Data prefetching on in-order processors

Cristobal Ortega[1,2], Victor Garcia[1,2], Miquel Moretó[1,2], Marc Casas[1,2], Roxana Rusitoru[3]
[1]Universitat Politècnica de Catalunya (UPC), [2]Barcelona Supercomputing Center (BSC-CNS), [3]ARM Ltd.
{cortega, vgarcia, mmoreto, mcasas}@{upc.edu, bsc.es}, {roxana.rusitoru}@{arm.com}

***Abstract*—Low-power processors have attracted attention due to their energy-efficiency. A large market, such as the mobile one, relies on these processors for this very reason. Even High Performance Computing (HPC) systems are starting to consider low-power processors as a way to achieve exascale performance within 20MW, however, they must meet the right performance/Watt balance. Current low-power processors contain in-order cores, which cannot re-order instructions to avoid data dependency-induced stalls. Whilst this is useful to reduce the chip's total power consumption, it brings several challenges. Due to the evolving performance gap between memory and processor, memory is a significant bottleneck. In-order cores cannot re-order instructions and are memory latency bound, something data prefetching can help alleviate by ensuring data is readily available.**

In this work, we do an exhaustive analysis of available data prefetching techniques in state-of-the-art in-order cores. We analyze 5 static prefetchers and 2 dynamic aggressiveness and destination mechanisms applied to 3 data prefetchers on a set of HPC mini- and proxy-applications, whilst running on in-order processors. We show that next-line prefetching can achieve nearly top performance with a reasonable bandwidth consumption when throttled, whilst neighbor prefetchers have been found to be best, overall.

***Keywords*—Data prefetching, in-order processor, High Perfomance Computing**

I. Introduction

Multi-core architectures are the main trend in current processor development. Processor vendors can improve performance without increasing clock rates or instruction level paralellism (ILP).

This trend has several issues. One is the added pressure on shared hardware resources in the chip multiprocessor (CMP). As such, adding more cores in a processor puts more pressure on the memory system, which is one of the main bottlenecks nowadays [1].

The High Performance Computing (HPC) field seeks to increase performance of current system to reach exascale, whilst maintaining a 20MW power budget [2]. This level of performance/Watt needs solutions with unprecedented levels of energy efficiency [3, 4]. Current commodity hardware such as embedded and mobile processors is designed to be energy efficient due to constraints such as battery life and overheating. Usually, these low-power processors contain in-order cores, which are significantly smaller (area, power) and lower performance compared to the typical desktop or server-class out-of-order processors. Nonetheless, low-power processors are promising due to their density for HPC, and they have previously been investigated for such purposes [5–8].

To alleviate the memory wall problem different solutions are available. Current out-of-order processors can reorder instructions to avoid being idle waiting for data. Therefore, the order of execution, within consistency constraints, is based on the availability of data instead of the original order of execution. Another option vendors use to reduce memory latency is to include a hardware data prefetcher. This device brings data to the processor's cache before it is needed, thus reducing stalls.

When we refer to low-power processors, more specifically those based on in-order cores, instruction reordering is not an option to increase performance, however, data prefetching is.

In this work, we perform an exhaustive analysis on how data prefetching affects in-order cores, whilst running a suite of HPC mini- and proxy-applications. We also implement different dynamic mechanisms for data prefetching to improve efficiency for in-order micro-architectures. We show that even relatively simple prefetching mechanisms, such as next-line, can achieve top or near top performance, whilst maintaining low bandwidth requirements.

Our main contributions are:

- We evaluate 5 hardware prefetchers on an in-order core in single and multi-core systems, with 2 state-of-the-art dynamic mechanisms, across 9 HPC mini- and proxy-applications.
- We show that dynamically reconfiguring the data prefetcher on in-order cores can speedup executions up to 1.4x in in-order cores.
- In the context of in-order cores, we show (1) that in *simple*[1] data prefetchers, cache pollution affects performance and they waste memory bandwidth by bringing more data than needed; (2) that in *complex* data prefetchers, memory bandwidth is used more efficiently, cache pollution is lower and prefetcher accuracy is higher. Yet, *complex* data prefetchers need more area on the chip, which can be a large percentatge of an in-order core and (3) that dynamic throttling mechanism can bring the benefits of complex prefetcher to simpler ones.

This paper is organized as follows: Section 2 provides the required background on data prefetchers for this work, Section 3 covers our dynamic mechanisms. Section 4 describes the experimental setup, while Section 5 and 6 shows the results of our experiments for single and multi core, respectively. Section 7 discusses the related work. We present the paper's conclusions in Section 8.

[1]We refer to data prefetchers with a straightforward algorithm with little or no decision making as simple data prefetcher.

II. Background on Data Prefetching

Hardware data prefetchers can reduce memory latency by bringing data to the processor's cache before it is needed. This reduces the stall time when the processor needs to perform a memory operation.

In the case of out-of-order cores, memory latency can be alleviated by data prefetching or by reordering instructions. On the other hand, in-order cores cannot re-order instructions, which makes data prefetching even more important.

Data prefetchers do not always improve performance [9]. Memory access patterns of the applications have a major impact on the data prefetcher performance. Predictable memory access patterns and spatial locality benefit from it, whilst random or unpredictable accesses can cause the prefetchers to thrash the caches, thus potentially leading to application performance degradation. Typically, complex data prefetchers offer more performance at the cost of more area and increased power consumption. Simpler data prefetchers occupy less area but they usually perform worse than a complex data prefetcher, due to being unable to recognize complex memory access patterns.

Ideally, a hardware data prefetcher brings the exact amount of data needed by the processor, in a timely manner and without evicting useful data. In practice, different application behaviors require hardware data prefetchers to tune their aggressiveness during application execution.

Several prefetchers can be tuned in such a way, although traditionally this needed to be done manually, due to the lack of automatic mechanisms. An incorrectly tuned prefetcher, such as a too aggressive one could waste enegy and lead to performance degradation, due to cache pollution. A non-aggressive prefethcer would degrade performance by prefetching old data.

Some configurable prefetcher parameters are:

- **Degree**: Number of cache lines transferred in every prefetch request. Increasing this parameter can help improve performance by bringing more cache lines. This also leads to an increased prefetcher aggressiveness, which can lead to the eviction of useful data.
- **Distance**: Number of cache lines ahead of the current memory address being accessed that will be prefetch. Increasing this parameter will prefetch cache lines further than the offended cache line, which can improve performance by prefetching ahead. Therefore, reducing latency and possible late prefetches. It can also happen that the prefetcher evicts useful cache data.

When measuring a prefetcher's performance, we need to take into account whether the prefetcher is bringing in useful data and whether it does so in a timely fashion without evicting useful data. Therefore, the performance of prefetchers is measured using the following metrics:

- **Accuracy**: if the issued prefetches are useful or not
- **Timeliness**: if the prefetched block does not arrive before the cache line is referenced.
- **Pollution**: if prefetched blocks evict useful blocks from the cache

A_{high}	A_{low}	$T_{lateness}$	$T_{pollution}$	$T_{congestion}$
0.75	0.40	0.05	0.001	0.005

TABLE I: Thresholds used in this work for the dynamic aggressiveness mechanism.

Hardware prefetchers are attached to a single cache. Therefore, it is possible to have different types of prefetchers in different cache levels and have different configurations. The prefetchers which we evaluate in this work are:

- **Nextline Prefetcher** A simple prefetcher that detects sequential access patterns and prefetches the next consecutive cache line [10].
- **Stream Prefetcher** In this prefetching scheme, the hardware prefetches consecutive cache blocks after a short training period during which it observes memory access streams. A stream is a group of data references in a short period of time that frequently repeat and are stable [11].
- **Stride Prefetcher** In this scheme, the hardware prefetcher calculates the distance between memory addresses (or stride) from the same instructions.
  When the prefetcher is trained, upon a cache miss of an instruction that is recognized, the missing cache line and the cache line with the same requested address plus the stride are returned [12].
- **Neighbor Prefetcher** In this scheme, the hardware prefetcher brings to the cache the surrounding cache lines of the demanded cache line [13]. The surrounding cache lines to be prefetched are called the neighborhood, which is composed by different cache lines near to the missing cache block (how near or far are the cache lines depends on the defined size of the neighborhood). The neighborhood needs a training phase to work properly.
- **Correlation** (Global History Buffer prefetcher) In this scheme, the hardware keeps an ordered list of memory addresses generated by the same memory instruction. This information is used in a training phase to observe possible correlations. Those correlations are used to prefetch cache blocks [14].

III. Dynamic mechanisms

We implement 2 dynamic mechanisms to modify dynamically the behavior of the different possible prefetchers.

Our dynamic mechanisms are checked and reconfigured at the end of an interval. An interval is defined when half of the blocks of the cache are evicted, which give us useful metrics based on cache activity. We use their global history of the metrics to take into account the global behavior of the application in order to reduce the noise of small application phases. We use an interval based on cache activity instead of fixed-time, as the cache-related data is more relevant to the prefetcher.

A. Dynamic prefetcher aggressiveness

We dynamically tune the prefetcher aggressiveness at execution time following the proposed algorithm in [15]. This dynamic mechanism can only be applied to *queue-based* prefetchers in gem5 due to design implementations in non-*queue-based* prefetchers, as explained in Section II.

At the end of each interval, metrics for that interval are collected. Based on the interval metrics and the global history metrics, prefetcher sets a configuration for its degree and destination. Original thresholds from the work by Srinath et al. [15] are changed to adapt them to our in-order system, which we show in Table I. We needed to increase the lateness threshold (T_{lateness}) and reduce the pollution one ($T_{\text{pollution}}$) in order to not lose performance since original thresholds were causing a reduction in prefetcher aggressiveness.

In total, there are 5 possible configurations for the prefetcher aggressiveness: (1) very conservative (distance: 4, degree: 1); (2) conservative (distance: 8, degree: 1); (3) middle-of-the-road (distance: 16, degree: 2); (4) aggressive (distance: 32, degree: 4); (5) very aggressive (distance: 64, degree: 4).

B. Dynamic destination

We implement a mechanism to decide dynamically where a cache stores the prefetched cache lines. This is done for 2 reasons: (1) to be able to prefetch even if the cache is blocked due to having too many demand accesses and (2) to alleviate the memory bandwidth requirements for the caches.

In this mechanism, we select a cache to be a master cache, which it will decide at each interval where to store the prefetched cache lines. A master cache uses the full memory access stream to train the prefetcher. Then, the master cache can decide based on several metrics to store the prefetched cache line into another cache level if the cache is polluted or to issue a prefetch from another prefetcher if the cache is congested.

This strategy allows the master cache to keep prefetching even if the master cache cannot prefetch more (cache is congested) or to send prefetched cache lines to other caches of the system if the master cache is congested or polluted. Therefore, increasing prefetch accuracy.

In our experiments, we use the L1 cache as master. Therefore, all L1 caches can send prefetches to their respective private L2 or to the shared L3 cache. We set the master cache to the L1 cache for several reasons: (1) L1 cache is the best performing one in terms of latency of all the caches due to its proximity to the core; (2) usually, it is the cache most limited by the memory bandwidth.

At the end of each interval, the master cache checks the pollution and congestion levels of the cache where current prefetches are being stored, starting with itself.

In the case the cache is polluted or congested, new prefetches will be send to the next level cache for the next interval (i.e. pollution level or congestion level are greater than $T_{\text{pollution}}$ or $T_{\text{congestion}}$, respectively). If that next cache level is polluted or congested, prefetches are sent to the last level cache. This happens irrespective whether the L3 cache is polluted or congested.

At the end of every interval, the master cache checks its own pollution and congestion levels in order to try to prefetch always to itself, which performs better since we set the master cache to the L1 cache, closest cache to the processor.

Configurable parameter	**DL1**	**L2**	**L3**
Size	32kB	256kB	2MB
Hit latency (cycles)	1	4	20
MSHRs	8	20	30
Associativity	4	16	16

TABLE II: Configurable parameters for the caches

C. Metrics

Our dynamic mechanisms are based on the following gathered metrics:

- Prefetch Accuracy: This measures if the prefetcher is bringing in useful data before the cache block is accessed. We define data as useful when it is accessed during its lifetime. We consider its lifetime as all the time that exists in a cache. It is defined as: $\frac{\textit{Number of useful prefetches}}{\textit{Number of issued prefetches}}$
- Prefetch Lateness: This measures whether the prefetch requests arrived in a timely fashion, in time to satisfy the demand access. It is defined as: $\frac{\textit{Number of late prefetches}}{\textit{Number of useful prefetches}}$
- Cache Pollution: This measures the useless data brought in by the prefetcher in the cache. It is defined as: $\frac{\textit{Number of misses caused by the prefetcher}}{\textit{Number of misses}}$
  In order to track useless data, we track cache lines that are brought by the prefetcher. If there is a miss on a cache line that was brought by the prefetcher, we consider it as useless data brought by the prefetcher.
- Cache Congestion: This measures the time that the cache's prefetcher cannot issue more prefetches due to the unavailability of free Miss Status Handling Registers (MSHRs). It is defined as: $\frac{\textit{Number of times cache is blocked due to a MSHR miss}}{\textit{Number of MSHR misses}}$

IV. Experimental setup

A. Simulation infrastructure

We use a customized iced version of gem5 [17] to simulate a 64-bit ARMv8 system. We report results whilst running in full-system mode with a 4-core configurations. These cores are ARM Cortex-A53-like. Each core has private L1 and L2 exclusive caches, and a shared, inclusive L3. The replacement policy for cache lines is least recently used (LRU). Table II summarizes the cache parameters. The system is configured such that prefetches occur on cache misses. The memory used is a DDR4 running at 2.4GHz, with 1 channel and a 16-byte bus.

On the simulated system, we run a Linux kernel version 3.16 with a configured base page size of 64kB. We use MPICH 3.2 [18] to run MPI benchmarks. Hardware-wise, the L1 data prefetcher is allowed to cross page boundaries if the accessed page is already translated in the translation lookaside buffer (TLB). Therefore, the accessed page is in memory.

B. Prefetchers hardware configuration

We evaluate 5 different prefetchers, including running experiments without any hardware prefetching. Every cache

	Description	Input	Heap usage (MB)
CoMD	Co-designed Molecular Dynamics: a classical molecular dynamics proxy application	20 -N 20 -T 4000	70
DGEMM	Double precision real matrix-matrix multiplication	5000	10
FFT	One-dimensional Discrete Fourier Transform	5000	35
PTRANS	Parallel matrix transpose	5000	65
STREAM	Sustainable memory bandwidth	5000	55
HPCG	High Performance Conjugate Gradient:preconditioned Conjugate Gradient method	24	55
mcb	Monte Carlo Benchmark: a simple heuristictransport equation using a Monte Carlo technique	320000	12
miniFE	Implicit Finite Elements: a proxy application for unstructured implicit finite element codes	860	95
pathfinder	Signature-search mini-application	medlarge1	15

TABLE III: Input and heap usage for the benchmarks used in the evaluation. Heap usage is measured on a physical 64-bit ARMv8 machine, using Valgrind [16]. Total memory footprint used exceeds the total cache size. gem5 reports for all benchmarks a high heap usage (>95%)

level can have a hardware prefetcher, and we perform a full design-space exploration, running every benchmark with all the possible combinations of prefetchers. The prefetchers we evaluate are described in Section II: Neighbor, Nextline, Correlation, Stride and Stream.

In gem5, prefetchers can be queue-based, which means they inherit from the Queue class[1] in gem5. If this is the case, they can be generically tuned with parameters such as distance or degree. If they are not queue-based prefetchers, the available parameter set relies on each prefetcher's implementation, which may not have distance or degree exposed. As such, we only apply our dynamic aggressiveness mechanisms on the queue-based prefetchers: Nextline, Stream and Stride.

C. Benchmarks

In order to evaluate the effectiveness of the different prefetchers implemented, we use benchmarks from different suites: Mantevo [19], HPC Challenge [20], Proxy applications [21], Trinity benchmarks [22] and the High Performance Conjugate Gradients (HPCG) benchmark [23]. These benchmarks are high performance computing-oriented benchmarks and parallelized using OpenMP or MPI.

Brief descriptions, including a list of parameters can be found in Table III.

We simulate the Region of Interest (ROI) of each benchmark until either the ROI finishes or a maximum number of instructions is reached. This maximum number of instructions is selected such that all cores do useful work. We select the ROI manually via source-code instrumentation, as the SimPoint methodology [24] cannot be applied to multi-threaded applications. Later, a gem5 checkpoint is created at the start of the ROI.

We warm-up the simulation for all benchmarks using 50M instructions. The standard detailed simulation interval is 500M (except in the FFT benchmark, which runs for 1B instructions). The number of instructions simulated is chosen taking into account that every thread must be doing useful work. We define useful work as progress on the execution, which ensures execution is not stalled in an idle loop waiting for data. We measure this via number of memory, scalar or floating operations executed.

[1]The Queue class in gem5 is a class for existing prefetchers in gem5 (Stride and Nextline prefetchers), which process every memory request ordered by age.

Benchmark	L2 Prefetcher	L3 Prefetcher
CoMD	Neighbor	Neighbor
DGEMM	Stride	Stride
FFT	Neighbor	Stride
PTRANS	Stride	Stride
STREAM	Neighbor	Stride
HPCG	Neighbor	Stride
mcb	Neighbor	Stride
miniFE	Neighbor	Neighbor
pathfinder	Stride	Stride

TABLE IV: L2 and L3 prefetchers used in the single core experiments. These configurations are the most performing ones in static experiments.

V. Results Single-Core

In this section, we cover results obtained using a single-core system, and all benchmarks were run using a single thread. All others components are the same as detailed in Section IV-A.

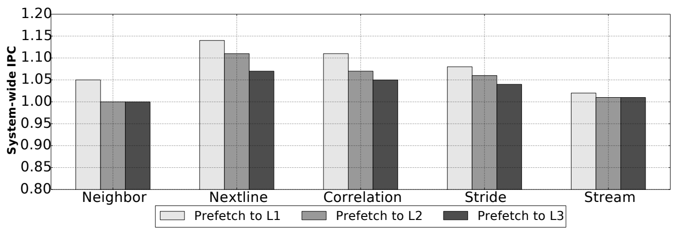

Fig. 1: System-wide IPC in static experiments with different prefetchers for the L1 cache against no prefetching. The L1 prefetcher can prefetch cache lines into the L1 (normal behavior), L2 or L3 caches in order to explore possible benefits using the dynamic destination explained in Section III-B

We start by evaluating the possible performance gains of dynamic destination mechanisms. Figure 1 shows the performance in terms of IPC of different L1 data cache prefetchers using dynamic destination, with respect to no prefetching at any cache level. Table IV lists which prefetchers were used for the L2 and L3 caches. Please note that these vary depending on the application, as we chose the best L2 and L3 prefetchers in each case.

We observe that as prefetcher inserts the prefetched lines into upper levels in the cache hierarchy, we reduce application performance, as upper cache levels have higher latency. The most complex the prefetcher, the most sensitive to prefetching into the upper cache levels. This can be observed for prefetchers such as Correlation and Neighbor. In the case of Neighbor, this technique is typically detrimental, which is caused by the way the prefetchers on all the cache levels interact in this experiment. We cover this in further detail later. In the case of simple prefetchers, this technique leads to a higher performance improvement since these bring in more data than the complex ones. Therefore, being able to increase a prefetcher's own effective cache capacity by using the L2 and L3 caches helps increase performance by overall reducing the load latency.

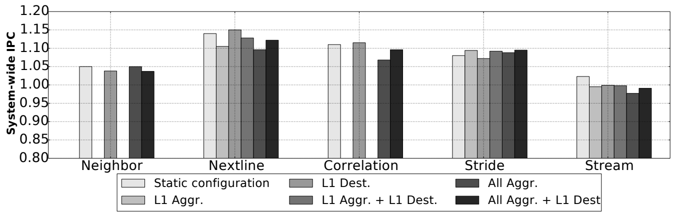

Fig. 2: System-wide IPC with different L1 prefetchers and different configurations with the dynamic mechanisms explained in Sections III-A and III-B. L2 and L3 prefetchers are fixed across the different configurations to the prefetchers specified in Table IV. Due to implementantation limitations, Neighbor and Correlation prefetchers cannot dynamically adapt their aggressiveness.

Figure 2 shows the performance of various L1 data cache prefetchers, using the configurations listed in Table IV, relative to no prefetching. The L2 and L3 configurations are as outlined in Table V. Results show that, typically, all prefetchers benefits from the dynamic mechanisms, however, at different rates.

The Neighbor prefetchers shows a small gain when the dynamic destination is enabled. As we explain in detail later, the Neighbor prefetcher has a high accuracy, which leads it to place useful prefetchers further away from the L1 cache, thus affecting the overall latency by keeping in the cache hierarchy useful cache lines.

The Nextline prefetcher gains the most out of the dynamic destination mechanism, as the prefetcher normally issues a large number of prefetches which, if all the prefetched cached lines placed in the same cache, would lead to increased levels of pollution. As such, by using this mechanism we can increase the prefetch data's utilization.

The Correlation prefetcher has performance gains of 10% across the different static and dynamic configurations against not using an L1 data cache prefetcher. The lowest gain is when all prefetchers have the dynamic aggressiveness enabled. This is caused by pollution added by the L2 and L3 cache prefetchers. The dynamic destination mechanism renders the highest performance.

The Stride prefetchers slightly benefits from all dynamic mechanisms. The dynamic destination, for example, reduces cache pollution, whilst the aggressiveness can help save memory bandwidth by not over-prefetching.

The Stream prefetcher's performance is lower than the other prefetchers. As we cover in further detail later, this is caused by the prefetcher not using sufficient memory bandwidth. The Stream perfetcher's performance lowers whenever any dynamic mechanism is used.

Configuration	L1 Prefetcher	L2 Prefetcher	L3 Prefetcher
1 - No prefetcher config.	-	-	-
2 - Static config.	-	-	-
3 - L1 Aggr.	DA	DA	DA
4 - L1 Dest.	DD	DD	DD
5 - All Aggr.	DA	DA	DA
6 - All Aggr + L1 Dest	DA & DD	DA & DD	DA & DD

TABLE V: Prefetcher configurations used in this work. DA is Dynamic Aggressiveness enabled. DD is Dynamic Destination enabled.

VI. Results Multi-Core

In this section, we show our experimental results whilst running a multi-core system.

We report Instructions per Cycle (IPC) of the overall system ($\frac{\textit{Total number of instructions}}{\textit{Total number of cycles}}$), memory bandwidth, cache misses, cache pollution and a classification in terms of used, late and unused prefetches for every prefetch issued.

We experiment with several prefetcher configurations, as shown in Table V.

Configuration 1 is a processor with no prefetcher in any cache level. Configuration 2 is a standard configuration for current in-order processors. In configuration 3, we enable the dynamic aggressiveness feature only in the L1 data cache prefetcher. In configuration 4, we instead enable the dynamic destination feature, whilst in 5 the dynamic aggressiveness is enabled for all prefetchers in the system. Finally, configuration 6 uses the dynamic aggressivenes features for all prefetcher levels and the dynamic destination for the L1 data cache prefetcher.

Results from the rest of the section have a specific prefetcher configuration for the second and last level cache. We set a Nextline prefetcher for the second level cache and a Stride prefetcher for the last level cache. This configuration is one of the most performing ones seen in our experiments, which offers a view of possible performance gains for the dynamic mechanisms.

A. Performance

Figure 3 shows the speed-up in terms of execution time and memory bandwidth of different prefetcher configurations with respect to no prefetching.

We observe that the main cause of performance increase is using a prefetcher on the L1 cache, which can speed-up execution time by 28% and up to 65% with the Stream prefetcher offering the least performance increase, whilst the Neighbor prefetcher increasing performance the most. As explained previously, the Stream prefetcher memory bandwidth usage is low compared to the other prefetchers.

The Neighbor prefetcher maintains constant performance with respect to the static configuration when enabling the dynamic destination and aggressiveness in all the cache levels. Yet, performance degrades a 2.0% when enabling both

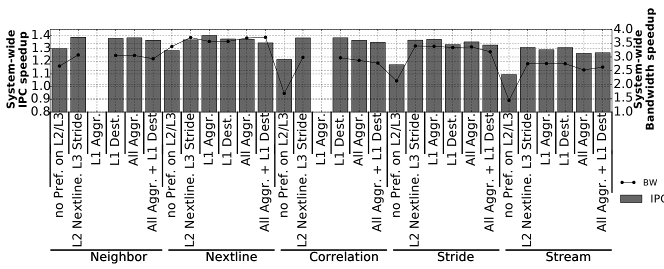

Fig. 3: System-wide IPC and system-wide bandwidth for several configurations with the dynamic mechanisms with respect to no prefetching on L1, L2, nor L3 cache. Due to implementation limitations, Neighbor and Correlation prefetchers cannot dynamically adapt their aggressiveness. Their degree and distance parameters are not reconfigurable.

dynamic features at the same time. At the moment there is pollution on the L1 cache, the dynamic destination kicks in and sends cache lines to the L2 or L3 cache, which can influence on their aggressiveness due to increased pollution. Potentially, lowering their aggressiveness hurts performance.

The Nextline prefetcher does not lose performance when enabling any dynamic mechanism. The dynamic aggressiveness mechanism improves performance by 3.2%. This is because the Nextline prefetcher reduces cache misses by bringing as many cache lines as possible before these cache lines are needed. Therefore, the dynamic aggressiveness mechanism chooses a performing configuration for the aggressiveness of the prefetcher while the dynamic destination mechanism reduces pollution in the L1 cache.

The Correlation prefetcher obtains a good speed-up by adding a L2 and L3 prefetchers. When the dynamic aggressiveness mechanism is enabled, performance degrades for 2% and 4%. This is caused by a higher miss cache rate in the L2 and L3 caches (3% higher miss cache rate when only the dynamic aggressiveness is enabled and 7% when the dynamic destination is enabled), whilst a lower aggressive is set due to the added pollution from the L1 prefetcher with the dynamic destination enabled.

The Stride prefetcher maintains performance across the different prefetcher configurations. The only exception is when both dynamic mechanisms are enabled, which leads to performance degradation due to increased pollution. Pollution is increased by 5% in the L3 cache with respect to only enabling the dynamic aggressiveness mechanism. Stride prefetcher shows a lower performance than Nextline prefetcher due to its lower memory bandwidth usage.

The Stream prefetcher suffers from a performance degradation when the dynamic aggressiveness mechanism is enabled, going from 1.30x speed-up to a 1.26x speed-up due to the mechanism choosing a less aggressive configuration. The Stream prefetcher shows a lower performance than Stride prefetcher due to lower memory bandwidth usage.

In the Figure 3, we observe that there is a trade-off between memory bandwidth usage and performance. Enabling our dynamic mechanisms, the same performance can be achieved while decreasing memory bandwidth, cache misses, and cache pollution.

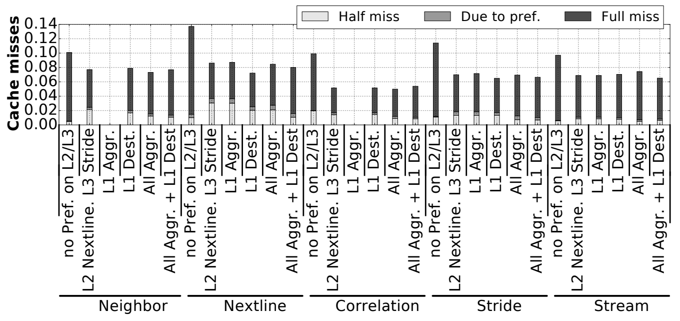

Fig. 4: Overall cache misses in the system. Cache misses in L1, L2 and L3 cache are taken into account. We report half misses: miss in cache, but it hits on the MSHR; due to prefetcher: cache misses that are caused by the prefetcher itself; full misses: miss in cache and the data must be bring from another location.

1) Cache misses: Figure 4 shows system-wide cache misses across all the cache in order to have a perspective of how data prefetchers affect the entire system.

Cache misses are classified in 3 categories: (1) Half miss, a prefetch was issued, but the prefetch has not arrived to the cache yet; (2) Due to prefetcher, cache misses caused by a prefetcher overwriting cache blocks for prefetched data; and (3) Full miss, the data is in memory and must be brought.

Cache misses are very similar across the different prefetcher configurations. We can see that misses due to the prefetcher are mainly seen in simple prefetchers. Also, it is interesting to see how the misses due to the prefetcher are mainly seen in the simple prefetchers. Overall, Nextline is the prefetcher that causes most system-wide misses compared to others (even compared to simple prefetchers).

In terms of the dynamic mechanisms, increasing the aggressiveness of the L1 prefetcher for the Nextline, Stride and Stream prefetchers does not increase performance since they do not reduce cache misses. Therefore, their aggressiveness level is not highly increased due to other constraints such as pollution or congestion. As we can see with the Neighbor and the Correlation prefetchers, they obtain similar performance compared to the simpler prefetchers, whilst lowering bandwidth usage (see Figure 3).

When enabling the dynamic destination, L1 cache misses are increased, yet, we can see that the overall cache misses decrease. This is not highly reflected in terms of performance due to a higher latency access to the L2 and L3 caches but it should impact on overall power consumption.

2) Issued Prefetches: We measure how useful are the issued prefetches in Figure 5. We classify every prefetch issued by the L1 prefetcher into: (1) unused, when a prefetched block is not addresses by the application; (2) late, the system performs a demand access to an address, that address misses in the cache and hits on a prefetch register (MSHR); and (3) used, when a prefetched block is addressed by the application.

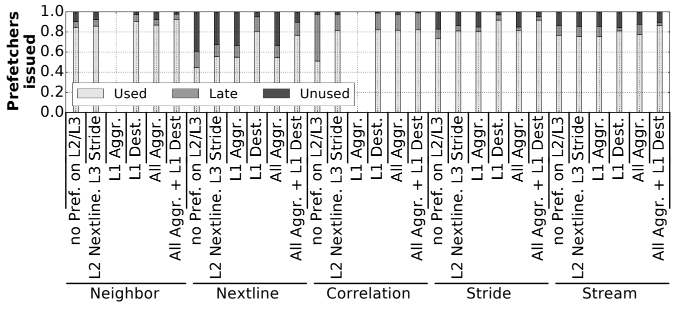

Fig. 5: Classification of the issued prefetchers for different prefetcher configurations. They are classified and unused: the cache block prefetched was not used by the processor; late: the cache line was accessed before the cache line arrived to the cache and used: the prefetcher brought a cache line that was used in time.

In Figure 5, the *Unused* component is mostly present in the simple prefetchers due to their simplistic nature.

The Nextline prefetcher without dynamic destination can waste up to 40% of the issued prefetches. When we enable dynamic destination, the Nextline *Unused* prefetches decrease by up to 5% due to cache lines can be prefetched into an upper cache level to be reused in the future. The Neighbor and Correlation prefetchers lead to a better utilization of the prefetched cache lines since they bring in fewer cache lines and in a more efficient way. They waste up to 10% of the issued prefetches.

In terms of lateness, the prefetcher issuing late requests is Correlation. This behavior is highlighted when there are no prefetchers in the L2 nor the L3 cache. This is casued by the training phase of the Correlation prefetcher, adding a L2 or L3 cache prefetcher can help to reduce the overall latency, and therefore reducing the training phase, which helps to increase timely prefetches.

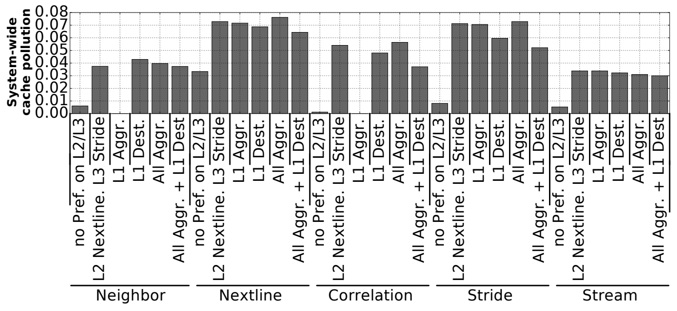

Fig. 6: Cache pollution of the system for different prefetcher configurations. It is measured as the ratio between cache misses caused by having a prefetcher and the total misses of the system.

3) Cache Pollution: In Figure 6, we report the overall cache pollution of the system for the different prefetcher configurations. The figure shows the ratio between misses caused by having a prefetcher and the total misses of the system.

Pollution is higher with the Nextline and the Stride prefetchers, both of them are simple prefetchers. The Neighbor and Correlation prefetchers offer the best rate performance to pollution rate.

When evaluating the dynamic mechanisms, we see that dynamic aggressiveness does not affect pollution negatively, unless all the prefetchers in the cache hierarchy have the dynamic aggressiveness enabled (such the case of the Nextline and Stride prefetchers). Dynamic destination can help to reduce cache pollution since cache line can be stored in other cache levels.

VII. Related work

Data prefetching is a known approach to solve the problem for the evolving gap between processor and memory. Therefore, it is a field that has attracted much attention from researchers.

Previous works propose prefetcher implementations to improve performance on, usually, out-of-order multi-core chips.

Several of these works implement hardware modifications in order to improve performance [25–27]. In this work, we focus on hardware prefetchers that can be found in current processors.

Previous works evaluated modifying the aggressiveness of the hardware data prefetcher at runtime. Srinath et al. improve performance by adjusting the prefetching based on several metrics [15]. Ebrahimi et al. provide mechanisms in order to improve performance and fairness of shared resources with data prefetching in a multi-core processor scenario [28–30]. Jimenez et al. modify the aggressiveness of the prefetcher based on the overall demands of the applications running on the system [31]. Nesbit et al. divide the memory address into equal-sized zones and detect patterns within each zone. Then, they adapt the prefetcher aggressiveness and the size of the zones [32]. Both works are developed in out-of-order cores, they evaluate serial or multi-programmed workloads and stick to one prefetcher analysis.

Previous work with parallel applications evaluated the dynamic reconfiguration of the prefetcher, Li et al. apply machine learning to reconfigure the data prefetcher, this needs of an offline training phase [33]. Prat et al. use a task-based runtime to manage the aggressiveness of the data prefetcher when running parallel applications [34]. Ortega et al. automatically coordinates at execution time the aggressiveness of the data prefetcher and the Simultaneous Multithreading level at execution time [35]. On these works, they work with only one hardware prefetcher and they improve it by setting different behaviors for different workloads. In this work, we explore several hardware prefetchers with dynamic mechanisms based on several cache and prefetch metrics.

VIII. Conclusions

Data prefetching in in-order cores has a major impact on the overall performance since it is a known technique to alleviate the evolving performance gap between processor and memory. There are several data prefetchers available, but research has been focused in out-of-order processors.

In this work we perform an exhaustive analysis of different data prefetchers in terms of performance, bandwidth and cache requirements. We implement 2 state-of-the-art dynamic mechanisms and evaluate them in our in-order core infrastructure.

Results show that there is a trade-off between complexity and memory bandwidth requirements. Simple data prefetchers have a higher memory bandwidth usage, which can be unaffordable for low-power processors. On the other hand, complex data prefetchers can be expensive in terms of area, which can be unaffordable for embedded processors.

Dynamically increasing the aggressiveness of the data prefetcher can increase performance at the cost of a higher memory bandwidth usage. While other mechanisms such as the dynamic destination can increase the efficiency of the prefetchers and the caches. Therefore, simple data prefetchers with dynamic mechanisms can match the performance of complex data prefetchers while using less area, which can meet the requirements for embedded and low-power processors.

Acknowledgment

We would like to thank Rekai González-Alberquilla for his help in setting up the experimental environment.

This work has been supported by the RoMoL ERC Advanced Grant (GA 321253), by the European HiPEAC Network of Excellence, by the Spanish Ministry of Science and Innovation (contracts TIN2015-65316-P), by Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272), by the European Union's Horizon 2020 research and innovation programme (grant agreements №671697 and №779877). M. Moreto has been partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Ramon y Cajal fellowship number RYC-2016-21104

References

[1] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *SIGARCH Comput. Archit. News*.

[2] S. Ashby, P. Beckman, J. Chen, P. Colella, B. Collins, D. Crawford, J. Dongarra, D. Kothe, R. Lusk, P. Messina *et al.*, "The opportunities and challenges of exascale computing," *Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee*.

[3] N. Adiga, G. Almasi, G. Almasi, Y. Aridor, R. Barik *et al.*, "An Overview of the BlueGene/L Supercomputer," ser. SC '02.

[4] S. Alam, R. Barrett, M. Bast, M. R. Fahey, J. Kuehn, C. McCurdy, J. Rogers, P. Roth, R. Sankaran, J. S. Vetter, P. Worley, and W. Yu, "Early evaluation of IBM BlueGene/P," ser. SC '08.

[5] N. Rajovic, A. Rico, N. Puzovic, C. Adeniyi-Jones, and A. Ramirez, "Tibidabo1: Making the case for an ARM-based HPC system," *Future Generation Computer Systems*.

[6] N. Rajovic *et al.*, "The Mont-Blanc Prototype: An Alternative Approach for HPC Systems," ser. SC'16.

[7] H. Nakashima, H. Nakamura, M. Sato, T. Boku, S. Matsuoka, D. Takahashi, and Y. Hotta, "MegaProto: 1 TFlops/10kW Rack Is Feasible Even with Only Commodity Technology," ser. SC'05.

[8] K. Fürlinger, C. Klausecker, and D. Kranzlmüller, "Towards energy efficient parallel computing on consumer electronic devices," in *International Conference on Information and Communication on Technology*, 2011.

[9] L. Jaekyu, K. Hyesoon, and V. Richard, "When prefetching works, when it doesn't, and why," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2012.

[10] A. J. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, 1978.

[11] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on.* IEEE.

[12] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride Directed Prefetching in Scalar Processors," ser. MICRO 25.

[13] D. M. Koppelman, "Neighborhood prefetching on multiprocessors using instruction history," ser. PACT 2000.

[14] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," *Software, IEE Proceedings-*, 2004.

[15] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers," ser. HPCA '07.

[16] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," *ACM Sigplan notices*, 2007.

[17] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*.

[18] "MPICH 3.2. https://www.mpich.org/."

[19] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," 2009.

[20] P. Luszczek, J. J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. Mccalpin, D. Bailey, and D. Takahashi, "Introduction to the HPC Challenge Benchmark Suite," 2005.

[21] R. Neely, M. Heroux, and S. Swaminarayan, "National security applications co-design: A frame-work for an asc tri-lab project," 2013.

[22] "Trinity benchmarks. http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/."

[23] J. Dongarra, M. A. Heroux, and P. Luszczek, "High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems."

[24] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," ser. ASPLOS X.

[25] X. Zhuang and S. L. Hsien-Hsin, "Reducing cache pollution via dynamic data prefetch filtering," 2007.

[26] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely, Jr., and J. Emer, "PACMan: Prefetch-aware Cache Management for High Performance Caching," ser. MICRO'11.

[27] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weems, "Guided Region Prefetching: A Cooperative Hardware/Software Approach," ser. ISCA '03.

[28] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Prefetch-aware shared-resource management for multi-core systems," ser. ISCA'11.

[29] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated control of multiple prefetchers in multi-core systems," ser. MICRO 42.

[30] E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems," ser. HPCA'09.

[31] V. Jimenez, A. Buyuktosunoglu, P. Bose, F. P. O'Connell, F. Cazorla, and M. Valero, "Increasing multicore system efficiency through intelligent bandwidth shifting," ser. HPCA'15.

[32] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith, "AC/DC: an adaptive data cache prefetcher," ser. PACT'04.

[33] M. Li, G. Chen, Q. Wang, Y. Lin, P. Hofstee, P. Stenstrom, and D. Zhou, "PATer: A Hardware Prefetching Automatic Tuner on IBM POWER8 Processor," ser. CAL'16.

[34] D. Prat, C. Ortega, M. Casas, M. Moretó, and M. Valero, "Adaptive and application dependent runtime guided hardware prefetcher reconfiguration on the IBM POWER7," ser. ADAPT'15.

[35] C. Ortega, M. Moreto, M. Casas, R. Bertran, A. Buyuktosunoglu, A. E. Eichenberger, and P. Bose, "libPRISM: An Intelligent Adaptation of Prefetch and SMT Levels," ser. ICS '17.