Understanding Memory Prefetcher Performance over Parallel Applications: From Real to Simulated

Journal:	*Concurrency and Computation: Practice and Experience*
Manuscript ID	CPE-20-0556
Editor Selection:	Special Issue Submission
Wiley - Manuscript type:	Special Issue Paper
Date Submitted by the Author:	20-Apr-2020

Complete List of Authors:	Girelli, Valéria; Universidade Federal do Rio Grande do Sul, Informatics Institute Moreira, Francis; Federal University of Rio Grande do Sul (UFRGS), Informatics Institute Serpa, Matheus; Federal University of Rio Grande do Sul (UFRGS), Informatics Institute Santos, Danilo; Universidade Federal do ABC, ; Universite Grenoble Alpes, Navaux, Philippe; Federal University of Rio Grande do Sul, Informatics Institute

Keywords:	Parallel Architecture, Computer Architecture, Prefetcher, Architecture Simulation

Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

RESEARCH ARTICLE

Understanding Memory Prefetcher Performance over Parallel Applications: From Real to Simulated

Valéria S. Girelli* | Francis B. Moreira | Matheus S. Serpa | Danilo Carastan-Santos | Philippe O. A. Navaux

Informatics Institute,Federal University of Rio Grande do Sul – UFRGS, Porto Alegre, Brazil

Correspondence
*Valéria S. Girelli, UFRGS, Bento Gonçalves 9500, Porto Alegre – RS, Brazil.
Email: vsgirelli@inf.ufrgs.br

Summary

Memory prefetcher algorithms are widely used in processors to mitigate the performance gap between the processors and the memory subsystem. Analyzing and developing new prefetcher algorithms is possible thanks to architecture simulators that model the highly complex interactions present in the hardware. When we account for parallel, High-Performance Computing (HPC) applications, understanding the prefetcher's contribution to performance, on both the real hardware and in the simulations, becomes challenging. In this paper, we provide a better understanding of the memory prefetcher's role in the performance of HPC applications, considering the prefetcher algorithms offered by both the real hardware and the simulators. We performed a careful experimental campaign, executing the NAS parallel benchmark (NPB) on a real Skylake machine, and as well in a simulated environment with the ZSim and Sniper simulators, taking into account the prefetcher algorithms offered by both Skylake and the simulators. Our experimental results show that: (i) prefetching from the L3 to L2 cache presents better performance gains, (ii) the memory contention in the parallel execution constrains the prefetcher's effect, (iii) Skylake's parallel memory contention is poorly simulated by ZSim and Sniper, and (iv) Skylake's non-inclusive L3 cache hinders the accurate simulation of NPB with the Sniper's prefetchers.

KEYWORDS:
computer architecture, parallel architecture, prefetcher, architecture simulation

1 | INTRODUCTION

In recent years, there have been significant advances in the performance of processors, exemplified by the reduction of transistor size and the increase in the number of cores in a processor. Conversely, the memory subsystem did not advance as significantly as processors, not being able to deliver data at the required rate. This problem is referred to in the literature as the memory wall [1]. For this reason, there is a keen effort in computer architecture research to overcome these memory limitations. An example of a technology used to mitigate the memory latency is the prefetcher, a technique that identifies access patterns from each core, creates speculative memory requests, and fetches data – that can be potentially useful – to the cache beforehand.

In High-Performance Computing (HPC) systems, there are many other problems besides those inherent to the processors and memory individually. HPC applications are highly parallel, with many threads communicating with themselves, mainly through

shared memory. For instance, (i) several threads access the same memory addresses, making necessary to keep data coherence in the several cache levels, and (ii) the memory interactions among different threads may also unpredictably change the data path through the memory hierarchy. When considering the memory hierarchy complexity of HPC systems along with the action of the prefetcher, the behavior of the processor's memory subsystem reaches a new level of complexity. The hindrance lies, therefore, in understanding how the prefetcher affects the processing performance of parallel, HPC applications.

To further complicate the matter, in computer architecture research, physical implementation and analysis are infeasible due to the high complexity and manufacturing costs. Consequently, architecture simulators are considered the primary mechanism to implement and evaluate new ideas[2], such as new prefetcher algorithms. To develop and analyze new ideas that attenuate problems arising from the inherent parallelism of HPC systems, we need multicore architecture simulators that support parallel workloads. Studies about the memory hierarchy in HPC systems require parallel architecture simulators that consider the interactions that occur among the different cores.

In this work, we seek to shed light on the following questions: *(i) how does the prefetcher affect the processing performance of parallel, HPC applications?*, and *(ii) how can state-of-the-art architecture simulators accurately simulate HPC applications, with and without prefetcher?*

More specifically, this work presents the following set of contributions:

- We developed a careful experimental campaign, executing the Numerical Aerodynamic Simulation Parallel Benchmark (NPB)[3] using different levels of parallelism, in an Intel Skylake machine, and with a simulated architecture environment in the ZSim and Sniper simulators;
- We performed a holistic analysis of the NPB execution performance on both real and simulated scenarios, taking into account many many prefetcher algorithms offered in Skylake and in the Sniper simulator;
- We show experimental evidence that an L2 memory prefetcher is more efficient in comparison with an L1 prefetcher, since avoiding excessive L3 cache accesses better contributes to performance, when comparing to accessing the L2 cache;
- We show evidence that the prefetchers' contribution to performance is limited by the level of parallelism of the application, mainly due to the increase in communication and memory contention as the level of parallelism increases;
- The Skylake and NPB simulation with ZSim and Sniper had poor accuracy in predicting the NPB applications performance, with and without simulating prefetcher algorithms, mainly due distinctions of the models and prefetcher algorithms present in ZSim and Sniper when compared to Skylake.

In a previous work[4], we performed a first step on understanding the prefetcher behavior over parallel applications and its consequences when simulating with ZSim[5] that does not simulate prefetcher. In this paper, we extend our previous work by:

- Adding the Sniper[6] simulator – that does support simulation of prefetcher algorithms – in the study;
- By performing a more detailed analysis of the NPB benchmark performance on both real and simulated architecture executions;
- By performing experiments with the Intel Skylake architecture, which is a more modern architecture, when compared to Intel Sandy Bridge used in the previous study;
- By adopting the instructions per cycle (IPC) metric to evaluate the performance of the NPB benchmark. The IPC is arguably a more representative performance metric when compared to the number of cycles used in our previous study.

We organized the remainder of this paper in the following manner: In Section 2 we provide a brief explanation of memory prefetcher, and we briefly describe the ZSim and Sniper simulators. Section 3, we present the related works about HPC benchmark applications, prefetcher algorithms, and architecture simulators. In Section 4, we describe the experimental campaign performed in this work. In Section 5, we present the experimental results, the main findings and observations obtained by our study, and in Section 6, we give our concluding remarks, and we discuss future research directions.

We follow a reproducible and open methodology in our investigation. A companion material of this work is publicly available at https://gitlab.com/msserpa/prefetcher-ccpe containing the application code, the data analysis code, and all data collected during experiments that culminated in this paper.

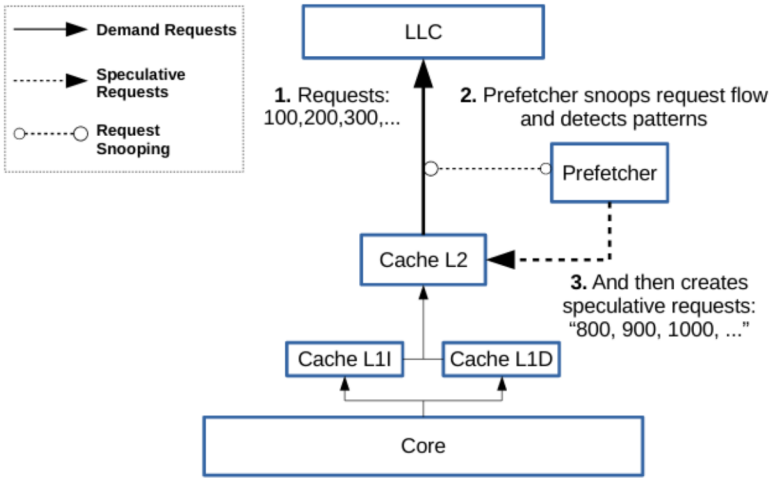

FIGURE 1 Abstraction of the prefetcher behavior.

2 | PRELIMINARY DEFINITIONS

2.1 | Prefetcher

The memory available for a processor is organized in different levels. Private levels closer to the processor have less storage capacity and provide more efficient data access. When a processor issues a request for data to the memory, this request is delivered to the first level of the processor's internal memory hierarchy, specifically to the first level data cache memory (L1). This memory is relatively small (32 kilobytes) and presents low latency (4 processor cycles). The translation lookaside buffer (TLB) is accessed in parallel to check whether the page is physically present in the main memory, and the virtual address to physical address translation. If the translation is not found in any TLB level, a page table walk must be performed. If the page is not in the page table, a page fault occurs, meaning the page does not currently reside in memory. Thus a request to a secondary storage device needs to be performed, incurring high latency.

If a copy of the requested data is not in the L1 data cache, it forwards the request to the next level of memory cache, which repeats the same procedure. Normally, this next level is larger to increase the probability of finding the data, but the increased size incurs higher latencies for cache line searches. A typical L2 has 256 KB and a 7-cycle latency. When added to the L1 latency, this means a memory request for data that misses L1 cache and hits L2 cache takes 11 cycles to be provide the data to the processor. On a miss in the last level of memory cache (Last Level Cache - LLC), it forwards the request to the main memory, which has even greater latency. Hence, while performing a memory request, finding the data in closer cache levels is preferable for the application performance, as the latency is lower.

To reduce the average data access latency, the prefetcher was created. Based on the pattern of accesses generated by the processor, a prefetcher speculates which might be the next address to be requested and performs the data request in advance. Thus, when the data block is actually requested by the processor, it will already be in cache levels closer to the processor. Therefore, prefetchers are prevalent in current architectures as a technique to mitigate the main memory access bottleneck, reducing load-to-use latency and spreading accesses through time to avoid contention[7]. Examples of common patterns are stride[8] and stream[9].

Figure 1 shows an example of a second level of cache (L2) prefetcher detecting a stride access pattern. The L2 cache forwards requests to the LLC (shown in Figure 1 as the event **1**). The prefetcher, in turn, intercepts these requests by snooping the cache interconnection (**2**) and identifies the access pattern being generated. Based on the pattern identified, speculative accesses are inserted into the L2 Miss Status Holding Register (MSHR) (**3**), a buffer that keeps track of miss events that still need to be handled. These speculative requests are made directly to the L2 cache in order to avoid a redundant cache fill if the speculated line already resides in the cache. These accesses are seen as regular requests made to the L2 by the prefetcher, so the L2 does not actually need to forward the response to the L1. If the speculated address is not present in the L2 yet, the next levels in the

hierarchy will forward the response to whoever requested it, as in a regular access. Thus, when the processor needs the data requested by prefetch, it will already be at a closer cache level (in this case, the L2 cache).

2.2 | Simulators

In computer architecture research, physical implementation and analysis are infeasible due to the complexity and high cost for manufacture. Consequently, architecture simulators are considered the primary mechanism to implement and evaluate a new idea in this research field. In High-Performance Computing (HPC) systems, there are many other problems besides those inherent to the architecture. To develop and analyze new ideas that attenuate problems arising from parallelism, we require multicore architecture simulators that support parallel workloads. For example, in systems with dozens of cores, problems such as the interference among the different threads and the communication costs among them must be modeled to obtain accurate results. Thread interactions occur mainly through shared memory, with several threads accessing the same memory addresses, making necessary to keep data coherence in the several cache levels. Thereby, parallel simulators must provide an accurate implementation of cache coherence protocols and ensure data consistency through the memory hierarchy.

In the following subsections, we provide some details about the implementations of the two simulators used in this work, ZSim[5] and Sniper[6].

2.2.1 | ZSim

ZSim was selected for our study due to its speed and accuracy, characteristics presented in its validation study[10]. It is an instruction-driven simulator that uses dynamic binary translation (DBT) to perform the instruction execution and dynamic instrumentation. The simulation uses a two-phase method called Bound and Weave. In the Bound phase, a few thousand cycles are simulated, ignoring the contention and applying a minimal latency for all memory accesses. In this phase, a trace of all memory accesses is recorded, including which caches lines were accessed, evicted, invalidated, and so on. In the Weave phase, a parallel simulation is performed, oriented to the events of the recorded interval to determine the actual latency of each memory request in each component. Once the interactions between memory accesses have already been identified in the first phase, this timing simulation of these accesses can be done efficiently, maintaining high precision.

The authors observed that, in an interval of a few thousand cycles, most of the concurrent accesses between different cores happen to unrelated cache lines. Therefore, simulating these unrelated accesses first out of order and ignoring contention and, later, simulating them in order respecting time constraints, is equivalent to simulating them entirely in order. However, when accesses are related, i.e., access the same cache line, it is necessary to maintain the coherence of the different copies of the data in the different cores. An example of this is when a core demands exclusive access over a shared cache line to write into it, causing the cache coherence protocol to invalidate the other copies of the line in the cache hierarchy system.

The set of requests and messages necessary to invalidate the other copies of the line to obtain it as exclusive is known as Request for Ownership - RFO. However, for being considered rare, the order of these accesses to the same line is not modeled by ZSim, which can change the path of this data in the cache hierarchy. Changing the path of the data through the cache can impact the number of cycles, misses, and coherence messages observed in the simulation. In addition, the generation and the paths of prefetch requests can also change, preventing the modeling of prefetcher in this simulation model.

In its validation, ZSim uses the benchmark PARSEC[11] and shows only that the speed-up is close to that obtained with real executions by varying the number of threads. Therefore, since the prefetcher is not simulated, we seek to understand the impact of its absence and evaluate the accuracy of the Bound and Weave simulation method used to simulate multiple structures and threads in parallel.

2.2.2 | Sniper

Sniper is a simulator that extends the original interval simulation model[12] to improve its handling of overlapping memory accesses through a more detailed dependency analysis of memory accesses[6]. The simulator presents the instruction-window centric (IW-centric) simulation model, a high-level core model that combines the interval modeling with a detailed simulation model of the instruction window, or reorder buffer (ROB). The IW-centric simulation model focuses on accurately simulating the parallelism and latencies of instructions' execution in the processor. This is done by modeling micro-op dependency and issue timing in detail, estimating the performance by processing micro-ops out of order, in a way similar to how a real processor

would issue them. However, the IW-centric model requires a longer simulation time. For that reason, in this study we make use of their improved interval simulation model.

The basis for interval analysis is the observation that, in the absence of miss events such as branch mispredictions and cache misses, a well-balanced superscalar out-of-order processor should smoothly stream instructions through its pipelines, buffers, and functional units[12]. Under ideal conditions the processor sustains a level of performance (instructions per cycle) roughly equal to the superscalar dispatch bandwidth[12]. However, the smooth dispatch of instructions is intermittently disrupted by memory request miss events and branch mispredictions. The effects of these events at the dispatch stage divide execution time into intervals, and these intervals serve as the fundamental entity for analysis and modeling[13]. The original interval modeling is itself a simulation model that allows the simulation of prefetchers in Sniper.

The interval simulation works with two structures named new window and old window. Each window contains as many micro-ops as would exist in the reorder buffer (ROB) of an out-of-order processor. The new window represents the upcoming micro-ops and is fully filled the entire time, and the old window contains a list of the most recently dispatched micro-ops. The instantaneous IPC is calculated as the number of micro-ops in the entire old window divided by the latency of the micro-ops on the critical path[12]. By repeating this process for each micro-op (and accumulating the leftover work for future micro-ops), Sniper can estimate the application's instruction level parallelism (ILP) during the non penalty portion of an interval.

In its validation, Sniper uses the benchmark SPLASH-2[14] and also shows that the average speed-up error is small when varying the number of threads. Although Sniper provides an implementation of the prefetcher's behavior, it has only been recently validated for the Cortex-A53 and Cortex-A72 ARM cores[15]. The prefetcher model is responsible for over half of the simulation error in the *povray* and *x264* benchmarks, as some implementation details of the real hardware are unknown. However, the validation used only sequential benchmarks. In contrast, this study contributes to a better understanding of the accuracy of the simulated prefetch model and how it behaves in simulations of parallel architectures.

3 | RELATED WORK

Due to the lack of explicit information that hardware companies employ to avoid competition and breaches of their intellectual property, it is difficult to obtain an accurate simulation that correctly presents all the characteristics of a processor and its architecture. Therefore, designers of a simulator seek a low relative error when validating their simulator, and tailor the details of the simulator to make it sensitive specifically to the desired area of design, e.g., memory hierarchy[16].

For instance, if we variate the amount of cache memory and try to simulate a target program that requires a large amount of cache memory, the simulator should indicate loss or gain of performance. If it does not, either it does not model the cache correctly or suffers from bottlenecks in other parts of the model.

3.1 | Improving Simulator Accuracy

Microbenchmarks can be used to decrease error in basic architecture structures and as a form of reverse engineering of architectural features[17]. In Desikan et al.'s work[18], the authors faced the difficulty of obtaining accurate information and studied the relevance of this information when validating the SimpleScalar simulator[19]. By obtaining more information about the Alpha 21264 processor model in direct contact with the manufacturers, the authors were able to reduce the experimental microbenchmarks error from 19.5% to 2%. With the SPEC-CPU 2000 workload, the average experimental error went from 36.7% to 18.2%. The authors then showed how the resources found generate bottlenecks in different parts of the system than previously modeled in several articles, invalidating ideas that presented performance gains due to bottlenecks that did not exist.

A more recent work by Walker et al.[20] in this direction automates the process of finding the source of simulator error. The main objective of the work is to obtain more accurate energy consumption models for gem5, but to do so they required a more accurate processor model. Using gem5 and the processor configurations provided by Gutierrez et al.'s gem5 validation[21], Walker et al. created GemStone, a framework to find the sources of simulation error based on empirical hardware performance monitor counters (PMCs) models. GemStone selects the events in gem5, correlating them with the PMC events, and at last, performs one regression analysis to approximate the relationship between hardware PMCs and gem5 error, and another to approximate the relationship between gem5 events and gem5 error.

3.2 | Prefetcher Simulation

The development of new prefetching techniques is a very active area. Mittal[22] provides a survey on recent development of prefetching techniques up to 2016. In the survey, the authors describe the relevant prefetch metrics, such as accuracy, coverage, and timeliness[23], as well as the different types of technique to improve prefetching, such as new pattern detection techniques[24], filtering prefetches[25], dropping prefetches[26], changing prefetches' priority on the memory controller[27], and so on. Given all of these different techniques and the active development of new techniques, one can hardly keep up with the design of prefetchers and additional techniques in the industry, which are not publicly disclosed. Thus, most simulator designers face difficulties when modeling prefetchers, as they cannot detail all the techniques used in real hardware. Interestingly, many of these techniques are not tested in multithreaded applications where communication plays a major role in the memory hierarchy latency[28,29,30], which we explore in this work.

3.3 | Simulator Survey

Since Desikan's work, several simulators have been created with along a validation effort to evaluate their accuracy. In Akram et al's research[31], the authors evaluated and compared the gem5[32], Multi2Sim[33], MARSSx86[34], PTLsim[35], Sniper[6], and ZSim[5] simulators. After thorough characterization of the simulators, the authors performed experiments with single core and multi-programmed workloads on each simulator. The results of each simulator were compared with the Intel's Haswell architecture[36]. The authors then highlighted the error sources of the simulators, their sensitivity to different architectural parameters, and their relative error. Thus, they concluded that the lack of validation of a simulator for the target architecture, no matter how popular it is (e.g., gem5, which was only validated for ARM[21]), leads to low accuracy and may render experiments invalid due to erroneous conclusions.

The two best simulators, in terms of accuracy, were Zsim and Sniper, which we selected for this work. However, Akram's work does not use multi-thread applications to verify the error relative to the number of threads and the accuracy of communication in these simulators.

4 | METHODOLOGY AND EXPERIMENTAL ENVIRONMENT

In this Section, we present details about the experimental environment and the prefetcher algorithms provided by the real machine hardware and by Sniper. Moreover, we describe the methodology applied in the experiments, the benchmark used and the different prefetcher combinations executed and simulated.

4.1 | Experimental Setup

To collect information about the application execution in the real machine, we make use of the perf[37] tool. Perf allows the obtaining of hardware counters values, a set of registers that provide information about CPU events such as the number of instructions and the number of cycles. In each real execution, only one hardware counter is evaluated, thus avoiding aggregations or approximations that the tool performs when calculating several metrics in the same execution. The performance and the real executions statistics were evaluated with 10 executions of each metric, for each benchmark. The execution environment was composed of an Intel Xeon Silver 4116 CPU, the Skylake microarchitecture[38]. The simulation environment is an approximation of the real hardware, respecting the simulators' limitations. Each simulation was executed only once, and all statistics were extracted from the same simulation.

Table 1 presents the configuration of the real machine and the processor simulated by ZSim and Sniper. The Sniper's out of order core model was based on the Nehalem architecture[39], while ZSim based its out of order core model implementation on the Westmere architecture[5], a process shrink of Nehalem. Therefore, both simulators present a 16 stages pipeline, while the real machine architecture may present between 14 and 19 stages[17]. Other parameters such as cache associativity and cache access latency are easy to configure in the simulators and can be found in[17,38].

In Table 2, we present the description of the prefetcher algorithms considered in this work. The algorithms found in the L1 cache hardware are the Data Cache Unit (DCU) Prefetcher[40] and the DCU IP Prefetcher[40]. The DCU Prefetcher, also known as the streaming prefetcher, is triggered by an ascending access to very recently loaded data. The processor assumes that this access is part of a streaming algorithm and automatically fetches the next line. The DCU IP Prefetcher keeps track of individual load

instructions (based on their instruction pointer's value). If a load instruction is detected to have a regular stride, then a prefetch is sent to the next address which is the sum of the current address and the stride.

The L2 Hardware Prefetcher[40] and the L2 Adjacent Cache Prefetcher[40] are the prefetcher algorithms found in the real machine L2 cache. The L2 Hardware Prefetcher monitors read requests from the L1 cache for ascending and descending sequences of addresses. Monitored read requests include L1 data cache requests initiated by load and store operations and also by the L1 prefetchers, and L1 instruction cache requests for code fetch. When a forward or backward stream of requests is detected, the anticipated cache lines are prefetched. This prefetcher may issue two prefetch requests on every L2 lookup and run up to 20 lines ahead of the load request. The L2 Adjacent Cache Prefetcher fetches two 64-byte cache lines into a 128-byte sector instead of only one, regardless of whether the additional cache line has been requested or not.

The prefetch algorithms provided by Sniper are the Simple and the Global History Buffer (GHB)[24]. Based on an analysis of the Simple prefetcher's code, we observed that it is similar to a strided prefetcher algorithm. Therefore, in our experiments we use the Simple prefetcher as the L1 cache prefetcher. The GHB prefetcher is an n-entry FIFO table that holds the n most recent L2 misses addresses. Each GHB entry stores a global miss address and a link pointer that is used to chain the GHB entries into address lists. Each address list is a time-ordered sequence of addresses that were issued by the same instruction pointer. Therefore, based on the information of the address lists, is possible to implement a correlation based prefetcher[41] and a stride prefetcher[24]. The algorithm found in Sniper corresponds to a GHB correlation based prefetcher, and in our experiments it is used as the L2 cache prefetcher.

Therefore, based on the aforementioned prefetchers, we conducted experiments on the real machine considering:

- All Skylake's L1 cache prefetchers, henceforth referred to as Real-L1 prefetcher, or simply as L1 prefetcher;
- All Skylake's L2 cache prefetchers, henceforth referred to as Real-L2 prefetcher, or simply as L2 prefetcher;
- All Skylake's prefetchers from both L1 and L2 cache, henceforth referred to as Real-L1-L2 prefetcher, or simply as L1+L2 prefetcher;
- No prefetcher, henceforth referred to as Real-NoPref.

Based on the Sniper's prefetchers, the following systems were simulated:

- Only the L2 cache prefetcher (GHB), henceforth referred to as Sniper-L2 prefetcher, or simply as Sniper's L2 prefetcher;
- Both L1 (Simple) and L2 (GHB) cache prefetchers, henceforth referred to as Sniper-L1-L2 prefetcher, or simply as Sniper's L1+L2 prefetcher;
- No prefetcher, henceforth referred to as Sniper-NoPref.

TABLE 1 Real machine, ZSim and Sniper configurations.

	Real	**ZSim**	**Sniper**
Processor Frequency	2.1 Ghz	2.1 Ghz	2.1 Ghz
Number of Cores	12	12	12
Pipeline Stages	18	16	16
Cache Line Size	64 B	64 B	64 B
L1 Data Cache	8-way 32 KiB	8-way 32 KiB	8-way 32 KiB
Latency	4	4	4
L1 Instruction Cache	8-way 32 KiB	8-way 32 KiB	8-way 32 KiB
Latency	4	4	4
Unified L2 Cache	16-way 1 MiB	16-way 1 MiB	16-way 1 MiB
Latency	ca. 14	14	14
Last Level Cache L3	Non-inclusive 11-way 16.5 MiB	Inclusive 11-way 16.5 MiB	Inclusive 11-way 16.5 MiB
Latency	ca. 60-80	77	77
Prefetchers	L1 IP Stride Adjacent Line prefetcher L2 DCU Stream prefetcher	No Prefetcher	Simple (Stride L1 Prefetcher) Global History Buffer (L2 Prefetcher)

TABLE 2 Prefetcher algorithms.

	Description	L1	L2	Real	Sniper
DCU Prefetcher	Streaming prefetcher, fetches the next cache line into L1-D Cache	✓		✓	
DCU IP Prefetcher	Strided prefetcher of next L1-D line based upon sequential load history	✓		✓	
L2 Hardware Prefetcher	Mid Level Cache (L2) streamer prefetcher		✓	✓	
L2 Adjacent Cache Prefetcher	Prefetching of adjacent cache lines into L2 Cache		✓	✓	
Simple	Strided prefetcher of L1-D line	✓			✓
Global History Buffer	L2 Prefetcher based on global miss addresses		✓		✓

Since ZSim does not model the prefetcher behavior, there are no variations of the simulated system. With ZSim we only perform simulations with no prefetcher, which we refer to as ZSim-NoPref.

4.2 | NAS Parallel Benchmarks

For this study, we used a known HPC benchmark in the literature called Numerical Aerodynamic Simulation Parallel Benchmark (NPB)[3]. This benchmark comprises ten applications, each of which encapsulates a certain type of computation that is often processed by HPC applications. We used nine of the ten NPB applications, namely:

- **CG**: *Conjugate Gradient* uses a conjugate gradient method to approximate the smallest eigenvalue of a defined, large and sparse, symmetric matrix. This application tests for irregular memory accesses and long-distance communication between threads, employing matrix multiplication by a structureless vector;
- **EP**: *Embarrassingly Parallel* solves the problem of generating pairs of Gaussians and tabulating their values in successive annuli squares. Unlike the other NPB applications, EP has virtually no communication, only stressing the performance of floating-point operations;
- **FT**: *Fourier Transform* solves the problem of the three-dimensional Fourier transform in a parallel way using the Fast Fourier Transform (FFT) algorithm. FT strictly tests high-distance communication with an all-to-all core communication pattern;
- **IS**: *Integer Sort* uses Bucket Sort to sort a vector of integer numbers. IS has random memory accesses and tests both the integer operations performance and communication;
- **MG**: *Multigrid* solves a simplified multi-grid calculation problem by testing short- and long-distance communications. MG requires highly structured long-distance communication;
- **UA**: *Unstructured Adaptive Mesh* solves a problem of heat transfer in a cubic domain, discretized in an unstructured adaptive mesh;
- **BT**: *Block Tridiagonal* solves a synthetic Computational Fluid Dynamics (CFD) problem with multiple 5x5 tridiagonal block equation systems with non-dominant diagonals. BT has more limited parallelism than other CFD applications (LU and SP);
- **LU**: *Lower Upper Gauss-Seidel Solver* solves a synthetic CFD problem by solving a system of sparse triangular linear equations, with 5x5 blocks. LU involves global data dependencies, with several short communications;
- **SP**: *Scalar Pentadiagonal* solves systems of pentadiagonal scalar equations with non-dominant diagonals. Like LU, SP involves global data dependencies, though with more infrequent and longer communications.

We discarded the Data Cube (DC) application because DC mainly stresses I/O operations, which are not modeled by the architecture simulators.

5 | RESULTS AND PERFORMANCE ANALYSIS

5.1 | Real Machine Execution

In this section, we present the main results obtained by the aforementioned experimental setup, taking into account the real execution of the NPB benchmark. More specifically, here we shed light to the following question: *How do the prefetchers behave when executing NPB over a real machine and how does their behavior affect the execution of NPB?*

Figure 2 shows the instructions per cycle (IPC) for each NPB application, taking into account different numbers of threads and prefetchers, and with hardware prefetcher disabled as well. The IPC can be understood as a general performance metric of the prefetchers when executing the applications, since a more efficient prefetcher will enable more processed instructions per cycle. The error bars represent the variability observed among both the cores of a certain execution and the 10 distinct repetitions of the executions.

As naturally expected, we can observe that any prefetcher in fact increases the execution performance of NPB, with the exception of the EP application. This expected, since the EP benchmark is known to not make much use of memory[3], thus processor stalls due to memory access latency rarely occur during execution. Therefore, there is little difference in the IPC given

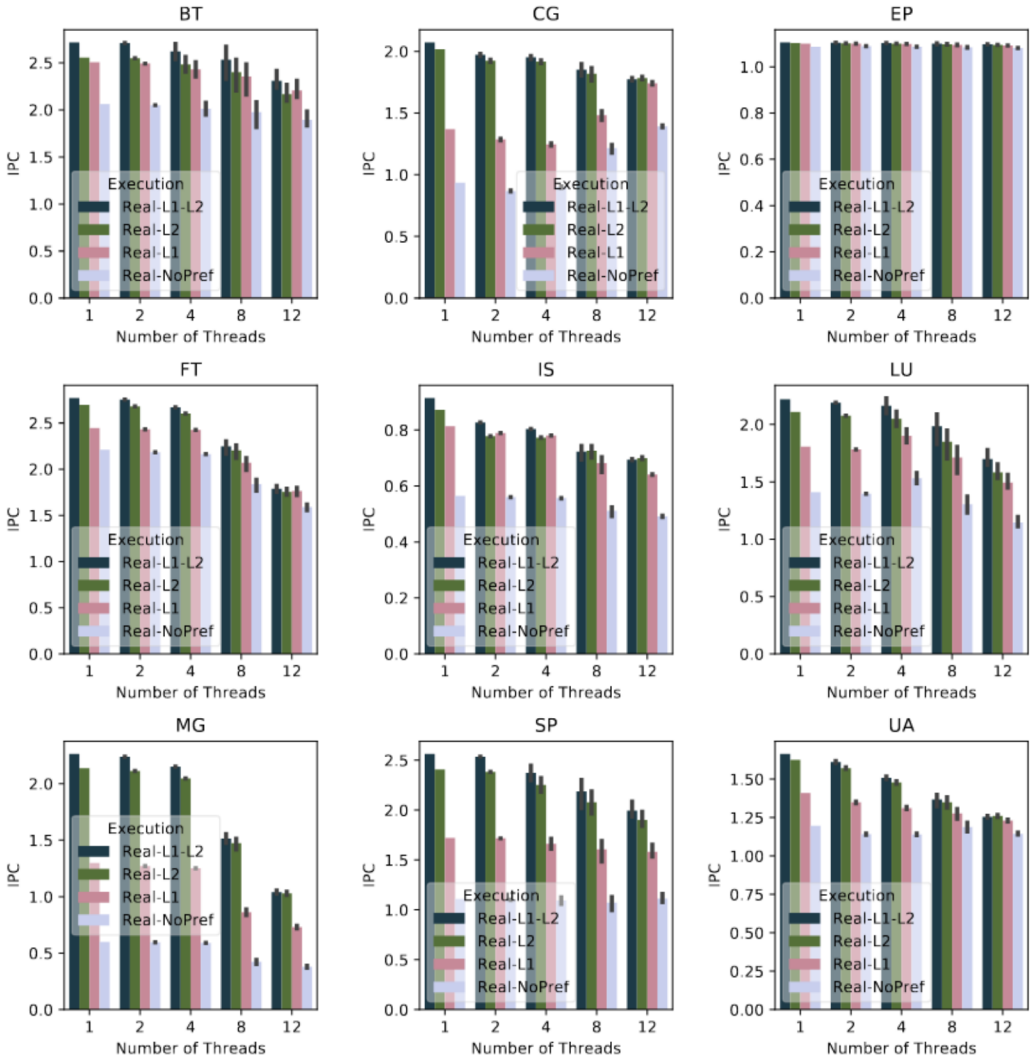

FIGURE 2 IPC results for the real execution of the NPB applications.

different prefetcher configurations for EP. For the other applications, memory prefetching significantly contributes to a better IPC, specially for executions up to four threads.

However, the increase in performance from a standalone L2 prefetcher to the combination of L1+L2 prefetchers is not as large, with an overall average of 3.3% of IPC improvement, when compared to the performance increase from no prefetcher to the standalone L2 prefetcher. The likely explanation to this phenomenon is that the L2 prefetcher is able to detect more relevant memory access streams, since it observes higher latency accesses that would go to the LLC, while L1 access patterns detect access streams that might be in L2. Since the difference in latency from L1 to L2 is small, and the latency from L2 to LLC is much larger, this means the main performance gains would be obtained by the L2 prefetcher and the associated access streams it detects. For concrete numbers, the L2 cache access latency is of 14 cycles, only 10 cycles higher than the L1 cache, while the LLC latency in Skylake is measured to be approximately between 60 and 80 processor cycles, presenting a much more substantial overhead and a higher probability of stalling the processor execution[42]. A complementary explanation for this small difference between the L1 and the L2 prefetchers is that the effect of the L1 prefetcher may occupy too many entries on the already limited line fill buffers present in the L1 hardware. This may create contention for demand accesses, which are more relevant for the immediate processor execution than the prefetched lines.

The L1+L2 prefetcher combination is often set as the default setting in the machine configurations. One of its main appeal is that it is the setting that provides the best performance. One of its drawbacks, however, lies in the interpretability of what is being performed by the algorithm, since the multi-hierarchy prefetches hinders the overall understanding. In its turn, the L2 prefetcher provides a more transparent understanding of the prefetches performed by the algorithm, with the drawback of not having the best performance. At the light of these results, it may be advisable to set the L2 prefetcher on its own, as opposed to the L1+L2 setting, since the lack of transparency of the L1+L2 prefetcher, the L1 line fill buffer contention, and the additional energy consumption of the L1 prefetcher may not outweigh its increase in performance.

Another interesting trend observed in Figure 2 is the decrease in the IPC for increasing number of threads in the majority of applications and prefetchers, with the exception EP application. Apart from EP – which, for the same reasons explained above, the IPC is stable for any number of threads on all cores – the NPB applications use memory accesses and inter-thread communications, that naturally increase in function of the total number of threads. With a large number of threads, these memory accesses and inter-thread communications generate contention that, it its turn, become a larger constraint in the IPC, and the performance provided by prefetchers becomes less predominant. This effect can be clearly seen in the IPC graph for the MG application: MG is known to use memory intensive operations, thus it largely benefits from memory prefetchers. However, as the number of threads increases, the communication – which is intensive in MG as well – becomes a larger constraint, resulting in lower IPC values.

In this regard, we can generalize a hypothesis that, in cases where the number of concurrent threads executed in a processor is very high, the performance benefit from memory prefetchers for applications that use intense memory and inter-thread communications would be negligible. These characteristics of the applications would need to be alleviated, in order for them to effectively benefit from memory prefetchers.

An interesting exception to the observation above is the executions of the CG application with the L1 prefetcher and without prefetcher (Figure 2), where the IPC increases in function of the number of threads. This effect may be due to the irregular

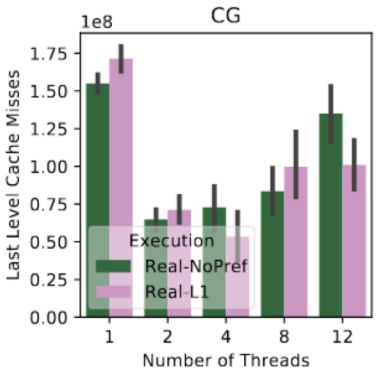

FIGURE 3 Last Level Cache (LLC) misses for the CG application, using no prefetcher and the L1 prefetcher.

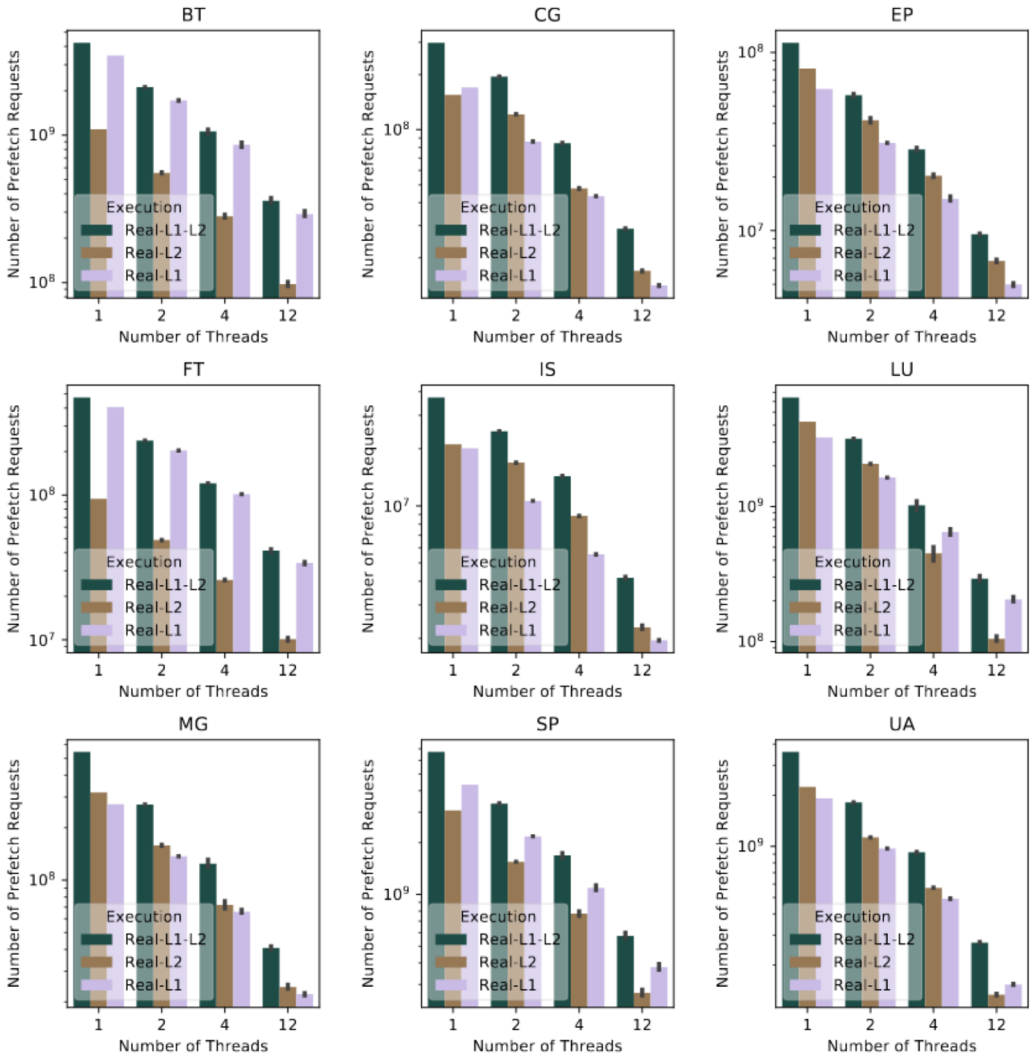

FIGURE 4 Number of prefetch requests of the real machine executions (log scale on *y* axis).

access nature of the CG application, as mentioned in Section 4.2: since memory accesses are irregularly distributed among the threads in CG, one thread may request a memory address that was already requested by another thread – or even by the effect of the another thread's prefetcher, that was not able to correctly predict the irregular access pattern. Thus, it will be likely that this memory address will be already in the LLC (which is shared among cores in Skylake), making it easier to be accessed from the L2 and L1 caches – and easier to be prefetched as well – by the other cores. This effect is less likely to happen in applications with regular memory accesses, where the memory addresses required by the threads are more uniformly partitioned.

To strengthen the above hypothesis, in Figure 3 we show the number of LLC misses for the CG application in function of the number of threads, and considering no prefetcher and the L1 prefetcher. We can observe that the number of misses falls for more than a half when we increase from 1 to 2 threads, exactly because one thread may bring data to the cache cache system that could be used by the other thread. From 2 to 12 threads the LLC misses increase because, although it is more likely that the data required by a thread will be in the LLC by the effect of other thread requests, if a thread modifies this data, the cache coherency protocol will invalidate that copy of the data in the cache system. Therefore, a new request for this data will result in a miss in the LLC.

In Figure 4 we present the total number of prefetch requests performed by the different prefetcher combinations in the real execution. An important observation is the small difference between the total number of prefetches performed by the L1 prefetcher and the execution considering both L1 and L2 prefetchers. Although the number of prefetches issued by the L1 prefetcher is

close to the number of prefetches issued by the other prefetches (notably the L1+L2 prefetcher), this number does not translate into performance in terms of IPC, as seen in Figure 2. This behavior is expected since the requests made by the L2 prefetcher are the most important in the application's performance by avoiding accesses in more distant cache levels, which present the largest access latencies.

5.2 | Simulation Results

As discussed in Section 1, parallel architecture simulation tools are necessary to develop and evaluate new techniques such as new prefetcher algorithms. In this regard, a key requirement is that these simulation tools can effectively simulate the parallel architecture and the parallel applications, bringing similar values of the metrics when compared to the real execution. In this section we therefore aim to clarify the following question: *Using the ZSim and Sniper simulators, how do they behave and how accurately do they simulate NPB, accounting distinct prefetchers when possible?*

While simulators are a useful tool to test new architecture techniques, one drawback is the large amount of time that is often required by the simulations. For instance, the time to simulate the SP and BT applications of NPB using Sniper took more than one week. Sinuca[42], which is another parallel architecture simulator, took even more time to simulate, exceeding the maximum execution time allowed by our computing infrastructure. In this regard, we had to discard Sinuca from the study and with Sniper we simulated only a subset of the NPB applications, notably CG, EP, FT, IS, LU, and MG. We also only analyzed the simulation of prefetchers with the Sniper simulator, since ZSim does not simulate memory prefetchers.

Figure 5 shows the obtained IPCs when no prefetcher is used for six of the NPB applications, namely CG, EP, FT, IS, MG, and LU. For that we disabled the prefetcher on both the real execution and the Sniper simulation. We did not make any specific changes to ZSim for this experiment, since ZSim does not support prefetcher simulation.

The only application where both simulators follow the real execution performance tendency is the EP. Since EP makes very little use of communication, it results in a simulation with very little contention events. This makes it easier for ZSim and Sniper to simulate EP, since simulating the effects of contention is arguably one of the most complex tasks in architecture simulation because of the out of order nature of contention events[5]. However, for applications where communication and contention are more predominant, we can notice discrepancies between the simulation and the real execution. This discrepancy can be quite extreme. For instance, ZSim's MG simulation with 12 threads resulted in a average IPC 1.88 times higher than in the real execution, and Sniper's CG simulation with 12 threads resulted in an average IPC 4.6 times lower than in the real execution. The CG application is an interesting case because both Sniper and ZSim do not seem to accurately simulate the parallel CG

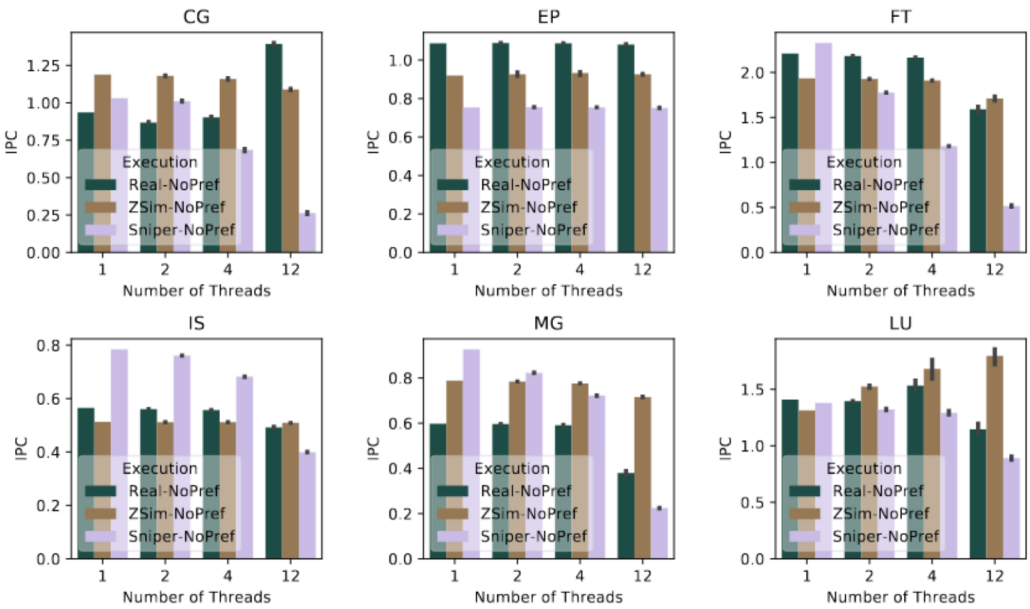

FIGURE 5 Experimental results comparing the performance of simulation and real execution without prefetcher.

execution (see Section 5.1). Both simulators predict a decrease in IPC as the number of threads increase, whereas in the real execution the trend is the opposite.

As aforementioned, accurately modeling contention is challenging. As mentioned in Section 2.2.1, in ZSim's case it is assumed that most concurrent accesses happen to unrelated cache lines when considering small time scales. In that way, ZSim was engineered in such a way that requests to the same cache line may be simulated in a different order than the observed in the real execution[5]. Therefore, the path of the data through the cache hierarchy may change, resulting in simulation inaccuracies. This may explain the discrepancies of ZSim when simulating CG, as it is straightforward to devise that the concurrent and irregular

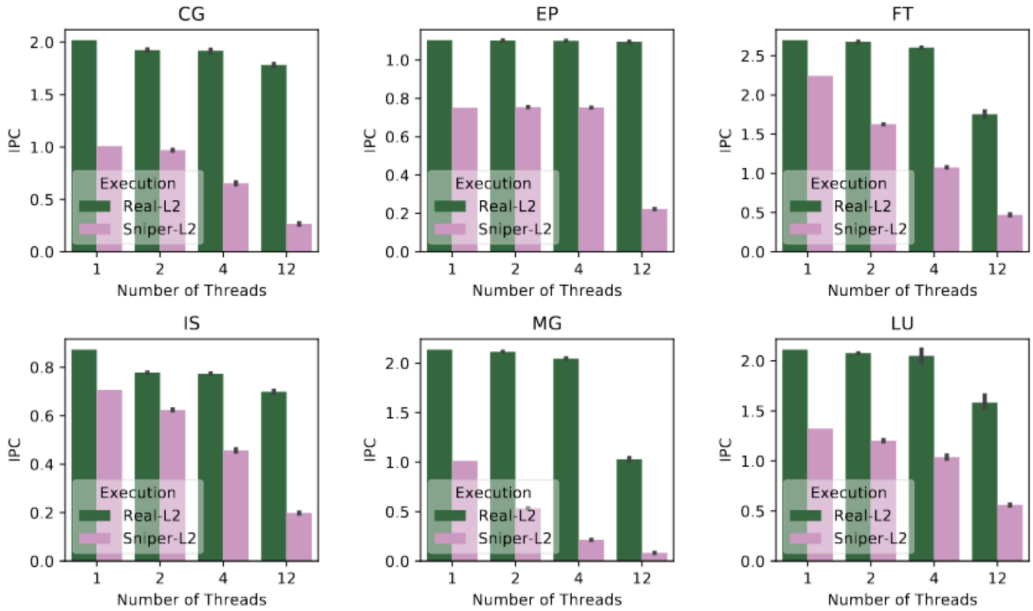

FIGURE 6 Performance results of Sniper's L2 prefetcher simulation and real hardware with L2 prefetcher.

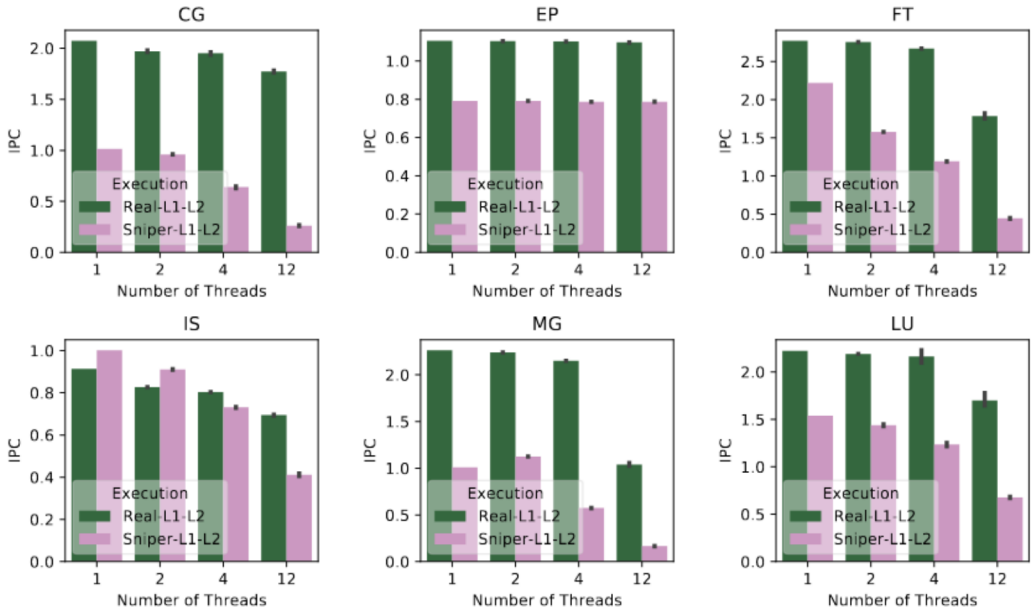

FIGURE 7 Performance results of Sniper's L1+L2 prefetchers simulation and real hardware with L1+L2 prefetchers.

memory accesses present in CG lead to concurrent accesses in same cache lines. The aforementioned explanation can also be the case for the LU application, though a more detailed memory access analysis of LU is required to attest this argument.

As a general trend, for NPB applications with communication and contention, ZSim tends to underestimate the contention effects, while Sniper tends to overestimate the contention effects as the number of threads in the simulation increases. For ZSim, this fact can be due to the above-mentioned design assumptions made during its development. For Sniper, in its turn, the reason can be more complicated, as it has been reported[6] that simulation errors in Sniper's memory model happen when simulating parallel applications with the interval model (see Section 2.2.2).

In Figures 6 and 7, we compare the performance of the Sniper's L2 prefetcher with the L2 prefetcher of the real machine, and the Sniper's L1+L2 prefetcher with the real counterpart, respetively. In all evaluated NPB applications, we can notice discrepancies between the Sniper simulation results and the real machine results, with Sniper mostly underestimating the IPC performance, and overestimating the communication and contention effects with increasing number of threads. Many reasons can explain this fact. First, as mentioned above, Sniper presents errors in its memory model when simulating parallel applications, even without considering prefetcher. These errors may be further amplified by adding a memory prefetcher in the simulation. Second, Global History Buffer (GHB)[24] L2 prefetcher implemented in Sniper differs from the Skylake's Stream[40] L2 prefetcher. In this regard, the GHB prefetcher offered by Sniper may not be suited to simulate Skylake. Modeling prefetcher algorithms with the same level of specificity of the real algorithms may be unfeasible, unfortunately, since manufaturers need to conceal key characteristics of their products (including prefetcher algorithms) in order to stay competitive. Another point that is worth noting is that Skylake implements a non-inclusive L3 cache, where the data present in the L2 cache may or may not be in the L3 as well. A non-inclusive L3 cache adds more complexity, since, for instance, the location of the data will depend on the particular access pattern of the executing application, the size of code and data accessed, and the inter-core sharing behavior. Similarly to other architecture simulators, Sniper implements an inclusive L3 cache. This difference can also contribute to the discrepancies between the real execution using Skylake and the Sniper simulation.

In Figure 8 we present the total number of prefetch requests performed by the Sniper's L1+L2 prefetcher, the Sniper's L2 prefetcher, and their real counterparts. The number of prefetch requests for the L1+L2 prefetcher counts the prefetches on both L1 and L2 caches. As expected at this point, Sniper's L2 prefetcher model was not capable to accurately simulate the Skylake's L2 prefetcher on the number of prefetch requests as well, presenting large discrepancies (notice the log scale on the y axis of Figure 8). On the other hand, Sniper's L1+L2 prefetcher presented a total number of prefetch requests closer to the real execution. However, the similar number of prefetch requests of the Sniper's L1+L2 prefetchers does not translate in similar estimations of the applications performance, as shown in Figure 7. This may be due to the fact that the prefetches performed by Sniper are

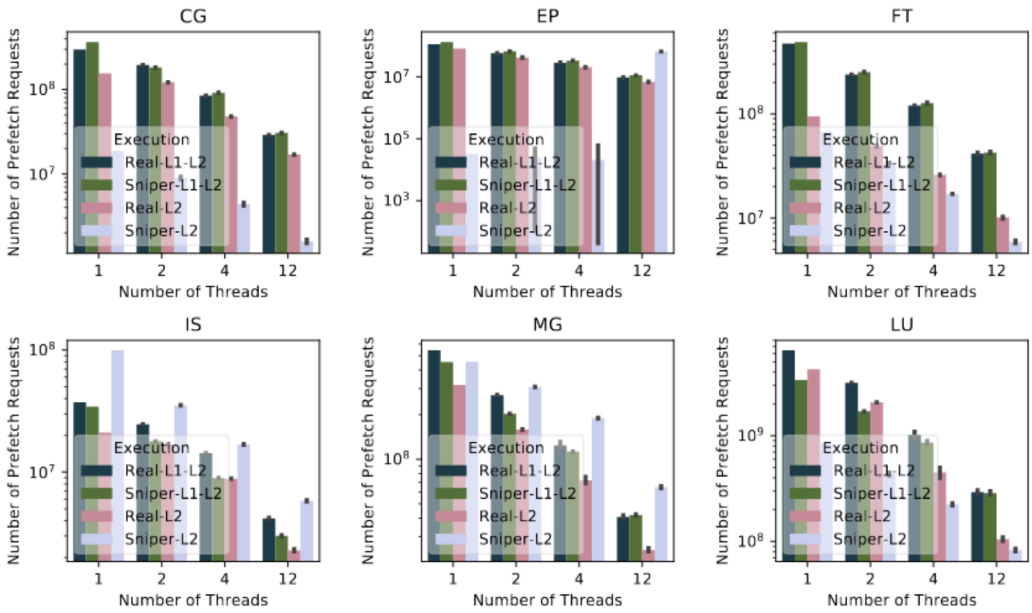

FIGURE 8 Total number of prefetches issued by the simulation and by the real hardware (log scale on y axis).

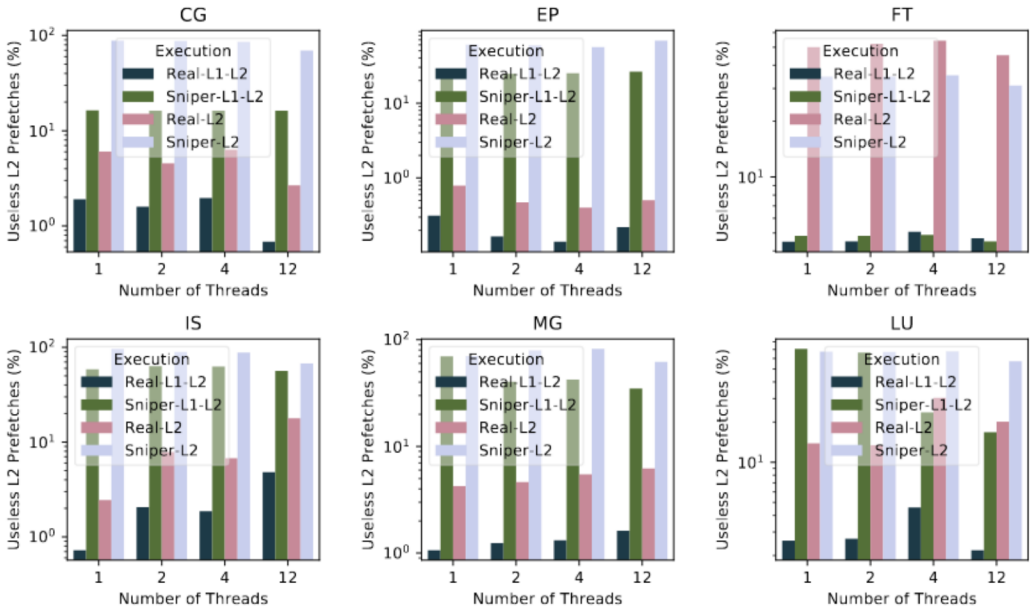

FIGURE 9 Useless prefeteches performed by the simulation and by the real hardware, in ratio of the total number of prefetches.

different in terms of usefullness. To attest this argument, in Figure 9 we show the mean percentage of prefetches that were not useful during the executions in the real machine and in the Sniper simulation. We can notice that the only application that sniper managed to accurately simulate the usefullness of the prefetches is the FT application. When simulating the other applications with Sniper, Sniper considered the majority of prefetch requests as useless, with useless prefetches close to 100% in some cases. This may indicate that Sniper's memory simulation module is having issues on simulating how the prefetched data is interacting with cache hierarchy. The non-inclusive L3 cache may also be a cause of this issue, since again Sniper considers an inclusive L3 cache.

6 | CONCLUSION AND FUTURE WORK

Throughout this work, several insights were obtained regarding methodological aspects, the behavior of the studied prefetcher models, and how researchers and end users should handle prefetchers. An important aspect highlighted throughout the entire work is the difficulty of working with the memory subsystem, with the prefetcher and its algorithms being a special case. The obscurity and confidentiality around the real implementation makes accurate models and algortihms impossible to be reproduced in simulators; moreover, for the same reasons above, simulator users have difficulties even in finding the proper parameters for the prefetcher models.

Regarding the observations we made in the experiments, we found several interesting characteristics in the studied prefetchers that may direct future works. We have discussed how the L1 data prefetcher snooping the L1 cache requests can create contention by occupying the limited line fill buffers in the L1 cache. Moreover, the obscurity of the L1 prefetcher implementation makes architecture studies that consider the L1 prefetcher harder to interpret. Analyzing applications' performance, therefore, becomes harder due to the lack of control over the prefetchers and their implementations.

We argue that the use of both prefetchers (L1+L2) does not necessarily warrant significant performance gains, which is not intuitive. When considering the L2 prefetcher, we obtained performance gains similar to when using both prefetchers, with the advantages of having more control over the experiments and the noise in the memory subsystem, faster simulation time, and thus less energy consumption due to the smaller number of memory requests being performed. This is comprehensible, as the current cache memory technologies allow a small difference in latency between the L1 and L2 caches. However, the access latency to the off-chip L3 cache memory is considerably higher (on average 7x, due to the mesh topology necessary to fully utilize the cache, as it is distributed in L2 off-core banks). Therefore, we advise the use of the L2 prefetcher as a standalone, instead of both

prefetchers (which is the default setting). The L2 prefetcher observes the more relevant access patterns, as it prefetches data that would always require an access to the L3, thus hiding the large L3 latency.

With the increase in application parallelism, the execution time naturally becomes smaller. However, we see that the performance per core decreases as we increase the level of parallelism, showing the impact of memory accesses and communication over the application's performance. As the amount of communication increases, the contention for the shared resources (interconnection network and LLC banks) increases as well. Thereby, the request buffers of the caches and main memory become full, and the prefetcher cannot sufficiently mitigate the memory latency, resulting in small IPC values. Thus, analyzing the communication pattern of applications is an important task to improve their performance, as applications which exert heavy memory pressure or communicate often can diminish the usefulness of a prefetcher.

High performance computing users should be aware of this. Highly parallel applications may benefit from turning off the prefetcher, as suggested by Dell[43], which claims 8% increased performance for the SAP NetWeaver Portal application when disabling the hardware prefetchers. In recent years, Intel implemented forms of reducing the prefetcher aggressiveness due to multiple reports such as Dell's, in order to avoid performance degradation. Our real execution results show that even with high parallelism, Intel's strategy works, as the prefetcher never loses performance compared to an execution without prefetchers. However, it would be interesting to test this assumption with simultaneous multithreading.

Nevertheless, our research shows that computer architecture researchers should always implement a prefetcher aggressiveness atenuator in the simulator model, and test their new implementations on highly parallel applications which exert memory pressure and intercore interference[27], so that the possible negative impacts of their design can be properly evaluated.

ACKNOWLEDGEMENTS

This work has been partially supported by Petrobras (2016/00133-9, 2018/00263-5) and Green Cloud project (2016/2551-0000 488-9), from FAPERGS and CNPq Brazil, program PRONEX 12/2014.

References

1. Wulf W, McKee S. Hitting the Memory Wall: Implications of the Obvious. *Computer Architecture News* 1996; 23.

2. Skadron K, Martonosi M, August DI, Hill MD, Lilja DJ, Pai VS. Challenges in computer architecture evaluation. *Computer* 2003; 36(8): 30–36.

3. Jin H, Jin H, Frumkin M, Frumkin M, Yan J, Yan J. The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. tech. rep., NAS System Division; NASA Ames Research Center: 1999.

4. Girelli VS, Moreira FB, Serpa MS, Navaux POA. Impacto do Prefetcher na Precisão de Simulações de Arquiteturas Paralelas. In: Camargo RY, Martins WS., eds. *2019 Symposium on High Performance Computing Systems (WSCAD)*. SBC. ; 2019: 382–393.

5. Sanchez D, Kozyrakis C. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. *SIGARCH Comput. Archit. News* 2013; 41(3): 475–486. doi: 10.1145/2508148.2485963

6. Carlson TE, Heirman W, Eyerman S, Hur I, Eeckhout L. An Evaluation of High-Level Mechanistic Core Models. *ACM Transactions on Architecture and Code Optimization (TACO)* 2014. doi: 10.1145/2629677

7. Bakhshalipour M, Shakerinava M, Lotfi-Kamran P, Sarbazi-Azad H. Bingo spatial data prefetcher. In: Louri A, Venkataramani G, Gratz P., eds. *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. ; 2019: 399–411.

8. Chen TF, Baer JL. Effective hardware-based data prefetching for high-performance processors. *IEEE transactions on computers* 1995; 44(5): 609–623.

9. Le HQ, Starke WJ, Fields JS, et al. Ibm power6 microarchitecture. *IBM Journal of Research and Development* 2007; 51(6): 639–662.

10. Sanchez D. ZSim Tutorial Validation. http://zsim.csail.mit.edu/tutorial/slides/validation.pdf; 2016. [Accesed in: 10 Sept. 2019].

11. Bienia C, Kumar S, Singh JP, Li K. The PARSEC benchmark suite: Characterization and architectural implications. In: Moshovos A, Tarditi D, Olukotun K., eds. *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM. ; 2008: 72–81.

12. Genbrugge D, Eyerman S, Eeckhout L. Interval simulation: Raising the level of abstraction in architectural simulation. In: Das C, cJacob M, Yang Q., eds. *Proceedings - International Symposium on High-Performance Computer Architecture*. IEEE. ; 2010: 1 - 12

13. Eyerman S, Eeckhout L, Karkhanis T, Smith JE. A Mechanistic Performance Model for Superscalar Out-of-Order Processors. *ACM Trans. Comput. Syst.* 2009; 27(2). doi: 10.1145/1534909.1534910

14. Woo SC, Ohara M, Torrie E, Singh JP, Gupta A. The SPLASH-2 Programs: Characterization and Methodological Considerations. *SIGARCH Comput. Archit. News* 1995; 23(2): 24–36. doi: 10.1145/225830.223990

15. Adileh A, González-Alvarez C, Ruiz JMDH, Eeckhout L. Racing to Hardware-Validated Simulation. In: Wenisch T, Agarwal N., eds. *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. ; 2019: 58–67.

16. Eeckhout L. Computer architecture performance evaluation methods. *Synthesis Lectures on Computer Architecture* 2010; 5(1): 1–145.

17. Fog A. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. *Copenhagen University College of Engineering* 2012: 02–29.

18. Desikan R, Burger D, Keckler SW. Measuring experimental error in microprocessor simulation. In: Stenstrom P., ed. *Proceedings of the 28th annual international symposium on Computer architecture (ISCA)*. ACM. ; 2001: 266–277.

19. Austin T, Larson E, Ernst D. SimpleScalar: An infrastructure for computer system modeling. *Computer* 2002(2): 59–67.

20. Walker M, Bischoff S, Diestelhorst S, Merrett G, Al-Hashimi B. Hardware-validated CPU performance and energy modelling. In: Nikolopoulos D, Supinski B, Delimitrou C., eds. *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. ; 2018: 44–53.

21. Gutierrez A, Pusdesris J, Dreslinski RG, et al. Sources of error in full-system simulation. In: Aamodt T, Hempstead M., eds. *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. ; 2014: 13–22.

22. Mittal S. A survey of recent prefetching techniques for processor caches. *ACM Computing Surveys (CSUR)* 2016; 49(2): 1–35.

23. Srinath S, Mutlu O, Kim H, Patt YN. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In: Louri A, Yousif M, Byrd G., eds. *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE. ; 2007: 63–74.

24. Nesbit KJ, Smith JE. Data cache prefetching using a global history buffer. In: Tirado F, Zapata E., eds. *10th International Symposium on High Performance Computer Architecture (HPCA'04)*. IEEE. ; 2004: 96–96.

25. Zhuang X, Hsien-Hsin SL. Reducing cache pollution via dynamic data prefetch filtering. *IEEE Transactions on Computers* 2006; 56(1): 18–31.

26. Lee CJ, Mutlu O, Narasiman V, Patt YN. Prefetch-aware DRAM controllers. In: Gonzalez A, Silvano C, Monchiero M, Galuzzi C., eds. *Proceedings of the 41st Annual IEEE/ACM international Symposium on Microarchitecture*. IEEE Computer Society. ; 2008: 200–209.

27. Ebrahimi E, Mutlu O, Lee CJ, Patt YN. Coordinated control of multiple prefetchers in multi-core systems. In: Albonesi D, Martonosi M, August D, Martinez J., eds. *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE. ; 2009: 316–326.

28. Jain A, Lin C. Rethinking belady's algorithm to accommodate prefetching. In: Bilof R., ed. *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. ; 2018: 110–123.

29. Wu H, Nathella K, Sunwoo D, Jain A, Lin C. Efficient metadata management for irregular data prefetching. In: Manne S, Hunter H, Altman E., eds. *Proceedings of the 46th International Symposium on Computer Architecture*. ACM. ; 2019: 449–461.

30. Bhatia E, Chacon G, Pugsley S, Teran E, Gratz PV, Jiménez DA. Perceptron-based prefetch filtering. In: Manne S, Hunter H, Altman E., eds. *Proceedings of the 46th International Symposium on Computer Architecture*. ACM. ; 2019: 1–13.

31. Akram A, Sawalha L. A Survey of Computer Architecture Simulation Techniques and Tools. *IEEE Access* 2019.

32. Binkert N, Beckmann B, Black G, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 2011; 39(2): 1–7.

33. Ubal R, Jang B, Mistry P, Schaa D, Kaeli D. Multi2Sim: a simulation framework for CPU-GPU computing. In: Yew PC, Cho S, DeRose L, Lilja D., eds. *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE. ; 2012: 335–344.

34. Patel A, Afram F, Ghose K. Marss-x86: A qemu-based micro-architectural and systems simulator for x86 multicore processors. In: Mueller W, Paderborn U, Petrot F., eds. *1st International Qemu Users' Forum*. IEEE. ; 2011: 29–30.

35. Yourst MT. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In: Albonesi D, Brooks D., eds. *2007 IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE. ; 2007: 23–34.

36. Hammarlund P, Martinez AJ, Bajwa AA, et al. Haswell: The fourth-generation intel core processor. *IEEE Micro* 2014; 34(2): 6–20.

37. Melo dAC. The New Linux 'perf' Tools. In: Kongress L., ed. *International Linux System Technology Conference*. Linux Foundation. ; 2010: 1–42.

38. Doweck J, Kao WF, Lu AKy, et al. Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake. *IEEE Micro* 2017; 37(2): 52–62. doi: 10.1109/MM.2017.38

39. Carlson TE, Heirmant W, Eeckhout L. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In: Lathrop S, Costa J, Kramer W., eds. *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. ; 2011: 1–12.

40. Intel . Intel® 64 and IA-32 Architectures Optimization Reference Manual. https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf; 2019. [Accesed in: 16 Jan. 2020].

41. Charney MJ, Reeves AP. Generalized correlation based hardware prefetching. https://www.semanticscholar.org/paper/Generalized-correlation-based-hardware-prefetching-Charney-Reeves/86b3c8787a63f4d1d30ff53ac115cbd8881a7af7; 1995. [Accesed in: 18 Apr. 2020].

42. Alves MAZ, Villavieja C, Diener M, Moreira FB, Navaux POA. Sinuca: A validated micro-architecture simulator. In: Qiu M., ed. *2015 IEEE 17th International Conference on High Performance Computing and Communications*. IEEE. ; 2015: 605–610.

43. Dell . Dell SAP NetWeaver Benchmark Info. https://www.dell.com/downloads/global/solutions/Dell_SAP_NetWeaver_Benchmark_Info.pdf; 2006. [Accesed in: 24 Mar. 2020].

How to cite this article: V. S. Girelli, F. B. Moreira, M. S. Serpa, D. Carastan-Santos, and P. O. A. Navaux (2020), Understanding Memory Prefetcher Performance over Parallel Applications: From Real to Simulated, *Concurrency and Computation: Practice and Experience*, *TBD*.