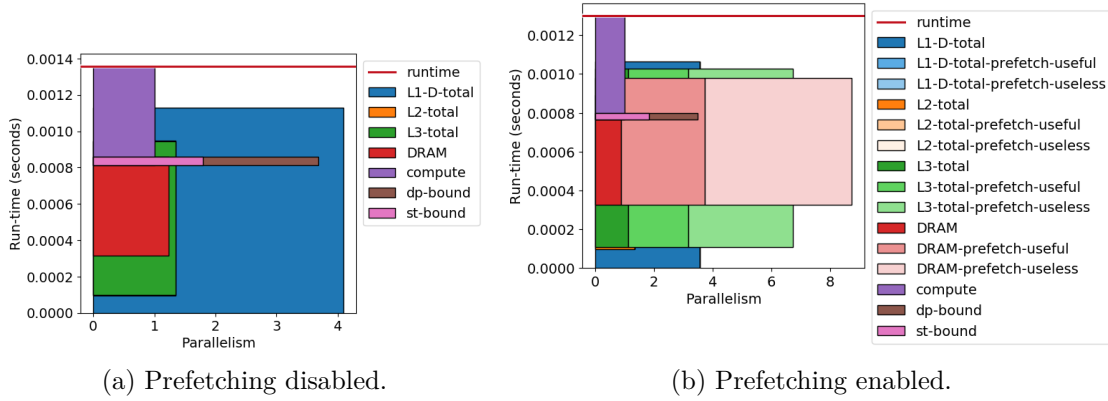


Figure A.1.: Per-core MLP Stacks for BFS in the 2-core configuration, with prefetching disabled and enabled.

A. Additional MLP Results

A.2. PageRank

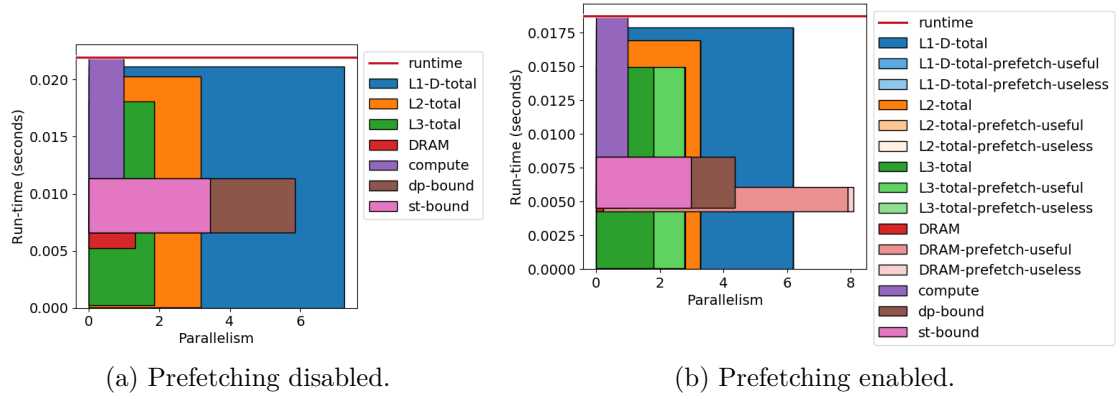


Figure A.2.: Per-core MLP Stacks for PR in the 2-core configuration, with prefetching disabled and enabled.

A.3. SPMV

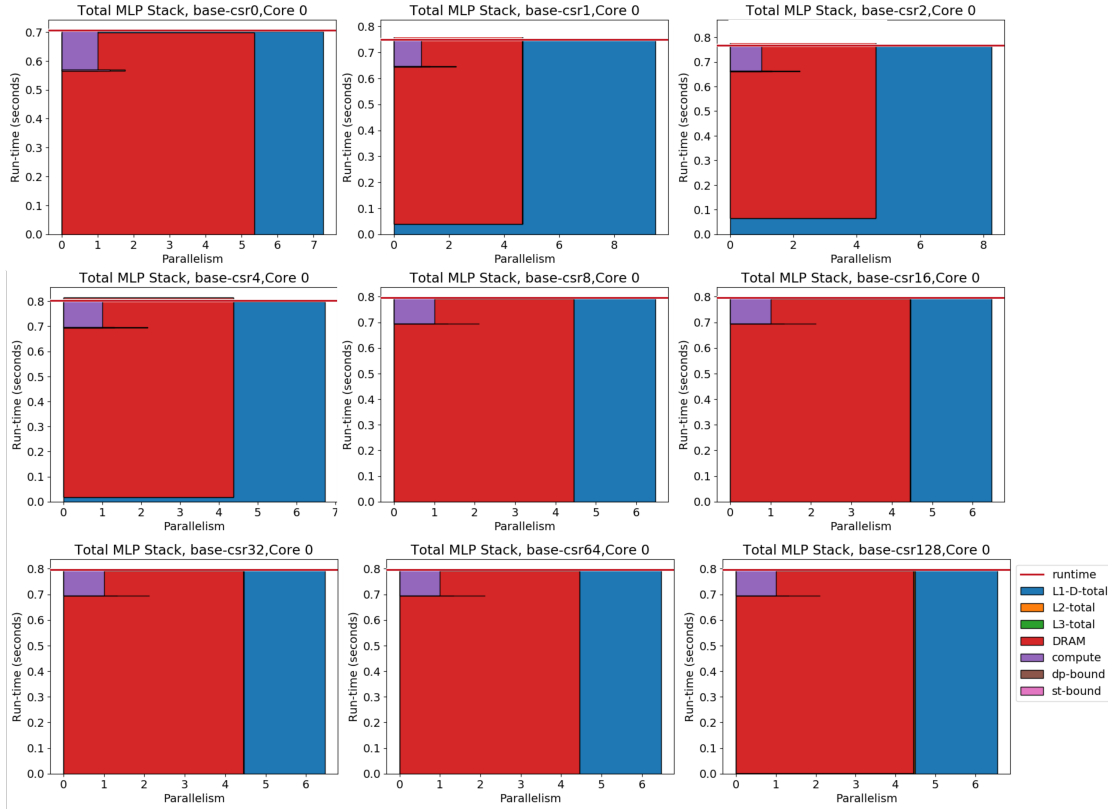
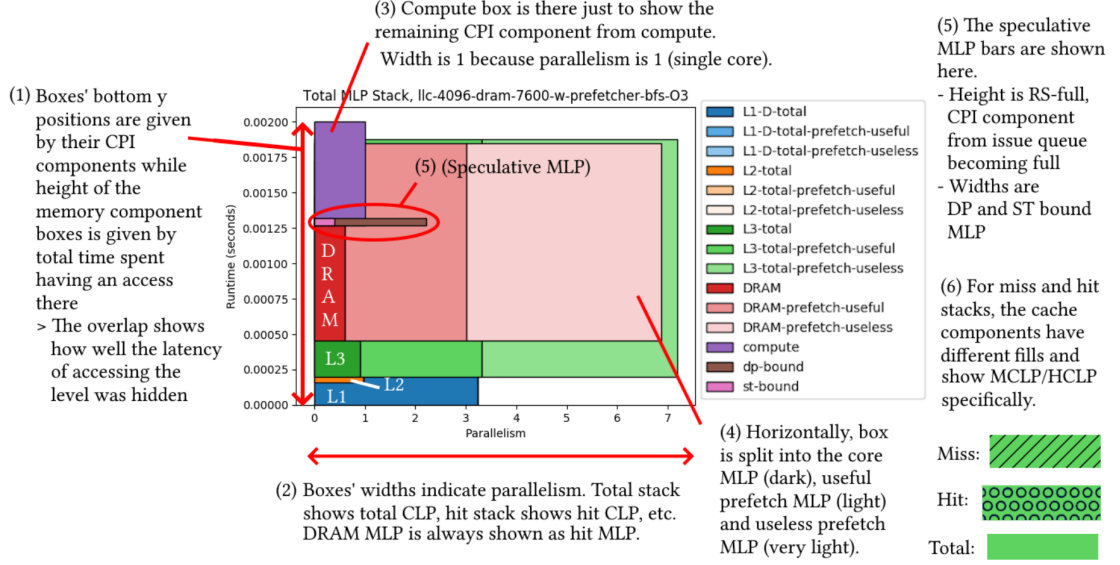


Figure A.3.: Total MLP Stacks for single-core SPMV with software-prefetch distances from 0 to 128. The distance denotes the number of loop iterations prefetched ahead, with 0 indicating no software prefetching.

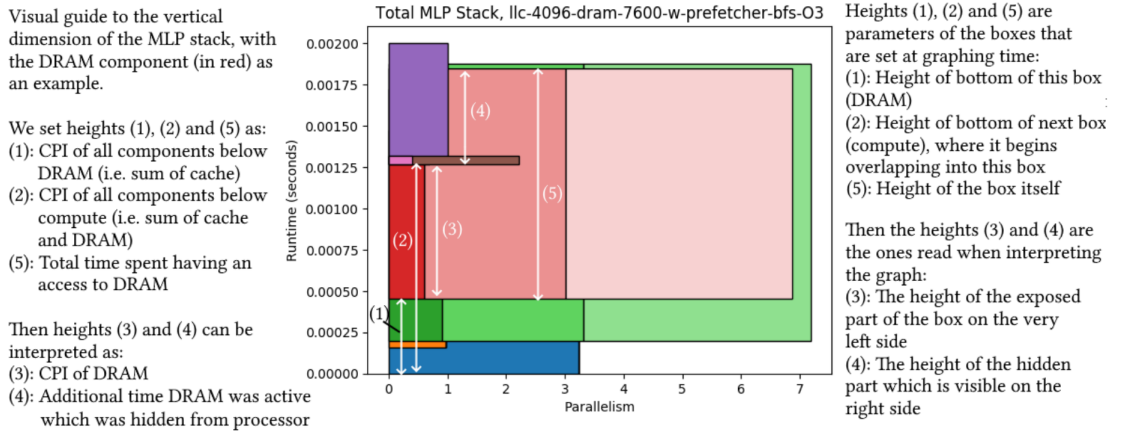
MLP Stacks Visualization Guide

This appendix reproduces the original visual guides for interpreting the MLP Stacks representation. Figure B.1 summarizes the overall visualization, including the horizontal parallelism dimension, the vertical organization of execution-time components, prefetch decomposition, and speculative MLP components. Figure B.1b provides additional detail on the interpretation of the vertical dimension and the overlap between memory activity and execution time.

B. MLP Stacks Visualization Guide



(a) Overall interpretation of the MLP Stacks representation.



(b) Detailed interpretation of the vertical dimension.

Figure B.1.: Visual guides to the MLP Stacks representation. Reproduced from [Hasnat \(2024\)](#).

Bibliography

- AI, N.; HE, W.; HE, H.; XIA, J.; LIAO, H.; AND ZHANG, G., 2025. Rich prefetcher: Storing rich information in memory to trade capacity and bandwidth for latency hiding. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*, MICRO '25, 170–183. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3725843.3756081. <https://doi.org/10.1145/3725843.3756081>. [Cited on page 11.]
- BEAMER, S.; ASANOVIĆ, K.; AND PATTERSON, D., 2015. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, (2015). [Cited on page 24.]
- BERA, R.; KANELLOPOULOS, K.; NORI, A.; SHAHROODI, T.; SUBRAMONEY, S.; AND MUTLU, O., 2021. Pythia: A customizable hardware prefetching framework using online reinforcement learning. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21 (Virtual Event, Greece, 2021), 1121–1137. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3466752.3480114. <https://doi.org/10.1145/3466752.3480114>. [Cited on page 10.]
- CARLSON, T. E.; HEIRMAN, W.; EYERMAN, S.; HUR, I.; AND EECKHOUT, L., 2014. An evaluation of high-level mechanistic core models. *ACM Trans. Archit. Code Optim.*, 11, 3 (Aug. 2014). doi:10.1145/2629677. <https://doi.org/10.1145/2629677>. [Cited on page 3.]
- CARLSON, T. E.; NILAKANTAN, S.; HEMPSTEAD, M.; AND HEIRMAN, W., 2015. Epoch Profiles: Microarchitecture-Based Application Analysis and Optimization. *IEEE Computer Architecture Letters*, 14, 01 (Jan. 2015), 30–33. doi:10.1109/LCA.2014.2329873. <https://doi.ieeecomputersociety.org/10.1109/LCA.2014.2329873>. [Cited on page 13.]
- CHAUDHURI, M., 2021. Zero inclusion victim: Isolating core caches from inclusive last-level cache evictions. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 71–84. doi:10.1109/ISCA52012.2021.00015. [Cited on page 9.]

Bibliography

- CHOU, Y.; FAHS, B.; AND ABRAHAM, S., 2004. Microarchitecture optimizations for exploiting memory-level parallelism. *SIGARCH Comput. Archit. News*, 32, 2 (Mar. 2004), 76. doi:10.1145/1028176.1006708. <https://doi.org/10.1145/1028176.1006708>. [Cited on pages 5 and 13.]
- DU BOIS, K.; EYERMAN, S.; AND EECKHOUT, L., 2013. Per-thread cycle accounting in multicore processors. *ACM Trans. Archit. Code Optim.*, 9, 4 (Jan. 2013). doi:10.1145/2400682.2400688. <https://doi.org/10.1145/2400682.2400688>. [Cited on pages 14 and 15.]
- EBRAHIMI, E.; LEE, C. J.; MUTLU, O.; AND PATT, Y. N., 2012. Fairness via source throttling: A configurable and high-performance fairness substrate for multicore memory systems. *ACM Trans. Comput. Syst.*, 30, 2 (Apr. 2012). doi:10.1145/2166879.2166881. <https://doi.org/10.1145/2166879.2166881>. [Cited on page 14.]
- EBRAHIMI, E.; MUTLU, O.; LEE, C. J.; AND PATT, Y. N., 2009. Coordinated control of multiple prefetchers in multi-core systems. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 316–326. doi:10.1145/1669112.1669154. [Cited on page 16.]
- EYERMAN, S.; HEIRMAN, W.; AND HUR, I., 2022. Dram bandwidth and latency stacks: Visualizing dram bottlenecks. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 322–331. doi:10.1109/ISPASS55109.2022.00045. [Cited on page 14.]
- FU, J.; PATEL, J.; AND JANSSENS, B., 1992. Stride directed prefetching in scalar processors. In *[1992] Proceedings the 25th Annual International Symposium on Microarchitecture MICRO 25*, 102–110. doi:10.1109/MICRO.1992.697004. [Cited on page 10.]
- GHASEMPOUR, M., 2015. *Workload-Adaptation in Memory Controllers*. Phd thesis, The University of Manchester. <https://research.manchester.ac.uk/en/studentTheses/workload-adaptation-in-memory-controllers/>. [Cited on page 8.]
- HASNAT, A., 2024. Analyzing and exploiting memory hierarchy parallelism with mlp stacks. COMP3770 Computing Research Project (R&D) report, Australian National University. [Cited on page 82.]
- HASNAT, A.; HEIRMAN, W.; AND AKRAM, S., 2025. Analyzing and exploiting memory hierarchy parallelism with mlp stacks. *IEEE Computer Architecture Letters*, 24, 1 (2025), 125–128. doi:10.1109/LCA.2025.3558808. [Cited on pages 1, 5, 7, 13, and 24.]
- HEIRMAN, W.; BOIS, K. D.; VANDRIESSCHE, Y.; EYERMAN, S.; AND HUR, I., 2018. Near-side prefetch throttling: adaptive prefetching for high-performance many-core processors. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, PACT '18 (Limassol, Cyprus, 2018)*. Association

- for Computing Machinery, New York, NY, USA. doi:10.1145/3243176.3243181. <https://doi.org/10.1145/3243176.3243181>. [Cited on page 16.]
- HELM, C. AND TAURA, K., 2020. Automatic identification and precise attribution of dram bandwidth contention. In *Proceedings of the 49th International Conference on Parallel Processing, ICPP '20* (Edmonton, AB, Canada, 2020). Association for Computing Machinery, New York, NY, USA. doi:10.1145/3404397.3404422. <https://doi.org/10.1145/3404397.3404422>. [Cited on page 14.]
- JAIN, A. AND LIN, C., 2018. Rethinking belady’s algorithm to accommodate prefetching. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 110–123. doi:10.1109/ISCA.2018.00020. [Cited on page 9.]
- KANELLOPOULOS, K.; SGOURAS, K.; BOSTANCI, F. N.; KAKOLYRIS, A. K.; KONAR, B. K.; BERA, R.; SADROSADATI, M.; KUMAR, R.; VIJAYKUMAR, N.; AND MUTLU, O., 2025. Virtuoso: Enabling fast and accurate virtual memory research via an imitation-based operating system simulation methodology. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS '25* (Rotterdam, Netherlands, 2025), 1400–1421. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3676641.3716027. <https://doi.org/10.1145/3676641.3716027>. [Cited on page 3.]
- KARKHANIS, T. AND SMITH, J., 2004. A first-order superscalar processor model. In *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, 338–349. doi:10.1109/ISCA.2004.1310786. [Cited on page 6.]
- KHADEM, A.; KAMALAKKANNAN, K.; ZHU, Z.; POPTANI, A.; GU, Y.; DOMINGUEZ-TRUJILLO, J. B.; TALATI, N.; FUJIKI, D.; MAHLKE, S.; SHIPMAN, G.; AND DAS, R., 2025. Dx100: Programmable data access accelerator for indirection. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture, ISCA '25*, 1641–1658. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3695053.3731015. <https://doi.org/10.1145/3695053.3731015>. [Cited on page 9.]
- KIM, Y.; SESHADRI, V.; LEE, D.; LIU, J.; AND MUTLU, O., 2012. A case for exploiting subarray-level parallelism (salp) in dram. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, 368–379. doi:10.1109/ISCA.2012.6237032. [Cited on pages 9 and 10.]
- KROFT, D., 1981. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual Symposium on Computer Architecture, ISCA '81* (Minneapolis, Minnesota, USA, 1981), 81–87. IEEE Computer Society Press, Washington, DC, USA. [Cited on page 8.]
- LIN, Y.; LIN, W.; CHEN, Y.; XU, J.; CAI, S.; YE, J.; WANG, Z.; AND CHEN, W., 2026. I-pop: Ignite positive prefetchers. In *2026 IEEE International Symposium on*

Bibliography

- High Performance Computer Architecture (HPCA)*, 1–16. doi:10.1109/HPCA68181.2026.11408454. [Cited on pages 16 and 53.]
- MEHTA, S., 2022. Performance analysis and optimization with little’s law. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 12–23. doi:10.1109/ISPASS55109.2022.00002. [Cited on page 13.]
- MUTLU, O. AND MOSCIBRODA, T., 2008. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *Proceedings of the 35th Annual International Symposium on Computer Architecture, ISCA ’08*, 63–74. IEEE Computer Society, USA. doi:10.1109/ISCA.2008.7. <https://doi.org/10.1109/ISCA.2008.7>. [Cited on page 10.]
- MUTLU, O.; STARK, J.; WILKERSON, C.; AND PATT, Y., 2003. Runahead execution: an alternative to very large instruction windows for out-of-order processors. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, 129–140. doi:10.1109/HPCA.2003.1183532. [Cited on page 5.]
- NATH, S.; NAVARRO-TORRES, A.; ROS, A.; AND PANDA, B., 2024. Secure prefetching for secure cache systems. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 92–104. doi:10.1109/MICRO61859.2024.00017. [Cited on page 9.]
- PANDA, B., 2016. Spac: A synergistic prefetcher aggressiveness controller for multi-core systems. *IEEE Transactions on Computers*, 65, 12 (2016), 3740–3753. doi:10.1109/TC.2016.2547392. [Cited on page 16.]
- PANDA, B., 2023. Clip: Load criticality based data prefetching for bandwidth-constrained many-core systems. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO ’23* (Toronto, ON, Canada, 2023), 714–727. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3613424.3614245. <https://doi.org/10.1145/3613424.3614245>. [Cited on page 15.]
- PANDA, B. AND BALACHANDRAN, S., 2014. Introducing thread criticality awareness in prefetcher aggressiveness control. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 1–6. doi:10.7873/DATE.2014.092. [Cited on pages 15 and 16.]
- PANDA, B. AND BALACHANDRAN, S., 2015. Caffeine: A utility-driven prefetcher aggressiveness engine for multicores. *ACM Trans. Archit. Code Optim.*, 12, 3 (Aug. 2015). doi:10.1145/2806891. <https://doi.org/10.1145/2806891>. [Cited on pages 16 and 53.]
- PUGSLEY, S. H.; CHISHTI, Z.; WILKERSON, C.; CHUANG, P.-F.; SCOTT, R. L.; JALEEL, A.; LU, S.-L.; CHOW, K.; AND BALASUBRAMONIAN, R., 2014. Sandbox

- prefetching: Safe run-time evaluation of aggressive prefetchers. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 626–637. doi:10.1109/HPCA.2014.6835971. [Cited on pages 12 and 16.]
- QURESHI, M.; LYNCH, D.; MUTLU, O.; AND PATT, Y., 2006. A case for mlp-aware cache replacement. In *33rd International Symposium on Computer Architecture (ISCA'06)*, 167–178. doi:10.1109/ISCA.2006.5. [Cited on pages 5 and 6.]
- SESHADRI, V.; KIM, Y.; FALLIN, C.; LEE, D.; AUSAVARUNGNIRUN, R.; PEKHIMENKO, G.; LUO, Y.; MUTLU, O.; GIBBONS, P. B.; KOZUCH, M. A.; AND MOWRY, T. C., 2013. Rowclone: fast and energy-efficient in-dram bulk data copy and initialization. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46 (Davis, California, 2013), 185–197. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2540708.2540725. <https://doi.org/10.1145/2540708.2540725>. [Cited on page 8.]
- SMITH, J. AND PLESZKUN, A., 1988. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37, 5 (1988), 562–573. doi:10.1109/12.4607. [Cited on page 6.]
- SRIDHARAN, A.; PANDA, B.; AND SEZNEC, A., 2017. Band-pass prefetching: An effective prefetch management mechanism using prefetch-fraction metric in multi-core systems. *ACM Trans. Archit. Code Optim.*, 14, 2 (Jun. 2017). doi:10.1145/3090635. <https://doi.org/10.1145/3090635>. [Cited on page 16.]
- SRINATH, S.; MUTLU, O.; KIM, H.; AND PATT, Y. N., 2007. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 63–74. doi:10.1109/HPCA.2007.346185. [Cited on pages 11, 12, and 16.]
- SUBRAMANIAN, L.; SESHADRI, V.; GHOSH, A.; KHAN, S.; AND MUTLU, O., 2015. The application slowdown model: quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48 (Waikiki, Hawaii, 2015), 62–75. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2830772.2830803. <https://doi.org/10.1145/2830772.2830803>. [Cited on page 14.]
- TANG, X.; KANDEMIR, M. T.; KARAKOY, M.; AND ARUNACHALAM, M., 2019. Co-optimizing memory-level parallelism and cache-level parallelism. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019 (Phoenix, AZ, USA, 2019), 935–949. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3314221.3314599. <https://doi.org/10.1145/3314221.3314599>. [Cited on page 13.]
- TUCK, J.; CEZE, L.; AND TORRELLAS, J., 2006. Scalable cache miss handling for high memory-level parallelism. In *2006 39th Annual IEEE/ACM International Symposium*

Bibliography

- on *Microarchitecture (MICRO'06)*, 409–422. doi:10.1109/MICRO.2006.44. [Cited on pages 5, 8, and 9.]
- VALSAN, P. K.; YUN, H.; AND FARSHCHI, F., 2016. Taming non-blocking caches to improve isolation in multicore real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 1–12. doi:10.1109/RTAS.2016.7461361. [Cited on page 9.]
- VAN DEN STEEN, S. AND EECKHOUT, L., 2018. Modeling superscalar processor memory-level parallelism. *IEEE Computer Architecture Letters*, 17, 1 (2018), 9–12. doi:10.1109/LCA.2017.2701370. [Cited on page 6.]
- WANG, X.; CHEN, S.; SETTER, J.; AND MARTÍNEZ, J. F., 2017. Swap: Effective fine-grain management of shared last-level caches with minimum hardware support. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 121–132. doi:10.1109/HPCA.2017.65. [Cited on page 9.]
- WU, C.-J.; JALEEL, A.; MARTONOSI, M.; STEELY, S. C.; AND EMER, J., 2011. Pacman: Prefetch-aware cache management for high performance caching. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 442–453. [Cited on page 9.]