Improving Prefetching Mechanisms for Tiled CMP Platforms

Martí Torrents

Computer Architecture Department

Universitat Politècnica de Catalunya - BarcelonaTech

This dissertation is submitted for the degree of

Doctor of Philosophy

September 2016

THESIS INFORMATION

Title of the thesis: Improving Prefetching Mechanisms for Tiled CMP Platforms

Candidate: Martí Torrents
Degree: Doctor of Philosophy

Advisors:
Raúl Martínez - Oracle Labs
Carlos Molina - Universitat Rovira Virgili

Tutor:
Antonio González - Universitat Politècnica de Catalunya

Department: Computer Architecture Department (DAC)

COMMITTEE MEMBERS

President: José Luis Sánchez - Universidad de Castilla-La Mancha

Secretary: Ramon Canal Corretger - Universitat Politècnica de Catalunya

Vocal: Pedro Marcuello Pascual - Broadcom Networks

Substitute: Francisco José Alfaro Cortes - Escuela Universitaria Politécnica de Albacete

Substitute: Josep Llosa Espuny - Universitat Politècnica de Catalunya

---

Submission date: Friday, 29th September 2016

I would like to dedicate this thesis to my loving parents, Florenci and Magda, and my brothers, Cesc and Esteve. I know that, although it has not been easy for them, because they did no understand what I was exactly doing, they gave me all their support when I was down in though times. Also to all my friends, Albert, Balma, Gerard, Laura, Marc Morales, Marc Corominas, Marta, Queralt and Xavi. You found fun in my sad moments and you have helped me week by week along these four years. All these little weekly oasis during our dinners, our "talkings" or simply the moments we spent together had been essential for having this work done today. At last but not least to you, Eli, because we have shared this experience day by day. Thanks for wiping away my tears when I was down, thanks for shining in the darkness and thanks for walking this long way next to me. I am sure that without all this lovely people it would have been impossible for me to achieve this milestone. Thanks to all of you.

M'agradaría dedicar aquesta tesi als meus estimats pares, Florenci i Magda, i als meus germans, Cesc i Esteve. Sé que, encara que no ha estat fàcil per ells perquè sovint no entenien allò que feia, m'han donat suport sempre i sobretot en els moments difícils. També a tots els meus amics, Albert, Balma, Gerard, Laura, Marc Morales, Marc Corominas, Marta, Queralt i Xavi. Vau arrancar-me somriures en els moments mes delicats, i setmana a setmana m'heu ajudat molt durant aquests 4 anys. Tots aquests petits oasis setmanals durant els sopars, els "talkings" o senzillament els moments que passàvem junts, han estat essencials per tenir aquest projecte acabat avui. Finalment, però no per això menys important, a tu, Eli, perquè amb tu he compartit aquesta experiència dia a dia. Gràcies per eixugar-me les llàgrimes quan estava més desanimat, gràcies per brillar a la foscor i per fer aquest llarg camí al meu costat. Estic segur que sense vosaltres, per mi, hagués estat impossible aconseguir aquesta fita. Gràcies a tothom.

Acknowledgements

My first contact on research was in 2008 when Prof. Ramon Canal suggested that I helped him in his investigations. He introduced me in the research and encouraged me to start a PhD. This was the beginning from everything. From that moment, I started to meet researchers who have helped a lot to me to finish this dissertation. For this reason, first of all, I would like to thank Ramon for introducing me to this world and showing me what has become my passion.

The second person I met when I was starting my research experience was Prof. Antonio Gonzalez and for him goes my second acknowledgement. Antonio proposed doing my Master Thesis in the Intel Barcelona Research Center. I was excited about it, thus, I accepted the proposal and there I met my Master advisors. One of them, Dr. Raúl Martínez, became my PhD advisor. When I finished my internship in Intel, I started the PhD. My tutor, Antonio Gonzalez, introduced to me my second advisor, Prof. Carlos Molina. My two last acknowledgments are for them, Raul and Carlos, who had advised me through this process. Carlos' positivity and Raul's good sense have been the key to overcome this PhD challenges. Without any doubt, the achievement of this milestone would not be possible without them.

To all of you, thanks for everything.

Abstract

Recently, high performance processor designs have evolved toward Chip-Multiprocessor (CMP) architectures to deal with instruction level parallelism limitations and, more important, to manage the power consumption that is becoming unaffordable due to the increased transistor count and clock frequency. At the present moment, this architecture, which implements multiple processing cores on a single die, is commercially available with up to twenty four processors on a single chip and there are roadmaps and research trends that suggest that number of cores will increase in the near future. The increasing on number of cores has converted the interconnection network in a key issue that will have significant impact on performance. Moreover, as the number of cores increases, tiled architectures are foreseen to provide a scalable solution to handle design complexity.

Network-on-Chip (NoC) emerges as a solution to deal with growing on-chip wire delays. On the other hand, CMP designs are likely to be equipped with latency hiding techniques like prefetching in order to reduce the negative impact on performance that, otherwise, high cache miss rates would lead to. Unfortunately, the extra number of network messages that prefetching entails can drastically increase power consumption and the latency in the NoC. In this thesis, we do not develop a new prefetching technique for CMPs but propose improvements applicable to any of them. Specifically, we analyze the behavior of the prefetching in the CMPs and its impact to the interconnect. We propose several dynamic management techniques to improve the performance of the prefetching mechanism in the system. Furthermore, we identify the main problems when implementing prefetching in distributed memory systems like tiled architectures and propose directions to solve them. Finally, we propose several research lines to continue the work done in this thesis.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Everything begins with an idea.

Earl Nightengale

1.1 Motivation

In the last years, processor designs have evolved toward architectures that implement multiple processing cores on a single die, commonly known as chip-multiprocessors or CMPs. Today, multi-core architectures are envisioned as the only way to ensure performance improvements after microprocessors designers have acknowledged that it is no longer efficient to rely on higher clock rates and/or exploiting greater levels of instruction-level parallelism (ILP). In this way, most of the industry would agree that multi-core is the way to go further and those designs with tens of cores on the die will be a reality in the near future. Additionally, future many-core CMPs with several tens (or even hundreds) of processor cores will be probably designed as arrays of replicated tiles connected over an on-chip switched direct network [89]. These tiled architectures have been claimed to provide a scalable solution for managing the design complexity, and effectively using the resources available in advanced VLSI technologies. Maybe, the best known examples of a tiled CMP architecture today from the main HPC developers are the Intel Xeon Phi [32], the Nvidia Tesla [50], the Tilera TILE64 [5], or the AMD FireStream [2] prototypes.

However, one of the greatest roadblocks to provide high performance in such tiled CMP architectures is the high cost of on-chip communication between cores [30]. This on-chip communication subsystem is what we know as the Network on Chip (NoC) [69]. A sign that suggests that this topic is envisioned as one of the current challenges by the research community is that in the last 20 years it has increased the number of publications. Moreover,

in the last 5 years it growing has stalled but it is still an active topic in the main conferences. (see Figure 1.1).

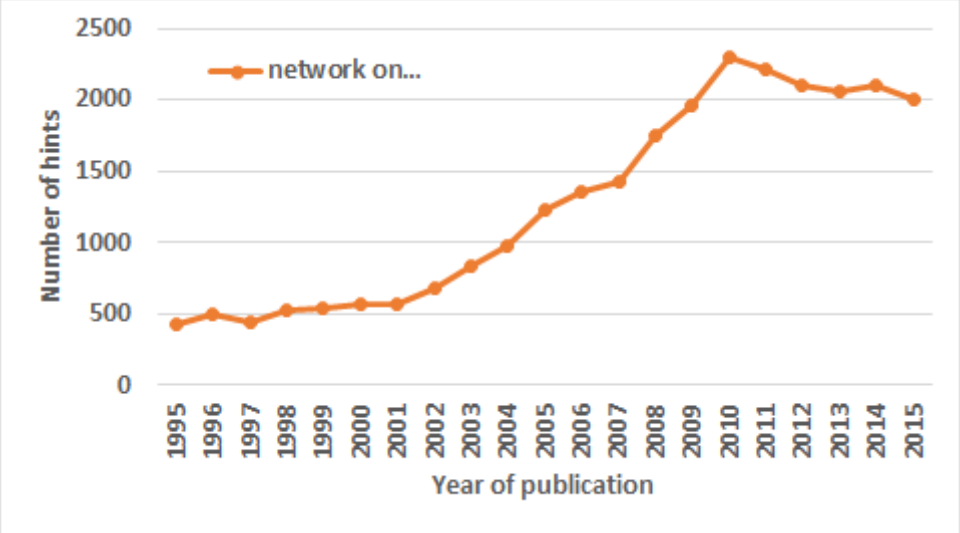


Fig. 1.1 Search hits for “network-on-chip” in IEEE Xplore archive.

One of the reasons why the NoC are becoming a bottleneck as the number of cores increases is the bad scalability of global communications as process technology advances. Figure 1.2 shows what it was outlined before: the smaller the technology node, the lower the gate delay and the local wire delay (delay from the wires in the same core). However, the global wire delay (delay from the communication between cores) increases exponentially. Although some improvements, such as adding repeaters, reduce the latency, they do not change the exponential trend. This issue is similar to the one related to the different scalability of memory access time and core processing speed [8], which is a problem in both, single and multicore processors.

One way to handle the increasing memory latency due to wire delay and memory access time is to use latency hiding techniques like prefetching, which eliminates some cache misses and/or overlaps the latencies of others. These mechanisms try to predict which data the processor is going to require and bring it to the nearest cache level before it is demanded by the application. Furthermore, prefetching is a technique which is implemented in most of the commercial processors [48] [76] [72] [29]. Each of these processors implements several techniques, however, the details of most of them are secret. An example is the case of the Intel Nehalem processor. In [48] they explain the type of prefetcher implemented in both, first and second cache level, but they do not provide implementation details of any of them.

Apart from providing a good performance in terms of execution time (speedup), an ideal prefetching algorithm should satisfy, among others, other important property: to produce a

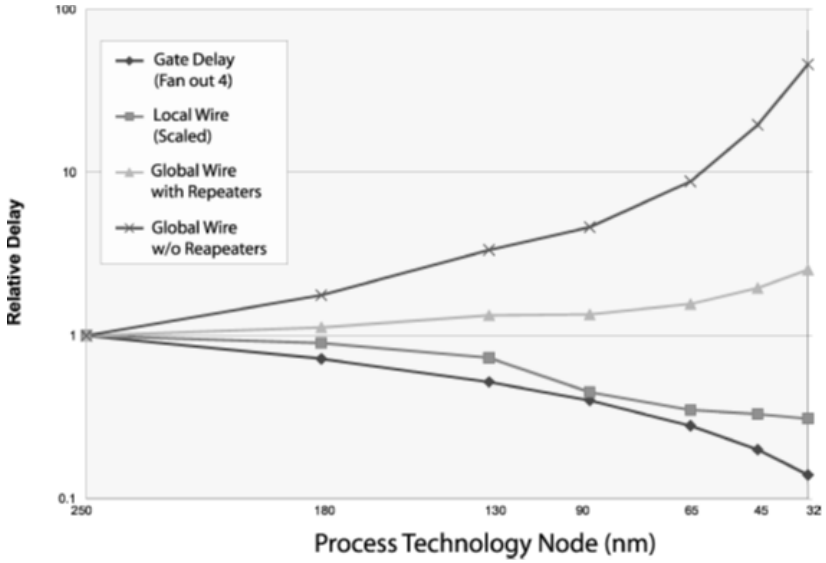


Fig. 1.2 Projected relative delay for local and global wires and for gates in technologies of the near future [8].

minimum increase of traffic in the memory interconnection. An increase in memory traffic entails higher power consumption and a higher degree of contention in the interconnection network. Note that prefetching generates a potentially high amount of memory requests in order to bring to the cores the data that they would require in advance. As an overall effect, the number of memory requests are in most of the cases going to be higher than in a system without prefetching. This is especially true if we are in a multi-core system because it increases on-chip communication since coherence between the L1 caches of the tiled CMP must be ensured. An increase in the congestion of the network will most probably increase the latency of not only prefetching requests but also regular memory operations [66]. Moreover, although, to the best of our knowledge, there is no specific literature in this regard, it is expected to be a straight relation between the overhead of prefetch operations in the network and memory system and the increment of dissipated power.

If the prefetcher would be perfect and all the prefetch requests would brought only the data blocks that are going to be used by the core, the core would not need to fetch them. This would entail that the prefetch traffic would overlap the traffic from the core demand operations and the same would happen regarding the power consumption. However, depending on the accuracy of the prefetcher for a given application, a certain portion of the data prefetched is not going to be actually used by the cores. Apart from the aforementioned increase in traffic and congestion, the unnecessarily prefetched block lines can displace beforehand other block lines that are going to be required after a short period of time and that have to be fetched again. This may negatively affect the performance of the application, diminishing

the improvement provided by prefetching in the worst case even degrading the performance compared with a system without prefetching. Nevertheless, it is very difficult for a prefetcher to predict which data block will be needed by its core. In order to increase the number of prefetching operations issue in time (and increase the speedup of the system), the prefetchers usually generates several prefetch requests each time it is triggered (to have a wider range of success), and this decreases the accuracy of the technique. Note that hardware prefetchers in commodity processors can be very aggressive, sometimes prefetching more than twice the data required for an application [38], [37]. Figure 1.3 shows some performance numbers obtained in [21] that also supports this argument. For three prefetching alternatives (one block lookahead [24], on miss prefetch [18], and tagged prefetcher [24]), the average improvement in terms of execution time (Figure 1.3.a) and the increase in the on-chip network power consumption due to the extra communication that prefetching entails for the parallel scientific applications considered in the study (Figure 1.3.b) is shown. As it can be observed, increases over 20% in the on-chip network power consumption are obtained for some of the prefetching techniques.

A lot of research has been done on proposing new prefetch techniques and incrementing their performance in single-core processors [64] [79]. However, research in the multicore environment has received much more less attention. Therefore, in this thesis, we will put a special emphasis on CMP environments, specifically tiled architectures, and on a key aspect of this kind of systems, the NoC. Moreover, we do not pretend to propose new prefetching engines for multicore but we will propose techniques valid for any prefetching mechanism that will improve the overall performance of the prefetching in CMP systems as a whole.

1.2 Objectives

Derived from the motivation previously exposed, we define the main general objective of this thesis as developing a better understanding of the implications of prefetching in tiled CMP systems and providing novel proposals to improve the performance in term of power and performance efficiency of prefetching mechanisms in this kind of systems. We have split this general objective in four secondary objectives that have guided all the work in this thesis:

- To characterize the NoC behavior of prefetching in tiled CMP platforms.
- To enhance the profiling of prefetching statistics used in dynamic management techniques to better handle prefetching traffic.
- To propose novel management techniques to improve the performance of prefetching techniques.

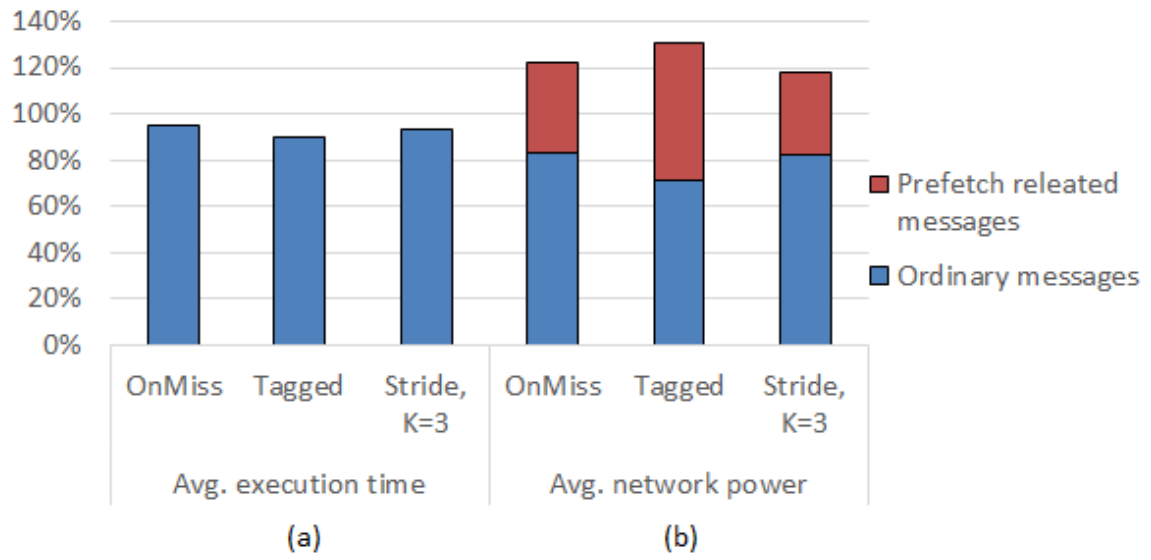


Fig. 1.3 (a) Normalized execution time and (b) network power consumption for a 16-core CMP with different prefetching mechanisms.

- To identify the challenges for prefetching in shared distributed memory systems and propose directions to address them.

In the following, we elaborate on what we pretend to achieve with each secondary objective, the specific motivation that leads each of them, and a first overview of the work that we have done to fulfill them.

Characterize the NoC behavior of prefetching in tiled CMP platforms

The aim of this objective is to focus on the behavior of the prefetching traffic in one of the main shared resources, as it is the NoC. Specifically, we want to emphasize the importance of taking into account the underlying NoC on the performance achieved by the prefetching mechanisms. To do this, we will analyze the traffic that is traversing the interconnect, identifying and characterizing the prefetching requests. With this analysis we plan to quantify the impact of the prefetching requests in the congestion of the interconnect.

Furthermore, implementing a simulation infrastructure able to simulate the prefetching traffic in the NoC will not be an easy issue. The reason for that is that we need a simulator which is able to simulate the CPU and the NoC in order to simulate the prefetching traffic in the NoC. The CPU generates the requests, that the prefetcher analyzes and injects them into the NoC. Therefore, a NoC simulator will not be able to generate prefetch requests, and a CPU simulator will not be able to simulate the NoC. There are several open-source

simulators that are able to work in these two areas. However, they do not have a prefetching infrastructure implemented. For this reason, another important point of this objective will be to adapt a current open-source simulator to integrate the prefetching in its simulation infrastructure.

Enhance the prefetching statistics profiling used in dynamic management techniques to better handle prefetching traffic

A better management of the prefetch requests could help in reducing or minimizing the effect from congestion on the interconnect and in general in the memory system. Ideally, to successfully manage prefetching requests in the system dynamically, we would have to know the impact of each of these prefetch requests in the system as soon and accurately as possible. Thus, predicting the impact of a prefetch request in the system is key. One of the most important things for making accurate predictions is the way that these requests are profiled.

For this reason, the idea of this objective is to enhance the strategy to collect the dynamic profiling information to help techniques that manage the prefetch generation. Therefore, we plan to develop a new mechanism to dynamically collect the profiling information from the prefetch requests and be able to generate accurate predictions of the usefulness of each potential request that may help decide the impact of the requests in the whole system.

Propose novel management techniques to improve the performance of prefetching techniques

The idea of this objective is to use the profiling techniques proposed in the previous objective to improve the dynamic management of prefetching requests, developing new techniques that reduce the congestion of the interconnect without losing performance, or losing the minimum performance.

Specifically, we plan to work in a technique that gives a different treatment to the requests according to its predicted impact in the system. Actually, in case the request is predicted with a negative impact, we plan to discard it. Otherwise, if the request is predicted to have a good impact in the system, it will have a higher priority treatment when accessing the shared resources.

Identify the challenges for prefetching in shared distributed memory systems and propose directions to address them

From the best of our knowledge, in all the prefetching studies for CMPs, the prefetcher works in the private and/or monolithic caches, but not in the shared and distributed ones, which are typical in the upper cache levels of tiled architectures. The reason is probably that implementing prefetching in shared distributed memories face challenges that, as far as we know, have not been properly addressed or even discussed in the literature.

For this reason, the aim of this objective is to identify the challenges of prefetching in distributed and shared memory systems and quantify its effect in the performance of the prefetchers. Moreover, we plan to propose some directions in how this challenges could be addressed.

1.3 Thesis organization

The remaining of this thesis is organized as follows: in Chapter 2, we introduce the more important concepts related to prefetching, networks on chip, techniques from the state of the art that manage the prefetcher generated requests or the traffic in the network on chip, and some information on multiprocessor simulators. After this related work, we continue with the contributions of this thesis that are detailed in the next four chapters. Chapter 3 shows our first contribution, where we explain our experimental framework, the modifications that we have introduced to the simulator and experimental results that shows that the traffic injected by the prefetcher to the network on chip is not negligible. There are several techniques in the state of the art that employ dynamic profiling information to offer a better management of prefetching. In Chapter 4, we analyze and evaluate several approaches to get information on the accuracy of prefetching requests and propose a novel confidence predictor for dynamic management techniques. In Chapter 5, we use this new predictor as the base for enhancing two techniques from the state of the art and propose novel management mechanisms. In Chapter 6, we focus on the challenges of prefetching on shared and distributed caches, typical from tiled architectures. We identify these challenges and propose some directions on how to solve them. Finally, in Chapter 7, we conclude the thesis by evaluating the objectives presented in this chapter and giving some ideas for future work.

Chapter 2

State of the Art

> Learn the rules like a pro so you can break them like an artist.
>
> *Pablo Picasso*

2.1 Introduction

In this chapter, we introduce the most important concepts related to prefetching on networks on chip that will be referenced along this thesis document. We will introduce the prefetching strategies as well as the related work done in techniques that dynamically improves its performance and the current state of the art on simulators that can simulate prefetch mechanisms.

This chapter is organized as follows: Section 2.2 introduces the main concepts about prefetching and the related work on this topic. Section 2.3 is about the networks on chip. What they are and what are the main contributions in this topic. Section 2.4 shows the main dynamic management techniques for prefetching requests that are presented in the literature. Finally, Section 2.5 introduces the main open-source simulators that currently are available in the literature to work in CMP.

2.2 Prefetching basic concepts

The memory hierarchy is one of the most important elements in the performance of nowadays processors. In order to reduce the latency of memory accesses, there are a lot of different ways to improve the memory hierarchy. One of the ways is make use of prefetching mechanisms.

These mechanisms try to predict which data the processor is going to need, and take it to the nearest cache level before it is demanded by the application.

In Figure 2.1, we can see the behavior of different prefetching methods. In this figure, we can see a time chart with 3 different cases. In each case, we can see the regular computation along the time, in this computation there are some memory accesses, that some of them can become a memory miss. Specifically, in Figure 2.1.a we can see the behavior of an execution where there is no prefetching, and whenever there is a memory access the processor has to idle until the access is finished. We can see the idyllic behavior of a prefetcher in Figure 2.1.b, where the prefetcher detects that some data is going to be required and generates a memory access that is processed in parallel with the processor computation and, in the moment that the processor needs that data, it is already in the cache module. However, nowadays prefetchers are working as in Figure 2.1.c because in very few cases is possible to exactly predict what a program is going to do enough cycles before to ask for the data in the correct moment.

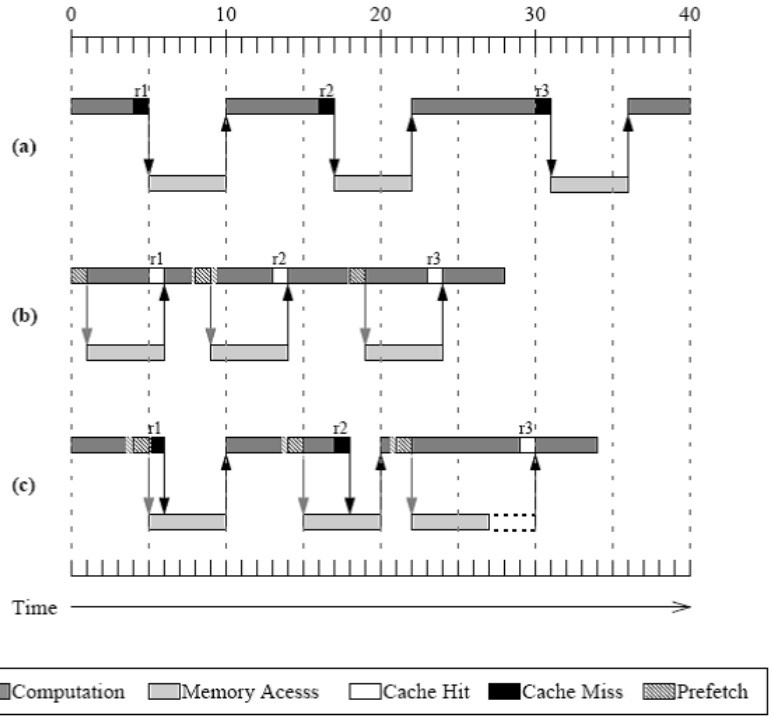


Fig. 2.1 Execution diagram assuming a) no prefetching, b) perfect prefetching, and c) degraded prefetching.

There are two main concerns that a good prefetcher must take into account. What to prefetch and when to prefetch in order to get the data timely.

If we do not have these concepts in mind, the prefetcher is going to have an unwanted behavior, so there are some issues we have to consider in order to avoid them. For instance, if the prefetch unit is taking wrong data in the cache module, could replace useful data, or if the data that the prefetch unit is looking for, it is already in the cache module, the prefetcher is going to waste power.

According to the scheme employed, prefetchers can be classified into two main categories: (1) software prefetchers, whose main components are composed of software with some hardware support [87], [46]; and, (2) hardware prefetchers, those for which pure hardware techniques are employed [68], [79]. In this thesis, we have focused our investigations on hardware prefetchers. However, some of the techniques presented in this thesis can be applied to both, hardware or software prefetchers. For this reason, the state of the art information will be focused mainly in the second group, but a brief introduction to software prefetchers is of interest in order to provide a global view of all the prefetching techniques. Although software prefetching falls outside the scope of this thesis, extending the research described here by examining these software prefetching algorithms would be of interest in the future.

- **Software prefetching**

  Prefetching operations are special instructions that are incorporated by a layer of code generation software (typically the compiler) into the binary. The aim of these instructions is to try to bring the data that is expected to be needed in the near future into a nearer level of memory. These operations are injected to the binary in accordance with the specific prefetching algorithm.

  The majority of software prefetchers analyzes the code during the compiling time and then injects the prefetching operations into the locations that the algorithm has identified as providing the best performance.

  The injected prefetching instruction must not stall the system if a miss occurs (i.e. the data is not present in the first level cache), as it must be dealt with in parallel with the execution of the program. For this reason, lock-free caches are required [40].

  One of the best well know software prefetching studies is [58] [57] [56]. In this study, the authors present a technique where the compiler injects prefetching requests. The technique analyses the code, and through complex mathematical operations tries to predict the behavior of the cache and when a memory request will miss. Therefore, it injects the request some instructions before in order to avoid the miss penalty.

  More recently, [67] and [35] published techniques that uses helper threads through the idle processors to execute the prefetching requests and avoid the instruction overheads.

This way, the main thread is assisted by the complementary threads and the prefetcher can be more aggressive without injecting overhead.

- **Hardware prefetching**

  The vast majority of hardware prefetching mechanisms are based on hardware structures that store the recent memory-related activity of the processor. Using this information and dedicated hardware, the prefetcher predicts which data the processor is going to request and physically injects the prefetch requests in the memory hierarchy when it is idle.

  The main advantages respect the software prefetching is that the hardware prefetcher knows about the hardware configuration of the memory and it does not inject any instruction overhead. However, making this predictions implies extra hardware, extra power, and extra time. For this reason, software prefetcher is able to do more complex predictions with higher processing time and more accuracy. But hardware prefetcher, can make faster predictions with lower penalty. Thus, as we have already said, in the rest of this thesis we will focus on hardware prefetching.

2.2.1 Prefetchers in current processors

Most current commercial processors implement several prefetching techniques, however, the details of most of them are secret. An example is the case of the Intel Nehalem processor. In [48] they explain that the Intel Core i7 processor has a four component hardware prefetcher. Two components associated with the second level cache and two components associated with the first level data cache. The two components of the second cache level hardware prefetcher are similar to those in the Pentium 4 and Intel Core processors [9]. There is a "streaming" component that looks for multiple accesses in a local address window as a trigger and an adjacency component that causes two lines to be fetched instead of one with each triggering of the "streaming" component. The L1 data cache prefetcher is similar to the L1 data cache prefetcher from the Core processors. It has another "streaming" component and a "stride" component that detects constant stride accesses at individual instruction pointers.

Nevertheless, the companies do not provide specific details on the heuristic used to generate new prefetchers. As we have seen, the provided information is always about the type of memory access patterns that they try to detect. The previously cited document talks about two kind of patterns that the Intel Core i7 tries to predict. These are the stream and stride patterns. Apart from this two patterns, the prefetchers can predict also analyzing the correlation between the accesses. Thus, as we will see with more details in the next section,

the prefetchers can be classified according to the pattern that they try to predict in three groups: *stream*, *stride*, and *correlation* prefetchers.

2.2.2 Implemented prefetchers

In this section we describe the three main kind of prefetchers according to its strategy to detect patterns. For our studies, we have selected one specific prefetcher from each group. These selected prefetchers are the ones that has been used along this thesis. We have selected these prefetchers because the literature seems to indicate that these mechanisms are the most representative in their categories [64].

Sequential prefetchers and the Tagged prefetcher

Most (but not all) prefetching schemes are designed to fetch data from main memory into the processor cache in units of cache blocks. It should be noted, however, that multiple word cache blocks are themselves a form of data prefetching. By grouping consecutive memory words into single units, caches exploit the principle of spatial locality [73] to implicitly prefetch data that is likely to be referenced in the near future.

Sequential prefetching can take advantage of spatial locality without introducing some of the problems associated with large cache blocks. The simplest sequential prefetching schemes are variations upon the one block look ahead (OBL) [73] approach which initiates a prefetch for block b+1 when block b is accessed. This approach differs from simply doubling the block size in that the prefetched blocks are treated separately with regard to the cache replacement and coherency policies. For example, a large block may contain one word which is frequently referenced and several other words which are not in use. Assuming an LRU replacement policy [73], the entire block will be retained even though only a portion of the block's data is in use. If this large block were replaced with two smaller blocks, one of them could be evicted to make room for more active data. Similarly, the use of smaller cache blocks reduces the probability that false sharing[1] occurs.

OBL implementations differ depending on what type of access to block b initiates the prefetch of b+1. In the following, we are going to discuss about the prefetch on miss and tagged prefetch algorithms. The prefetch on miss algorithm simply initiates a prefetch for block b+1 whenever an access for block b results in a cache miss. If b+1 is already cached,

[1] **False sharing**: false sharing occurs when a process attempts to periodically access data that will never be modified by another process, but that data shares a cache block with data that is modified, the caching protocol may force the first process to reload the whole unit despite the lack of logical necessity. The caching system is unaware of activity within this block and forces the first process to bear the caching system overhead required by true shared access of the resource.

no memory access is initiated. On the other hand, the tagged prefetch algorithm associates a tag bit with every memory block. This bit is used to detect when a block is demand-fetched or a prefetched block is referenced for the first time. In either of these cases, the next sequential block is fetched.

Smith found that tagged prefetching reduced cache miss ratios in a unified (both instruction and data) cache by between 50% and 90% for a set of trace-driven simulations. Prefetch on miss was less than half as effective as tagged prefetching in reducing miss ratios. The reason prefetch on miss is less effective is illustrated in Figure 2.2, where the behavior of each algorithm when accessing three contiguous blocks is shown. Here, it can be seen that a strictly sequential access pattern will result in a cache miss for every other cache block when the prefetch on miss algorithm is used but this same access pattern results in only one cache miss when employing a tagged prefetch algorithm. However, as we have said, it have to be an strictly sequential access pattern, other way the tagged prefetcher will send more useless prefetching operations than the other ones.

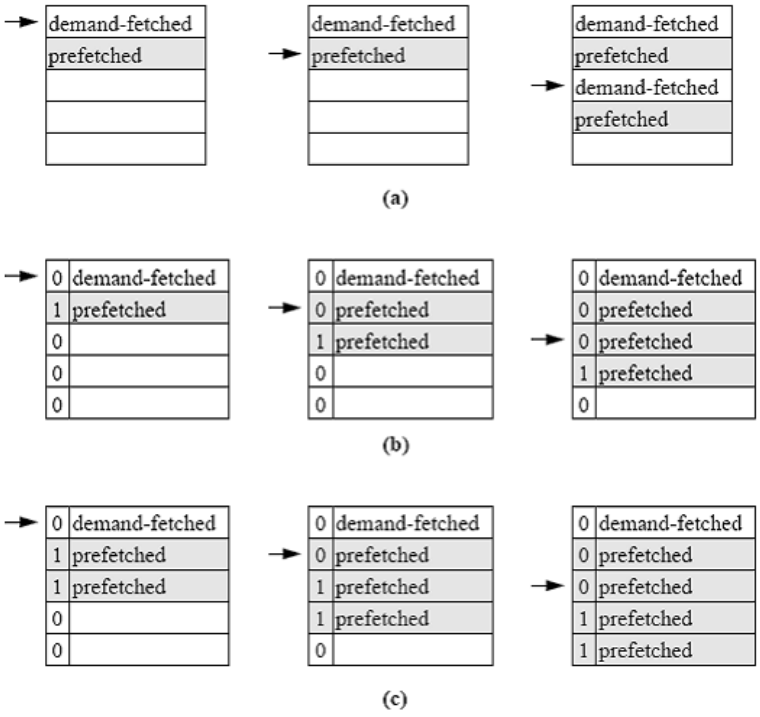


Fig. 2.2 Three forms of sequential prefetching: a) Prefetch on miss, b) tagged prefetch and c) tagged prefetch with K = 2.

Note that one shortcoming of the OBL schemes is that the prefetch may not be initiated far enough in advance of the actual use to avoid a processor memory stall. A sequential access stream resulting from a tight loop, for example, may not allow sufficient time between

the use of blocks b and b+1 to completely hide the memory latency. To solve this problem, it is possible to increase the number of blocks prefetched after a demand fetch from one to K, where K is known as aggressiveness. Prefetching K > 1 subsequent blocks aids the memory system in staying ahead of fast processor requests for sequential data blocks. As each prefetched block, b, is accessed for the first time, the cache is interrogated to check if blocks b+1, ... b+K are present in the cache and, if not, the missing blocks are fetched from memory. Note that when K = 1 this scheme is identical to tagged prefetching.

Although increasing the degree of prefetching reduces miss rates in sections of code that show a high degree of spatial locality, additional traffic and cache pollution are generated by sequential prefetching during program phases that show little spatial locality. This overhead tends to make this approach unfeasible for values of K larger than one as shown in [65].

Simulations of a shared memory multiprocessor found that adaptive prefetching could achieve appreciable reductions in cache miss ratios over tagged prefetching. However, simulated run-time comparisons showed only slight differences between the two schemes. The lower miss ratio of adaptive sequential prefetching was found to be partially nullified by the associated overhead of increased memory traffic and contention[77].

Sequential prefetching requires no changes to existing executables and can be implemented with relatively simple hardware. Compared to nowadays sequential prefetching policies, tagged prefetching appears to offer both simplicity and performance. Compared to software-initiated prefetching, sequential prefetching will tend to generate more unnecessary prefetch demands. Moreover, non-sequential access patterns, such as scalar references or array accesses with large strides, will result in unnecessary prefetch requests because they do not exhibit the spatial locality upon which sequential prefetching is based. To enable prefetching of strided and other irregular data access patterns, several more elaborate hardware prefetching techniques have been proposed, such as the ones explained in the next point.

Stride prefetchers and the Reference Prediction Table prefetcher

The idea for this mechanism is to identify a strided array referencing pattern, the simplest way to do it is to explicitly declare when such a pattern occurs within the program and then pass this information to the hardware. This is possible when programs are explicitly vectorized so that computation is described as a series of vector and matrix operations by the programmer, as is commonly done when programming machines which contain vector processors [28]. Given such a program, array address and stride data can be passed to prefetch logic which may then use this information to calculate prefetch addresses. This is what was in mind of Fu and Patel, in their scheme [23], a miss to an address as the result from a vector load or store

instruction with a stride of Δ prompts the prefetch hardware to issue fetches for addresses $a, a+\Delta, a+2*\Delta, ..., a+K*\Delta$, where K is again the degree of prefetching. Unlike sequential prefetching techniques, Fu and Patel's approach was found to be effective for values of K up to 32 due to the more informed prefetches that vector stride information affords.

When such high-level information cannot be supplied by the programmer, prefetch opportunities can be detected by hardware that monitors the processor's instruction stream or addressing patterns. Lee, et al.[47] examined the possibility of looking ahead in the instruction stream to find memory references for which prefetches might be dispatched. This approach requires that instructions be brought into a buffer and decoded early so that the operand addresses may be calculated for data prefetching. Instruction look ahead is allowed to pass beyond branches using a branch prediction scheme so that some prefetches will be issued speculatively. This speculation results in some unnecessary prefetches if the branch prediction is incorrect, in which case the buffer holding the speculatively loaded instructions must be flushed. Although this scheme allows for prefetching of arbitrary data access patterns, the prefetch window is limited by the depth of the decoded instruction buffer.

Several techniques have been proposed that employ special logic to monitor the processor's address referencing pattern to detect constant stride array references originating from looping structures [3] [23]. This is accomplished by comparing successive addresses used by load or store instructions. The scheme proposed by Chen and Baer [11] is perhaps one of the the most aggressive thus far. To illustrate its design, assume a memory instruction, m_i, that references addresses a_1, a_2, and a_3 during three successive loop iterations. A prefetch for m_i will be initiated if

$$(a_2 - a_1) = \Delta \neq 0$$

where Δ is now assumed to be the stride of a series of array accesses. The first prefetch address will then be $A_3 = a_2 + \Delta$, where A_3 is the predicted value of the observed address, a_3. Prefetching continues in this way until the equality $A_n == a_n$ no longer holds true.

Note that this approach requires the previous address used by a memory instruction to be stored along with the last detected stride, if any. Recording the reference histories of every memory instruction in the program is clearly impossible. Instead, a separate cache called the reference prediction table (RPT) holds this information for only the most recently used memory instructions.

The organization of the RPT is given in Figure 2.3. Table entries contain the address of the memory instruction, the previous address accessed by this instruction, a stride value for those entries which have established a stride, and a state field which records the entry's current state.

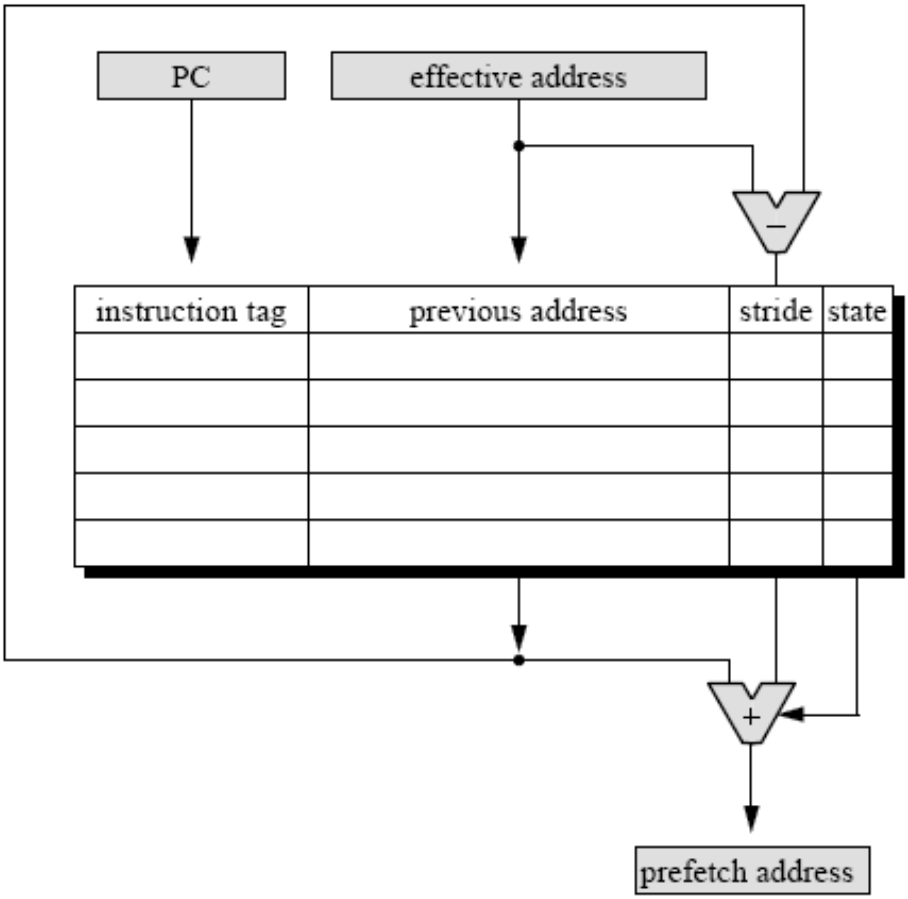


Fig. 2.3 The organization of the reference prediction table.

The RPT improves upon sequential policies by correctly handling strided array references. However, the RPT still limits the prefetch distance to one loop iteration. To remedy this shortcoming, a distance field may be added to the RPT which specifies the prefetch distance explicitly. Prefetch addresses would then be calculated as

$$effective\ address + (stride * distance)$$

The addition of the distance field requires some method of establishing its value for a given RPT entry. To calculate an appropriate value, Chen and Baer decouple the maintenance of the RPT from its use as a prefetch engine. The RPT entries are maintained under the direction of the Program Counter (PC) as described above but prefetches are initiated separately by a pseudo program counter, called the look-ahead program counter (LA-PC) that is allowed to precede the PC. The difference between the PC and LA-PC is then the prefetch distance, δ.

This prefetch mechanism consists of a table of RPT_SIZE entries indexed by the PC. As we have already said, each entry in the table has the structure showed in Table 2.1.

Instruction Tag	Previous address	stride	state

Table 2.1 Entries in the reference prediction table

When memory instruction m_i is executed for the first time, an entry for it is made in the RPT with the state set to initial, what means that no prefetching is yet initiated for this instruction. If m_i is executed again before its RPT entry has been evicted, a stride value is calculated by subtracting the previous address stored in the RPT from the current effective address.

Correlation prefetching. Global History Buffer

The correlation prefetching techniques make use of a history table to record consecutive address pairs. When a cache miss occurs, the miss address indexes the correlation table (see Figure 2.4). Each entry in the correlation table holds a list of addresses that have immediately followed the current miss address in the past. When a table entry is accessed, the member(s) of its address list are prefetched, with the most recent miss address first. The left side of Figure 2.4 illustrates the state of the correlation table after processing the miss address stream shown at the top of the figure.

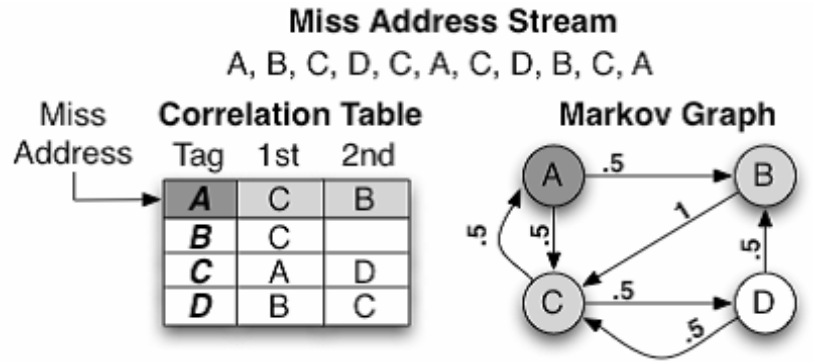


Fig. 2.4 Markov prefetching.

There are two main correlation prefetching techniques, both of them use an algorithm to get the correlation between cache misses, and then they use another algorithm to launch the prefetching operations. This second algorithm is exactly the same in both mechanisms. What makes them different is the way to calculate the correlation. The first one is the Markov Prefetching technique and the second one is a generalization of the Markov Prefetching and

it is called Distance Prefetching. In order to have a good point of view for the correlation prefetching methods, we are going to introduce these two mechanisms, and then how they are implemented making use of a Global History Buffer (GHB) [59].

Markov Prefetching [34]: The Markov prefetching mechanism models the miss address stream as a Markov graph – informally, a probabilistic state machine. Each node in the Markov graph is an address and the arcs between nodes are labeled with the probabilities that the arc's source node address will be immediately followed by the target node address. Each entry in the correlation table represents a node in an associated Markov graph, and its list of memory addresses represents arcs with the highest probabilities. Hence, the table maintains only a very crude approximation to the actual Markov probabilities. The right side of Figure 2.4 is the Markov transition graph that corresponds to the example miss address stream.

Distance Prefetching [36]: The Distance prefetching mechanism is a generalization of Markov Prefetching. Originally, Distance Prefetching was proposed for prefetching the translation lookaside buffer (TLB) [73] entries, but the method is easily adapted to prefetching cache lines. In this adaptation, Distance Prefetching uses the distance between two consecutive global miss addresses, an address delta, to index the correlation table. Each correlation table entry holds a list of deltas that have followed the entry's delta in the past. Figure 2.5 illustrates the address delta stream for the miss address stream in the previous example (Figure 2.4), and the state of the correlation table after processing the delta stream. Distance Prefetching is considered a generalization of Markov Prefetching because one delta correlation can represent many miss address correlations. By generalizing Markov Prefetching, Distance Prefetching is capable of prefetching most of the reference patterns that Markov Prefetching can. Plus, with the data it has available, it can also detect and prefetch delta access patterns that occur in the global miss address stream.

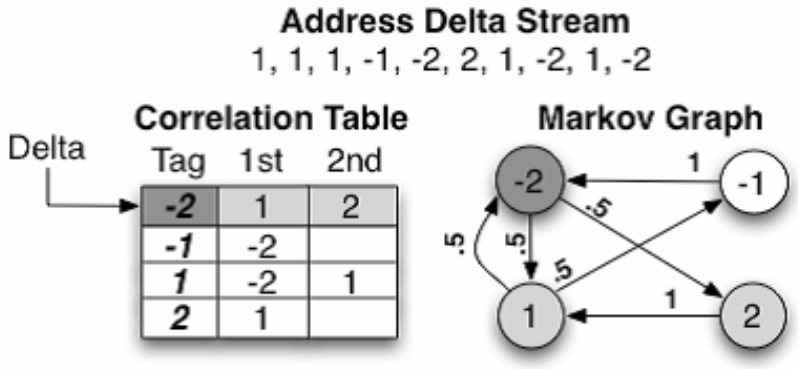


Fig. 2.5 Distance Prefetching.

Like all correlation prefetching methods, Distance Prefetching can be modeled with a Markov graph (as shown in Figure 2.5). However, unlike Markov Prefetching, Distance

Prefetching's state predictions are not prefetch addresses. To calculate prefetch addresses, the predicted deltas are added to the current miss address. If we do this, we will have, as in Figure 2.4, from the current miss, which is the probability that the next miss is one of his neighbors. Then it is time for the launching algorithm to chose which is going to launch.

The Global History Buffer (GHB) [59], which was proposed by Nesbit and Smith, is one way of implement different correlation prefetching techniques like the Markov Prefetching and Distance Prefetching. Before explaining what a global history buffer is and how does it makes use of the prefetching correlation techniques, it is necessary to explain the drawbacks of using a common table to implement the techniques explained before.

In general, prefetch tables store prefetch history inefficiently. First, table data can become stale, and consequently reduce prefetch accuracy (the percent of prefetches that are actually used by the program before being evicted). Second, tables suffer from conflicts that occur when multiple access keys map to the same table entry. The most common solution for reducing table conflicts is to increase the number of table entries. However, this approach increases the table's memory requirements and the percentage of stale data held in the table. Third, tables have a fixed (and usually a small) amount of history per entry. Adding more prefetch history per entry creates new opportunities for effective prefetching, but the additional prefetch history also increases the table's memory requirements and its percentage of stale data, which together can negate the advantages.

To provide more efficient prefetchers the Nesbit and Smith proposal is to use an alternative prefetching structure that decouples table key matching from the storage of prefetch-related history information. The overall prefetching structure has two levels (Figure 2.6):

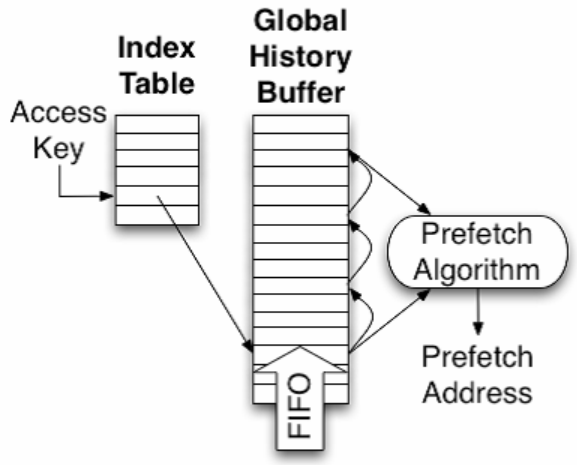


Fig. 2.6 Global History Buffer Prefetch Structure.

- An Index Table (IT) that is accessed with a key as in conventional prefetch tables. The key may be a load instruction's PC, a cache miss address, or some combination. The entries in the Index Table contain pointers into the Global History Buffer.

- The Global History Buffer (GHB) is an n-entry FIFO table (implemented as a circular buffer) that holds the n most recent cache miss addresses. Each GHB entry stores a global miss address and a link pointer. The link pointers are used to chain the GHB entries into address lists. Each address list is the time-ordered sequence of addresses that have the same Index Table key.

Depending on the key that is used for indexing the Index Table, any of a number of history-based prefetch methods can be implemented. In the following, we illustrate how the GHB can be used to implement correlation prefetching.

To simplify the discussion and illustrate the relationship between methods, the names given to prefetching methods follow a consistent taxonomy. Each method is denoted as a pair: X/Y, where X is the key used for localizing the miss address stream and Y is the mechanism used for detecting addressing patterns. We consider one localizing method: Global (G), and two detection mechanisms: Delta Correlation (DC), and Address Correlation (AC). So we are going to explain first, how is implemented the Markov prefetching using a G/AC and then the distance prefetching using G/DC.

We first use Markov Prefetching (G/AC) to illustrate a GHB prefetcher. See Figure 2.7. When a cache miss occurs, the miss address indexes the Index Table. If there is a hit in the Index Table, the Index Table entry will point to the most recent occurrence of the same miss address in the GHB. This GHB entry is also at the head of the linked list of other entries with the same miss address. For each entry in this linked list, the next entry in the FIFO ordered GHB is the miss address that immediately followed the current miss address when it occurred in the past. These "next" global miss addresses are prefetch candidates. With the bottom-to-top orientation of the GHB in Figure 2.7, the "next" GHB entry is immediately below the current entry. To better illustrate the method, the current memory address is shaded with a darker gray and the prefetch memory addresses are shaded with a lighter gray. In the example, prefetch candidates generated by walking the address list are C and B, the same addresses held in the conventional Markov correlation table in Figure 2.4.

The Markov model can serve as a basis for a number of global correlation prefetching methods. As earlier mentioned, the nodes are addresses and an edge connecting two nodes indicates the probability that one address is followed by the other in the global miss address stream. Prefetch algorithms can use this information in a number of ways to predict future addresses.

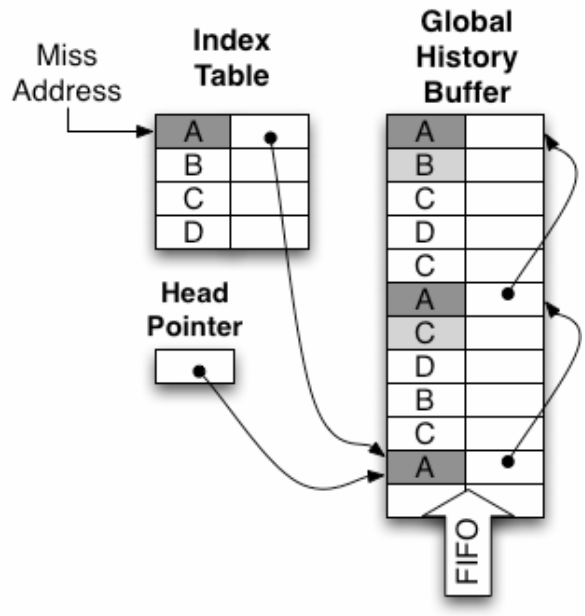


Fig. 2.7 GHB Global / Address Correlation.

In terms of the Markov graph, existing Markov (G/AC) and Distance Prefetching (G/DC) methods essentially start at a node and prefetch using one or more adjacent nodes – but only the immediately adjacent nodes. We refer to this as width prefetching, and it is "wired in" to the table structures that implement Markov and Distance Prefetching. In terms of the original Markov graph, however, one can also consider depth prefetching; i.e. beginning with the current miss address, the sequence of the most likely edges is followed, with prefetching initiated at each node along the path. Of course, one can also consider hybrid methods that use a combination of width and depth, i.e. the sequence of the most likely arcs edges prefetched, then the second most likely, and so on.

For most workloads, a problem with relying only on width is that the effective look-ahead distance is relatively short and prefetches have poor timeliness (whether prefetches are issued early enough to prevent processor stalls). On the other hand, depth prefetching allows the prefetcher to run farther ahead of the actual address stream.

The GHB method can be used for either width or depth prefetching as well as hybrid combinations. Following the address linked list alone gives width prefetching; using the sequential GHB entries beginning at a member in the address list adds depth.

Figure 2.8 illustrates how the GHB can prefetch depth in the global miss address stream using delta correlations, Global / Delta Correlation depth (G/DC depth) prefetching. The lighter "Deltas" box shown in the figure does not exist in GHB hardware, but is extracted by finding the difference between miss addresses in the GHB. As shown in the figure, prefetch

addresses are generated by taking the miss address and accumulatively adding deltas; a valid prefetch address is created from each addition.

With the GHB approach, one can often get a better estimate of the actual Markov graph transition probabilities than with conventional correlation methods. In fact, the GHB allows a weighting of transition probabilities based on how recently they have occurred. For example, if the last five Markov transitions from state A were to C, B, B, B, and B, where C is the oldest transition, a GHB prefetching method (with a width of two) would examine the first two transitions and prefetch B. In contrast, a correlation table prefetching method would prefetch B and C, no matter how long ago the transition to C occurred. In practice, the GHB's width prefetching method results in improved prefetch accuracy.

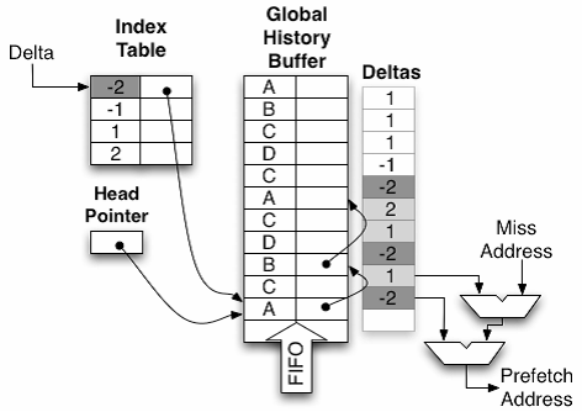


Fig. 2.8 GHB Global / Delta Correlation.

2.2.3 Performance implications

The effectiveness of prefetching depends greatly on the correct prediction of future memory accesses of the specific mechanisms used, but also on other parameters like the timeliness of the generated requests. Note that prefetching a request may pollute the cache by substituting useful memory lines with other lines that may or may not be needed in the future. There are several metrics that are used to measure prefetching effectiveness. The most important according to the literature [74] are the following: IPC speedup, accuracy, lateness, pollution, and aggressiveness. A brief description of each of these is given below.

- **IPC Speedup**: This is the most common metric used to evaluate performance. Its value indicates how much faster is the system when prefetching is going on it:

$$IPC\ Speedup = \frac{(IPC\ simulation\ without\ prefetching)}{(IPC\ simulation\ with\ prefetching)}$$

Were the **IPC simulation without prefetching** is the number of instructions per cycle executed by the simulated processor without the prefetching module activated and the **IPC simulation with prefetching** is the number of cycles executed by the simulated processor under the prefetcher influence.

- **Accuracy**: This is a metric whose value is between 0 and 1, and it represents the proportion of all the prefetching operations launched which have proved to be useful. Accuracy is calculated using this formula:

$$accuracy = \frac{(useful\ prefetches)}{(total\ number\ of\ prefetches)}$$

On the one hand, the *useful prefetches* are the number of prefetching operations that put a cache line into the cache that is subsequently used by a demand operation. On the other hand, *the total number of prefetches* is all the memory operations launched by the prefetch module.

- **Lateness**: This is a measure of how timely the prefetch requests generated by the prefetcher are with respect to the demand accesses that require the prefetched data. Lateness is also a value between 9 Late prefetches will hide part of the miss latency, but not all. Lateness is measured with the following formula:

$$lateness = \frac{(late\ prefetches)}{(useful\ prefetches)}$$

A *late prefetch* is counted when a load or store operation looks for a line in the cache and the line is not yet in the cache, but it is due to enter the cache as part of a prefetch operation.

- **Pollution**: Cache pollution generated by the prefetcher is a measure of the disturbance caused by prefetched data in the cache module. Pollution is measured using this formula:

$$pollution = \frac{(Number\ of\ demand\ misses\ caused\ by\ the\ prefetcher)}{(Total\ of\ demand\ misses)}$$

The *number of demand misses caused by the prefetcher* is the total number of misses that would not have occurred if the prefetcher was not activated. The *total number of demand misses* concerns the misses produced by any load or store operation.

- **Aggressiveness**: Rather than being an output statistic, aggressiveness is an input parameter. This parameter is used to determine the number of extra prefetch requests that are generated every time the prefetcher engine decides to generate a request. Increasing the aggressiveness increases the chances of obtaining useful prefetches, but may decrease accuracy and increase pollution.

Note that, an ideal prefetching algorithm should satisfy, among others, two important properties that are to produce a minimum increase of traffic in the memory interconnection and to have a low computational and implementation complexity. Notice that the increase in the memory traffic entails higher power consumption and a higher degree of contention in the interconnection network, specially, if we are in a multiprocessor system.

Moreover, in a CMP environment, cores share system resources beyond some level in the memory hierarchy, such as the Network on Chip (NoC) interconnect, the last level cache, and the main memory bandwidth. Due to interferences in these shared resources, aggressive prefetching on the different cores and levels of the hierarchy may lead from diminished beneficial effect from prefetching, to actually hurting the global system performance. These topics are explained in more detail in the following points.

2.2.4 Prefetching in CMPs

As we have said, all the concepts previously commented are harder to control when working in multicore [10]. This makes more difficult to do good prefetch operations. Moreover, wrong prefetching penalties are higher in multicore than in single-core. Examples of this are the potential increase of the natural latencies, the increase in power consumption of the network, or the interference in other core memory operations produced by unnecessary traffic that traverses the network. So in multicore is much more important to have accurate and timeliness prefetchers. Opposite to the single-core scenario, the number of papers that address the issues of prefetching in multicore are less. However there are some very interesting proposals that approach the problem from different perspectives.

In this section, we are going to comment some representative techniques that tries to improve the prefetching mechanisms from different perspectives. These techniques can be applied to the vast majority of the current prefetchers and improve them, which as we have already said, is one of the objectives of this thesis.

- In [42], this study is focused on an adaptive DRAM controller that self-adapts priorities of prefetching requests to minimize the negative impact of useless prefetches and maximize the benefits of useful prefetches. The aim behind this proposal is based on the observation that some controllers treat prefetch requests the same as demand

requests or prioritize demand over prefetch. The problem is that treating prefetches and demands equally can lead to performance loss if prefetches are useless. In a similar way, prioritizing demands over prefetches can also degrade performance if prefetches are useful.

- The second one [19] is a global management approach that proposes a low-cost mechanism to make prefetching between L2 shared memory and off-chip memory more effective, minimizing interferences among cores. The idea is to dynamically tune the aggressiveness of core prefetchers by means of a global control system that accepts or overrides decisions made by local control systems.

- The last mechanism [21] makes use of the heterogeneous interconnect to deal with the extra number messages that prefetching introduces in the interconnection network that significantly impacts on power consumption. This approach assumes low-power wires for dealing with prefetching messages meanwhile rest of them travels on regular baseline wires. This is one of the few papers that address the topic of prefetching in multicore from the NoC perspective and this is of special interest for our work.

2.3 Networks on chip

As stated in the introduction, NoC is becoming a research hot topic [69]. The three critical challenges for NoC according to [61] are: power, latency, and CAD compatibility. Note that, as the number of cores increases, communication becomes more important and thus, the contribution to the global power and performance of the NoC is significantly increased. It is critical thus, to implement NoC with proper power/performance characteristics. On the other hand, CAD compatibility refers to avoid on-chip network circuits and architecture techniques that are incompatible with modern design flows and CAD tools, making them unsuitable for use in NoCs [31]. To achieve these challenges, according to [60] the research areas in NoC can be classified in three main areas: Communication infrastructure, Communication paradigm, and Application mapping. These areas would encompass the following topics:

- **Communication infrastructure**: topology and link optimization, buffer sizing, floor-planning, clock domains and power.

- **Communication paradigm**: routing, switching, flow control, quality-of-service and network interfaces.

- **Application mapping**: task mapping/scheduling and IP component mapping.

A NoC consists of routers, links, and network interfaces. Routers direct data over several links (hops). Topology defines their logical lay-out (connections) whereas floorplan defines the physical layout. The function of a network interface (adapter) is to decouple computation (the resources) from communication (the network). Routing decides the path taken from source to the destination whereas switching and flow control policies define the timing of transfers. Task scheduling refers to the order in which the application tasks are executed and task mapping defines which processing element (PE) executes certain task. IP mapping, on the other hand, defines how PEs and other resources are connected to the NoC.

In the following subsections, we will review some interesting works on the area of communication infrastructure and communication paradigm that address the issue of providing differentiated treatment to different kind of traffic.

2.3.1 Traffic differentiation in the communication infrastructure

In the communication infrastructure area, we have focused in the sub-area of interconnects at the microarchitecture level. [12] Introduces to the literature the idea of heterogeneous interconnects. This heterogeneous on-chip interconnection network is done by links that are comprised of wires with varying physical properties. By tuning wire width and spacing, it is possible to design wires with varying latency and bandwidth properties. Similarly, by tuning repeater size and spacing, it is possible to design wires with varying latency and energy properties. Specially, in this paper they propose to have two new wire implementations apart from baseline wires (B-Wires): power optimized wires (PW-Wires) and bandwidth optimized wires (L-Wires) in order to fit the latency and bandwidth requirements of the different kinds of coherence messages. On the other hand, [20] proposes another mechanism to make use of the heterogeneous networks proposed in [12]. In this study, the authors explore such an approach by proposing the use of an address compression scheme in this context that allows most of the critical messages, used to ensure coherence between the L1 caches of a CMP, to be compressed in few bytes and transmitted using very-low-latency wires meanwhile the rest of messages are transmitted using baseline wires. Another idea based on the area of the interconnections at the microarchitecture level, and very similar to the already presented heterogeneous interconnect mechanisms is the **secondary parallel networks** mechanism. In [4], the authors claim that including a second parallel network can increase performance while improving efficiency. Specially, they evaluate two strategies for distributing traffic over the sub-networks: (1) The heterogeneous architecture deals with one sub-network to transport short packets (read requests, write replies) and the other to transport long packets (read replies, write requests). (2) The homogeneous architecture deals with one

sub-network to transport packets associated with read transactions (requests and replies) and one to transport packets associated with write transactions.

2.3.2 Traffic differentiation in the communication paradigm

In order to have a good overview of this sub-area, we have selected four different techniques that show different ways to improve performance by taking into account traffic characteristics.

1. The first one [53] proposes a novel adaptive **routing algorithm**, which they call Destination-Based Adaptive Routing (DBAR). The objective of their proposal is to increase performance by employing more accurate information to select the path through the network. Moreover, they claim to provide a higher isolation among different applications running on different regions of the processor. Specifically, the algorithm mitigates intra- and inter-application interference by using only congestion information of the nodes in the minimum quadrant defined by the current and destination nodes. In this way, the routing algorithm employs both local and non-local congestion network information to decide the output link in router traversed along the path between the source and the destination. In order to communicate the congestion information, they propose a low-cost congestion information propagation network..

2. The following technique is another mechanism that suggests hardware modifications to the router in order to guarantee a certain level of Quality of Service (QoS). [44] presents **Globally-Synchronized Frames** (GSF), to implement QoS for multi-hop on-chip networks. GSF provides guaranteed and differentiated bandwidth as well as bounded network delay without significantly increasing the complexity of the on-chip routers. In a GSF system, time is coarsely quantized into "frames" and the system only tracks a few frames into the future to reduce time management costs. Each QoS packet from a source is tagged with a frame number indicating the desired time of future delivery to the destination. At any point in time, packets in the earliest extant frame are routed with highest priority but sources are prevented from inserting new packets into this frame. GSF exploits fast on-chip communication by using a global barrier network to determine when all packets in the earliest frame have been delivered, and then advances all sources and routers to the next frame. The next oldest frame now attains highest priority and does not admit any new packets, while resources from the previously oldest frame are recycled to form the new future frame.

3. In the next technique [16] the responsible of guaranteeing the QoS is a **prioritization** criterion. The authors of this technique not only propose prioritization policies,

they also propose architectural extensions to NoC routers that improve the overall application-level throughput, while ensuring fairness in the network. The prioritization policies in this technique are application-aware, distinguishing applications based on the stall-time criticality of their packets. The idea is to divide processor execution time into phases, rank applications within a phase based on stall-time criticality, and have all routers in the network prioritize packets based on their applications' ranks.

4. In line with the previous idea, [17] propose a different approach of this concept. They present a NoC architecture with support for packet prioritization based on latency slack. The objective of this architecture is not to provide network-level QoS but optimize system throughput. It proposes a heuristic to approximate latency slack, modifications in the network queues (NIC, switches, and memory controller) to support several ties of priorities. It uses batching intervals to avoid starvation of low priority packets and it presents an estimation of the priority arbiter delay based on adder delays and results of average system throughput and network fairness.

2.3.3 Traffic differentiation in the application mapping

Although this is not one of the main topics in this thesis, so we are not going to enter in too much detail in it, it affects in some of the techniques previously cited. For example, the DBAR routing technique commented in the previous section tries to increase the fairness in the execution of the applications that are simultaneously sending traffic through the network. To be effective, this technique relies in a certain mapping of the applications. Furthermore, it uses the results provided in [14] and [15] which show that mapping each application to a near convex region provides the optimal NoC configuration for a multicore system in which multiple applications are currently running.

In [14] the authors propose to make use of the user behavior information in the application resource allocation process. This allows the system to better respond to real-time changes and to adapt dynamically to different user needs. Several algorithms are proposed for solving the task allocation problem while minimizing the communication energy consumption and network contention. In [15] they address precisely the energy and performance aware incremental mapping problem for NoC with multiple voltage levels and propose a technique (consisting of region selection and node allocation) to solve it. Moreover, the proposed technique allows adding new applications to the system with very low inter-processor communication overhead.

2.4 Dynamic management

Given that the effectiveness of prefetching, in a high degree, depends on workload and run-time effects, several techniques have been proposed to dynamically adapt the behavior of the prefetching mechanism based on run-time profiling information of the previously presented parameters. In fact, given the extra interaction complexity inherent to CMP systems, research on this environment has, therefore, focused more on improving the overall performance achieved by existing prefetching mechanisms than on proposing new prefetching engines to detect new kinds of patterns. Some of the most relevant techniques that have been proposed are summarized in the following sections.

2.4.1 Filtering

The action of this technique relies on avoiding specific prefetching requests according to some heuristic. For instance, the Pollution Filter approach [91], [90] is a set of filtering techniques that are implemented through a standalone module that examines the addresses generated from the prefetcher. The prefetch engine generates the prefetching request and reroutes it to the prefetch pollution filter to check if the request should proceed. If the prefetch pollution filter rejects the prefetch, this prefetch operation will be terminated and no prefetch will be issued to the L1 cache. Otherwise, the prefetch is issued to the prefetch queue. The heuristic used in order to decide if a prefetch operation may proceed or not, may vary according to the filtering technique. Nevertheless, the heuristic presented in the literature, is based on a branch predictor technique. This heuristic, tries to predict the confidence of the prefetch requests triggered by a certain static instruction. This is done by using a table indexed by the static instruction address. Each entry of this table contains a two but counter that is increased when a prefetch request triggered by this static instruction is useful and decreased when it is useless.

2.4.2 Throttling

This technique relies on modifying the aggressiveness of the prefetching engine according to some heuristic. The aggressiveness of a prefetcher refers to the degree of speculation and number of prefetch requests that are generated each time the prefetching engine is triggered. The difference with filtering is that prefetching requests are not discarded after being generated. Instead of that, throttling increases or decreases the aggressiveness of the prefetching mechanism. For instance, Feedback Directed Prefetching (FDP) [74] estimates some characteristics as the accuracy, prefetcher timeliness, and prefetcher-caused cache

pollution, and it makes use of all these data to adjust the aggressiveness of the data prefetcher dynamically. With this aggressiveness adjustment the system can increase the performance improvement provided by prefetching as well as reduce the negative performance and bandwidth impact of prefetching. Hierarchical Prefetcher Aggressiveness Control (HPAC) [19] is a global throttling management approach that proposes a low-cost mechanism to make prefetching between L2 shared memory and off-chip memory more effective, minimizing interferences among cores. The idea is to dynamically tune the aggressiveness of core prefetchers by means of a global control system that accepts or overrides decisions made by local control systems. Moreover, filtering and throttling are both orthogonal, so they can be applied at the same time.

2.4.3 Prioritization

Traditionally, memory systems do not make a difference between prefetch requests and demand requests. Lately, there are several approaches that try to give different priorities to both types of requests as it has been shown that delaying demand requests may degrade performance, particularly if prefetch requests are not accurate [42]. For instance, approaches as issuing prefetches only when the memory channels are idle [49] or an insertion policy in the Feedback Directed Prefetching (FDP) mechanism [74] always prioritize demand requests over prefetch requests. Nevertheless, this rigid prioritizing does not always provides the best performance as a delayed useful prefetch request may turn into a completely useless prefetch if arrives too late. Moreover, traffic congestion and energy consumption would be increased for nothing. Prefetch-Aware DRAM Controller (PADC) [42] focus on an adaptive DRAM controller that self-adapts priorities of prefetching requests to minimize the negative impact of useless prefetches and maximize the benefits of useful prefetches.

There is another approach that differentiates prefetch requests from the rest by means of a heterogeneous interconnect [21]. The idea is to assume low-power wires for dealing with prefetching messages meanwhile the rest of them travels on regular baseline wires. Finally, prioritization of demand packets over prefetch packets inside the NoC routers [13], [45] has been also considered. Now, the idea is to apply a regular arbitration (round-robin or age-based) among all the demand packets and only when there are no demand candidates, a round-robin arbitration among the prefetch packets will be applied. The main difference between these two works is that in [45] accuracy information is also used to determine the priority of the prefetch requests.

2.4.4 Summary of other dynamic management techniques

In [51] the authors have proposed analytical models for bandwidth partitioning to identify when prefetching can help in improving system performance. In [71] the authors used reuse-distance based cache modeling to insert non-temporal prefetch instructions to cache bypass the data that is not reused from the lower level caches. Similarly, in [41] the authors proposed a runtime mechanism to find opportunities to insert non-temporal prefetch instructions in batch applications to conserve LLC space so that userfacing applications' performance in data-centers remains predictable. In [33] they implemented a run-time mechanism for exploring and adjusting hardware prefetcher configuration on a POWER7 processor to maximize performance. The POWER7 processor allows the prefetcher aggressiveness to be configured at 7 different levels. Their runtime method explores the best hardware prefetcher settings on per-core basis (for two cores only) and applies the one that performs best.

2.5 Open-source multiprocessor simulators

In order to accurately simulate the effects of prefetching, it is necessary to emulate the cores, the memory system, and the NoC in a realistic way. Table 2.2 summarizes the main open-source simulators that we have analyzed in order to assess whether they possess the features needed to accurately simulate a CMP system with prefetching. MARSSx86 [62] possesses a full simulation environment with which to simulate a multi-core system and memory hierarchy. Moreover, in the memory hierarchy it offers a prefetching module with different prefetching engines whose performance in the hierarchy can be tested. However, it does not support the simulation of the NoC. The second simulator, Simics-GEMS [54], possesses most of the requirements, but is almost obsolete because the researchers who were working on the GEMS part of Simics-GEMS abandoned this project some years ago in order to build the newer gem5 simulator [7]. This latter full system simulator is able to simulate a simple (Classic) or a detailed (Ruby) memory hierarchy. The two memory models of this system have been placed on a different row in Table 2.2, which shows that, with Ruby, gem5 can simulate a detailed NoC system. Note that, the classic memory system from gem5 models a simplistic bus between cores and the shared cache, and between the cache and main memory. This bus is also able to show statistics about contention. When the number of cores increases, it is not realistic to use a bus to connect them. We need to make use of the Ruby memory system, which is able to model a more realistic switched NoC. This detailed memory system does not support any kind of prefetching. Therefore, to the best of our knowledge, there is no open-source simulator that, in its official release form, is capable of the accurate simulation of prefetching in multi-core environments with large number of cores.

	Core	Memory	NoC	Up-to-date	Prefetch
MARSSx86	X	X		X	X
Simics-GEMS	X	X	X		
gem5-Classic	X	X		X	X
gem5-Ruby	X	X	X	X	

Table 2.2 Requirements for multi-core simulators.

2.5.1 Benchmarks

Several benchmark suits are supported to work with this simulator (SPLASH, SPEC, and PARSEC). From this set of benchmarks, we chose PARSEC because is the multithreaded suite with the most wide range of workloads. According to [6] these are the five requirements for a benchmark suite:

- **Support for multi-threaded applications**: Shared-memory CMPs are already ubiquitous. The trend for future processors is to deliver large performance improvements through increasing core counts on CMPs while only providing modest serial performance improvements. Consequently, applications that require additional processing power will need to be parallel.

- **Support for emerging workloads**: Rapidly increasing processing power is enabling a new class of applications whose computational requirements were beyond the capabilities of the earlier generation of processors. Such applications are significantly different from earlier applications. Future processors will be designed to meet the demands of these emerging applications and a benchmark suite should represent them.

- **Need to be diverse**: Applications are increasingly diverse, run on a variety of platforms and accommodate different usage models. They include both interactive applications like computer games, offline applications like data mining programs and programs with different parallelization models. Specialized collections of benchmarks can be used to study some of these areas in more detail, but decisions about general-purpose processors should be based on a diverse set of applications. While a truly representative suite is impossible to create, reasonable effort should be made to maximize the diversity of the program selection. The number of benchmarks must be large enough to capture a sufficient amount of characteristics of the target application space.

- **Employ state-of-art techniques**: A number of application domains have changed dramatically over the last decade and use very different algorithms and techniques. Visual applications for example have started to increasingly integrate physics simulations to

generate more realistic animations. A benchmark should not only represent emerging applications but also use state-of-art techniques.

Program	Application Domain	Parallelization		Working Set	Data Usage	
		Model	Granularity		Sharing	Exchange
blackscholes	Financial Analysis	data-parallel	coarse	small	low	low
bodytrack	Computer Vision	data-parallel	medium	medium	high	medium
canneal	Engineering	unstructured	fine	unbounded	high	high
dedup	Enterprise Storage	pipeline	medium	unbounded	high	high
facesim	Animation	data-parallel	coarse	large	low	medium
ferret	Similarity Search	pipeline	medium	unbounded	high	high
fluidanimate	Animation	data-parallel	fine	large	low	medium
freqmine	Data Mining	data-parallel	medium	unbounded	high	medium
raytrace	Rendering	data-parallel	medium	unbounded	high	low
streamcluster	Data Mining	data-parallel	medium	medium	low	medium
swaptions	Financial Analysis	data-parallel	coarse	medium	low	low
vips	Media Processing	data-parallel	coarse	medium	low	medium
x264	Media Processing	pipeline	coarse	medium	high	high

Fig. 2.9 Qualitative summary of the inherent key characteristics of PARSEC benchmarks. The pipeline model is a data-parallel model which also uses a functional partitioning. PARSEC workloads were chosen to cover different application domains, parallel models and runtime behaviors.

- **Support research**: A benchmark suite intended for research has additional requirements compared to one used for benchmarking real machines alone. Benchmark suites intended for research usually go beyond pure scoring systems and provide infrastructure to instrument, manipulate, and perform detailed simulations of the included programs in an efficient manner.

In Figure 2.9, we present a qualitative summary of the PARSEC Benchmarks key characteristics. PARSEC workloads were selected to include different combinations of parallel models, machine requirements and runtime behaviors. As stated before, PARSEC meets all the requirements outlined in the previous point and because of this reason this has been the chosen benchmark suite to be used in the experiments of this thesis.

Chapter 3

Prefetching evaluation in multi-core platforms

> Good ideas are not adopted automatically. They must be driven into practice with courageous patience.
>
> *Hyman Rickover*

3.1 Introduction

It is not easy to accurately simulate the effect of a prefetcher in a system. The main element affected by the prefetching mechanism is the memory system. However, given that prefetching is sensitive to the timing of the memory request workload, in order to be really effective [75], a trace-based simulator will not provide high accuracy. This is the reason why cores must be simulated to at least a certain level of detail, yet the prefetching mechanism becomes more complicated to simulate when we move to a multiprocessor environment.

One of the main challenges when simulating prefetching mechanisms in a multiprocessor environment concerns bandwidth consumption. Because the number of cores increases, the network which connects the processing and memory elements becomes more and more important for global performance and the power of the global system [8]. This on-chip communication subsystem is widely known as the network-on-chip (NoC) [70]. The effect of prefetching over the NoC is a key issue because, depending on the specific prefetching technique used and the workload, it may increase the number of memory requests considerably. This directly affects the amount of traffic that traverses the network due to both the

memory requests themselves but also the coherence traffic that is generated as a result, which is also an important factor in network traffic.

However, no current open-source simulator allows the simulation of prefetching, the core, the memory system, and the NoC at the same time. This is a roadblock for any researcher trying to work on prefetching in distributed memory multiprocessors. This is the motivation behind our decision to modify one of the best well known open-source simulators in order to obtain a simulation framework in which the cores, NoC, and the memory hierarchy are emulated to a high level of detail.

Therefore, in this chapter we show our work in two directions: firstly, to provide an infrastructure that can perform complete prefetching simulations for multi-core systems; secondly, to show the importance of simulating the NoC when working with prefetching, due to the significant effect of network congestion on performance. It is important to note that this will be even more critical as the number of cores incorporated in future multiprocessors increases. Having studied the different simulators, as it is shown in Section 2.5, we decided to develop a prefetch module for gem5-Ruby, converting it into a prefetching-aware simulator. Thus, in this chapter we show the following contributions:

- We implement a generic prefetching module which enables a prefetching engine to be attached to any cache level. This module has been implemented with the aim of easily incorporating specific prefetching engines for both L1 and L2 caches. A statistics module has been also developed for the provision of specific prefetching statistics. This chapter describes the main implementation details for these capabilities.

- We have implemented three of the most characteristic prefetchers from the literature. An initial set of specific prefetching statistics has been incorporated to enable the level of success of the generated prefetching requests to be determined. This chapter presents a full set of performance results for the usage of these prefetchers at different levels of aggressiveness and in different network environments.

- We analyze the effect of prefetching-generated traffic on the NoC and its direct contribution to application performance. We use these results to show the significance of simulating all the relevant components in order to obtain accurate performance numbers for prefetching.

The rest of the chapter is organized as follows. Section 3.2 gives a brief introduction to the open-source simulator used in this study. The changes made to the simulator are described in Section 3.3, while Section 3.4 demonstrates the functionality of the prefetching framework implemented. Finally, the main conclusions are summarized in Section 3.5.

3.2 The official release version of the gem5 simulator

The gem5 simulator has two main components: the CPU and the memory. Each of them can be sub-divided into many other blocks. This model allows the user to choose from several options, shown in Figure 3.1. At the top is the CPU module and it is composed of the ISA and the CPU model. Below is the memory system, which can be chosen to act as a classic memory system or as the detailed (Ruby) memory system. It is only with the Ruby memory system the user is able to choose between the NoC options. The reason for this is that, in gem5, the NoC is considered to be part of the memory system and, in particular, part of the Ruby memory system. Further information on each of these modules is given in the following subsections.

It is important to mention that gem5 is a relatively new simulator [7], so it is currently not completely finished. Therefore, some of the combinations that the simulator will allow in the future are not ready at the present time. The gem5 website [27] contains a status matrix displaying the current state of development of the simulator.

3.2.1 CPU module

The CPU module contains two main areas: (1) the ISA and (2) the CPU model. Gem5 can work with 6 different ISAs: ARM, ALPHA, X86, SPARC, Power PC, and MIPS. The user can also choose the level of detail used to model the micro-architectural aspects of the CPU. As shown in Figure 3.1, there are two levels of detail: simple and detailed. Furthermore, there are two CPU models for each level of detail. It should be noted that a higher level of detail entails a longer simulation time.

3.2.2 The memory system

The memory module behaves as the CPU. The simulator allows users to choose between two memory systems: Classic (simple) and Ruby (more detailed), Ruby takes much more simulation time than Classic, but it also has much more detail and versatility. It is important to remark that Classic already has a prefetching module implemented with several prefetching engines, but this memory system cannot simulate the NoC. Therefore, all our modifications have been carried out using Ruby (the detailed memory system).

The internal structure of Ruby is shown in the memory model box in Figure 3.1. Ruby is a complete memory simulator: it includes the cache modules, the coherence controllers, and the interconnection network between the different tiles of the system, details of which will be discussed in section 3.2.3. Ruby can model inclusive/exclusive cache hierarchies with

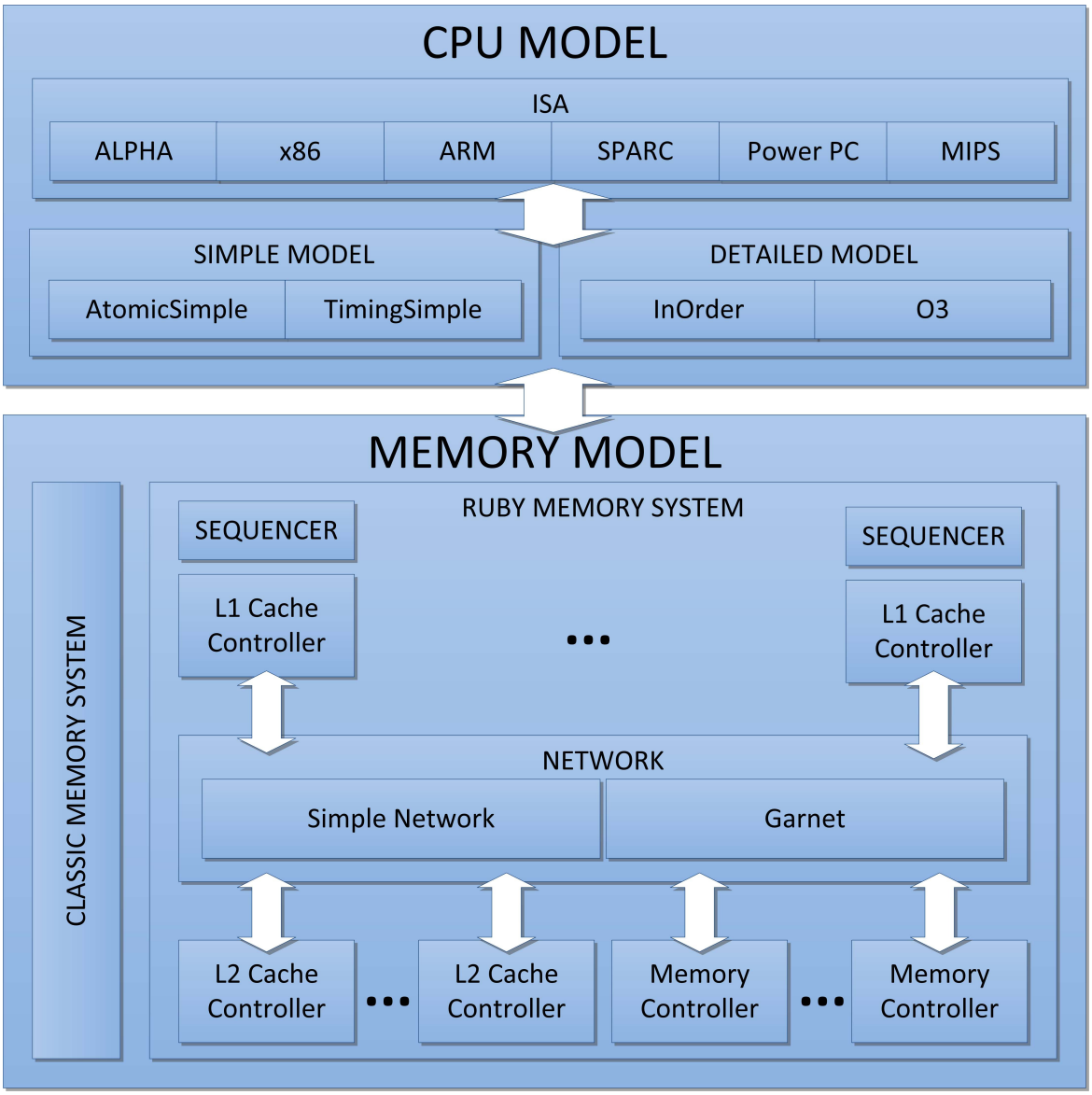


Fig. 3.1 Internal structure of gem5, composed of two main modules: CPU and Memory.

various replacement policies, coherence protocol implementations, interconnection networks, DMA, memory controllers, and various sequencers (initiating memory requests and handling responses). Ruby is implemented as a combination of different modules, making it flexible and configurable. There are three main modules: the implemented C/C++ classes, the network, and the protocol. Ruby also allows any aspect related to the memory hierarchy functionality to be configured, while also enabling any modification in the memory controller or memory protocol to be made by means of the SLICC code without modifying anything in the implemented C++ classes.

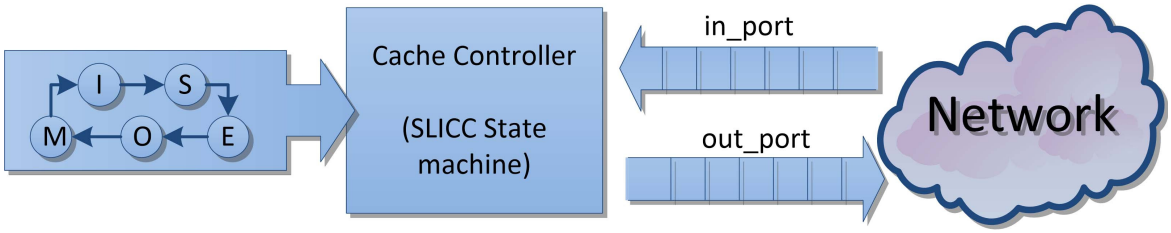


Fig. 3.2 High level view of the connections between the SLICC state machine and network in/out ports.

This code, **SLICC** (*Specification Language for Implementing Cache Coherence*), is a domain specific language that is used for specifying cache coherence protocols. In essence, a cache coherence protocol behaves like a state machine and SLICC is used for specifying the behavior of the state machine. Since the aim is to model the hardware as closely as possible, SLICC allows the user to impose specifiable constraints on the state machines. For example, SLICC can impose restrictions on the number of transitions that can take place in a single cycle. Apart from protocol specification, SLICC also combines some of the components in the memory model. As can be seen in Figure 3.2, the state machine takes its input from the input ports of the interconnection network and queues the output at the output ports of the network, thus, tying together the cache / memory controllers with the interconnection network itself. The official release version of gem5 already has 6 full functional protocols implemented in SLICC (MI example, MESI CMP directory, MOESI CMP directory, MOESI CMP token, MOESI hammer, and Network test). We have focused on the MOESI CMP directory protocol, which is probably the most complete. Thus, this is the protocol we have modified to add prefetching support. Full details on this protocol and the others implemented can be found in [25].

In the Ruby module of the implemented C++ classes, four independent components can be distinguished: the Sequencer, the Cache Memory structure, the Cache Replacement Policies, and the Memory Controller. **The Sequencer** class is responsible for feeding the memory subsystem (including the caches and the off-chip memory) with load/store/atomic memory requests from the processor. Every memory request, when completed by the memory subsystem, also sends back the response to the processor via the Sequencer. There is one Sequencer for each hardware thread (or core) simulated in the system. The **Cache Memory** can model set-associative cache structures of variable size, associativity, and replacement policies. L1, L2, and L3 caches in the system (where applicable) are instances of Cache Memory. The **Cache Replacement** policies are kept modular, separate from the Cache Memory, meaning that different instances of Cache Memory may use different replacement policies. Currently, the release version has two replacement polices (LRU and Pseudo-LRU).

The Memory Controller is responsible for simulating and servicing any request that misses on any of the on-chip caches of the simulated system. The **Memory Controller** is currently simple, but models DRAM bank contention and DRAM refresh faithfully. It also models a close-page policy for the DRAM buffers.

3.2.3 Interconnection network

As with the previous modules, in the network module, several configurations can be chosen. As can be seen in Figure 3.1, two network options can be selected: (1) a simple network or (2) GARNET (a detailed network). Both of them allow the user to model any network topology. Some of the most common network topologies are already implemented, but the user can easily model any new topology using the network files. The main difference between the two options is again the level of detail. GARNET, in addition to the capabilities of the simple network, can accurately model contention and router resource utilization, and can generate statistics on the power consumption in the network, among other things. In the detailed GARNET configuration, each router in the network works as a 5-stage pipelined virtual channel router, fully modifiable by the user.

Figure 3.3 shows the elements of this typical virtual channel router and Figure 3.4 shows the common 5-stage pipeline stages which the routing process is divided. The most important logics from the router are the following ones:

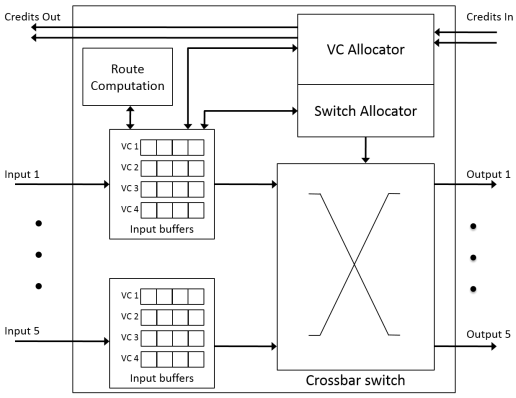


Fig. 3.3 Design of the 5-stage virtual channel router implemented in gem5.

Fig. 3.4 Pipeline stages for the gem5 virtual channel router.

- **The input units:** The most important elements that compose these units, are the input port and the Virtual Channels (VC). The input port is connected to the output ports of other routers or tho the Network Interface (NI) of the cache controllers. And the VCs buffer the flits that are waiting to be processed.
- **Route computation logic:** Whenever a flit is injected in the router, this logic uses its routing tables to calculate the output port of this flit.
- **VC and switch allocator logic:** These logic modules uses the prioritization policies to select the flits that will traverse the router each cycle.
- **Crossbar Switch:** Each cycle this switch connects the input units selected by the VC and switch allocator logic to the output ports.
- **Output ports:** The output ports are the links that connect this router to the input units of the neighbor routers.

Moreover, each of these units are directly related to one of the pipeline stages. In the next points we explain the behavior in these stages:

- **Buffer Write and Route Compute (BW/RC):** In this stage, the incoming flit is buffered to the determined VC and the route computation logic computes its output port.
- **VC Allocation (VA):** In this stage, the buffered flits in the VCs, allocate for the VCs in the next routers. This process is done in two steps:
    1. Each of the input VCs is associated to an input VC in the next router. To build this association, the input VC in the next router must have credits enough to fit the flit that is going to be send. If more than one channel is available for this flit, the prioritization policy decides the VC.
    2. The VCs in the next routers with more than one associated input VC, breaks conflicts through the prioritization policy.
- **Switch Allocation (SA):** In this stage, the router will decide for each output port, which input unit is going to use it. As in the previous stage, this process is done in two steps:
    1. Firstly the prioritization policy will chose, for each input unit which of its VC is going to be candidate to traverse the crossbar switch. To be candidate, the VC

must have at least one flit to send and this flit must have allocation in one VC in the input unit of the next router (resolved in the previous stage).

2. As the result of the previous step, an output port can be associated to one or more candidates. In this step, the prioritization policy will break conflicts for each output port.

- **Switch Traversal (ST):** In this stage, the selected flits will traverse the switch crossbar.
- **Link Traversal (LT):** In this stage, the flits coming from the crossbar will traverse the links to reach the next routers.

3.3 Converting gem5 to a prefetching-aware simulator

In order to convert gem5 to a prefetching-aware simulator, some new modules needed to be added to the simulator, as well as making some changes to the current files. Figure 3.5 gives a schematic representation of the modified (blue) and new (green) modules added to Ruby. Almost all the modifications that we have carried out on the system are focused on Ruby. Although Ruby has been modified specifically to work with gem5, some other simulators have mechanisms which also make use of it. This means that any other simulator which is able to use Ruby will be able to take advantage of the modifications presented here. The prefetching module has been included as a protocol-independent component. For this reason, a new simulation object (SimObject according to the gem5 nomenclature) was added as a wrapper between the cache controller and the prefetcher. It is important to note that the prefetching module can be used in conjunction with any protocol and that it is able to interact with the memory hierarchy at L1, L2, or both of them at the same time. Also, as stated earlier, we focused our attention on modifying the MOESI CMP directory. Therefore, protocols other than this one will need to be modified accordingly in order to work with the prefetching module.

3.3.1 New SimObject

As shown in Figure 3.5, the SimObject acts as a wrapper between the simulator and the prefetcher. The SimObject is required to enable the cache controller to make use of the prefetcher. Furthermore, as we want to make an adaptive prefetching module where any prefetching engine can be attached, we have built another abstract class which is explained in Section 3.3.2. This abstract class is responsible for holding the prefetch engine activity. The prefetch SimObject has the following responsibilities:

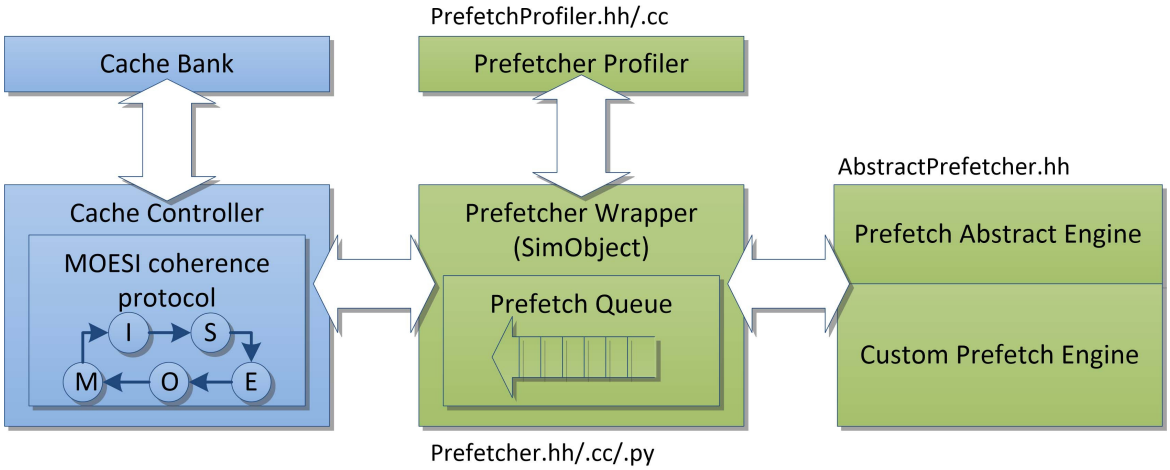


Fig. 3.5 Modified and new elements added to the simulator for prefetching support.

1. **Creation of the prefetch engine**: Every SimObject has restrictions that must be respected when implementing a new instance of it. The most important of these are that: (1) SimObjects must be created by means of a concrete function (`SimObject::create()`) and (2) they must be initialized after being created by means of another concrete function (`SimObject::init(const Params *p)`). During the initialization period, the prefetch SimObject will receive, through the command-line, the options that will determine which prefetch engine the system will use and the parameters that specify how this prefetching engine will behave.

2. **Communication between the prefetch engine and the cache controller**: The main requirement for the prefetcher to be able to make its predictions is for it to be aware of the activity in the cache module. The prefetching module has to track three different events: (1) read access, (2) eviction, and (3) insertion. Any time one of these events occurs, the wrapper must communicate the corresponding information to the prefetcher. The modifications needed to ensure this have to be made to the cache controller and these will be explained in Section 3.3.3. However, the wrapper must implement the functions that will be required by the cache controller. These functions will control the incoming message and will prepare it for communication and reception by the prefetching engine.

3. **Management of the prefetching queue**:The prefetch SimObject is where the prefetch queue resides. This queue holds the prefetching requests already generated before they are actually issued to the memory system. It works as a simulation structure, but can also be used to emulate the restrictions existing in a real hardware prefetch engine. Whenever the prefetch engine generates a new request, it is stored in the prefetch

queue. The cache controller will be responsible for picking up this request and issuing it. Furthermore, when a prefetch request is generated, a ready bit is also activated with a certain delay (depending on the delay of the prefetcher). If the request is in the queue but the ready bit is not activated, the cache controller will not be able to process the request.

4. **Collecting the prefetching statistics**: The aim is to use the wrapper as a centralized point to collate the statistics from the prefetcher. To this end, the wrapper must have a list of the memory blocks in the cache memory. This list is used to identify which blocks have been carried to the cache because of a prefetch request or a demand request. This list makes it possible to determine, for example, when a prefetch has been useful or not. More information about the statistics can be found in Section 3.3.4.

3.3.2 Abstract prefetcher and custom prefetch engines

On the right of Figure 3.5, the custom prefetch engine class can be seen, which inherits from an abstract class: `AbstractPrefetcher`. We have made this structure to make it easier to add new prefetching engines. The abstract class contains the declaration of all the functions that must be defined by the custom class. These functions must be defined in order to ensure correct communication between the cache controller and the engine. All these functions are defined as virtual functions, so the compiler will check that all of them are defined in the custom class; otherwise, the compilation will fail. These functions are detailed in [80].

On the other hand, the `AbstractPrefetch` class already implements the function to generate and enqueue prefetch operations to the prefetching queue. Therefore, users may employ the currently implemented function or, if there is any special behavior that the prefetcher must have when enqueuing the prefetch requests, they may redefine its behavior in the custom prefetch engine class. This function, `pushPrefetch(addr a)`, in its current implementation, checks if the request fits in the queue and enqueues the address to the prefetch queue. If the queue is full, the oldest address is erased to fit the new one in, creating a circular queue. Finally, if the address generated by the prefetch engine is already in the queue, the address is discarded (by performing a certain amount of filtering).

As we have said, almost any prefetcher can be used as a custom prefetcher in this scheme. We have implemented three prefetching engines in this thesis. To do so, we have done back engineering from the papers that described these techniques. However, there are some implementation details that are not clear in the papers, but we needed to specify to make them work. In the following points, we explain these details as well as its storage cost.

Tagged prefetcher

This prefetch unit is based on adding a bit (initially set to 0) at every cache line. In every hit in the cache, the tagged bit is examined. If it is set to 0, the bit is set to 1 and the next line is prefetched, otherwise nothing is done.

On the other side, if the demand operation has produced a miss in the cache, when the new line is placed on the cache, its tagged bit is set to 1 and a prefetch operation to the next line is launched.

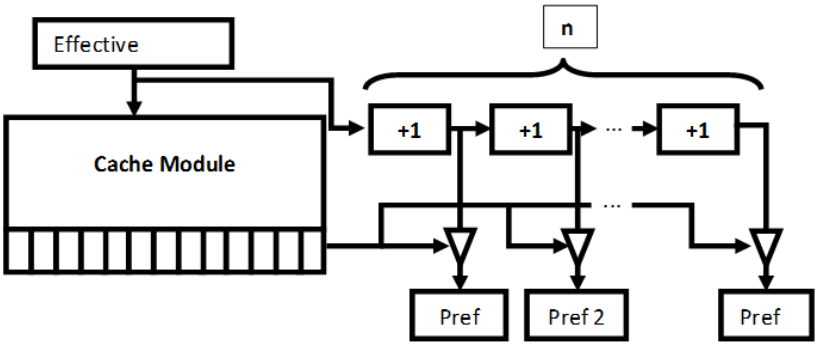


Fig. 3.6 Tagged prefetcher schema.

Moreover, the only storage required for this prefetcher is a bit for each cache line. So the size (in bits) is going to be equal to the number of cache lines in the cache level it is implemented.

RPT prefetcher

In order to decide if a memory access generates a prefetch request, this engine keeps a table indexed by the PC of the memory instructions. Each time that there is a cache miss (or a first access to a prefetched line) the table is accessed and, based on the memory address referenced by the current memory instruction (current address) and the information held on the corresponding table entry, the entry is updated appropriately and a prefetch request may be generated. This table contains the following fields: PC (used to index the table), Previous address (memory address accessed by the memory operation the previous time that the table was accessed), Stride (predicted stride), State (state of this entry as we will describe later), and LRU (information used for replacing entries on the table).

The particularity of the implementation of this prefetcher is found on the evaluation of the prediction. This will be done considering that, if $current\ address = previous\ address + stride$, then this is a good prediction. If not, the prediction is wrong. Depending on the current state for the PC entry and if the prediction is right or wrong, the state will remain the

same or change to a different one. The state machine that rules the behavior of the prefetch is described in Figure 3.7 and commented below.

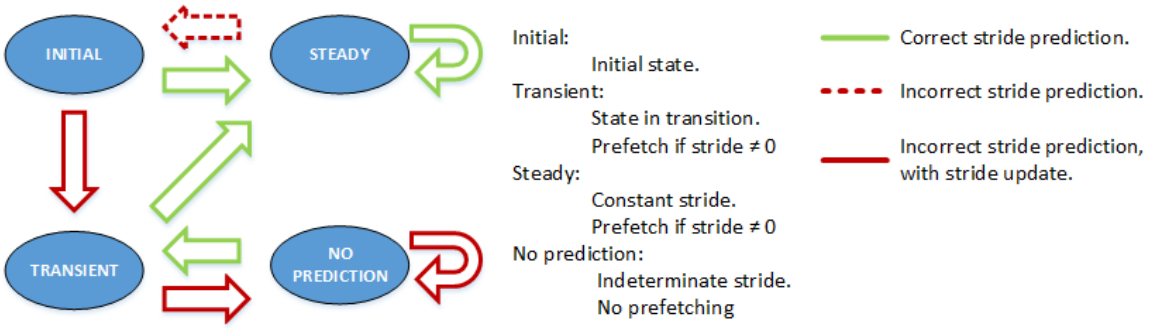


Fig. 3.7 State transition graph for reference prediction table entries.

- The starting state is “initial”. When a memory instruction PC that is not already included in the prediction table is encountered, a new entry on the RPT is created for that PC (substituting an old one following the LRU algorithm if required). The previous address field is set to the current address of the memory operation, the stride is set to 1, and the state is set to "initial". If we are already in the "initial" state and the prediction is right, we move to the "steady" state launching a prefetch operation, if its wrong, the entry moves to "transient" state and the stride is updated: $New\ stride = Current\ Address - Previous\ Address$.

- In "steady" state, every time that the prediction is right, a prefetching operation is generated. If the prefetching operation is wrong, we come back to the initial state, but the stride is not updated.

- In the "transient" state, we move to "Steady" state with a right prediction and to "no prediction" with a wrong one (updating the stride).

- In the "no prediction" state, the prefetcher will keep updating the stride until a good stride is detected in this case the state will be changed to transient.

In terms of memory structures requirements or storage, the prefetcher only makes use of the reference prediction table, so we consider the size of this table as the size of the prefetcher. Each table entry size (for a 32 bit processor) is:

PC	Previous address	Stride	State	LRU
32 bits	28 bits	64 bits	2 bits	6 bits

Note that for *previous address*, we only require the address of the cache line and not the full memory address. This field size depends on the line size of the cache level. In all our configurations, the RPT is running in cache with cache line size of 64 bytes. This means that:

$$Previous\ address\ field\ size = address\ size - bloc\ id\ size = 32 - 6 = 28\ bits$$

Finally, we have to multiply the size of each table entry for the number of entries of the RPT. In our tests we have used a 256 entry RPT, for this reason its size will be:

$$RPT\ Size = RPT\ entries * 132\ bits = 256 * 132 = 8192\ bits = 1\ KByte$$

GHB prefetcher

To implement this mechanism, first of all, we decided which technique we wanted to use using the global history buffer. We actually chose to implement the most generic one (distance prefetching), because it is the one that provides the best performance according to the authors of the GHB [59].

The implementation makes use of two different structures: the Index Table (IT) and the Global History Buffer (GHB). The entries in these tables depend on the way we implement the prefetching algorithm.

The main idea of this prefetching mechanism is to catch the correlation between misses, in order to predict which is going to be the next cache miss and prefetch it before it occurs. So this prefetcher starts working with the first cache miss. In the first cache miss, we just push back the miss address in the global history buffer, after that, and for every other miss, we subtract from the new miss address, the last miss address (that it is the last entry in the GHB) and then we have a Delta, once we have this delta, we look for it in the Index Table, if we do not find it, we make a new entry in the IT (applying LRU replacing policy). If we find it, we look at the GHB entry related to this IT entry and we start, first the depth exploration and then the width exploration. Subtracting the GHB entries, we find the deltas, which are added to the new miss address in order to create the prefetching operations.

To be more accurate, we are going to explain more in detail each step:

1. Once we have found the entry in the IT, we get the related entry in the GHB.

2. We look “depth” times to the next entries getting new deltas. For example with depth set to 2 and supposing that we are in the n position of the GHB, the new deltas will

be: $GHB[n] - GHB[n+1] and GHB[n+1] - GHB[n+2]$. Then, we add the obtained deltas to the miss address to get a new prefetching address for each delta.

3. Once we have looked in "depth", we must look in "width", so we go to the previous entry in the GHB related with the current one and we start a new "depth" exploration to get some new deltas and prefetch them.

4. We have already generated all the prefetching operations, now we must add the new miss address to the GHB, so we push it back, we connect the new entry with the previous entry in the GHB (the one linked with the delta found in the IT) and we connect the delta in the IT with this new entry.

The IT is implemented as a totally associative cache with LRU replacement; each entry has two fields, the delta, which is the key of the table, and the index of the related entry in the GHB.

The Global History Buffer is a circular buffer using FIFO replacement. Note that, on the original GHB paper, there is no extra behavior associated with the replacement of old entries in the table by new ones.

The size computation for this prefetcher consists on getting the size of each type of table entry and multiply it by the number of entries in that table. For the IT:

Delta	**GHB entry**
28 bits	$log_2(num\ entries\ GHB) bits$

And for the GHB:

Miss Address	**Previous**
28 bits	$log_2(num\ entries\ GHB) bits$

So, its size will be:

$$\begin{aligned} GHB\ Size = &(28 + log_2(num\ entries\ GHB)) * (num\ entries\ IT) + \\ &(28 + log_2(num\ entries\ GHB)) * (num\ entries\ GHB) = \\ &(28 + log_2 1024) * 256 + (28 + log_2 1024) * 1024 = \\ &(38 * 256) + (38 * 1024) = 6080\ bytes \sim 6\ Kbytes \end{aligned}$$

3.3.3 Modifications in the cache controller and the coherence protocol

Although the behavior of a prefetch operation is similar to a regular load operation, there are significant differences that must be properly considered by the memory coherence protocol.

However, there is little information available on how the MOESI coherence protocol behaves in the presence of a prefetcher and the details are not widely known. Therefore, in this section, we present the implementation that we have devised. Our solution requires, as we will see, the addition of several new states and transitions. Another issue to take into account is that, in Ruby, the cache coherence protocol is defined in combination with the cache controller: they are defined together as a state machine using the SLICC language. Apart from the implications this has for design, this means that it is not possible to discuss each component separately. However, for the sake of clarity, we have tried to describe the general modifications required for each of them in the following subsections. For more details on the actual implementation, see [80].

Changes done to the cache controller

The cache controller is responsible for checking whether there are pending prefetch requests in the prefetching queue and arbitrating, if necessary, between these pending requests and the regular requests arriving at each memory cache. For each private memory cache in the system, whenever there is no request from the core to process, the cache controller will check if there are ready requests in the prefetch queue. If so, the oldest of these prefetch operations will be issued. If the coherence state for the requested cache line is already valid, the request is discarded. Otherwise, the request will trigger a *Prefetch* event in the coherence protocol (as we will see in next section) and will bring the data from upper levels of the memory hierarchy or another L1 private cache if available. The behavior of the cache controller for the shared caches is similar to that described in the case of the private ones. Priority is given to regular demand requests and when a prefetch request is issued, if the state for the cache line is invalid (i.e. not present in the shared cache nor in any private cache), a *Prefetch* event is triggered; otherwise, the request is discarded. However, the caveat here is that the prefetching engine that is running in the current tile may generate and queue requests that have to be processed by another tile, i.e. the cache line address does not correspond to the current tile. For this reason, when the prefetch operation is issued in the shared memory, before any coherence state is checked, a check will be made to ensure that this is the owner tile. If it is not, the request will be forwarded to the corresponding tile. Once in the corresponding tile, it will be queued as a regular request but treated as a prefetch from the coherence protocol perspective. It should be noted, however, that a prefetch request generated in a private cache level will be considered as a completely regular load request in the upper levels of the hierarchy.

Changes made to the MOESI coherence protocol

As was said earlier, our prefetching framework is able to work in both the private cache (L1) or in the shared cache memory (L2). However, the behavior of the coherence protocol is a little different in each type of cache. Before describing the required modifications, what follows is a description of the original behavior of a memory load that misses in each of the two cache memory levels that we consider here.

Original MOESI protocol

When a load reaches the private cache memory, the controller will check whether it is in this cache and if it is in a valid state. If it is not, a new request is propagated to a higher cache level and the cache line state is changed to the P (prefetched) transient state while it is waiting for the data. When the data arrives and is stored in the cache module, the requesting core is notified and the state of the cache line changes to a new stable state. This state will be shared (if other private caches share the data) or exclusive (if not). This behavior is shown in Figure 3.8.

By the side of the shared memory, it is important to remember that, every shared memory fragment is responsible for just a part of the memory space which stores the coherence information of each of the lines present in this shared fragment and also the lines in the private memories that corresponds to the memory space of that shared memory fragment. When a load operation reaches the shared memory, and the state of the requested cache line is invalid (and also invalid in all the private caches), the request is propagated to a higher memory level in the hierarchy and the state of the cache line is changed to a transient state. However, as the shared memory that we are working on is non-inclusive, a load operation does not store any data in the shared memory. For this reason, when the data reaches the shared memory, it will be sent directly to the private memory which has requested it. When the data arrives and is stored in this cache, the state of the memory block in the shared cache changes to a stable state, reflecting the fact that the block is invalid in the shared memory, but valid and shared or exclusive in a local cache. Figure 3.9 shows a summarized version of the state machine that defines the behavior of the coherence protocol in the shared cache level.

Prefetching-aware MOESI protocol

In the private cache, the behavior of a prefetch operation is very similar to a regular load. The main difference is that the core is not notified of the data arrival when the cache block is finally stored in the memory. The reason for this is that, whereas the system requires a response from a load operation, it will not wait for any response in the case of a prefetch

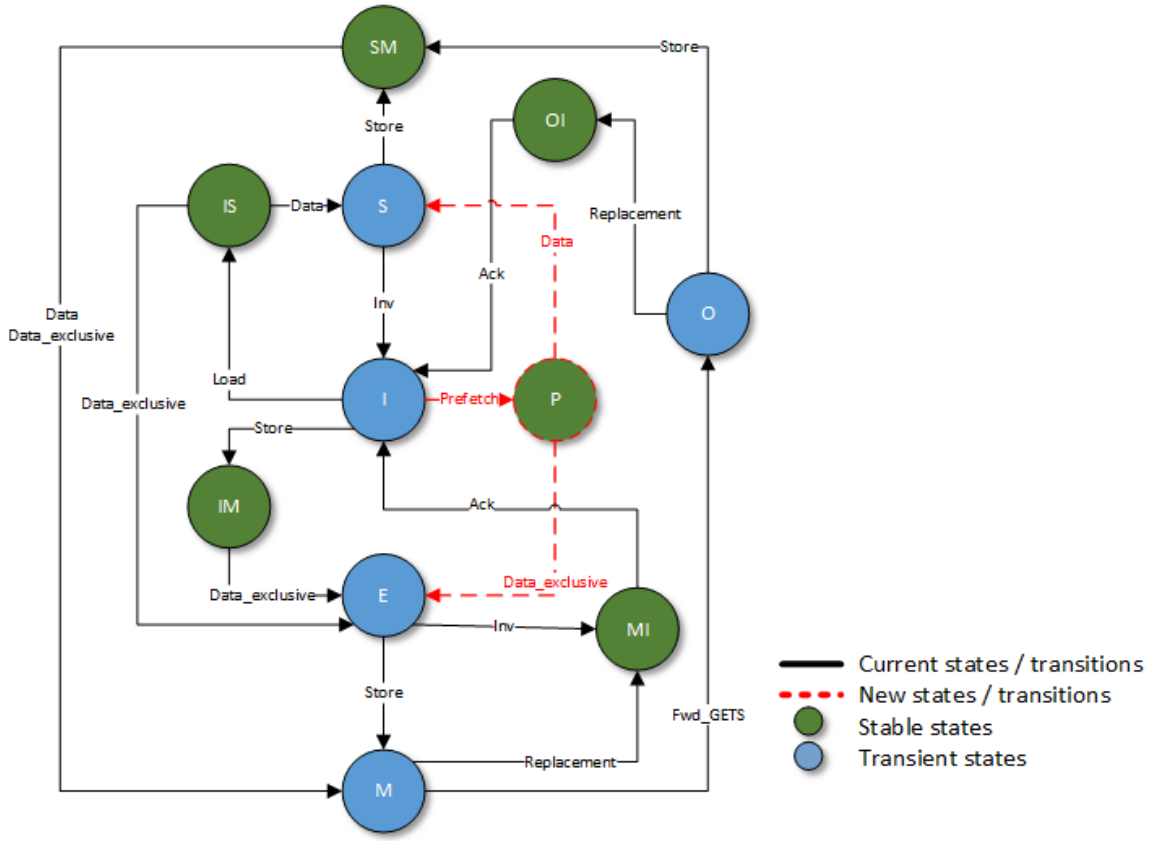


Fig. 3.8 Modifications done in the L1 state machine that defines the MOESI CMP directory protocol. Stable states detailed in Table 3.1.

States	Description
M	The cache block is held exclusively by this node and is potentially modified.
O	The cache block is owned by this node. It has not been modified by this node. No other node holds this block in exclusive mode, but sharers potentially exist.
E	The cache block is held in exclusive mode, but not written to.
S	The cache block is held in shared state by 1 or more nodes. Stores are not allowed in this state.
I	The cache block is invalid.

Table 3.1 Stable states of the private cache state machine.

operation. In order to accommodate this slightly different behavior, we have replicated part of the state machine. A new state has been added P (Prefetched) that is reached by replicating the *Load* transitions for the prefetching side and changing the event for a *Prefetch* event. This state will be active while a prefetching operation is in flight. Figure 3.8 shows the new state and transitions added to the state machine. Note that, when a *Prefetch* event arrives to the memory and the cache line is invalid or not present, the request is propagated to an

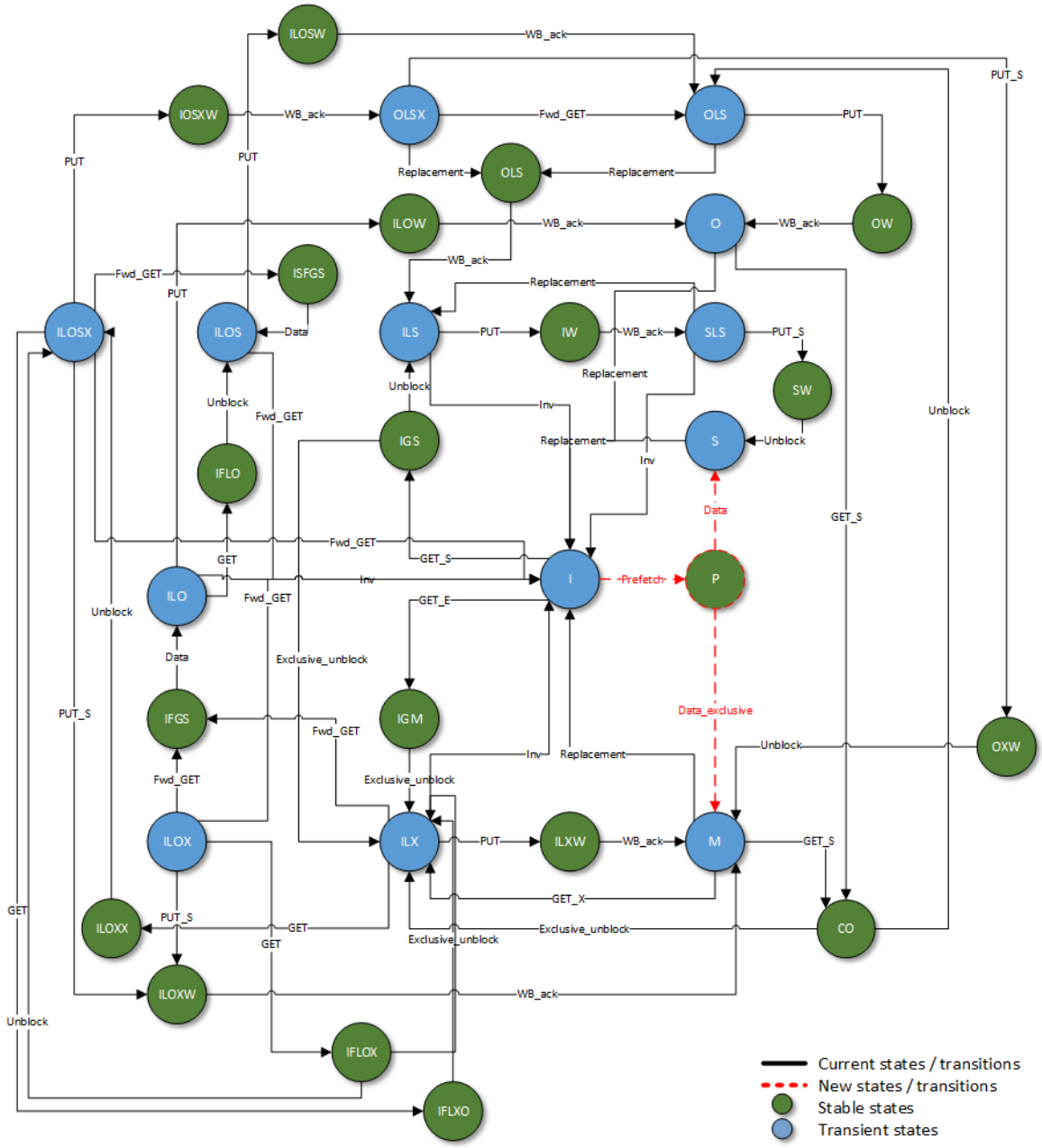


Fig. 3.9 Modifications done in the L2 state machine that defines the MOESI CMP directory protocol. Stable states detailed in Table 3.2.

upper cache level, and the status changes to P (Prefetched). When the data arrives, if the data is shared with other nodes, the *Data* event is processed and the S (Shared) state is reached. However, if there are no other sharers, a *Data_Exclusive* event is processed and the cache line moves to E (Exclusive) state.

Intra-chip Inclusion	Inter-chip Exclusion	States	Description
Not in any L1 or L2 at this chip	May be present at other chips	I	The cache block at this chip is invalid.
Not in L2, but in 1 or more L1s at this chip	May be present at other chips	ILS	The cache block is not present at L2 on this chip. It is shared locally by L1 nodes in this chip.
		ILO	The cache block is not in L2 on this chip. Some L1 node in this chip is an owner of this cache block.
		ILOS	The cache block is not in L2 on this chip. Some L1 node in this chip is an owner of this cache block. There are also L1 sharers of this cache block in this chip.
	Not present at any other chip	ILX	The cache block is not in L2 on this chip. It is held in exclusive mode by some L1 node in this chip.
		ILOX	The cache block is not in L2 on this chip. It is held exclusively by this chip and some L1 node in this chip is an owner of the block.
		ILOSX	The cache block is not in L2 on this chip. It is held exclusively by this chip. Some L1 node in this chip is an owner of the block. There are also L1 sharers of this cache block in this chip.
In L2, but not in any L1 at this chip	May be present at other chips	S	The cache block is not in L1 on this chip. It is held in shared mode at L2 on this chip and is also potentially shared across chips.
		O	The cache block is not in L1 on this chip. It is held in owned mode at L2 on this chip. It is also potentially shared across chips.
	Not present at any other chip	M	The cache block is not in L1 on this chip. It is present at L2 on this chip and is potentially modified.
Both in L2, and 1 or more L1s at this chip	May be present at other chips	SLS	The cache block is in L2 in shared mode on this chip. There exists local L1 sharers of the block on this chip. It is also potentially shared across chips.
		OLS	The cache block is in L2 in owned mode on this chip. There exists local L1 sharers of the block on this chip. It is also potentially shared across chips.
	Not present at any other chip	OLSX	The cache block is in L2 in owned mode on this chip. There exists local L1 sharers of the block on this chip. It is held exclusively by this chip.

Table 3.2 Stable states of the shared cache state machine.

Note that, we have added a transient state in the protocol as well as a new transition. In order to simplify the diagram of the protocol shown in Figure 3.8, we have only shown the main transitions to the states (we have not drown all the loop transitions). However, note that the *Prefetch* event can interact with any of the states of the protocol. However, if this event is triggered and the state of the cache line is not invalid, it means that the data is already in the cache. For this reason, the prefetch request will be simply discarded and the prefetch event will behave as a loop event. On the other side, the interaction of the other events with the P (prefetched) state works in a similar way. In the definition of the MOESI protocol, the transient states are only prepared to receive a few messages (the ones that allow to finish the action that they were invoked for). If another event interacts with the cache line that is in a transient state, this will wait until the action is finished to be processed. For example, if a *Load* event is triggered to a line that is in P state, the Load will wait until the Data arrives to the cache and the state of the line changes to S (Shared) or E (Exclusive).

In the shared cache, the required behavior of a prefetch operation is slightly different from the behavior of a regular load. Unlike the regular Load, the prefetch operation actually stores the requested data in the shared memory instead of sending it to a private cache. Therefore, we have added new states and transitions to the shared cache protocol that are similar to those employed in the private cache. It is worth noting that, as Figure 3.9 shows, when *L1_GETS* is received, the resulting state is ILS (Invalid in L2 and Shared in L1) or ILX if the data is sent in an exclusive mode, meaning that the data is not present in the L2. However, when the new *Prefetch* event is triggered, the resulting state is S (Shared) or M (Modified) if the data is sent in an exclusive mode, meaning that the data is physically stored in the cache. As in the private cache, the *Prefetch* event may interact with any other state. In this case, the request will be discarded. The new event, P (Prefetched) is a transient state that will stall the messages until it receives the *Data*. After that, the new state for the cache line will be stable and will process the stalled events (if there are). More detail about the states in the protocol can be found in [26] .

3.3.4 Prefetching statistics

Statistics in gem5 are collated in a particular way. Any SimObject collating statistics may have a profiling class associated with it. We have also implemented a profiling class for the prefetcher. This class is attached directly to the prefetching SimObject (see Figure 3.5), so statistics are taken directly from the prefetcher activity. An initial set of prefetching statistics has been implemented. These statistics make it possible to classify the success of prefetching requests. Statistics measuring the time that prefetching requests take in the memory hierarchy have also been added.

Any new statistic can be easily included in the profiling module of the prefetcher. To do this, a new variable, which will hold the statistical information, has to be declared in the profiling class. In the printing function, the variable has to be defined in such a way that the statistic is displayed. It is also necessary to implement the function for the updating of this statistic. The last step is to insert, in the prefetching SimObject, the function calls for the statistic update function (defined in the profiling module) in every location where this function is required, in order for the statistics to be generated from the data.

3.4 Performance study

In order to test our prefetching framework and highlight the importance of making an accurate evaluation of the network when studying prefetching-related techniques, we implemented the three prefetching engines mentioned earlier (Tagged prefetcher, RPT, and GHB). Each prefetching engine was implemented with a configurable degree of aggressiveness. The aim behind this was to be able to analyze how much traffic the prefetcher could inject into the network without degrading performance. It should be noted that the execution time shown by the gem5 simulator that incorporates our prefetching framework is subject to two main factors:

1. The impact of prefetching on the cache modules. By increasing the aggressiveness, the prefetcher increases the probability of generating useful prefetches, tending to reduce the miss ratio. However, the probability of evicting useful data from the cache also rises, tending to increase the miss ratio.

2. The impact of prefetching on the NoC. By increasing the aggressiveness, the prefetcher will increase congestion in the network. This can have a dramatic effect, because the prefetch operations flow through the network with all the other operations, potentially increasing global congestion. For this reason, injecting prefetch operations into the network increases the latency for all memory requests.

The following sections will illustrate the effect that these two factors have on the system. It is important to note that, in order to assess the second effect described above, the use of the framework developed in this study is essential.

3.4.1 Experimental framework

The system simulated in this performance evaluation is represented in Figure 3.10. It consists of a tiled mesh with private first level data and instruction caches, and a shared L2 cache.

The L2 cache is composed of several banks and each of these banks is associated with a local tile. Each memory address is deterministically associated with a given L2 location. This means that there is no replication in the L2 cache, although a given block can be in several L1 caches. L2 and L1 are non-inclusive. As stated before, the coherence protocol employed is the MOESI CMP directory. The hardware specifications are shown in Table 3.3. The benchmark suite in this performance study was the PARSEC 2.1 [6] benchmark suite.

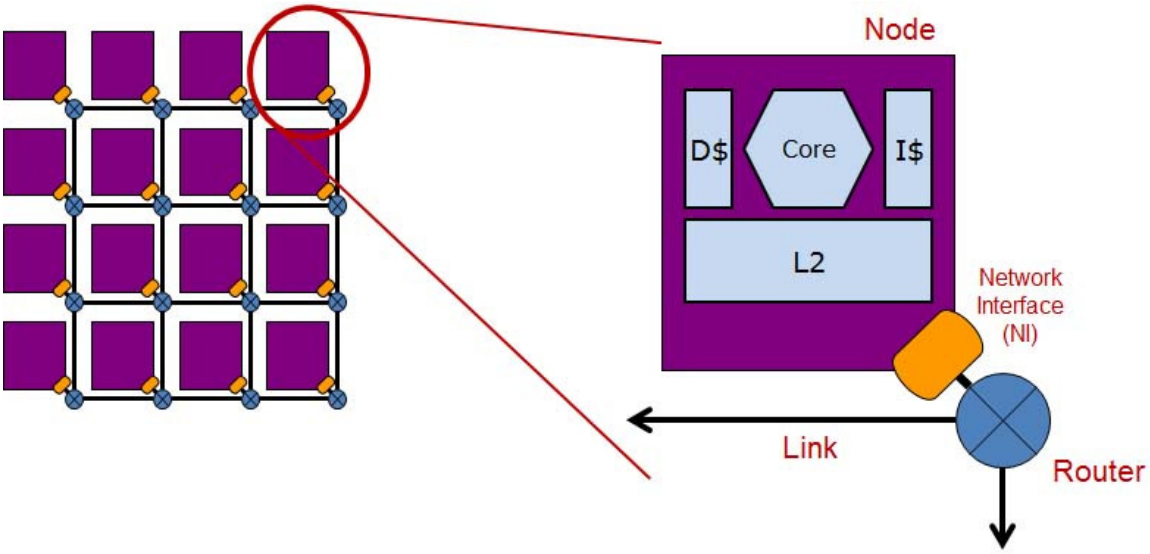


Fig. 3.10 Tile configuration of the simulated environment.

Moreover, to analyze the effect of the prefetcher in the network, we undertook two different studies varying the aggressiveness of the prefetchers and the number of available Virtual Channels (VCs). Varying the aggressiveness of the prefetcher, we sought to analyze the amount of prefetching requests that the prefetcher can inject into the network without congesting it. By varying the number of VCs, we studied what the behavior of the simulator would be when varying the network resources. In the first study, the prefetcher specifications were grouped according to the different levels of aggressiveness (low, medium, and high). More details of these configurations are shown in Table 3.4. This first experiment is also shown at two levels of detail: (1) a comparison of the averages obtained and (2) detailed per benchmark results for all the prefetch mechanisms. In the second study, we varied the use of 1, 2, and 4 VCs.

3.4.2 Aggressiveness analysis for all prefetchers: average results

In the following Figures we can see the behavioral statistics of the three prefetch engines for the different levels of aggressiveness using our modified version of gem5. The baseline for the

Hardware specifications	Values
ISA	x86
CPU model	TimingSimple
Tile number	16
L1 Data/Instruction cache	16Kb each tile
L2 size	8MB
Network	Garnet
Topology	Mesh
Virtual Networks	3
Virtual channels per virtual network	2
FlitBuffer size	6 flits
Bandwidth factor	16 bytes
Data paquet size	64 bytes
Ctrl paquet size	8 bytes
Prefetcher cache level	L1
Simulated cycles	350 millions of cycles

Table 3.3 Simulation environment specifications.

Pref	Param	Low Aggr	Medium Aggr	High Aggr
TAGGED	**AGGR**	1	2	4
RPT	**AGGR**	2	4	8
GHB	**WIDTH**	2	4	8
	DEPTH	2	4	8

Table 3.4 Detailed definition of the level of aggressiveness for each prefetch mechanism.

experiments was the same system with no prefetching mechanism. Figure 3.11 displays the behavior of the memory module, showing the MPKI (number of Misses Per Kilo-Instruction) in each scenario. This metric shows the average number of L1 misses every 1000 instructions. In a simulator in which the effect of the prefetcher is only represented in the memory model, variation of the MPKI from the baseline configuration would proportionally modify the speedup in the experiment. However, this is not necessarily the case if the NoC effect is also considered. As can be seen, in most cases, as the aggressiveness of the prefetcher increases, there are more chances of bringing in useful data, thus reducing the MPKI. GHB and RPT show this behavior, although increasing the aggressiveness of RPT from medium to high does not provide more benefit. However, in the case of the Tagged prefetcher, when the aggressiveness of the prefetcher is increased, the MPKI also increases.

Figure 3.12 provides a detailed illustration of this, showing the number of prefetch operations issued per kilo-instruction, categorized using the statistics generated from the

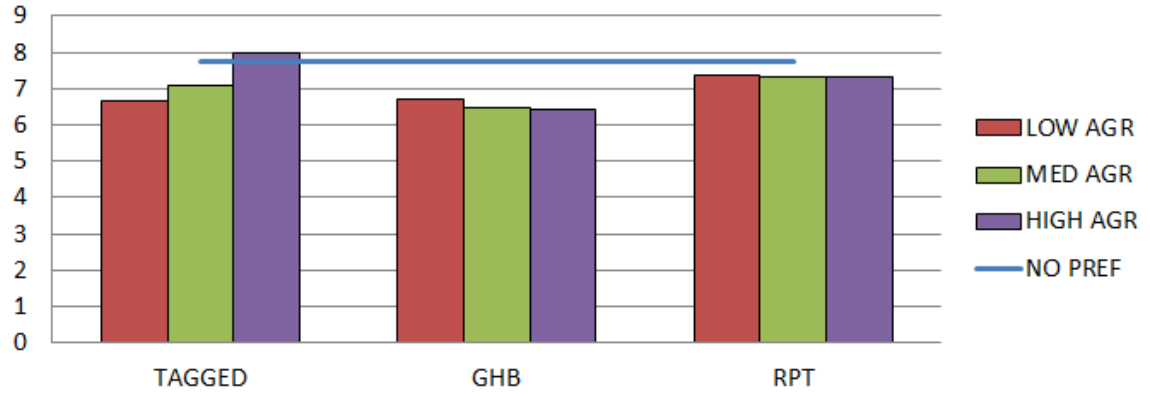


Fig. 3.11 Average MPKI calculated with the gem5 simulator and the prefetching module.

prefetcher-profiling module described in Section 3.3.4. While the Tagged prefetcher increases the number of prefetching operations issued in proportion with the increase in aggressiveness, the RPT and the GHB prefetchers do not. The reason for this is that the Tagged prefetcher works in a more simplistic way. As the Tagged prefetcher with medium or high aggressiveness issues a lot of prefetch operations and the accuracy is low, the cache becomes filled with a lot of useless data – pollution – which evicts useful data. This effect leads to an increase in the number of misses for this cache level.

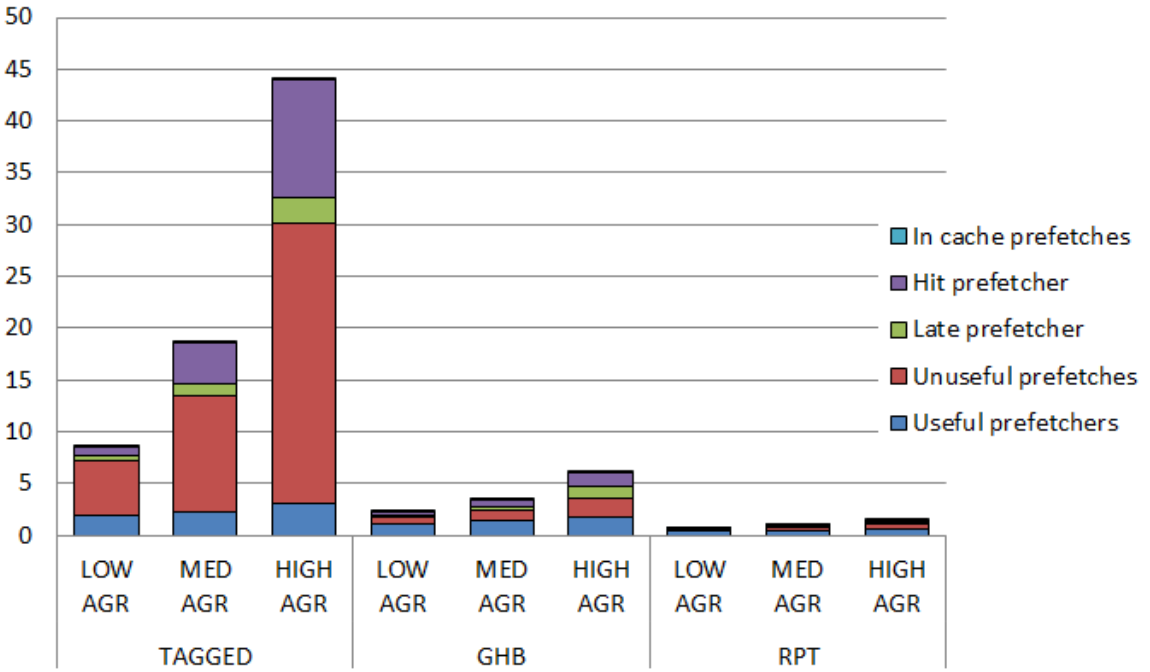


Fig. 3.12 Average prefetch request distribution.

Figure 3.13 shows the consequences of this increase in requests on the NoC. This figure shows the average latency (in cycles) for a miss in L1. If we compare Figure 3.12 with Figure 3.13, it is easy to see a strong correlation between the increase in the number of operations issued by the prefetchers and the increase in memory latency. The higher the level of aggressiveness, the more the L1 misses latency increases. An explanation for this may be that the higher the memory requests, the greater the number of conflicts in the buffers and the higher the waiting time in the routers. For this reason, memory requests spend more time traversing the network.

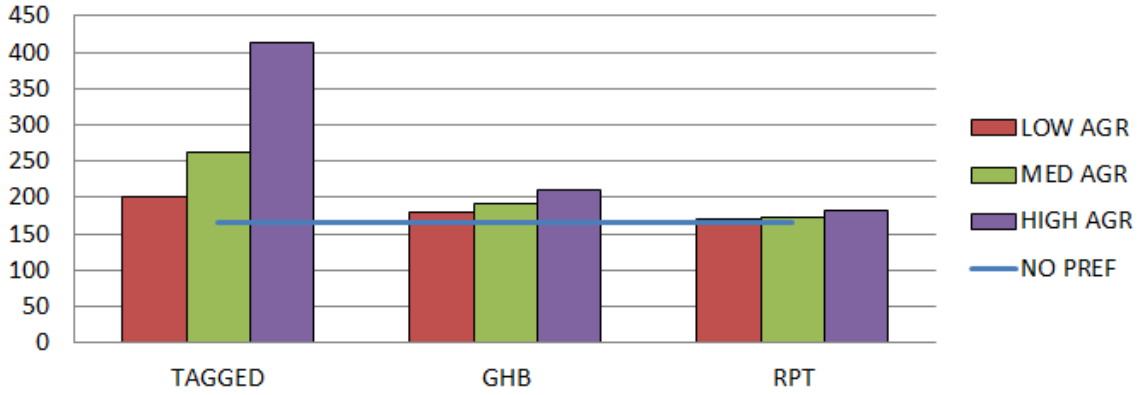


Fig. 3.13 Average L1 miss latency (cycles) calculated with the gem5 simulator and the prefetching module.

This information shows the benefits and the drawbacks of a prefetcher in a multi-core system. To show the combination of these benefits and drawbacks, Figure 3.14 shows the IPC speedup for each aggressiveness level. The IPCs were calculated by dividing the amount of x86 instructions executed by all the cores between the simulated cycles. It can be seen that there is no a direct link between the speedup and the MPKI. This conclusion is especially important as this would not be the case with other simulators which do not take the NoC into account. For example, the Tagged prefetcher with medium aggressiveness reduces the MPKI, yet it degrades performance in comparison with a system without prefetching. A conclusion to draw from these numbers is that performance studies using prefetching techniques (or any other mechanisms that affect the traffic in the network) which do not take the NoC effect into account, may make erroneous conclusions. For example, if one were to select the best mechanism and aggressiveness for the framework described in this study, taking into consideration only the results from the memory hierarchy, using the MPKI, the wrong choice would be made. It is important to note that, as one can see in Figure 3.11, the mechanism with the greatest reduction in MPKI is GHB with high aggressiveness. However, Figure 3.14 shows that the prefetch mechanism which provides the greatest speedup is GHB with low

aggressiveness. Thus, this shows that taking more than just the MPKI reduction into account ensures that more suitable and well-founded decisions can be made.

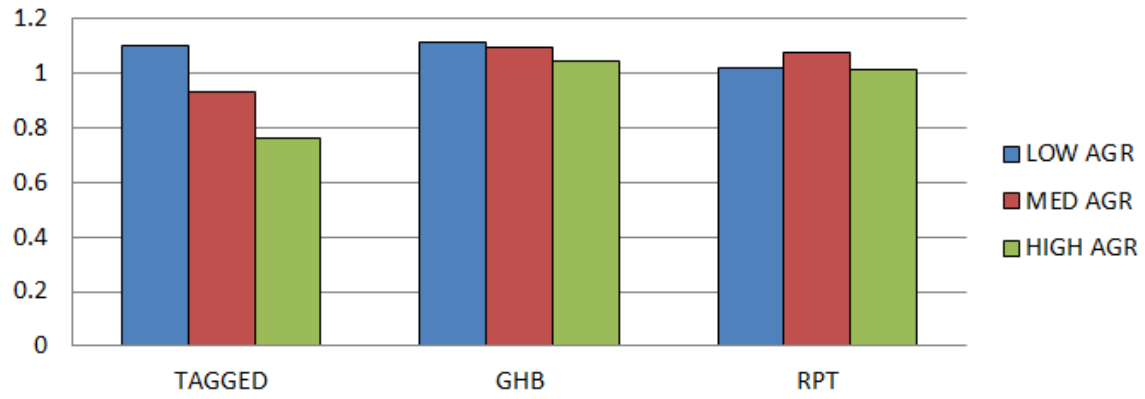


Fig. 3.14 Average IPC Speedup calculated with the gem5 simulator and the prefetching module.

The statistics provided by gem5 with the prefetching framework extension clearly show that the Tagged prefetcher works by means of brute force. As the aggressiveness increases, the number of generated requests grows by the same proportion. In contrast, the RPT prefetcher uses a very strict confidence strategy that greatly reduces the number of prefetch requests generated by the prefetcher. Even when the aggressiveness is increased in the same way as with the Tagged prefetcher, the number of requests generated by RPT with high aggressiveness is less than 50% of the number generated by the Tagged prefetcher at the lowest level of aggressiveness, and the percentage of useful prefetches is higher. This means that the RPT prefetcher is less aggressive and more accurate than the Tagged prefetcher. As mentioned earlier, GHB is a correlation prefetcher, which does not generate a large number of requests until the prefetcher has saved enough information to generate the requests. For this reason, the number of requests generated by GHB grows as the aggressiveness increases, but to a smaller degree than the Tagged prefetcher does. It is worth noting that GHB is a prefetch engine capable of capturing more complex memory patterns (at higher hardware and computational costs) and thus, the number of useful prefetches that it is able to produce is higher than for the other prefetchers.

To conclude this first analysis, it is important to highlight that the number of operations generated by the prefetcher has a direct impact on the speedup (e.g. MPKI is reduced). For this reason, the MPKI should not be the only statistic considered for the purpose of assessing performance: statistics concerning both speedup and MPKI should be considered at the same level of detail. In this way, we will be able to take better-informed decisions. Tools such

as the framework developed in this study are therefore of considerable usefulness for this purpose.

3.4.3 Aggressiveness analysis on Tagged: detailed results

The aim here is to provide detailed results for each benchmark instead of average values. This enables a more in-depth and detailed analysis of the the results. We will start with the Tagged prefetcher results and follow up with the RPT and GHB prefetchers.

As we can see in Figure 3.15, in most of the benchmarks, the number of misses every one thousand instructions decreases for low aggressiveness tests and increases when the aggressiveness is higher. Depending on the effectiveness of the prefetcher, the optimal aggressiveness can be low or medium, but there are several benchmarks which the Tagged prefetcher is not able to generate useful prefetches. This effect increases the number of mnisses due the polution injected in the cache. One of this examples is the canneal benchmark. In this case, almost all the prefetches generated by the engine are useless, what means that as more aggressiveness, higher will be the MPKI. Nevertheless, the swaptions benchmark works in opposite direction because its behavior is really easy to predict and Tagged prefetcher can do a great work on that. In average, the low aggressiveness is the aggressiveness that better manages to reduce the number of misses.

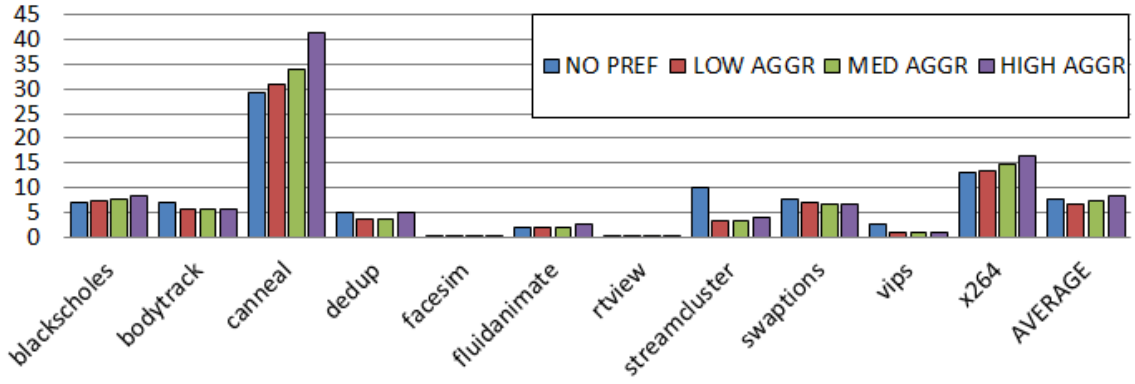


Fig. 3.15 Per benchmark MPKI results generated with the gem5 simulator and the prefetching module using the Tagged prefetcher.

As we have said before, the increment of injections of requests in the network entails to increase the congestion of the network. In almost all the benchmarks the latency is incremented in correlation with the aggressiveness. As well is shown in the average value. The only difference that can be appreciated is in the facesim benchmark which the latency is reduced with the higher aggressiveness. This is because in this benchmark there are

very few memory requests, thus the prefetcher activity is very low although in the higher aggressiveness. For this reason, the number of requests sent to the network is very low and the congestion is not increased.

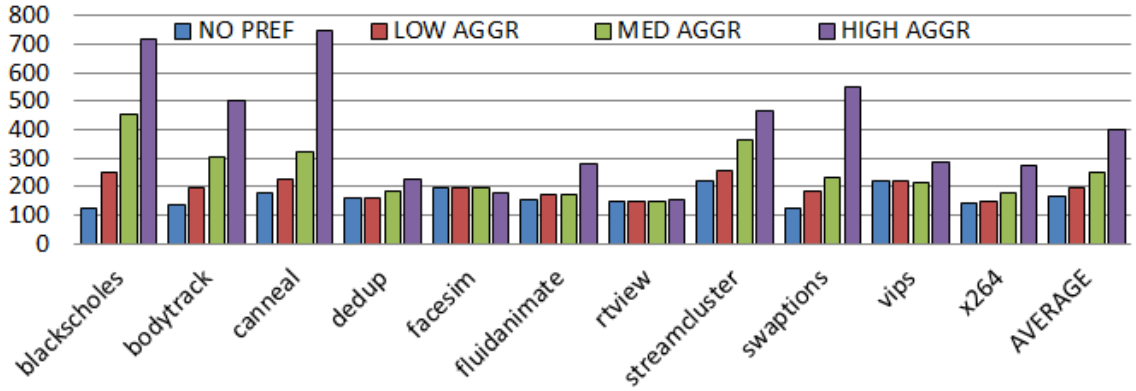


Fig. 3.16 Per benchmark average L1 miss latency (cycles) results generated with the gem5 simulator and the prefetching module using the Tagged prefetcher.

In Figure 3.17 we can see the IPC of the benchmarks without prefetching or with prefetching and their different aggressiveness. As we can see although some benchmarks such as blackscholes or bodytrack got a reduction on the MPKI, here we see that there is a downgrade in the performance. This is again because the MPKI reduction generates less speedup than the slowdown produced by the injection of prefetch requests in the network.In other benchmarks, as facesim, fluidanimete or rtview, the result is similar. Finally, streamcluster, vips or x264 gets some speedup. This is again because of the combination of the two variables that we have already commented. In average the low aggressiveness is the configuration that manages to improve the performance of the system in a higher degree. In this case, the result matches with the result of the MPKI analysis.

Figure 3.18 shows in stacked the behavior of the different prefetch requests. The sum of all of them represents the total of prefetch requests generated by the prefetcher. We can see that, as much we increase the aggressiveness, more useless requests we generate (which are the worst for the system). Moreover, the total number of requests increases in the same proportion that the aggressiveness do.

3.4.4 Aggressiveness analysis on RPT: detailed results

As we have seen in the average analysis, the RPT is the prefetcher that generates less activity from the three prefetchers that we have analyzed. In this detailed analysis we can see that

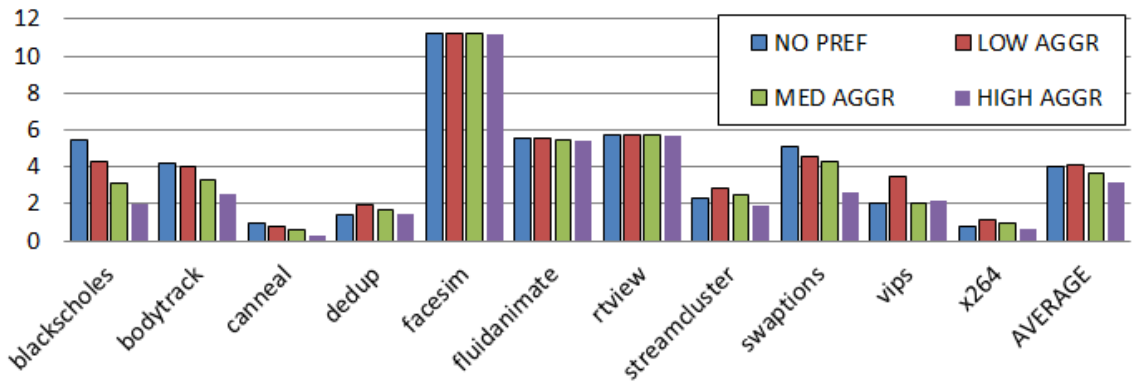


Fig. 3.17 Per benchmark IPC results generated with the gem5 simulator and the prefetching module using the Tagged prefetcher.

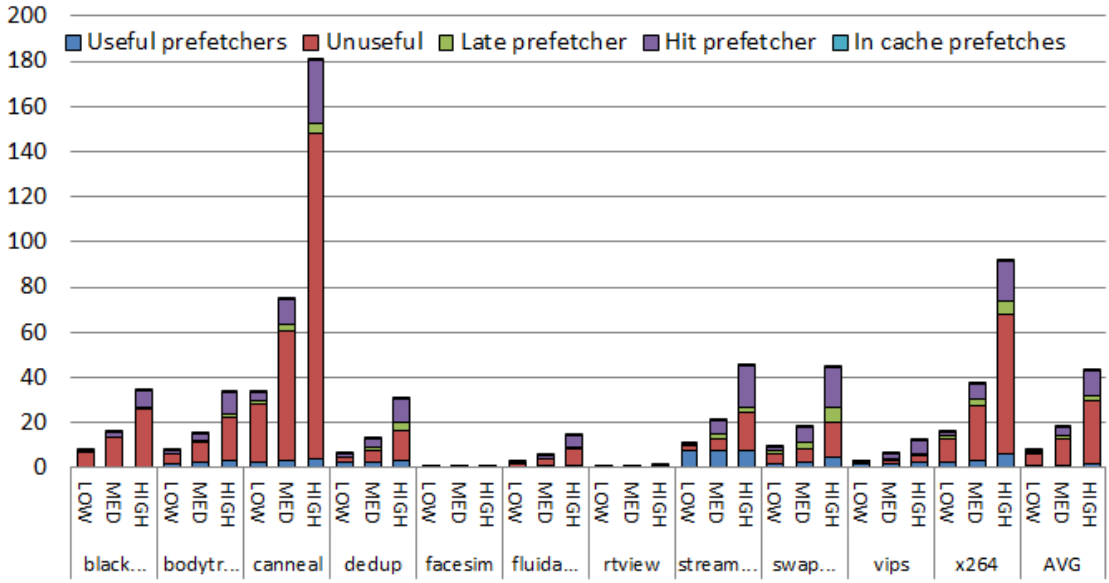


Fig. 3.18 Tagged prefetcher requests distribution every one thousand instructions.

the mpki reduction of low aggressiveness and the medium aggressiveness is very similar. In average the high aggressiveness is the configuration that decrements in a higher degree the

Figure 3.20 shows the increment of the congestion in the network. As we can see the increment compared to the Tagged prefetcher is less important. nevertheless there is a straight relation between the increment of aggressiveness and the increment of latency.

In Figure 3.21 we can see a comparison of the IPC between the simulation witout prefetching, and the simulation with several aggressiveness. In this case, we see that the medium aggressiveness is in almost all the cases the most successful. This is because the

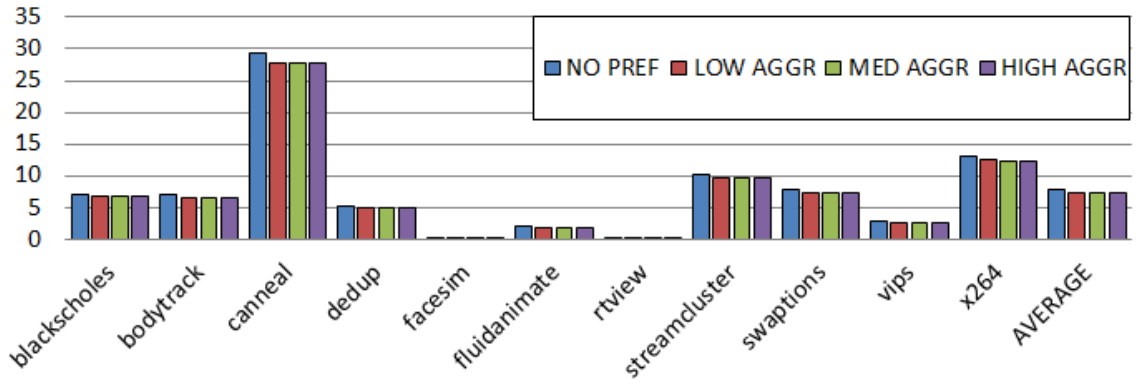


Fig. 3.19 Per benchmark MPKI results generated with the gem5 simulator and the prefetching module using the RPT prefetcher.

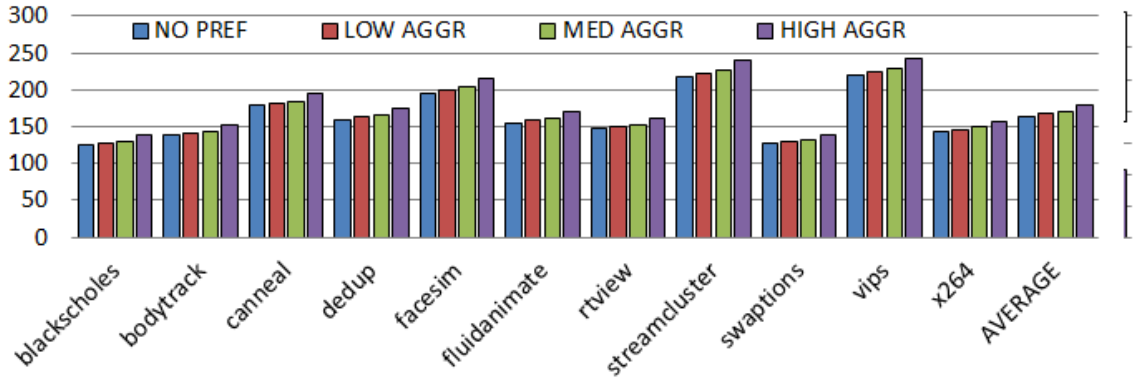


Fig. 3.20 Per benchmark average L1 miss latency (cycles) results generated with the gem5 simulator and the prefetching module using the RPT prefetcher.

increment of latency due to the injection of prefetching increases more slowly than in the case of the Tagged prefetcher. The medium aggressiveness reduces the MPKI in a higher degree than the low aggressiveness do.

In the last chart of the comparison of the RPT prefetcher we can see that the total number of requests increases as well as the aggressiveness increases. Nevertheless, we see that compared to the Tagged prefetcher, the RPT has a higher percentage of of useful requests and a lower percentage of usless requests, what contributes in being a more accurate prefetcher.

3.4.5 Aggressiveness analysis on GHB: detailed results

Figure 3.23 shows that GHB prefetcher does not reduce the MPKI for 5 of the 11 benchmarks, but two of them have almost no misses (facesim and rtview). Thus, GHB is able to reduce misses in 6 of the 9 benchmarks when high MPKI values are involved. Figure 3.26shows the

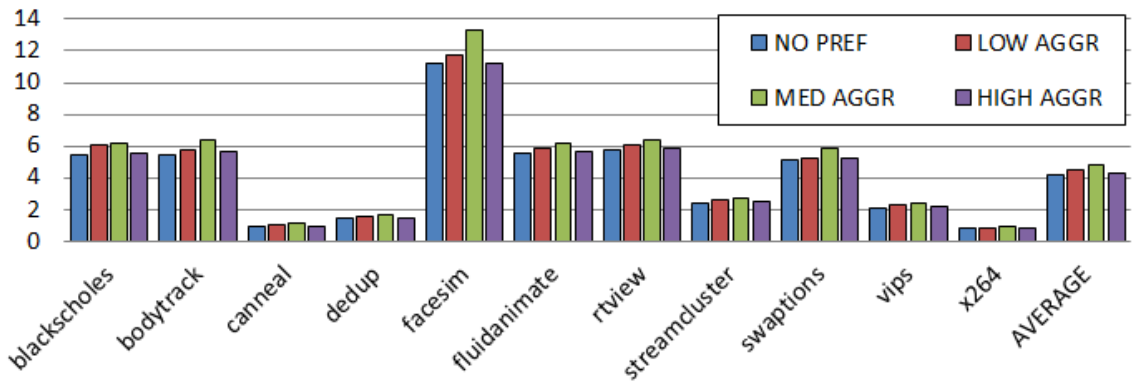


Fig. 3.21 Per benchmark IPC results generated with the gem5 simulator and the prefetching module using the RPT prefetcher.

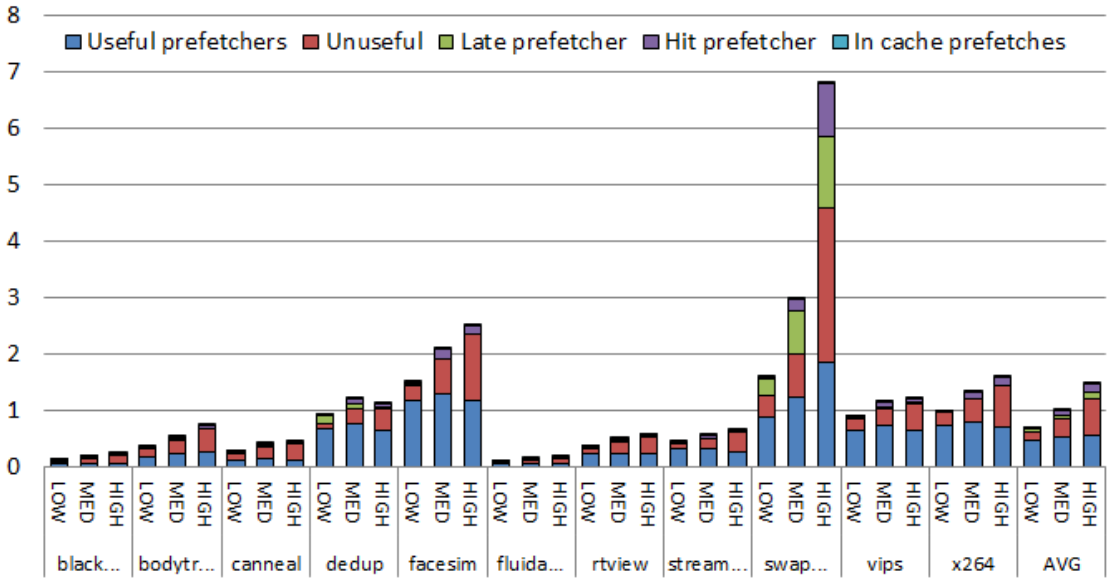


Fig. 3.22 RPT prefetcher requests distribution every one thousand instructions.

increase in requests generated by the prefetcher. It is important to note that in most cases in which the number of misses does not fall, the prefetcher does not increase the number of requests. The reason for that is that when the GHB prefetcher does not find a correlation, it is not able to generate requests. In Figure 3.24, it can be seen that the fact that there are fewer request operations contributes to a reduction in network congestion, thereby ensuring that the latency of the requests traversing the network does not increase. In Figure 3.25, we can see that there are only a few occasions in which the prefetcher leads to an IPC slowdown, all of these occurring when the prefetcher is working at the highest level of aggressiveness. With

this information, it can be said that the GHB prefetcher does not provide significant speedup in some of the benchmarks because it is unable to find a correlation in the memory access pattern of the application. However, it does not increase the number of useless requests in as extreme a manner as the Tagged prefetcher does, for example, which would cause greater network congestion and cache module pollution.

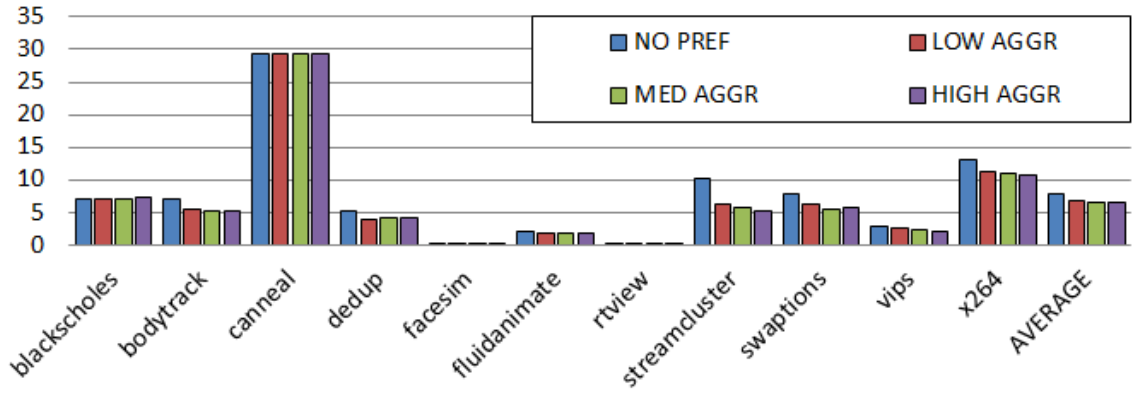


Fig. 3.23 Per benchmark MPKI results generated with the gem5 simulator and the prefetching module using the GHB prefetcher.

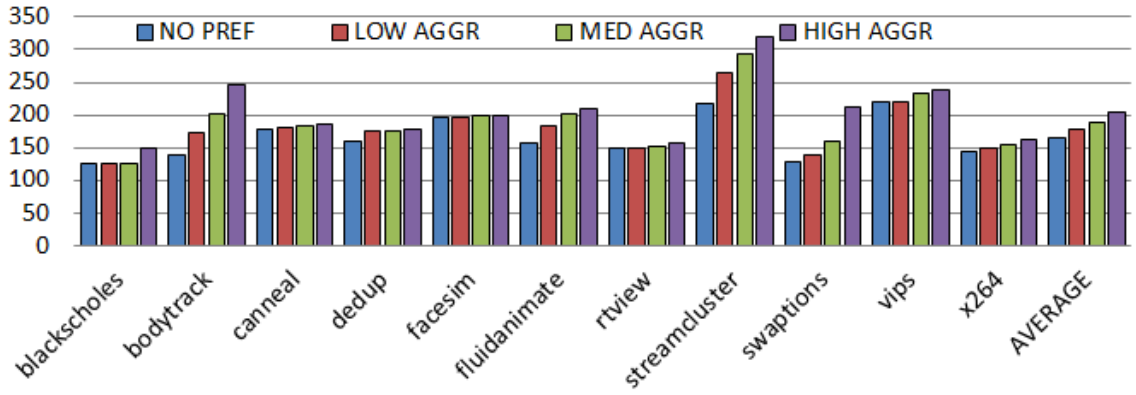


Fig. 3.24 Per benchmark average L1 miss latency (cycles) results generated with the gem5 simulator and the prefetching module using the GHB prefetcher.

3.4.6 Analysis of the network resources effect

In this analysis, we have modified the number of virtual channels (VCs) in order to test out several network environments. Note that the considered baseline employs two VCs assigned to each of the three virtual networks (as stated in Table 3.3). This number of virtual

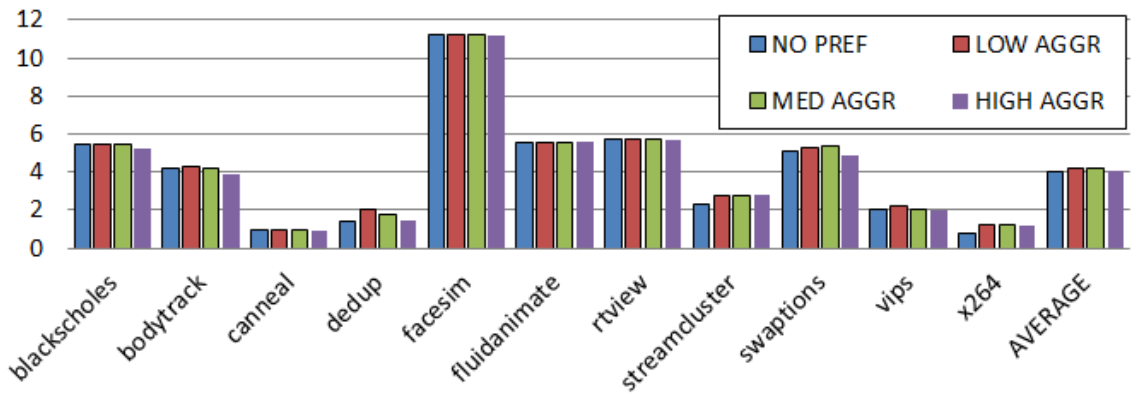


Fig. 3.25 Per benchmark IPC Speedup results generated with the gem5 simulator and the prefetching module using the GHB prefetcher.

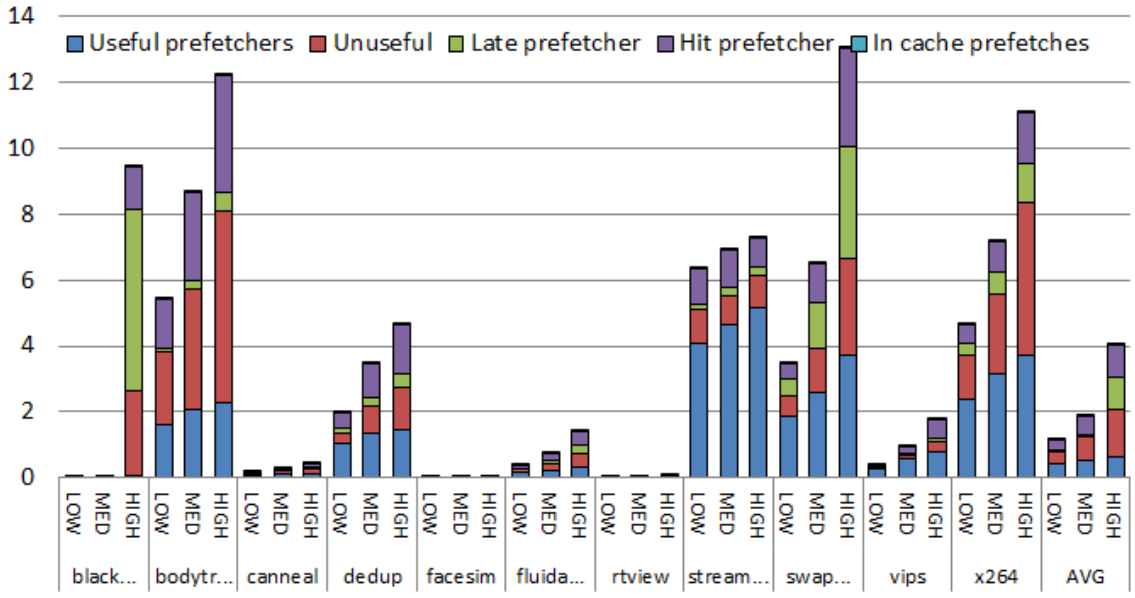
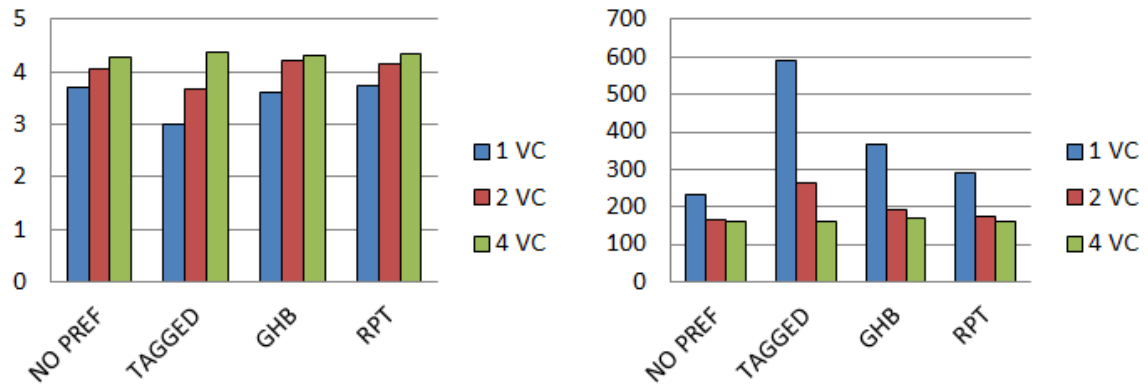


Fig. 3.26 GHB prefetcher requests distribution every one thousand instructions.

networks is required to avoid deadlocks in the memory coherence protocol. Each of the virtual networks holds a different kind of traffic: (1) request messages between L1 and L2, (2) request messages between L2 and Main Memory, and (3) response or data messages among all the cache levels. Modifying the number of VCs per virtual network has a direct impact on the network capability to handle contention. Note that VCs allow packets to overcome other packets in the same path. The higher the number of VCs, the more traffic that can manage the network and the less problematic is the head of line blocking. However, note that



(a) IPC.

(b) Avg latency of the L1 misses (cycles).

Fig. 3.27 Network resources analysis.

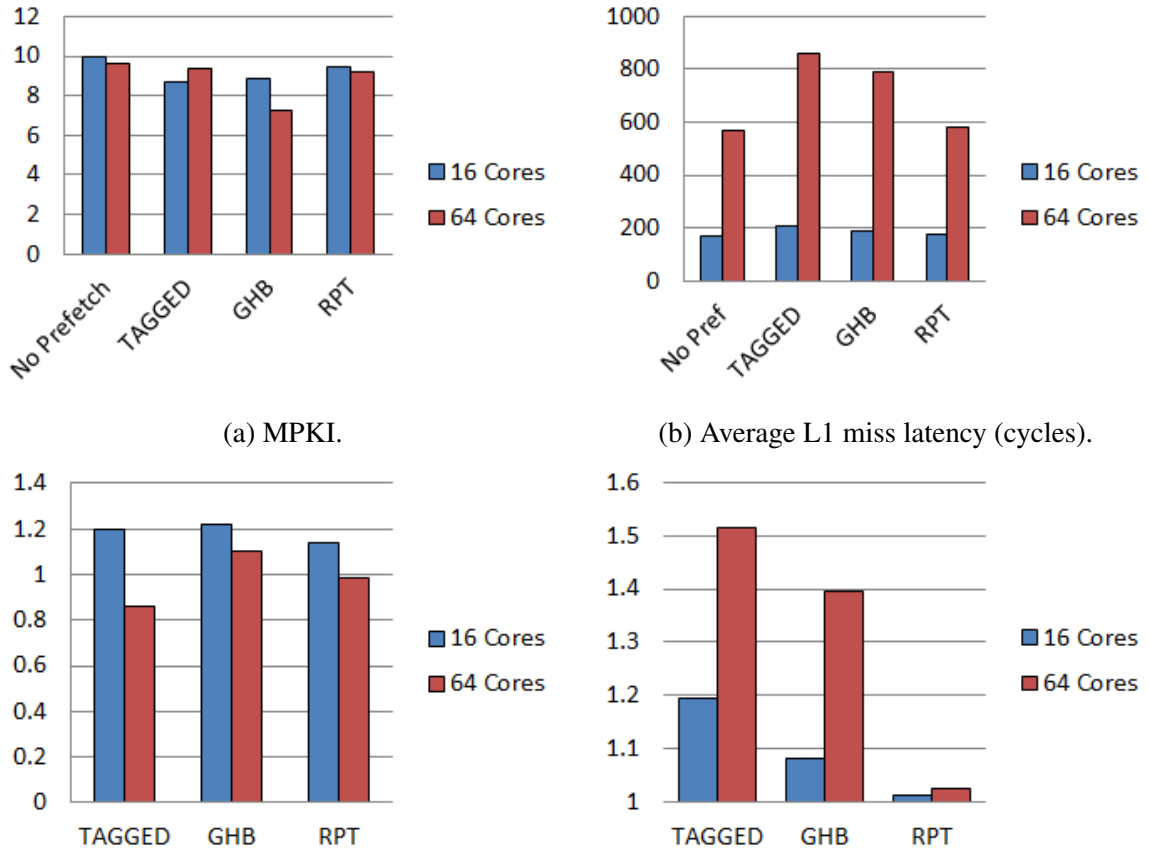
increasing the number of VCs implies more buffer space and increased complexity in the network interface and router implementation.

The aim of this analysis was to test the soundness of the entire framework, to show how network parameters affect global performance, and to show that studies carried out without evaluating network congestion may be considering networks with unrealistic resources. Figure 3.27.a shows the IPC of the different prefetch engines when varying the number of VCs. When the resources are increased, the congestion is reduced and the prefetch techniques can work without limitations: therefore, they perform better. As we can see in Figure 3.27.b, the request latency does not increase when the network resources are high. For this reason, the final speedup is only affected by the MPKI. If the network resources are reduced to more realistic or low values (for instance, in a low area design), the network has a critical impact on the final performance of the prefetcher. This provides support to the idea that when a CMP design is studied in a simulator with a prefetcher, the network should also come under analysis. Otherwise, the experiments will take place in an unrealistic environment.

3.4.7 Scalability analysis

The aim in this part of the study was to analyze the behavior of the module in an environment with a higher number of tiles and thus, also a higher number of cores. In this new scenario, the communication between cores increased and it was possible to evaluate the criticality of the network contention introduced by the prefetcher. In this study, we used the parameters of the best configurations observed in the previous study, but with 64 tiles instead of 16. Note that as each tile has 512KB of shared memory, the total L2 shared memory will be 32MB instead of 8MB. Moreover, some benchmarks did not work in the environment of 64 tiles.

For this reason, we have used a subset of the benchmark suite to make this analysis in both of the experiments (16 and 64 tiles). Four benchmarks could not be simulated: bodytrack, facesim, raytrace, and swaptions. However, as can be seen in the previous experiments these benchmarks do not change the trends of the behavior of the memory system. In addition, to compare both systems, we have maintained the same number of cycles simulated for each core, but, obviously, the simulations of 64 tiles have executed longer regions of code. For this reason the regions of code executed by both systems may be different.



(a) MPKI.

(b) Average L1 miss latency (cycles).

(c) IPC Speedup (vs. no prefetching).

(d) Increment of the L1 miss latency.

Fig. 3.28 Scalability analysis results with the prefetching module.

By increasing the number of tiles, the resources are also increased, but there are also more consumers. This affects the system in two ways: on the one hand, the private resources, such as L1, will preserve its number of consumers, but they will have less activity, because the input set will also be divided into more parts. Figure 3.28.a shows the difference for L1 MPKI with the 16-tile simulation and that with the 64-tile simulation. It can be observed that, for each prefetching mechanism, the MPKI for 16 tiles is very similar to the MPKI for 64 tiles. As we have seen, the reason for this is that, although the number of cores was increased, the

number of the private caches was also increased in the same proportion, and so the working sets fitted in the same way into the cache. Only very little changes can be observed because, as it was stated before, different regions of code are executed in each of the simulations. On the other hand, this increase in consumers affected the shared resources, such as the network, which came under more pressure. Moreover, this effect on the network was critical because the network bandwidth was not increased (the size of the network was increased, in the sense that it was now a 8x8 tile network, but the same bandwidth was maintained). This effect is shown in Figure 3.28.b where we can see that the L1 memory miss latency increases dramatically compared to the simulation with 16 tiles. This increment is explained because there is a higher number of hops that the requests must do to reach its destination. Moreover, this effect is aggravated by the prefetcher, which is injecting requests into the network. Although all the prefetching techniques manages to reduce the MPKI, Figure 3.28.c shows slowdown in two of the three prefetching techniques and the only technique which is able to get some speedup, is getting less speedup than the simulations with 16 tiles. The reason of this effect is the increment of traffic injected by the prefetcher. Figure 3.28.d shows the L1 miss latency increment for the 16 tiles system and the 64 tiles. This increment has been calculated towards the simulation with no prefetching of 16 and 64 tiles respectively. In this figure we can observe that the traffic injected for each prefetch engine is always higher in the 64 tiles system. For this reason, we can assure that the prefetching techniques tested in this study do not scale properly and some improvements must be done in this field.

3.5 Conclusions

In this chapter, we have shown the importance of accurate NoC simulation when considering prefetching techniques. We have developed and presented the modifications required to transform gem5-Ruby into a prefetching-aware infrastructure. As we have seen, this requires a number of changes, including changes to the state machine that controls the memory system. This modified simulator enables the accurate simulation of the combined effect of prefetching and the NoC. In this sense, our prefetching module allows the user to extract statistics from the memory system and the cores (as most other simulators do), but also further statistics regarding the behavior of prefetching requests and their impact on the network. Furthermore, this module has been designed to allow prefetching engines to be attached to and switched in the simulator in the easiest way possible. Thus, researchers can test their own prefetching engines in the simulator without needing to have in-depth knowledge of the whole prefetching module. Moreover, we believe that the resulting simulation infrastructure is useful not only for the evaluation of new prefetching-related techniques in CMP environments, but for any

microarchitecture-related study in a multi-core environment, as it can significantly increase the accuracy of these.

Finally, we have employed this simulation infrastructure to demonstrate that the interaction of prefetching mechanisms with the NoC can be an important factor in multi-core systems. The results obtained using our simulation infrastructure show that the requests generated by a prefetcher have a considerable impact on the global performance of the system, as the requests generated by the prefetcher are traversing the network together with all the other memory requests. For this reason, we believe that it is essential to take into account the prefetching effect on the network when simulating a CMP system. This will enable better-informed decisions to be taken for real systems.

Chapter 4

Confidence predictor mechanisms for prefetching in CMPs

> Prediction is very difficult, especially if it's about the future.
>
> *Niels Bohr*

4.1 Introduction

Non-useful requests have a large negative impact on the system. This is because they contribute to pollute the cache, evicting data with potential temporal and spatial localities, that could be reused later. They also increase congestion in the memory system and waste power by sending and storing unnecessary requests. In CMP and specially many-core systems, this problem is aggravated by the effects of unnecessary traffic in network on chip (NoC), which bad prefetchers can easily congest, possibly leading to both performance and power degradation [81]. In order to avoid these large number of useless requests, prefetchers used by commercial processors are very conservative. They only issue requests when they are almost certain that they will be useful. This methodology greatly increases the accuracy of the prefetcher, but limits its performance.

Several proposals in the literature [90] [45] [19] make a prediction of the probability of usefulness of the prefetch requests generated by a given prefetching engine, and use this information to apply several optimizations to improve performance and/or reduce power. However, we will show that the common approach to predicting the accuracy of a prefetch request is quite naïve, predicting the same accuracy for all requests generated by a prefetching

engine based on the accuracy observed during a previous interval of time, and may even not provide a good prediction.

In this chapter, we put forward and evaluate novel confidence predictor mechanisms for prefetching requests. Particularly, the contributions are as follows:

- We propose several new confidence predictors for the prefetching.
- We evaluate and compare techniques in order to find the one most capable of making accurate predictions in a realistic simulator, enabling the combined effect of prefetching and the NoC in a CMP to be accurately simulated.
- We suggest a feasible hardware implementation for the most accurate technique.

The chapter is organized as it follows. Section 4.2 exposes the different confidence predictor that we propose and compare in this chapter. In Section 4.3, we show the hardware implementation for the best confidence predictor proposed in this chapter. Section 4.4 presents the methodology followed to evaluate the different techniques and the results of the study. Finally, Section 4.5 summarizes our main conclusions.

4.2 Proposed confidence predictors for prefetcher requests

The success of the dynamic prefetching techniques is directly related to their capacity to predict the performance of a request before it has been issued. Thus, if the technique predicts that the request will be useful, its aggressiveness might increase, it might not be filtered, or it might gain in priority (depending on the technique applied). If, on the other hand, it is predicted to be useless, the aggressiveness can be lowered, the request can be filtered, or its priority decreased. Therefore, accurately predicting the behavior of a particular request is a key point for the success of these kinds of techniques.

To carry out the prediction, many of these techniques assume that the accuracy of a given prefetcher engine varies slightly in execution time (or to be more precise, that its behavior at a given point in time will be similar to its behavior in the recent past). Actually, as is shown in Figure 4.1 to calculate the confidence of a new phase $(x+1)$, the confidence predictor used for the prioritization technique from the state of the art [45] uses the accuracy from the last phase. The example is completed with the following formula that explains how the confidence is calculated by this predictor:

$$confidence(x+1) = accuracy(x)$$

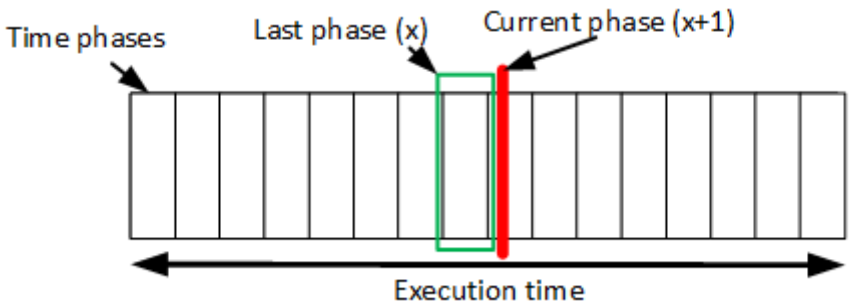


Fig. 4.1 How the last phase heuristic predicts its confidence.

Although, this techniques have shown significant performance, we believe that this predictor is too coarse-grained and could be improved so as to become more accurate and fine-grained, thereby increasing the effectiveness the dynamic prefetching techniques.

With this objective in mind, we have focused on several heuristics that try to infer the probability of usefulness, i.e. the accuracy of a prefetch request at its moment of generation. In the following subsections, we describe five new techniques that we have developed to improve confidence prediction.

4.2.1 Phase history heuristic

As stated earlier, we take the *last phase heuristic* [45] as the baseline and develop our confidence predictor as an upgrade of this technique. The execution time is divided into phases of a certain number of cycles and every time a phase expires, it is tagged with the accuracy calculated by the profiler during the phase. However, instead of using the accuracy of only the last phase to predict the accuracy of the new one, as it is shown in Figure 4.2 in this approach, the predictor uses the accuracies calculated in the previous N phases. To do this, it calculates the average of the last N phases and uses this number as the confidence of the new batch. Therefore, to calculate the confidence of the phase $x+1$ the following formula is used:

$$confidence(x+1) = \frac{\sum_{i=0}^{N-1} accuracy(x-i)}{N}$$

The reason for this modification is that we have observed that there is no clear correlation between the behavior of one phase and the previous one. We therefore try to soften the effect of the last phase and achieve a more progressive change on the prediction from phase to phase.

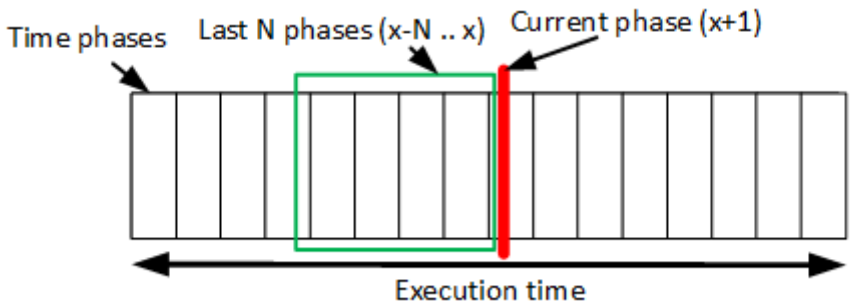


Fig. 4.2 How the phase history heuristic predicts its confidence.

4.2.2 Balanced phase history heuristic

Although the previous heuristic improves prediction in some cases, in other cases the prediction could still be inaccurate. According to our observations, this is because not all the last N phases should have the same impact when calculating the confidence prediction for the new phase. We believe that the closer the phase is to the new phase, the more impact it should have in the prediction. That was the idea behind this second heuristic.

As in the case above, this heuristic is an upgrade of the previous one. The execution time is again divided into phases of a certain number of cycles that are tagged with the accuracy calculated by the profiler in that phase. However, when predicting the confidence for a new region, instead of calculating the average of the accuracy in the last N phases, in this approach, we calculate the confidence by giving more importance to the phases that are closer to the new one. Hence, to calculate the confidence of a new phase $(x+1)$ we use the following formula:

$$confidence(x+1) = \frac{\sum_{i=0}^{N-1} accuracy(x-i) * (N-i)}{\sum_{i=0}^{N-1} N-i}$$

4.2.3 Stream position based profiling

In the accuracy predictions performed by the different phase-based heuristics, all the requests generated from the same prefetching module (i.e. the prefetcher in a core of a CMP) are assigned the same profiling information and thus, the same prediction. Discrimination is therefore performed in a coarse-grained manner, between prefetching modules, but not between requests generated by the same prefetcher.

In this subsection, unlike with our two previous proposals, we do not try to improve prediction accuracy by modifying the way we consider the profiled data from previous phases,

but by changing the granularity with which we perform the profiling. Thus, the resulting algorithm is not an upgrade of the previous version but a completely different approach.

This heuristic focuses on the stream of requests generated by the prefetcher. When a prefetch predicts a memory address as the next one that the core will need, it usually generates a small stream of requests depending on the aggressiveness of the prefetcher. This is commonly done to increase the possibilities of having successful prefetches. We have identified a direct relation between the position of a prefetching request in the stream of prefetch requests and its accuracy. With this information, we want to design a technique able to identify different predictions for requests in the same stream.

This stream of prefetching requests is totally dependent on the prefetch engine and its aggressiveness. Thus it is the prefetch engine that is responsible for deciding the order of the requests in this stream. Moreover, in the stride, and stream prefetchers the number of requests generated per stream is usually directly related to the aggressiveness. However, in the correlation prefetcher it may depend on other parameters. Stride and stream prefetchers generate small streams very frequently, whereas the correlation prefetchers generates much longer trains, but less frequently.

For this reason, the requests that are generated in a certain position of the stream may have more accuracy than requests generated in other points of the stream. We have therefore taken advantage of this insight to design a heuristic that is able to perform a more fine-grained prediction of the accuracy of a generated request that will allow us to distinguish between several levels of accuracy among the different prefetching engines in the system (probably located in different cores of the multiprocessor), but also among the requests generated by the same prefetcher.

To be more precise, Figure 4.3 shows how this technique stores the information. Using this structure, this technique profiles the accuracy for every position in the stream request. The mechanism makes use of a table whose size is the maximum number of prefetch requests generated by a single stream. Each of the positions in the table contains the accuracy of the prefetch requests issued from this position. Whenever a new request is generated in the same position of the stream, the prediction of confidence is made by consulting the accuracy value in the table.

Although, we could also have used a phase-based approach to gather and generate the prediction for each stream position, in the performance evaluation section we have employed the accumulated value of accuracy for the time the application has been running to determine the predicted accuracy of the prefetch requests generated.

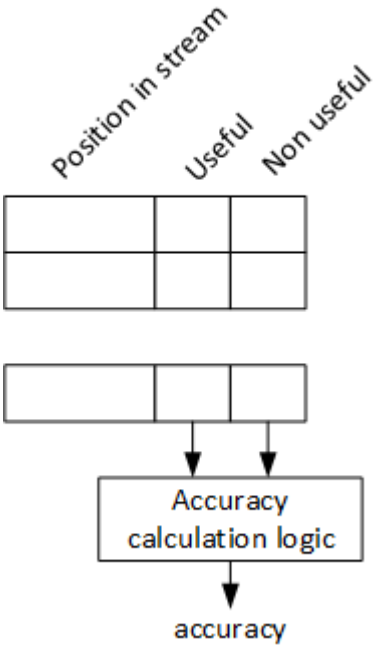


Fig. 4.3 Design of the stream based predictor.

4.2.4 Code region dynamic profiling

This technique is based in the software-hardware co-designed approach presented in [55]. In order to go one step further and make a more fine-grained prediction, we analyze accuracy according to the executed region of code when the request is issued. To do this we profile the region of code responsible for triggering a stream of prefetch requests. We assume that the accuracy of a prefetch request triggered by the same region of code will have no significant variations. In line with this hypothesis, we profile the regions of code by updating the number of useful and useless prefetch requests generated by the prefetchers triggered when the control is in this region of code. We do so by making use of the structure showed in 4.4. When we are again in that region, we can predict the accuracy of the prefetching requests triggered by the memory operations of the region based on its previous executions.

To implement this technique we use a hardware mechanism to detect the code region for every memory request and assign it a memory region identifier. When one of these demand requests triggers one or more prefetch requests (depending on the aggressiveness of the prefetcher), the feedback related to the profiling information generated by these requests is stored in a table indexed by that code region identifier. Thus, when a request triggers the prefetcher, the table with the profiling information of the region is accessed and the prefetcher can use this information to predict the confidence of the request.

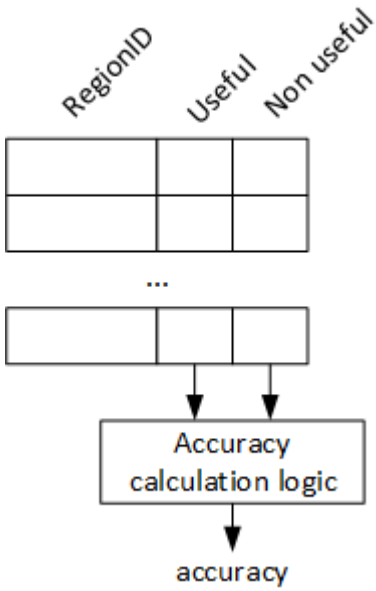


Fig. 4.4 Design of the code region predictor.

We have used basic blocks as our code regions for evaluation purposes, but the regions used could be any others, such as superblocks, hyperblocks, functions, inner loops or even individual instructions. We have not used the latter option because it would increase the hardware cost of profiling and maintaining the confidence information, and we have observed that it does not significantly increase the accuracy of the prediction compared to the basic block approach.

4.2.5 Heuristics combination

This final technique is the result of combining the two previous ones. This confidence predictor tries to predict the accuracy of a request by analyzing the previous accuracy determined by two variables: (1) the position in the train of requests generated by the prefetcher at the trigger time, and (2) the granularity of the profiled code. Figure 4.5 shows how the structure used by this predictor is the combination of the two proposed before.

As the techniques presented in the two previous subsections are orthogonal, both can be applied at the same time to improve the confidence of our predictor. That is therefore what we have done. In the profiling table, for of each region, we have included an array to track the number of useful and useless requests generated by each point in the stream of requests generated by the prefetcher. Thus, we have the profiling information sorted according to the generation point in the stream of prefetch requests and the region of code where the request

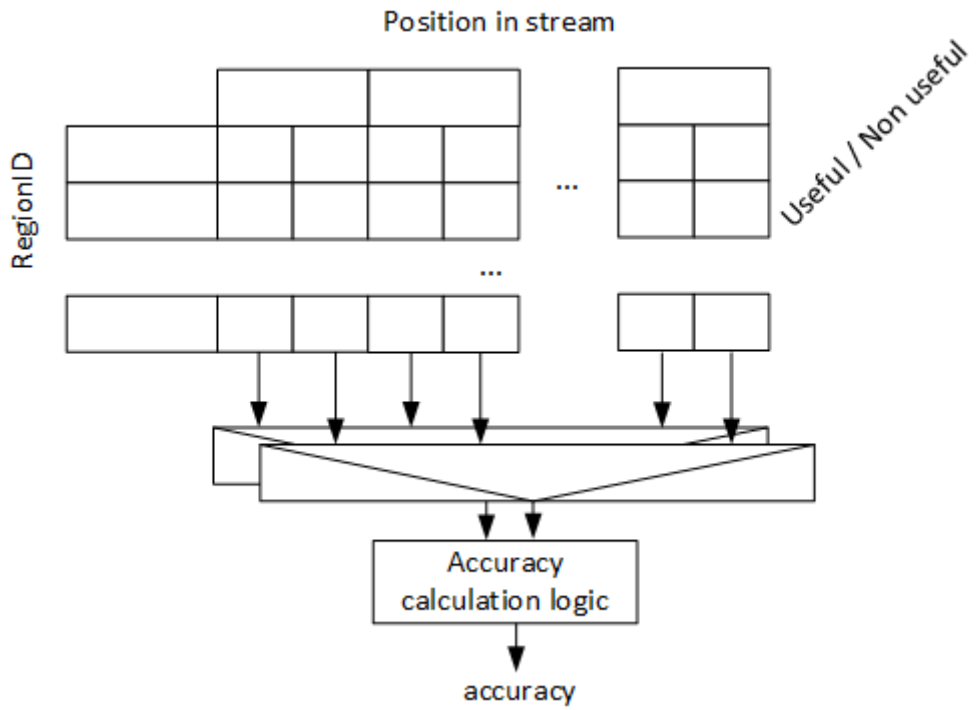


Fig. 4.5 Design of the combined predictor.

was triggered. More details on the hardware implementation of this technique are shown in the next section.

4.3 Hardware implementation

In this section, we explain the extra hardware needed to make it feasible to implement the combined confidence predictor and how it is used.

4.3.1 New hardware description

We introduce two new hardware tables into the system. Their specifications are listed below:

- **Prefetch Profiler Table**: The main objective of this table is to reduce the overhead of the bits to be added to every cache line. Each entry represents a prefetch request and the idea is to have, in the best case, one entry per prefetched line in the cache. If there are not enough entries in the table to maintain information about all the prefetched lines that have not been used, then as we will see, it will only store information about the last blocks brought into the cache. Each entry in this table is indexed by the

address of the prefetched memory line. Note that, as the profiling of the prefetching is not in the critical path of demand memory requests, this table does not need to be accessed in a fast way. Therefore, we can implement a linear search on it or maintain an ordered structure to perform the search and access. Moreover, This table contains the address of the memory line it represents, a bit to indicate if the entry is active, the identifier of the region that has generated the request, and the position in the stream of requests. When a memory line marked as prefetched is used by a demand request or evicted without being used, we can use the information in this table to increase the count of useful and non-useful prefetch requests associated with the corresponding memory region and position on the stream in the Region Profiler Table (see below). To determine the number of entries in the table and its replacement policy, we carried out some experiments with an unbounded table. Elements in this table were evicted only when they were replaced from the cache. We classified the elements that lasted more than a short period of time in the table as useful or non-useful. The results are shown in Figure 4.6, where we see that most of the prefetch requests that were not used in a brief period of time, were useless. Therefore requests that stay too long in the table, could be considered non-useful. This means that if we apply a FIFO replacement policy to this table, we can assume that the replaced requests will be non-useful. Some further experiments were done to calculate the number of entries in the table. The optimal result for the workloads tested is to use a table with 64 entries, which would be the value used in the performance evaluation section. Instead of using a the prefetch profile table to store the information on a limited nuber of a prefetch requests, a naïve implementation of this mechanism could, store the identifier of the code region and the position in the stream of prefetchers in each line of the cache itself. However, this would require a large number of bits to be added to all the cache lines. Our proposal is to include just a single bit per line to indicate whether the cache line has been prefetched and to use the Prefetch Profiler Table as a secondary table, which is a much more efficient approach.

- **Region Profiler Table**: The aim of this table is to keep and update profiling information on of the hot regions of the code. For this reason, each entry on the table is indexed by the region of code and contains an array with two counters for every possible stream position (the maximum number of positions depends on the type of prefetching and how aggressivene it is). These two counters store the number of useful and non-useful prefetchers generated by each position on the stream and memory region. The ratio between the useful and the useful + non-useful counters represents the profiled accuracy for that region of code and the position on the prefetching stream

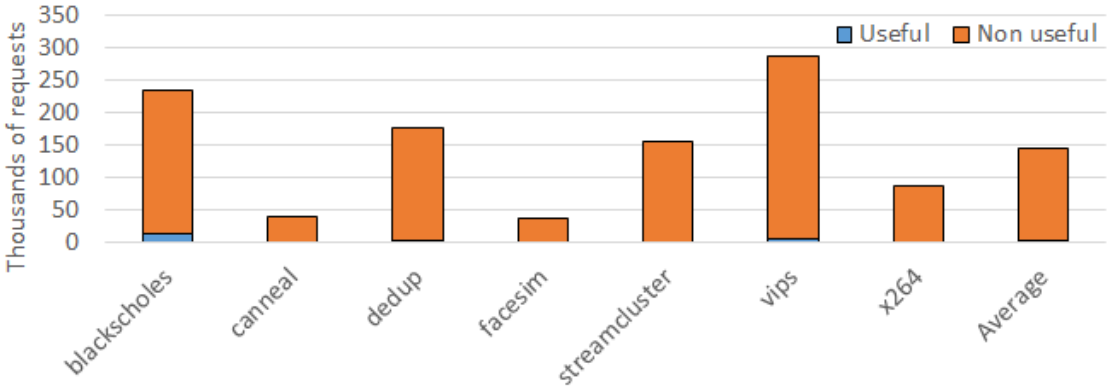


Fig. 4.6 Useful and non-useful classification of prefetch requests that last for more than 100 cycles in the Prefetch Profiling Table (using an unbounded table).

tuple or generation point. As we want to keep the most active regions in the table, we use a Last Recently Used (LRU) replacement policy. Again we carried out several tests to find the most suitable size for this table. The optimal result for the workloads tested was to use a table with 32 entries, which will be the value used on the performance evaluation section. As mentioned in the previous section, we considered using individual instructions as the memory region (with the PC as the region identifier). This produced slightly more accurate predictions but at the cost of requiring more table entries to hold the information at such a fine-grained level of granularity. We therefore, rejected this option because it was less effective.

4.3.2 Prediction flow

In this subsection, we present the interaction between the hardware tables described above and the methodology of the combined predictor. Figure 4.7 shows the block diagram for each of the steps described in this methodology. Note that, each step is represented by a number which description is in the following enumeration.

1. **Memory access**: The core issues a memory request that goes to the data cache.
2. **Memory stream analysis**: Depending on the specific prefetching algorithm and the kind of pattern that it tries to correlate, the prefetcher engine analyzes data access, using information such as: the accessed address, whether it was been a hit or a miss, the PC of the instruction, etc.
3. **Prefetch trigger**: Some of the demand memory requests that are analyzed by the prefetcher may trigger the prefetcher and generate a stream of prefetching requests.

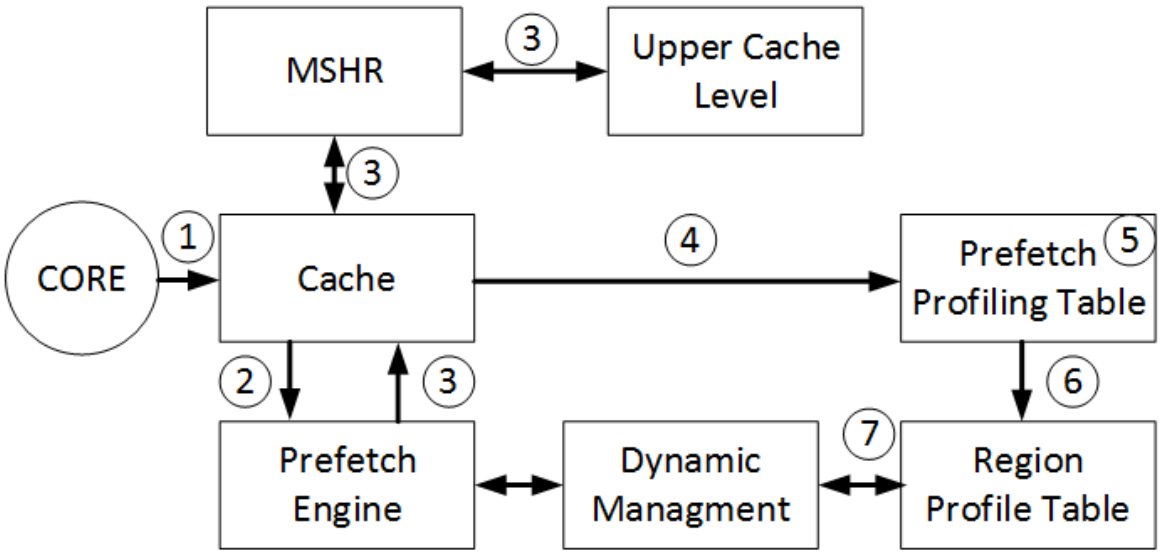


Fig. 4.7 Block diagram for the hardware implementation of the combined predictor.

When generating these requests, the prefetcher tags them with the identifier of the code region, which will be common to all the requests generated in the same stream, and the position on that stream. If the memory line requested is already in the cache, the request is discarded. If not, these tags are stored on the miss status holding register (MSHR) until the prefetched line is brought from the upper levels of the memory hierarchy.

4. **Prepare profiling**: Once the prefetched data reaches the cache, the memory line is tagged with an extra bit to indicate that this is a prefetched request. The Prefetch Profiling Table will store the profiling information related to this prefetch request (address, region identifier, and generation point in the stream of requests).

5. **Prefetching request evaluation**: Each prefetching request is evaluated when the corresponding entry in the Prefetch Profiler Table is evicted. This eviction can be due to several reasons and the evaluation will vary depending on the reason. (1) The prefetched memory line is evicted from the cache without being used: the prefetch requests is categorized as useless. In this case, the profiled accuracy of the corresponding generation point will decrease. (2) The prefetched memory line is used by a demand request: once these data are used, the line is marked as a non-prefetch request and the corresponding prefetch request is evaluated as useful. The profiled accuracy of the generation point that triggered the request will increase. (3) Finally, an entry on the Prefetch Profiler Table may be evicted because it is a limited size hardware table and the entry has to be replaced by another new request. In this case, the evicted prefetch

request will be evaluated as useless (for the reason already explained at the beginning of this section), and the profiled accuracy on the Region Profile Table for this request will decrease.

6. **Region Profile Table Update**: As mentioned early, this table is updated when there is an eviction from the Prefetch Profiler Table. Depending on the evaluation of the corresponding prefetch request, the appropriate counters will be updated to keep track of the profiled accuracy.

7. **Confidence prediction**: When a dynamic prefetching management technique requires a confidence prediction to be performed for a given prefetch request (given the memory region of the trigger request and the position in the stream of the prefetch request to be generated), that technique will access the Region Profile Table, on the basis of the accuracy represented by the counters on that table, will categorize the prefetch request to be generated with a confidence level. The simplest way of doing this would be based on an accuracy threshold. If the profiled accuracy is higher than that threshold the request will be predicted to have a high confidence level. If not, it will have a low confidence level. This is the approach followed by most of the dynamic management techniques. However, for comparing the different confidence predictors we have used 3 levels of confidence (low, medium and high) instead of just 2. We have done this for two reasons. The first is that we think that 3 levels of confidence gives a finer granularity to some of the dynamic mechanisms. In prioritization, for instance, it could be interesting to use three levels of priority for prefetching requests instead of two. The second is that, from a qualitative point of view, 3 levels of confidence enable the high category to be associated with the concept of being almost sure that those prefetchers are going to be useful, that the ones in the low category are going to be non-useful, and that we do not know or are not sure about the ones in the medium confidence category.

Note that for the benchmarks we have used in the evaluation section, the limited number of entries in the described tables is enough to give acceptable results. Nevertheless, there could be other benchmarks with many regions which would require tables too large to be effective. If that were the case, we would suggest using a third table that could dynamically detect hot regions in the code. The confidence predictor would only gather statistics that were previously detected as hot. Using this table as a kind of filter would greatly reduce the number of regions processed by the hardware described in this section, thereby increasing their scalability.

4.4 Performance evaluation

In this section, we explain the framework used in the research and the experiments we designed.

Hardware specifications	Values
ISA	x86
Tile number	16
L1 Data/Instruction cache	16Kb each tile
L2 size	8MB
Network	Garnet
Topology	Mesh
Prefetcher cache level	L1
Simulated cycles	250M of cycles
Prefetcher Profiler Table	64 entries
Region Profiler Table	32 entries
Lower accuracy threshold	20%
Upper accuracy threshold	60%
Code region granularity	Basic Block

Table 4.1 Simulation specifications.

4.4.1 Experimental framework

The system simulated in this performance evaluation consists of a tiled mesh with private first level data and instruction caches, and a shared L2 cache. L2 and L1 are non-inclusive. The coherence protocol employed is the MOESI CMP directory. The prefetchers used to evaluate the predictions are the Tagged prefetcher [24] (being the most common stream prefetcher), the RPT [77] (being the most common stride prefetcher), and the GHB [59] (being the most common correlation prefetcher). The hardware specifications are shown in Table 4.1. The simulator employed for the analysis was the gem5 [7] and the benchmark suite in this performance study was a subset of the PARSEC 2.1 benchmark suite [6].

To test the potential of our predictors, we performed several experiments. We divided the results into two groups showing two different points of view. In the first, we compare the accuracy of the requests according to the confidence prediction. As stated before, we distinguish between three levels of confidence: high, medium, and low. The thresholds between them are shown in Table 4.1. In the second group of experiments, we compare the percentages of useful and non-useful requests according to the confidence prediction. To shortcut the names of the techniques in the figures we have used acronyms: last phase (LP),

phase history (PH), balanced phase history (BP), code region (CR), stream position (SP), and the combination of code region and stream position (COMB).

4.4.2 Tagged prefetcher analysis

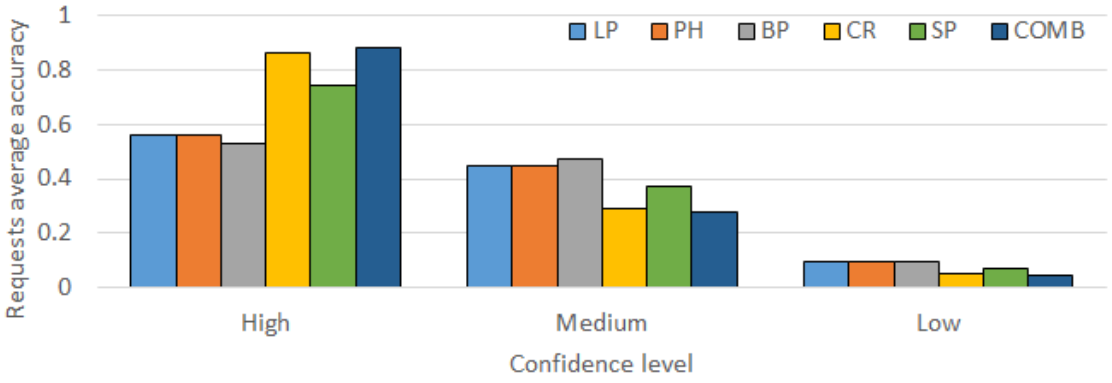
In the following, we will show the average results for the Tagged prefetcher and the per benchmarks results.

Average analysis

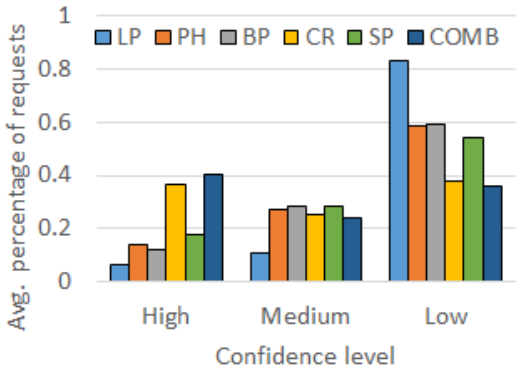
We evaluate the confidence of our technique by getting the average accuracy of the requests classified in each group. Figure 4.8a shows the result of this prediction for the Tagged prefetcher. In this Figure, we can see that the techniques derived from the baseline (PH and BP) present very similar accuracy for each of the three groups. We can also see that the predictions for these techniques are pretty fair. The requests classified as high confidence have an average accuracy of 55% for the high confidence, 45% for the medium confidence and finally, 10% for the low confidence. We can also see that for this case the BP has about 2% less of accuracy in the high confidence predictions which is found in the medium confidence group. This is because the predictor in this case works a little bit worse than the rest. Thus, some requests with high accuracy has been wrongly predicted as medium. However, we can see that the results for the code region predictor stand out from the others. Most of the wrongly predicted requests by the other predictors (mainly high confidence, predicted as medium confidence) are predicted accurately. The accuracy of the requests predicted with high confidence is more than 80%. Medium confidence requests had an average accuracy about 30% and low confidence requests are about 5% accuracy. Although the results of the stream based predictor are better than the baseline, for this prefetcher, this technique does not work better than the code region predictor. Nevertheless, the combined predictor, manages to take profit of the two techniques that form it. This is seen because is the approach that has better results for this prefetcher.

However, the accuracy does not take into account the total number of requests. This means that if a technique classifies only one request as high confidence and considers this request useful, this predictor would be 100% accurate in its high confidence prediction. But the predictor would be far from working properly. Hence, we contrast the accuracy evaluation with a study of the useful and useless requests for each of the confidence levels.

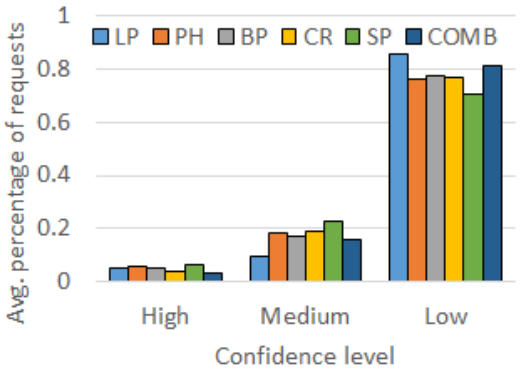
Figure 4.8b shows the distribution of useful requests among the confidence predictions. We can see that all the proposed techniques improve the number of useful requests classified with high confidence. However, most of them still classify many useful requests as medium



(a) Accuracy prediction.



(b) Useful requests prediction.



(c) Non-useful req. prediction.

Fig. 4.8 Results from evaluating the confidence predictors with the Tagged Prefetcher as the stream prefetcher

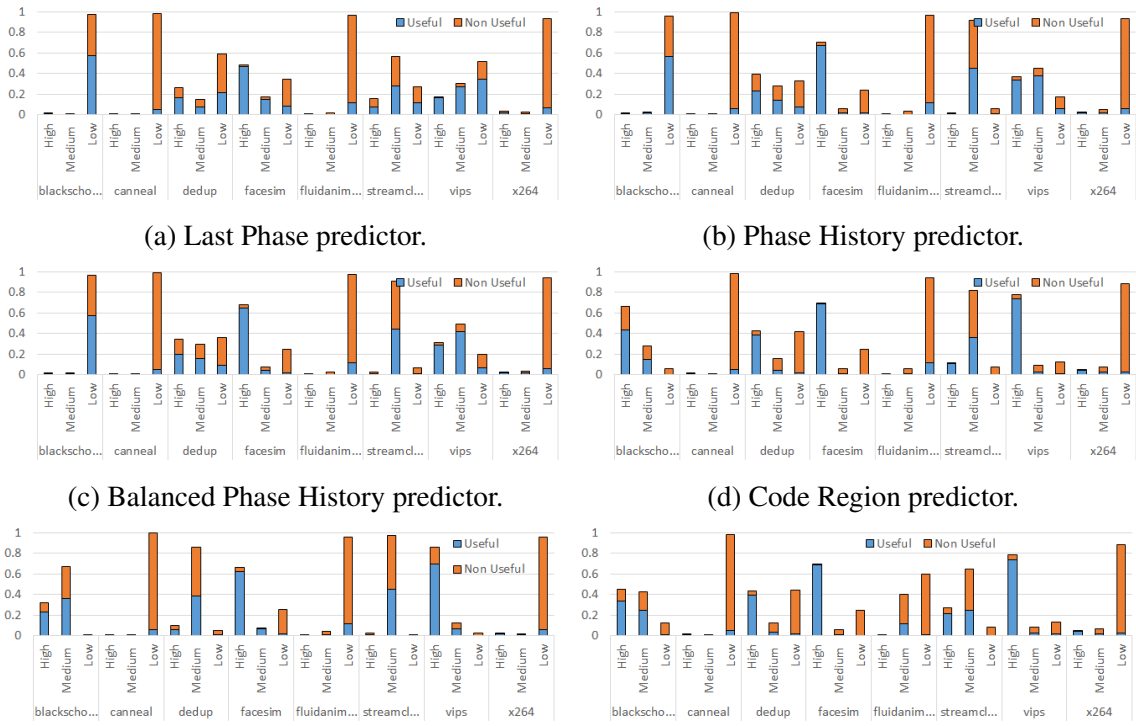
or low confidence. As in the previous case, the code region predictor is the one that works best, followed by the stream-based predictor. Again, the combined predictor benefits from both of the predictors that comprise it. Nevertheless, although it seems to be the best proposal for this configuration, it classifies as many useful requests as low confidence as it does high confidence. This happens because of the large number of non-useful requests that the prefetcher issues. Note that this version of the tagged prefetcher issues two requests per trigger. It also triggers once per miss or useful prefetch. This means that it would generate approximately twice as many requests as misses. Thus, in the best of scenarios, if all the misses were prefetched, the prefetcher would generate 50% useful requests and 50% non-useful requests. Figure 4.8c shows the distribution of these non-useful requests. As we can see, all the techniques manage to classify over 70% of these non-useful requests as low confidence. The percentage of useful requests is so small compared to non-useful that it makes it very difficult to accurately classify them. Nevertheless, the results show that

the code region still classifies most of the non-useful requests as medium or low priority, followed by the stream-based predictor. And the combined version still makes the most of the two techniques and classifies the majority of requests as medium or low priority.

Per benchmark analysis

In order to provide a more detailed analysis, we have analyzed each benchmark with each confidence predictor. We have analyzed in a qualitative manner which benchmark´s prefetching requests can each heuristic classify with better accuracy. Remember that, one of the reasons to use 3 confidence levels is that from a qualitative point of view, this enables the high category to be associated with the concept of being almost sure that those prefetchers are going to be useful, that the ones in the low category are going to be non-useful, and that we do not know or are not sure about the ones in the medium confidence category. Therefore, we are going to consider that a confidence predictor has done a good job classifying the requests from a given benchmark if there are more useful prefetchers classified as high confidence than as medium confidence, and more useful prefetchers classified as medium confidence than low confidence.

Let´s start with commenting the behaviour of the baseline, the Last Phase predictor, which per benchmark results are shown in Figure 4.9a. We can see that this heuristic is only able to classify with high accuracy the facesim benchmark. It is specially doing wrong predictions with the blacksholes, dedup, or vips benchmarks, in which is predicting with low confidence a lot of useful prefetches. In Figure 4.9b, we see the predictions of the Phase History technique, which improves a little bit the prediction of the facesim benchmark, being the resulting classification almost optimal. Nevertheless, it still predicts wrongly the same benchmarks as the baseline does. The next heuristic is the Balanced Phase History, its results are shown in Figure 4.9c. As we can see, the resulting chart is very similar to the Phase History predictor without the balancing, what means that for this prefetcher, the optimization applied on this heuristic is irrelevant. If we look at the results of the Code Region predictor in Figure 4.9d, we can see important improvements on the classification of the previous heuristics. The Code Region predictor is able to increase the number of useful prefetchers classified in high priority in almost all the benchmarks. Moreover, this heuristic succeed in classifying the requests from benchmarks that were wrongly classified with the previous predictors such as blacksholes, dedup, or vips. As can be seen in Figure 4.9e, the Stream Position predictor also improves the predictions from the baseline predictors. However, the classification of the requests is not as accurate as the Code Region predictor is. Finally, as we can see in Figure 4.9f, the Combined predictor gets advantage for each of the predictors that compose it, predicting the most accurate classification for each of the benchmarks.



(a) Last Phase predictor.

(b) Phase History predictor.

(c) Balanced Phase History predictor.

(d) Code Region predictor.

(e) Stream Position predictor.

(f) Combined predictor.

Fig. 4.9 Percentage of useful and non useful requests generated by the Tagged prefetcher classified with high, medium, or low priority by the different predictors.

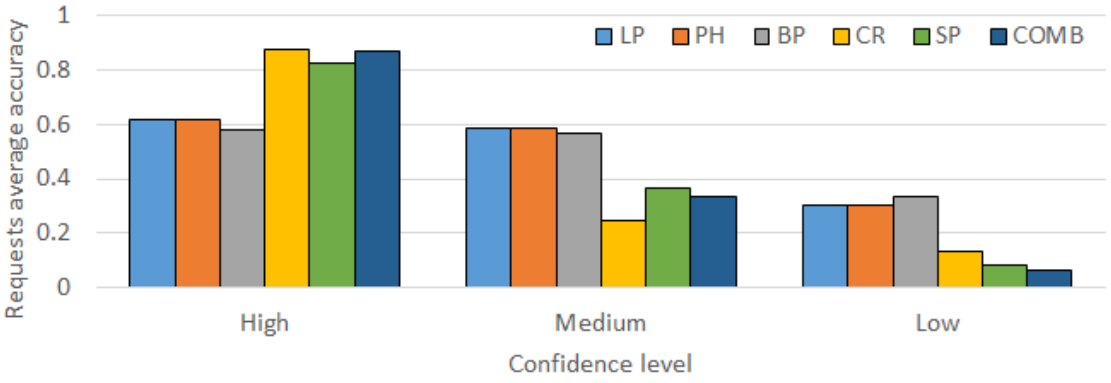
4.4.3 Reference Prediction Table prefetcher analysis

In the following we will show the average results and the per benchmarks results for the Reference Prediction Table prefetcher.

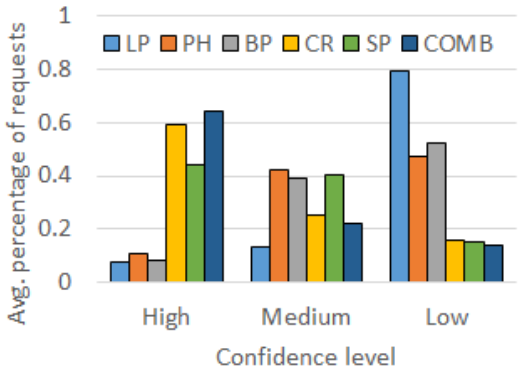
Average analysis

In the next prefetcher, the reference prediction table, the behavior of the predictors is very similar to the tagged prefetcher. As we can see in Figure 4.10a, the baseline predictor, the phase history and the balanced phase history predictors were about 10% better as regards the accuracy of high confidence requests than the tagged prefetcher. However, they are still a long way behind the code region, the stream-based and the combined predictors. Again the balanced phase predictor does not work quite as well as the baseline or the phase history predictor that works with more or less the same accuracy as the baseline. Also, the code region is the predictor that achieves greater accuracy with high confidence predictions, followed by the stream-based predictor. However, the predictor that combines both techniques combines the best of each and makes the most accurate predictions.

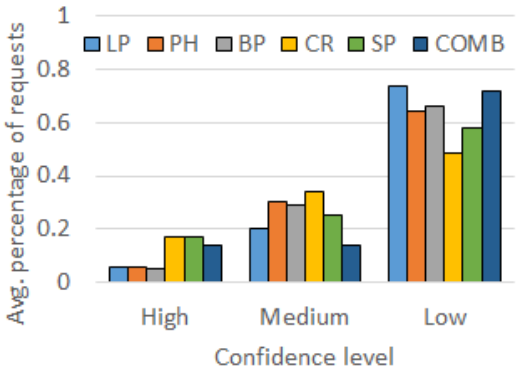
As with the tagged prefetcher, Figure 4.10b shows the percentage of useful requests predicted with high, medium, and low accuracy. However, for the RPT, we see a change in the trend of this classification. The techniques derived from the baseline classify up to 80% of useful requests as low confidence. This can be fixed with the phase history and balanced phase history upgrades. Nevertheless, these techniques classify more useful requests as medium confidence than high confidence and also more useful requests as low priority than medium priority. This is the trend that changes with respect to the code region, the stream-based, and the combined predictors. With any of these techniques, we can see that there are more useful requests classified as high confidence than medium and that there are more requests classified as medium confidence than low. Moreover, the code region predictor and the combined predictor manage to classify about 60% of useful requests as high confidence.



(a) Accuracy prediction.



(b) Useful requests prediction.



(c) Non-useful req. prediction.

Fig. 4.10 Results from evaluating the confidence predictors with the RPT as the stride prefetcher.

The classification of the non-useful requests is shown in Figure 4.10c. Here, we can see that most of the predictors again manage to classify most non-useful requests as low

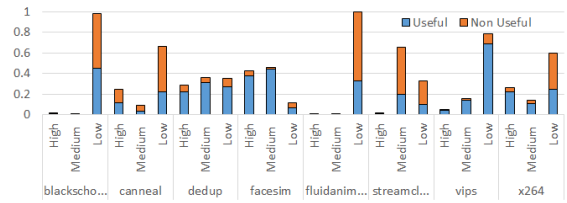
confidence. However, in this case the baseline predictor and its upgrades classify fewer non-useful requests as high confidence than the others. This is because the code region, the stream-based, and the combined predictors classify fewer non-useful requests as medium confidence. The combined approach, meanwhile, takes advantage of the two predictors that comprise it. Moreover, in this figure, we see that the baseline manages to classify more non-useful requests as low confidence. This is because the baseline classifies many requests as low, as seen in Figure 4.10b. Thus, this technique generally classifies many requests as low, which is good for non-useful requests but bad for useful ones. We can evaluate whether it is worth achieving this percentage of low predictions for the non-useful requests when predicting 80% of useful requests with low priority by looking at Figure 4.10a, which shows the predicted accuracy for each. There, we can see that the answer is no, because the average accuracy for the low confidence requests predicted by the baseline is 25% higher than the combined predictor. Moreover, the combined technique predicts high confidence requests with 25% more accuracy than the baseline.

Per benchmark analysis

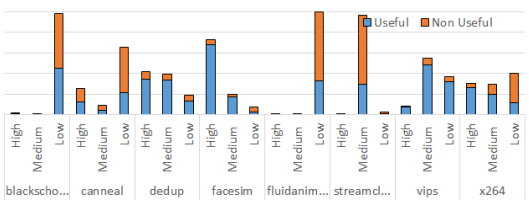
The analysis per benchmark done in the RPT provides similar conclusions than the ones from the Tagged prefetcher analysis. As can be seen in Figure 4.11a, the Last Phase heuristic is only providing good confidence predictions with the facesim benchmark. However, in the majority of the cases the predicted classifications are wrong. As happened in the Tagged prefetcher, the Phase History predictor improves a little bit the prediction of the facesim benchmark as can be seen in Figure 4.11b, but the rest of the benchmarks are still predicted wrong. Figure 4.11c shows that the Balanced Phase History predictor provides more or less the same accuracy for all the benchmarks in its prediction. In Figure 4.11d, we can see that the Code Region predictor improves the predictions for the facesim. But also improves the prediction in the canneal, dedup, vips, and x264 benchmarks. The next technique, the Stream Position predictor, improves the prediction of the remaining benchmarks as can be seen in Figure 4.11e. Finally, the Combined predictor, as shown in Figure 4.11f, gets the best of the two predictors that compose improving or keeping the right predictions for all the benchmarks.

4.4.4 Global History Buffer prefetcher analysis

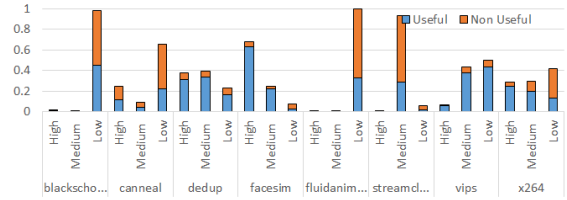
In the following we will show the average results and the per benchmarks results for the Global History Buffer prefetcher.



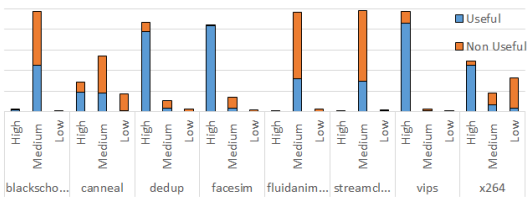
(a) Last Phase predictor.



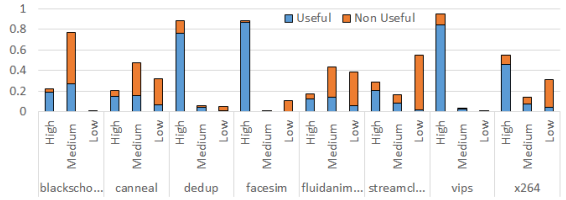
(b) Phase History predictor.



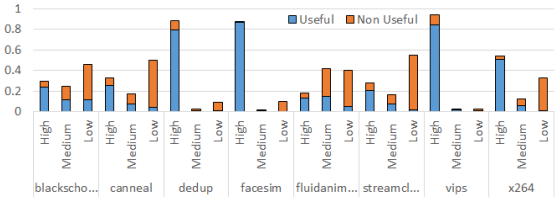
(c) Balanced Phase History predictor.



(d) Code Region predictor.



(e) Stream Position predictor.



(f) Combined predictor.

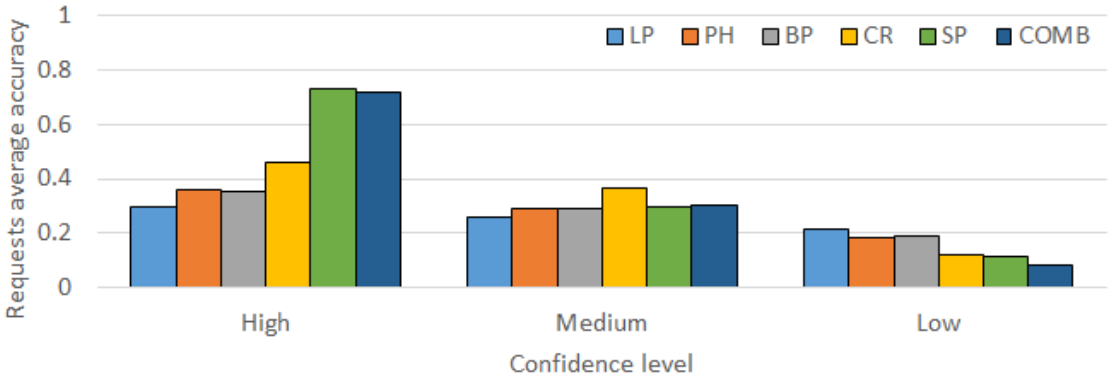
Fig. 4.11 Percentage of useful and non useful requests generated by the RPT prefetcher classified with high, medium, or low priority by the different predictors.

Average analysis

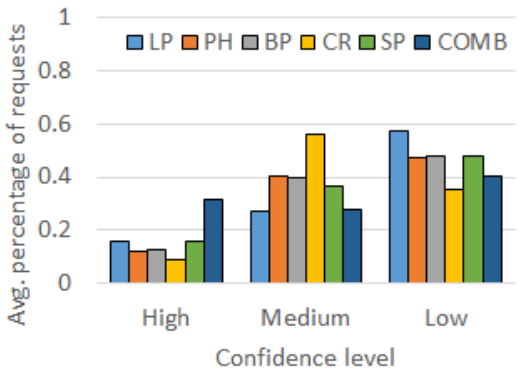
The last prefetcher in our study is the GHB, the correlation prefetcher. The main difference between this and the previous experiments is the accuracy of the stream-based predictor. With this prefetcher, the code region predictor does not work as well as the stream-based predictor. Nevertheless, the combined predictor again takes advantage of the two predictors that comprise it and achieves an accuracy of over 70% with the requests that it tags as high priority.

Figure 4.12b shows the classification of useful requests among the three levels of confidence. As we can see, the complexity of the GHB makes it harder to distinguish between them. For this reason, the percentage of useful requests classified as low confidence is higher than the number of useful requests classified as high confidence. Nevertheless, the combined predictor is the technique that manages to tag more useful requests as high confidence.

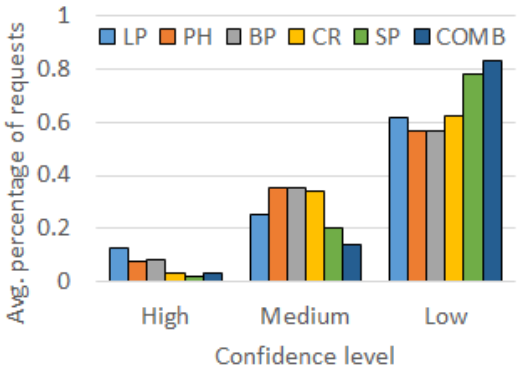
Figure 4.12c shows the percentage of non-useful requests tagged with different levels of confidence. All the techniques manage to tag most requests as low confidence. However, the combined predictor manages to classify more than 80% of non-useful requests as low confidence.



(a) Accuracy prediction.



(b) Useful requests prediction.



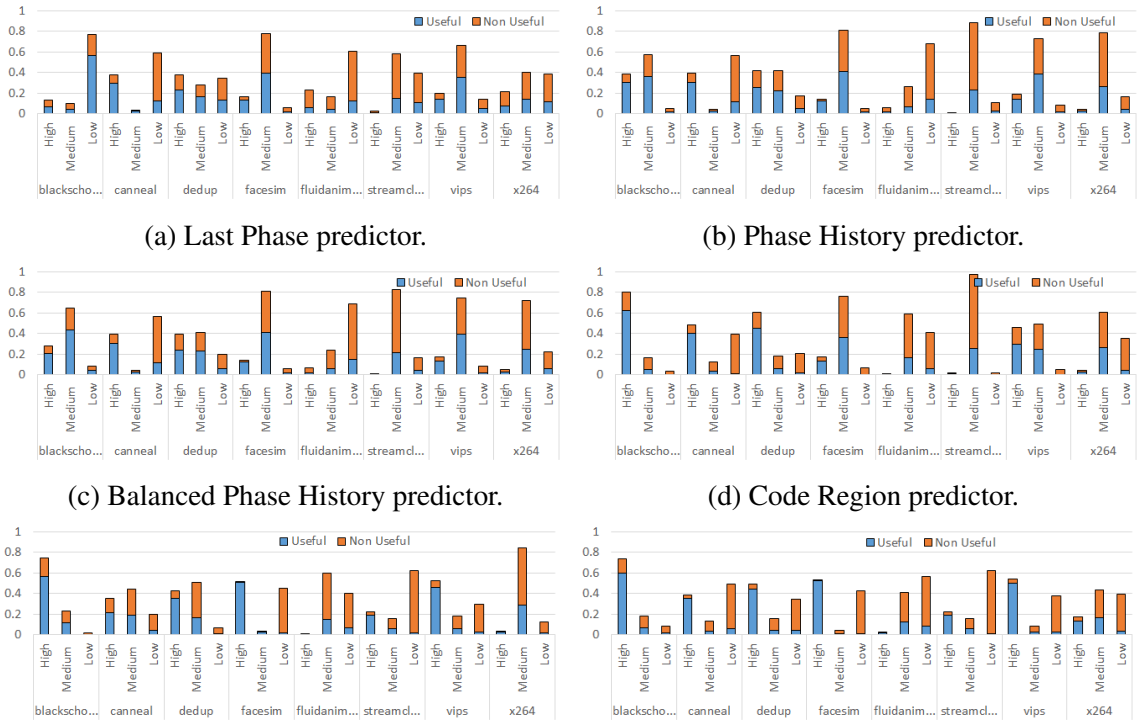
(c) Non-useful req. prediction.

Fig. 4.12 Results from evaluating the confidence predictors with the GHB as correlation prefetcher.

Per benchmark analysis

The analysis per benchmark for the GHB is slightly different from the other prefetchers that we have analyzed. To start with, as can be seen in Figure 4.13a, the Last Phase predictor is not able to predict with accuracy any of the benchmarks. It is true that in average the prediction may be better than in the other cases, but in particular none of the benchmarks has its requests classified accurately. In Figure 4.13b, we can see the predictions done by the Phase History heuristic. There is a slightly improvement of the confidence prediction in the fludanimate benchmark. Again, the Balanced Phase History predictor works in a very similar way that the Phase History. Its classification is shown in Figure 4.13c. Where we can see that more or less the result is the same as in the Phase History. The Code Region predictor manages to be more accurate. As can be seen in Figure 4.13d, in the classification of the requests of some benchmarks such as blackscholes, canneal, or dedup. Moreover, its

classification of the other benchmarks is not less accurate than in the baseline. The Stream position predictor improves the accuracy in some benchmarks but decreases the accuracy in some others compared to the Code Region predictor. Figure 4.13e shows that canneal, dedup, and fluidanimate are less accurate than the Code Region prediction. But facesim, streamcluster, vips, or x264 are more accurate. Furthermore, any of the predictions is less accurate than in the baseline. Finally, the Combined prefetcher can predict the behavior of the request with the best of the two predictors that compose the tool. Figure 4.13f shows that for all the benchmarks the predictions are more accurate.



(a) Last Phase predictor.

(b) Phase History predictor.

(c) Balanced Phase History predictor.

(d) Code Region predictor.

(e) Stream Position predictor.

(f) Combined predictor.

Fig. 4.13 Percentage of useful and non useful requests generated by the GHB prefetcher classified with high, medium, or low priority by the different predictors.

Summary of results

To summarize, we can say that the combined predictor, which is a combination of the code region and the stream-based predictors, seems to be the technique that provides more accurate predictions for all the prefetching engines. We arrive at this conclusion because the requests classified by the predictor as high priority had more accuracy than medium priority, and the requests with low priority were the ones with least accuracy. Another interesting conclusion

is that, depending on the prefetching technique, the prediction is to a greater or lesser degree successful. The code region predictor, for instance, is very accurate for stream and stride prefetchers. However, with the correlation prefetcher it is not as accurate as the generation point predictor. It is for this reason that the combined technique works with the best of both solutions and provides very accurate confidence predictions.

4.5 Conclusions

In this chapter, we have presented five new confidence predictors for generic prefetch engines. They are able to successfully work with any of the main prefetching strategies (stream, stride and correlation). We have compared the five proposals and suggested one, the combined stream-position and code-region based approach for which we propose a feasible hardware implementation, as the best solution. This mechanism provides a more accurate prediction of the confidence level of each generated prefetch request, which in terms of performance and power reduction should improve the effectiveness of most of the proposed dynamic prefetching management techniques existing in the literature, such as filtering, throttling and prioritization.

Chapter 5

Improving the Prioritization and Filtering of Prefetch Requests

> Theory is splendid but until put into practice, it is valueless.
>
> *James Cash Penney*

5.1 Introduction

As stated in the previous chapter, the success of the prefetching management techniques presented in the state of the art depends on the confidence prediction of the prefetch requests. We have proposed a novel confidence predictor and compared it with the state-of-the-art predictors, showing that our mechanism provides more accurate predictions. In this chapter we propose three dynamic management techniques that employ our enhanced 3-level confidence predictor: a filtering technique, a network-on-chip (NoC) prioritization technique and a combination of both techniques. Specifically, this dynamic manafement technique filters requests with least confidence and prioritizes high and medium confidence requests. We also compare the performance of our techniques against state-of-the-art filtering and prioritization mechanisms.

The contributions of this chapter are as follows:

- We propose a confidence predictor based filtering mechanism.
- We propose a confidence predictor based prioritization mechanism.

- We propose a combined confidence predictor based filtering and prioritization techniques.

- We provide details on the hardware implementation required for these techniques, including some changes done in the confidence predictor in order to improve its performance.

- We evaluate and compare our confidence predictor with the predictors of the filtering and prioritization techniques in their native prefetching management techniques. Moreover, we do this comparison in a realistic simulator which enables the accurate simulation of the combined effect of prefetching and the NoC in a CMP.

The rest of the chapter is organized as follows. Section 5.2 explains our filtering technique based on the confidence predictor. Section 5.3 shows how we have implemented a prioritization technique based also in our predictor. Next Section 5.4 shows how the techniques exposed in the previous sections can be combined to work together. In Section 5.5, we show the hardware modifications that should be done to implement this proposal. Section 5.6.1 presents the methodology followed to evaluate our proposal with the other ones and the results of the study. Finally, Section 5.7 summarizes our main conclusions.

5.2 Confidence predictor based filtering

In order to use our heuristic for the filtering technique, when a request is generated the filtering logic assigns a confidence to this request. This confidence is obtained from the region profiler table according to the region of code that triggered the request and its position in the stream of requests generated by the prefetcher. In line with our proposal, confidence can be high, medium or low. Whenever confidence is low, this would mean that most probably, according to our confidence predictor, the request will not be useful. Therefore, when our filtering technique is applied, the request will be discarded and not sent to the prefetching queue. As we use the region of code and stream position to decide if a request will be useful, we do not even need to compute the address of the prefetch request, but can discard it before it has actually been generated. Other prefetchers in the same stream of requests might have a higher confidence according to our predictor and thus they will not be discarded.

5.2.1 Warmup period

The filtering technique has one big drawback that is not actually addressed in the previous filtering proposals . When the confidence predictor detects a useless request and this request is filtered, it will not be issued any more. This means that its confidence prediction will no longer be updated. Hence the mechanism is not able to recover from an erroneous filtering decision. Should the behavior of the code executed by the program change for a few instructions, or the memory structures become different or any other change occur in the execution flow, some prefetch requests that were useful may became non-useful for a long enough period of time for the confidence of that generation point to go down and for the mechanism to decide that it has to be filtered. If this happens, when the program goes back to its usual behaviour, then the previous useful requests would now be filtered. In this situation we believe that it is better to launch a short useless stream and keep launching the other accurate requests with the same confidence. In fact there is only one way to recover from this situation, and this is when the filtered entry is evicted from the table. However, if it is a very active entry (corresponding to a hot region of code), it will last a long time in the filter table and will filter requests that are potentially useful.

A specific case of this problem is the instability of the confidence counters when the mechanism starts to profile a generation point. Imagine that when a region is first executed, a few initial prefetching requests associated with that region and position on the stream are unuseful but the following ones are useful. If the mechanism decides too soon that the generation point has a low level of confidence and starts to filter it, it will never generate enough requests for it to realize that the confidence of that generation point should actually be higher.

To solve this situation, we have been working with the hypothesis that a generation point, i.e. a tuple of <region of code, position in the stream of prefetch requests>, after a long enough number of iterations, will have a similar accuracy for the rest of the execution. Thus, when a new region is detected, the confidence predictor updates the information for all its stream positions during a certain period of time (warmup). When this period is over, we assume that the behaviour for that region of code has stabilized or at least that the gathered profiling information is enough representative of its long term behaviour. Then, the confidence predictor assigns one of the 3 levels of confidence to each tuple and locks them. Therefore, the prediction for that region will not be updated anymore until it is evicted from the Region Profile Table. Moreover, during this warmup period, and this is key, the filter will not discard any request.

Note that by locking the values of the Region Profile Table, we not only alleviate the problem presented at the beginning of this paragraph, but we also save power and entries

from the Prefetch Profiling Table. Note that one we have assigned confidence predictions to the stream position of a region of code, it is not necessary to profile the requests generated for that region anymore and thus, we don not need to use entries from the Prefetch Profiling Table for that.

5.2.2 Dynamic warmup strategy

For this mechanism to be effective, the time destined to the warmup period is a key parameter. For this reason, we have implemented a dynamic warmup strategy to improve the capacity to detect when the value of a generation point in a region becomes stable. As the accuracy is a value that can have small variations in small periods of time, we have counted the number of iterations that a position in the stream of generated prefetches from a region has the same confidence level. We have chosen the confidence because its variability is lower than the accuracy. Small changes in accuracy may not imply changes in confidence. Nevertheless, big changes in accuracy, will change the confidence values. For example, in a configuration where we have a warmup of 5 iterations and where the threshold to change the confidence is 20% accuracy. If we have 7 accesses to a tuple of region and position in the stream of generated prefetches composed by: 30%, 25%, 19%, 16%, 19%, 18%, and 17%. The warmup would lock that entry in this last access. Table 5.1 shows the warmup process until the lock of the entry. As we can see, in the last access, where the accuracy is 17%, the warmup is over because it reaches its limit and the region is locked with Low confidence. Note that between the 1st and the 2nd access there is a change in the confidence level, thus the warmup is reset. We have done several experiments to figure out which would be the best number of dynamic iterations with the same confidence to finish the warmup. They are shown in Section 5.6. Note that, from now on, we will use the concept of warmup iterations as this dynamic value of iterations of the warmup.

Access	Accuracy	Confidence	Warmup iterations
0	30%	High	1
1	25%	High	2
2	19%	Low	1
3	16%	Low	2
4	19%	Low	3
5	18%	Low	4
6	17%	Low	5

Table 5.1 Example of the warmup process of an entry in the Region Profiler Table.

5.3 Confidence predictor based prioritization

To make use of our confidence predictor to prioritize prefetch requests we have used a prioritization mechanism based on that described in [45]. In this technique the prioritization is based on the request being a demand or prefetch request and the level of priority of the prefetch request, combined with a starvation avoidance priority.

5.3.1 Levels of priority

In this technique, the prioritization is done in three steps. For this reason, we can consider the priority as a tuple of three values: < Time Phase, Prefetch, Confidence >. According to this tuple, the first thing that is evaluated is the time Phase (this is for starvation purposes), followed by the prefetch flag (differentiate between demand and request requests), and finally the confidence of the prefetch request (level of priority among the prefetch requests). In case that a request had exactly the same priority in these three points, the final distinction is done via Round Robbin policy.

- The first thing that will be compared is the **phase**. This refers to the time phase where a request is created. Thus a request injected to an early time phase has higher priority than a request injected in a late time phase. This priority level is evaluated so as to avoid the starvation of requests due to prioritization. As this is the first level of prioritization, we can ensure that the starvation avoidance mechanism means that all the requests will be processed at some point in time.

- The second level of prioritization is a boolean indicator. Non-**prefetch** requests are prioritized over prefetch requests. As demand operations are more critical than prefetch operations, we prioritize them. Thus prefetch requests will only be processed when there are no non-prefetch requests for the resource.

- The last level of prioritization is the **confidence** level for a certain request. This value is the confidence that the predictor has given to the request at its generation time. Requests with a lower confidence level will naturally have lower priority than requests with a higher confidence level.

5.3.2 Points of prioritization

When a prefetch request is generated, it is tagged with the priority tuple. It is then sent to a prioritized prefetch queue, which is divided into two smaller queues, one for each priority.

After that, when the request is processed from the prefetch queue, it is processed by the cache controller and injected to the network. In the network the prioritization is basically done at two points: (1) the network interface from the cache controller, and (2) the routers.

Figure 5.1 shows how the prefetch queue and the network interface work. As we said earlier, the prioritization in the prefetch queue is done by dividing the queue into two subqueues. This does not mean the prefetch queue increases in size. Instead, we use two half-sized subqueues. When picking up requests from the prefetch queue, the requests in the higher priority subqueue are picked before the others.

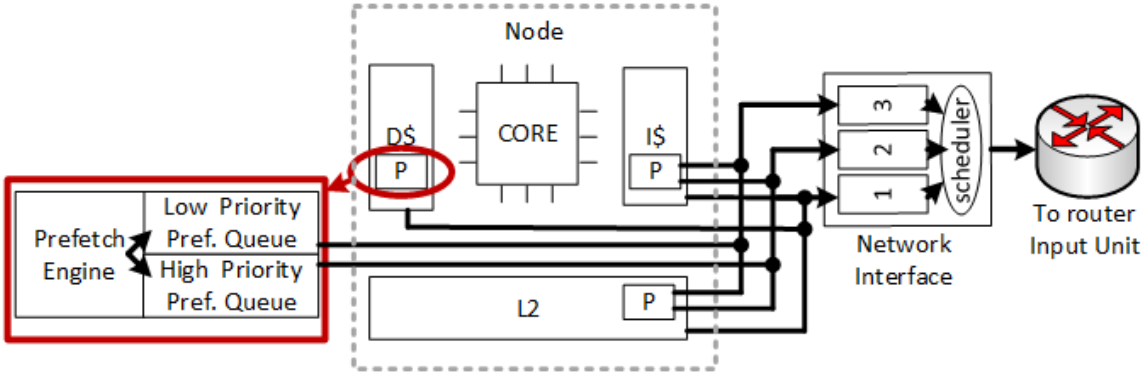


Fig. 5.1 Implementation of the prioritization in the network interface.

The network interface (NI) is the point where the requests to be issued to the network are queued to the input buffers. To process these messages the NI divides the whole message into flits and allocates them to the VCs of the input unit to which it is linked. In order to prioritize at this point, we have split the input buffer from the network interface into three smaller buffers. Each represents a different priority. The highest priority buffer is filled with the non-prefetch requests, while the other two are filled with prefetch requests in order of priority. The scheduler is responsible for choosing the highest priority queue with requests awaiting processing.

Applying prioritization in the 5-stage virtual channel router is rather more complex than in the two previous structures. As we said in Chapter 3, the router we worked with is the classical 5-stage pipeline virtual channel router. Just to recap the names, these 5 stages are: (1) buffer write and route computation, (2) virtual channel allocation, (3) switch allocation, (4) crossbar traversal, and (5) link traversal. However, prioritization cannot be applied to all stages. This is because the scheduling is done in only two stages when allocating resources. Once the resources are assigned, the flit traverses the router without stalling at any stage. These resource allocation stages are, from those listed above, (2) virtual channel allocation (VA) and (3) switch allocation (SA). In these stages the allocation is done in two steps. In the first step the flits request the resources they need and these resources are assigned. In the

second step these resources are guaranteed by breaking the conflicts via the output arbiters, where the prioritization policy is implemented.

With this information we can assume that the prioritization policy is applied in three different structures: the prefetch queue, the network interface and the router. It is important to note that, in the case of a draw where two requests have the same priority, the highest priority flit is selected using a round-robin policy. The round robin is one of the best-known scheduling algorithms. According to how the term is generally used, time slices are assigned to each process in equal portions and in circular order, with all processes handling without priority (also known as cyclic executive). Round-robin scheduling is simple and starvation-free. Moreover, prior work has shown that the area and power impact of prioritizing in networks-on-chip is small [88], [63].

5.4 Combined confidence predictor based filtering and prioritization

In the confidence predictor based filtering and prioritization techniques presented in the previous sections, the confidence predictor only had to assign two levels of confidence: high and low. For this, a threshold is defined to determine the level of profiled accuracy required to classify the generation points. This means that at a certain point in time, the ratio of useful requests versus useful plus unuseful requests is evaluated to produce a profiled accuracy for the generation point. Based on the threshold, the accuracy is translated on a high or low confidence prediction.

In the case of the combined filtering and prioritization technique, two thresholds are required to adding 3 levels of confidence: High, medium, and low. In order to carry out this prediction, the warmup technique described for the filtering mechanisms is employed. Once the generation point has been assigned one of the 3 levels of confidence, the requests corresponding to low level confidence generation points will be filtered. On the other hand, the medium and high confidence requests will be respectively assigned low and high priorities for the prioritization technique. During the warmup period, prefetch requests that would have been assigned a low confidence level, and thus filtered, are assigned, as the medium confidence requests a low priority for the prioritization mechanism perspective.

We show more details on the way this technique works in the following section.

5.5 Hardware implementation

In this section, we explain the hardware required in order to make the implementation of this technique possible. Note that some modifications were needed to be done in the hardware specified for the confidence predictor in the previous chapter. In the following points we explain this modifications. Moreover, we include a small evaluation about the overhead that this hardware would represent to the system.

5.5.1 Execution flow

In this section, the implementation is explained as an upgrade of the implementation of the confidence predictor. As in the previous chapter, we have used a block diagram to explain the behavior of the hardware for this technique. Figure 5.2 shows the block diagram tagged with the numbers that are specified in the following points:

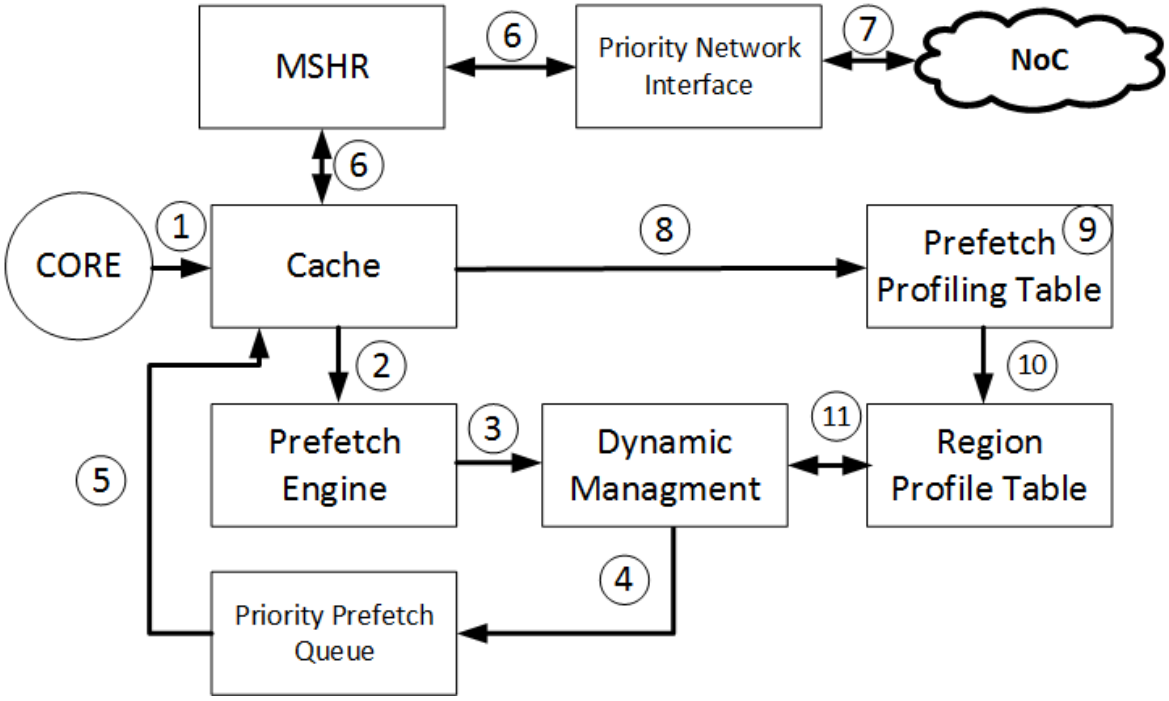


Fig. 5.2 Block diagram for the hardware implementation of the prioritization and filtering technique using the combined confidence predictor.

1. **Memory request issue**: The core requests some data to the cache memory through a demand operation, the code region of the request is identified and sent to the memory hierarchy with the request.

2. **Cache access**: The prefetcher analyses this demand operation and uses its heuristics to infer which will be the next request that the core will need.

3. **Prefetcher trigger**: When the prefetcher decides to trigger a stream of requests, the Dynamic Management logic tags these requests from the stream with a certain priority based on the confidence prediction. Remember that we have defined three levels of confidence for prefetch requests: high, medium and low. Thus, in this point, the low confidence requests will be filtered instead of being queued to the prefetch queue. If the warmup has not finished for a generation point or there is no information about that region on the Region Profile Table, the requests are assigned the medium confidence (low priority for the prioritization mechanism) and tagged to be profiled.

4. **Prefetch queue**: Each prefetch request is queued to the priority queue. Remember that these structure is composed by two small subqueues with different priorities. Each request is queued to one of these subqueues according to its confidence.

5. **Process the prefetch request**: When the cache controller does not have any demand operation to process, it peeks up the request at the head of the high priority queue and process it.

6. **Miss status holding register**: If the requested memory block is not in the cache, the prefetch request is added to the Miss Status Holding Register (MSHR) and submitted to the corresponding input buffer from the priority network interface. If the request has been tagged to be profiled, a bit is set in the corresponding entry of this structure to profile the success of the prefetch request when the memory block arrives from the memory hierarchy.

7. **Network interface**: The arbiter from the priority network interface is the responsible of applying the prioritization policy and choosing the flit from the input buffer with the highest priority (with pending requests) that will be processed. Once it is injected in the network, the rest of the prioritization is done by the routeres in the way that was specified in the previous Section.

8. **Prepare profiling**: When the miss is solved the data is stored in the cache module waiting to be requested by the processor or being evicted by other request. Moreover, if the prefetch request had been tagged for profiling, the memory line is tagged with an extra bit to indicate that this is a prefetched request, and the Prefetch Profiling Table stores the profiling information related to this prefetch request (address, region identifier, and generation point in the stream of requests).

9. **Prefetching request evaluation**: Each prefetching request is evaluated when the corresponding entry of the Prefetch Profiler Table is evicted. This eviction can be due to several reasons and according to these reasons evaluation will differ. (1) The prefetched memory line is evicted from the cache without being used: the prefetch requests is categorized as non-useful. In this case, the profiled accuracy of the corresponding generation point will decrease. (2) The prefetched memory line is used by a demand request: Once this data is used the line is marked as a non-prefetch request and the corresponding prefetch request is evaluated as useful. The profiled accuracy of the generation point that has triggered this request will increase. (3) Finally, an entry of the Prefetch Profiler Table may be evicted because it is a limited size hardware table and the entry has to be replaced by another new request. In this case, the evicted prefetch request will be evaluated as non-useful, and the profiled accuracy on the Region Profile Table for this request will decrease.

10. **Region Profile Table Update**: Each entry in this table contains the confidence related to a position in the stream of prefetch requests and the region of code. Moreover, this table is updated when there is an eviction in the Prefetch Profiler Table. Depending on the evaluation of the corresponding prefetch request, the appropriate counters will be updated to maintain the track of the profiled accuracy. However, when the warmup period finishes, the confidence in the Region Profile Table will be locked and it will not be updated anymore until it is evicted. Note that, when a new entry is allocated in the table, it always must wait for a warmup period until being locked, it does not matter if the entry was previously in the cache and it was evicted, There is always this warmup period for all the entries. Moreover, the replacement policy for this table is LRU and the use of the entries is updated when the prefetcher access the table to get the priority for a certain region.

11. **Confidence prediction**: When the dynamic prefetching management technique requires to perform a confidence prediction for a given prefetch request (given the memory region of the trigger and the position on the stream of the prefetch request to be generated), that technique will access the Region Profile Table and will get the confidence for this entry. Nevertheless, if the warmup period is not over, the confidence will be high or medium. This is done in this way, because if low priority would be provided during the warmup period, the requests may be filtered, and its profiling information would not be updated properly. For this reason, if a request would be tagged with low confidence under the warmup period its confidence is increased to medium.

5.5.2 Hardware overhead evaluation

In order to evaluate the hardware overhead required to implement the confidence predictor, we evaluate the two hardware structures that we have included.

- **Prefetch profiling table**: This table was created with the aim of reducing the hardware overhead of our technique. To implement our confidence predictor in a straightforward way, each cache entry would need to have: 4 extra bits to determine the position in the stream of generation and 26 extra bits (size in bits of the block address) to determine the generation point identifier. In a 32KB cache (with 512 blocks), this would represent a table of 1.875 bytes. The following equation shows the calculation of the table size:

$$Size = (\frac{32KB}{64bytes/block})blocks * (26+4)bits = 512 * 30 = 14336bits = 1.875bytes$$

  For this reason, we use the prefetch profiling table, which is a 64 entries FIFO table where each entry has the following bits: 26 bits per cache address, 26 bits per region identifier, and 4 bits to determine the position on the generation stream (in the worst case, the prefetcher with a longer stream is the GHB with a maximum stream of 16 requests, which can be coded in 4 bits). In total 52 bits per entry, what represents a 464 bytes sized table. The following equation shows the calculation of the table size:

$$Size = (26+26+4)bits * 64entries = 56 * 64 = 3584bits = 448bytes$$

  Note that, this new table is 3 times smaller than the naïve solution.

- **Region profiling table**: This is a 32 entry LRU table. Each entry of this table is composed of: 26 bits to code the region identifier and a set of bits to determine the useful and non useful requests of each position stream during the warmup. We have evaluated that 8 bits per useful and 8 more per non useful are enough to hold these values during the warmup. Thus, we have to multiply these 16 bits for the 16 positions in the generation stream. Finally we need 5 extra bits for the LRU. The result of this operation is a total of 1.1 KB size table. The operation can be found in the following:

$$Size = (26+((8+8)*16)+5)bits * 32entries = 287 * 32 = 9184bits = 1.1kB$$

Note that, this table is only 10% bigger than the proposed in the filtering technique. We are able to drastically reduce the number of entries in the table, because we work at basic block level instead of working at static instruction level. Therefore, with almost the same size of table, we can collect more information and be more accurate in the predictions.

Although it is not necessary the architecture and worloads that we have tested, it could be necessary to scale this design if it is implemented in other architecture or workload with a higher scale of code regions or/and memory operations that trigger the prefetcher. For this reason, we propose a technique to reduce the size of the region profile table. The idea is to use the current region profile table as a warmup table and include a new smaller table as the region profile table. This new region profile table would hold the confidence of the region once it is warmed up. This new table would be a LRU table where each entry would have only 2 bits per position in the stream of requests. With this two bits, the entry indicates the confidence level (1-3) and, if it is not ready, it can be marked with 0. In case the entry is not ready, the warmup table might be consulted for an early evaluation. This should allow the system to drastically reduce the number of entries in the warmup table.

Moreover, a part from the hardware used in the tables, it is important to note that at any time that the accuracy wants to be calculated, at least one division has to be used, and the hardware complexity of this operation is not negligible. To simplify this overhead, we propose a mechanism to perform the divisions for any threshold (from 10% to 90%) with adders and shifters. The formula used to evaluate if a certain accuracy is greater than a certain threshold is the following:

$$Threshold > \frac{Useful}{Useful + Useless}$$

If this operation is simplified, we can easily eliminate the division:

$$(Threshold * Useful) + (Threshold * Useless) > Useful$$

Finally we can join the variables and:

$$Threshold * Useless > Useful - Threshold * Useful$$
$$Threshold * Useless > (1 - Threshold) * Useful$$

In this way, we substitute the division for two multiplications. However, two multiplications are still an important overhead. For this reason, we apply another simplification

to substitute the multipliers for shifts and adders. To apply this simplification, we need real threshold values. In the following example, we will use a threshold of 70%, which according to Table 5.2 is one of the thresholds that has more hardware requirements. Using this threshold in the previous equation, we can simplify it in the following way:

$$0.7 * Useless > 0.3 * Useful$$
$$7Useless > 3Useful$$
$$(2^3 Useless - Useless) > (2^1 Useful + Useful)$$
$$((Useless << 3) - Useless) > ((Useful << 1) + Useful)$$

According to this result, the division and the addition required to calculate the accuracy can be substituted by two right shifts and two adders. Which is negligible cost, compared to the cost of the divisor. Table 5.2 shows the hardware requirements for all the thresholds from 10% to 90% and the formulas used to evaluate them.

Threshold	**ADD**	**SHIFT**	**CMP**	**Simplified Formula**
10%	1	1	1	$((U << 3) + (U)) > (N)$
20%	0	1	1	$(U << 2) > (N)$
30%	2	2	1	$((U << 3) - (U)) > ((N << 1) + (N))$
40%	1	2	1	$((U << 1) + (U)) > (N << 1)$
50%	0	0	1	$(U) > (N)$
60%	1	2	1	$(U << 1) > ((N << 1) + (N))$
70%	2	2	1	$((U << 1) + (U)) > ((N << 3) - (N))$
80%	0	1	1	$(U) > (N << 2)$
90%	1	1	1	$(U) > ((N << 3 + (N)))$

Table 5.2 Required hardware to calculate the accuracy according to the different thresholds. U: Useful, N: Useless

5.6 Performance evaluation

In this section, we evaluate the performance of our proposals. We have divided the performance analysis into several subsections with different types of experiments. As we are testing our confidence predictor with two existing techniques, we have done experiments to test our heuristic with the ones defined in the literature. Thus, we have divided the experiments in three parts: (1) Filtering experiments, (2) Prioritization experiments, and (3) Combined filtering and prioritization experiments.

5.6.1 Experimental framework

As the other experiments in this thesis, the system simulated in this performance evaluation consists of a tiled mesh with private first level data and instruction caches, and a shared L2 cache. L2 and L1 are non-inclusive. The coherence protocol employed is the MOESI CMP directory. We have tested our technique using one of the most common stride prefetcher. We have chosen this kind of prefetcher because we have implemented the prioritization in the first level cache and these prefetchers are the most commonly used in this cache level [39] [52]. In fact, the prefetcher used for this study has been the RPT [77] prefetcher. The hardware specifications are shown in Table 5.3.

Hardware specifications	**Values**
ISA	x86
Tile number	16
L1 Instruction cache	16Kb each tile
L1 Data cache	32Kb each tile
L2 size	8MB
Network	Garnet
Topology	Mesh
Virtual Networks	3
VCs per virtual network	2
Bandwidth factor	16 bytes
Data paquet size	80 bytes
Ctrl paquet size	16 bytes
Prefetcher cache level	L1
Simulated cycles	250M of cycles
Prefetcher Profiler Table	64 entries
Region Profiler Table	32 entries
Lower accuracy threshold	20%
Upper accuracy threshold	60%
Code region granularity	Basic Block

Table 5.3 Simulation specifications.

5.6.2 Implementation of the state of the art heuristics

In this subsection, we explain how we have implemented the heuristics described in the state of the art for each of the techniques exploited in this chapter. Note that, these heuristics may not be implemented exactly in the same way than they were implemented in their original proposals. Nevertheless, our implementations are inspired in the descriptions we found in

the papers they were published. Therefore, in the following subsections we describe the way we have implemented them.

Filtering heuristic

The heuristic of the Filtering technique, is based on the typical 2-bit branch predictor. It consist of a 4096 entries table indexed by the static address of the instruction that has triggered one or several prefetch requests (what represents a table of 1KB). Each entry of the table has a two bit counter. When evicting a prefetch request from the cache the static address that triggered this prefetch request is used to access the table. If the request has been useful, the counter in that entry is increased, if not, it is decreased.

When a new prefetch request is generated, the static address that generated this request is used to access the table and get its related value. If the value is 0, the request is filtered (or discarded) thus, it is not injected to the prefetch queue. If the value is greater than 0, the request is pushed to the prefetch queue.

Although in the paper the authors does not mention the replacement policy, we think that the most reasonable policy to use is the Last Recently Used (LRU) policy. The reason is that we think that the objective of this table is to have the static addresses that are most frequently used. Therefore, the most common replacement policy that holds the active entries a longer time in the table is the LRU.

Prioritization heuristic

The heuristic of the Prioritization technique is an heuristic that tries to predict the accuracy. According to the predicted accuracy, a priority is given to each request. Moreover this heuristic is based on time phases. The whole execution time of the application is divided into phases of a certain number of cycles. The idea of this heuristic is that in the majority of cases, the average accuracy calculated in a phase will be very similar to the accuracy in the next phase. Therefore, when the execution enters to a new phase, the accuracy from the previous phase is calculated. This value is used as the prediction of accuracy for all the requests that are generated in this new phase.

To determine the priority we need a threshold to distinguish between high and low accuracy. The authors does not talk about any concrete threshold. We did a small sensitivity study to find this threshold and we fixed this value to 20%. Thus, the requests with predicted accuracy higher than 20% are high priority. While, below 20% their priority is low.

5.6.3 Filtering experiments

In this subsection, we will evaluate the performance of our confidence predictor based filtering technique. However, before evaluating it, there are some aspects that should be analyzed. The first variable that we will evaluate in the following points is the potential of the accuracy increment for each confidence predictor. Note that if we filter requests with low level confidence, which are expected to result in non useful prefetches, the resulting level of accuracy of the prefetching technique should improve. After that, we made a study to estimate the optimum period of warmup for each region. Finally, we compare the filtering technique from the state of the art with our technique.

Estimated accuracy increment

To calculate the estimated increment of accuracy, for each predictor, we have calculated the accuracy without filtering any requests, and the accuracy when filtering the low confidence requests according to each predictor. Note that, this experiment has been done without any hardware restriction. For this reason, the difference between them gives us the theoretical accuracy that the filtering technique with a certain predictor should obtain. The predictors that we have compared are: Last Phase (LP), Filtering state of the art technique (FT), Code Region (CR), Stream Position (SP), and the combined of Code Region and Stream Position (COMB).

Figure 5.3 shows the estimated accuracy for the predictors after filtering the low priority requests from the RPT prefetcher. All the heuristics improve their accuracy compared with the baseline. Moreover, the accuracy predicted for the baseline heuristics is always below the accuracy predicted for the combined (COMB) technique which is our proposal. A better accuracy should result in a more efficient technique (less cache pollution, traffic in the network, and power to issue and resolve the unuseful requests). For this reason, we expect that with this prediction, our technique will improve the current state of the art technique.

Stability analysis

In order to get the best warmup period value for the filtering analysis, we have done the following experiment: We have counted the percentage of wrong predictions done after locking a region for several warmup values. We consider a prediction to be wrong if the predicted value is not the same as it would be predicted without locking the the region.

In Figure 5.4, we see the results of this experiment. We see that, the longer the warmup period is, the less number of wrong predictions are done. Nevertheless, for values longer than 50 warmup iterations, the number of wrong predictions increases. This is because

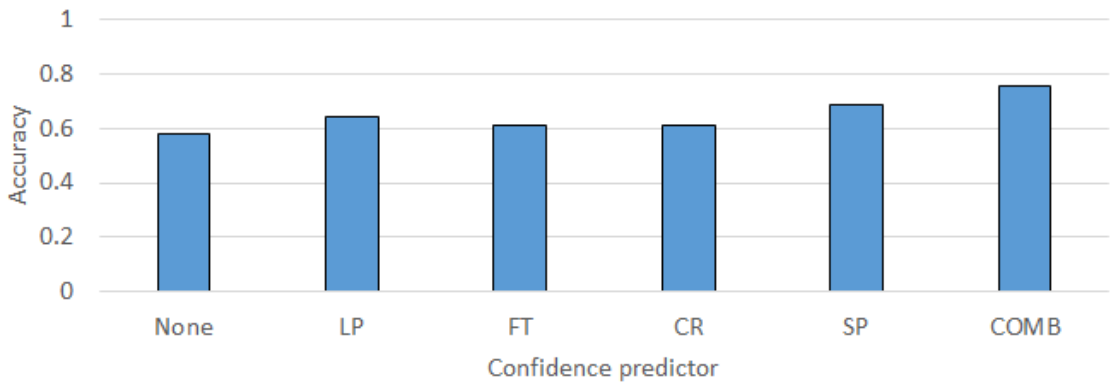


Fig. 5.3 Accuracy increment estimation after filtering the low confidence requests.

during the warmup period, the low confidence requests are always predicted as medium confidence requests. At some point in time, the region profiling is enough stable to take right decisions. Thus, forcing the low confidence predictions to be treated as medium confidence increases the number of wrong predictions. As we can see in the figure, this value is around 50 iterations. For this reason, we have selected this number as the optimal value for the parameter of our dynamic warmup technique.

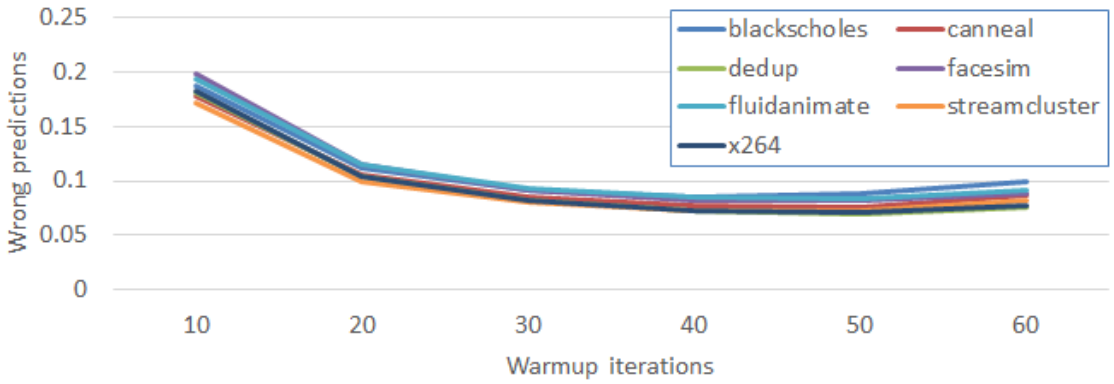
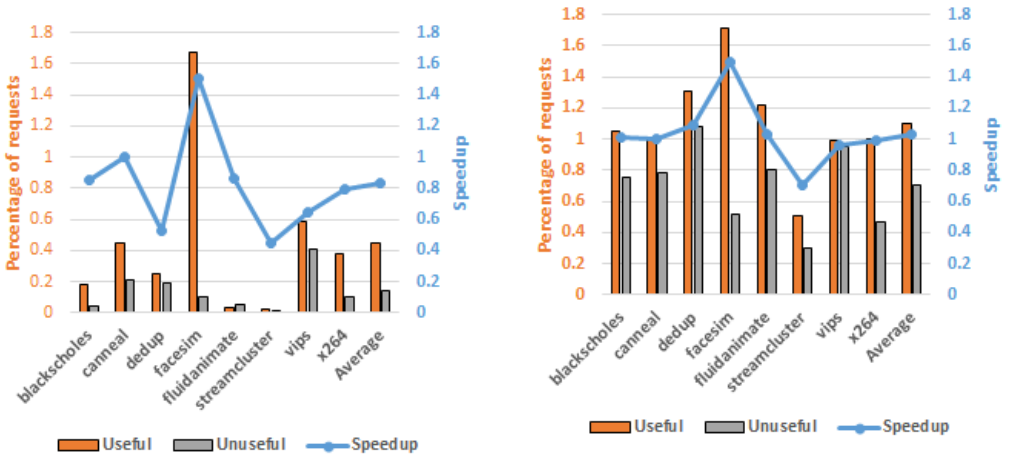


Fig. 5.4 Percentage of regions that change their confidence after the warmup.

Filtering performance evaluation

In this subsection, we compare the performance of our combined confidence predictor based filtering technique with the filtering technique from the state of the art. Figure 5.5a shows the filtering results when simulating both mechanisms. To get this figures, we have done an initial execution without filtering and another one with the filtering activated. The bars from the

figures represent the average percentage of prefetch requests issued with the filtering activated respect the average number of prefetch requests issued without filter (100% indicates the same number for both runs). The line represents the IPC speedup comparing the filtered simulation with the non filtered one (more than 100% indicates a higher IPC with filtering). In Figure 5.5a we can see the results for the baseline mechanism. We can see that the baseline heuristic is able to reduce a lot the prefetching requests. It manages to eliminate about 87% of the non-useful requests in average. However, due to the filter aggressiveness of this heuristic, it also filters a lot of useful requests. In average, this heuristics filters more than 56% of the useful requests, what have an important impact on the speedup. As we can see, the IPC is reduced in around 18% in average. Another important observation that can be done regarding this figure is about facesim benchmark. As we can see, the number of useful requests is about 167%, this means that the execution with filtering issues more useful prefetch requests than the non filtered one. The reason for this is found on the behavior of the prefetcher. When the prefetcher works more accurately it is able to issue more requests. The reason for this is in the behavior of the RPT. If the requests issued by the RPT are not useful, it is auto-regulated and it reduces the number of requests issued for a certain static address. If these requests are useful the reduction is not applied and they are issued. Moreover the filter will let them pass, and consequently we can have results like the facesim ones, which are directly transformed in important speedup improvements. On the other hand, there are benchmarks such as fluidanimae or streamcluster which percentage of filtered requests is nearly 99%, which is directly reflected in negative speedups.



(a) Baseline (state of the art).

(b) Combined confidence predictor based technique.

Fig. 5.5 Results for the filtering technique: Filtered requests vs. IPC speedup.

On the other hand, the results for the filtering technique using our combined heuristic methodology are shown in Figure 5.6. As we can see, our technique is much less aggressive than the baseline one, However, we reduce the non-useful requests by 30% and in average the prefetcher is able to increment the useful ones in 10%. This effect is directly translated into an average 3% of speedup. We can see that our technique is able to reduce the non-useful requests in all the benchmarks except dedup and allow the prefetcher to increase the useful requests in most of them. This big difference between the baseline and our proposal is because of the increased effectiveness of our combined confidence predictor and the dynamic warmup methodology that makes our technique more flexible and capable of adapt itself more efficiently to the program behaviour.

5.6.4 Prioritization experiments

In this subsection we analyze the performance provided by our confidence predictor based prioritization mechanism and compare it with the performance of the state of the art mechanism.

In order to compare the effectiveness of both prioritization techniques, we have done, like with the filtering case, first a simulation without prioritization, and then we have compared it to the state of the art technique and our combined heuristic. Figure 5.6 shows the results for this experiment. As we can see, the performance is very similar for all the benchmarks. However, the combined technique gets an 8% of speedup in average while the baseline technique is about 6%. These results shows that prioritization has a small effect in the performance of prefetching. For this reason, we want to combine this technique with filtering in order to improve the global results. This is what is shown in the next subsection.

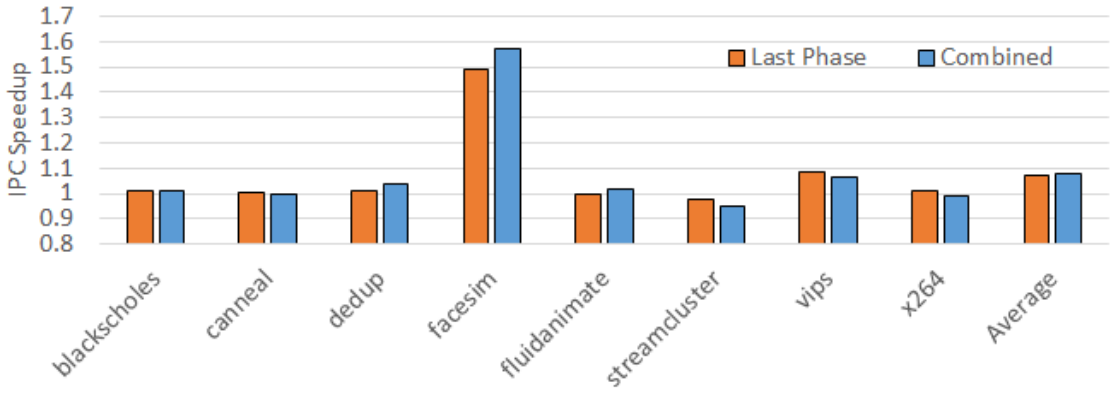


Fig. 5.6 RPT IPC Speedup after applying prioritization.

5.6.5 Combined filtering and prioritization experiments

In this subsection, we show the evaluation of our confidence predictor with the combined technique of filtering and prioritization.

Figure 5.7 was obtained by doing a first simulation without the filtering and prioritization techniques and compare its result with the same simulation with prioritization and filtering. In this figure, we can see the same trends that are shown in Figure 5.5. Actually the speedup is almost negligible. The reason for this small speedup increment, is that although the both techniques can be applied in parallel, their performance does not increase in the same proportion. This is because both techniques tries to solve similar problems. Thus, as the filtering reduces the congestion in the interconnect, the impact of the prioritization is smaller. Nevertheless, the speedup is higher in all the benchmarks and in average the combined technique is 1% faster than the filtering technique alone.

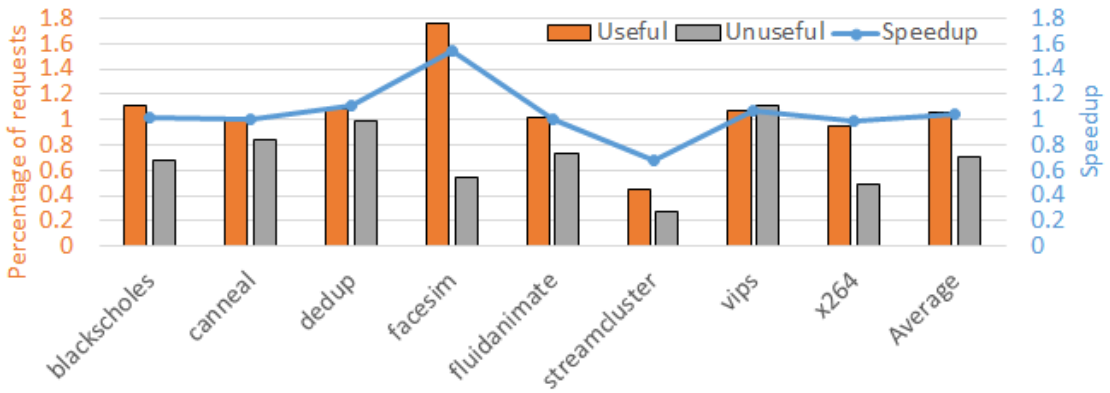


Fig. 5.7 RPT Request reduction vs. IPC speedup .

In order to evaluate the effect of the hardware limitations, we compare the accuracy of our combined technique with the predicted accuracy from the Figure 5.3. Remember that, in the experiment of Figure 5.3 there is where no hardware limitations. Figure 5.8 compares the accuracy of applying our technique with the hardware restrictions (basically the number of entries of the several tables) and the idealized case without hardware restrictions. We can see that our technique manages to get 65% accuracy in average. In Figure 5.3, we saw that the potential of this technique was about 73%, what means that we only lose less than 7% of accuracy due to the hardware limitations.

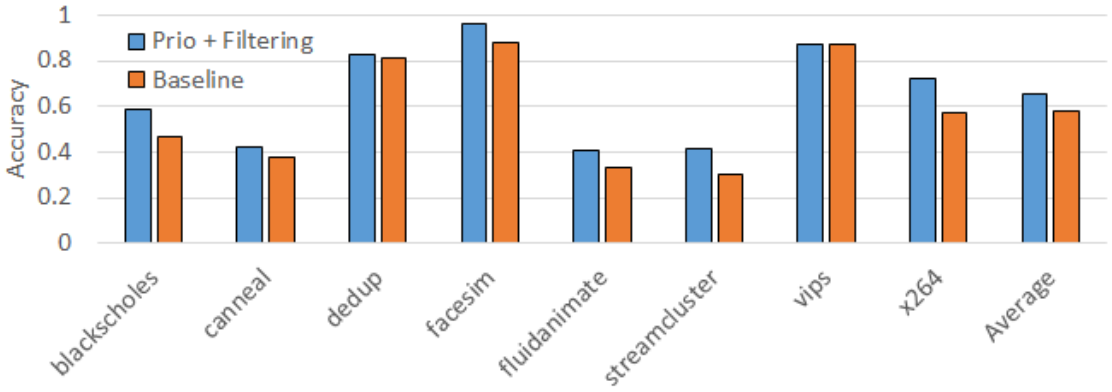


Fig. 5.8 RPT accuracy increment.

5.7 Conclusions

In this chapter, we have shown the potential of our confidence predictor heuristic applying it to two techniques that manages the prefetching according to their confidence, filtering and prioritization. We have proposed a feasible hardware implementation for these techniques and we have evaluated their performance by comparing them with the techniques from the state of the art.

In the results, we have seen that our technique is more accurate than the one from prioritization. It is able to do better confidence predictions and gets more speedup when prioritizing. We have also seen that the current filtering technique is very aggressive, what filters many requests. Our technique manages to reduce the filtered requests without generating slowdown. The combined technique improves both, the prioritization and the filtering techniques, what means that both techniques can work together successfully. Moreover, we have seen that the hardware limitations of our technique had a negligible effect in the performance obtained.

Chapter 6

Prefetching Challenges in Distributed Shared Memories for CMPs

> It's lack of faith that makes people afraid of meeting challenges, and I believed in myself.
>
> *Muhammad Ali*

6.1 Introduction

Processor design techniques have evolved toward architectures that implement multiple processing cores on a single die, commonly known as chip multiprocessors (CMPs). However, in those chips, the network on chip that connects all the cores with only one unified memory, becomes a new bottleneck. When incrementing the number of cores, the contention in the port of the unified memory and the time required to send a request from one core in the chip to the memory port, makes the solution of the unified memory hierarchy not the best choice. For these reasons, when the number of cores grows, the idea of a unified memory hierarchy becomes unfeasible. In this way, as it is shown in Figure 6.1, future multi-core CMPs with tens (or even hundreds) of processor cores will probably be designed as arrays of replicated tiles (with shared and distributed memories), connected over an on-chip switched direct network [78]. These tiled architectures are reported to provide a scalable solution for managing design complexity and the effective use of resources in advanced VLSI technologies.

As stated before, prefetching works as a latency-hiding strategy, as well as multithreading does. For this reason, the potentiality of prefetching is lower when we combine both

techniques. Nevertheless, there is already a lot of work to do in multithreading. Nowadays, performance does not scale proportionally to the number of threads. For these reason Surendra Byna et al. in [10] shows us that smart prefetching mechanisms can already improve performance in multithreading processors.

As shown in Figure 6.1, the most common memory hierarchy organization of these architectures lies on one or two private cache levels per tile and, as a last level of cache, a Distributed and Shared Memory (DSM). This DSM holds a banked organization with one bank on each tile. Moreover, each cache block-sized unit of memory is statically mapped to one of the banks based on its address in an interleaved way.

In these systems, if prefetching is allocated on a DSM cache level, the prefetcher also has to be distributed in such a way that each tile of the CMP contains a prefetch engine. In a straightforward implementation derived from unified memory systems, these prefetch engines would work independently from each other by analyzing the set of memory accesses

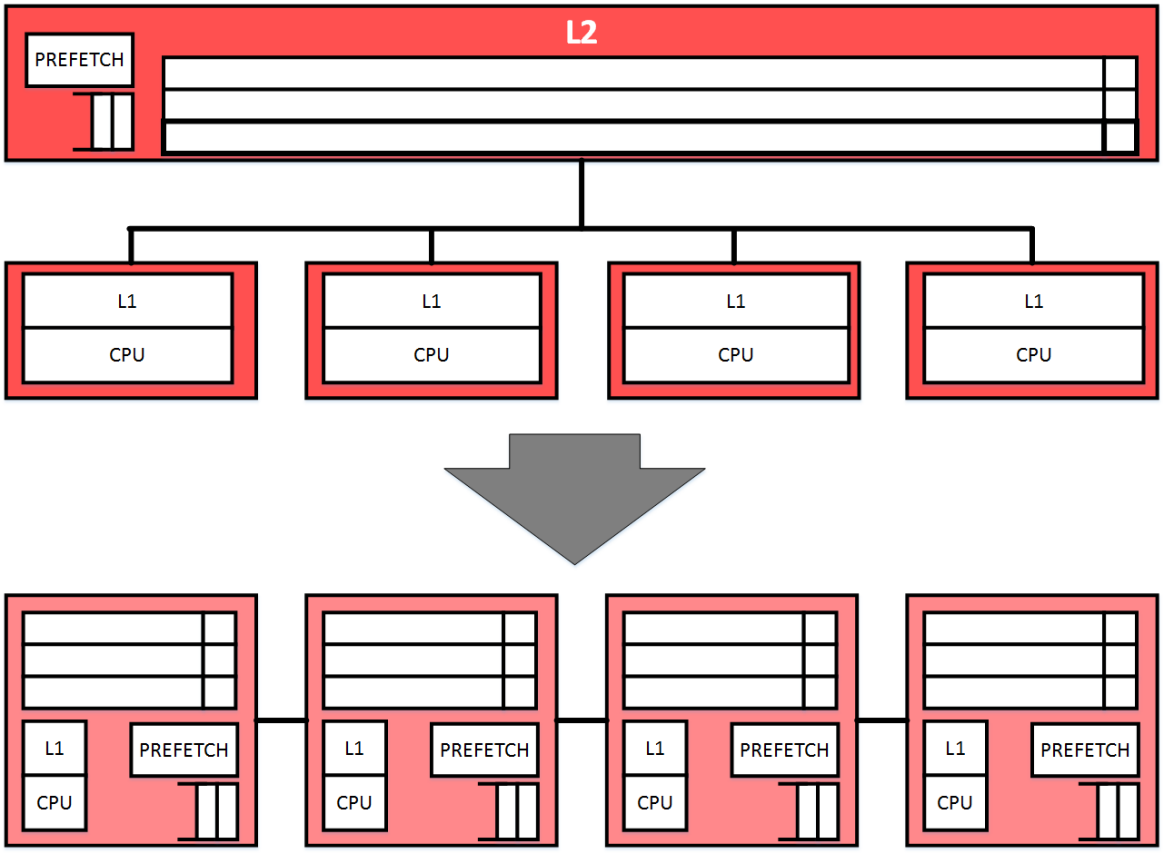


Fig. 6.1 Top: CMP with unified memory hierarchy. Bottom: Tiled CMP architecture with distributed and shared memory system.

that arrives at the attached piece of cache and potentially generates prefetching requests targeted to any other tile on the system depending on the target address.

We have observed that this distributed behavior entails several challenges that are not present when the cache is unified. These challenges are: (1) **pattern detection**, (2) **prefetching queue filtering**, and (3) **dynamic profiling**.

- **Pattern detection** derives from prefetchers that attempt to predict the future memory access pattern only from the partial view of the accesses that are addressed to their tiles.
- **Prefetching queue filtering** is related to local prefetching queues that attempt to filter redundant requests without having global knowledge of generated requests.
- **Dynamic profiling** refers to the difficulty of obtaining dynamic statistics, such as prefetching accuracy, for a specific prefetching engine by analyzing the behavior in a memory bank that receives requests from various prefetching engines.

These challenges will affect the effectiveness of prefetching, the overheads (in terms of unnecessary requests traversing the system), and the usability of dynamic control techniques that rely on prefetching statistics. Although several approaches [43] [19] [22] are related to prefetching in CMPs, all of them focus on the unified cache levels of the CMP memory hierarchy. In this chapter, we identify, analyze, quantify, and hint on how to face the effects of these three challenges.

The rest of the chapter is organized as follows. Section 6.2 presents the challenges studied in this chapter. Section 6.3 shows the modifications performed in the simulator to evaluate the challenges. Section 6.4 describes the evaluation framework and quantifies the impact of these challenges. Section 6.5 provides some research directions for solutions to the challenges. Finally, Section 6.6 summarizes our main conclusions.

6.2 Challenge identification

In this section, we identify the challenges when incorporating prefetch mechanisms into a DSM. To better describe the problem, we divide the work done by a prefetcher into three phases: (1) analysis, (2) request generation, and (3) evaluation. Figure 6.2 shows four tiles, which represent the behavior of the prefetcher in each phase. Note that we have tagged the arrows with a number that represents the behavior in each phase.

1. **Analysis phase**: The prefetcher collects the information related to the memory accesses and analyzes it. To do this, most kind of prefetchers use internal structures

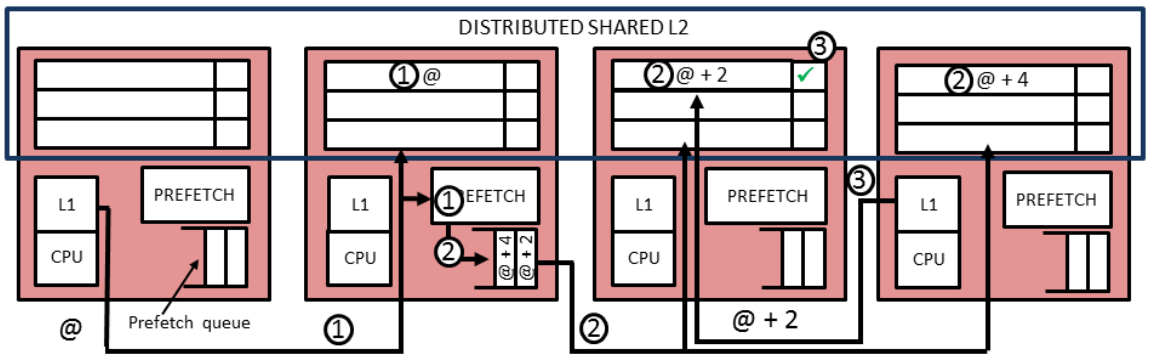


Fig. 6.2 Phases from the prefetcher in a distributed memory system: (1)analysis, (2) request generation, and (3) evaluation.

that are filled with information related to the memory accesses. The analysis then determines whether prefetch requests must be generated. If prefetch requests have to be generated, the analysis phase is responsible for deciding the number of requests that have to be generated and also the blocks that must be prefetched.

2. **Request generation phase**: At the beginning of this phase, the prefetcher generates the requests decided by the analysis phase. In this generation phase, the requests are queued into the prefetch queue. Before being queued, the prefetch queue checks whether the requests are already in it. If they are, they are merged and not queued. Once in the queue, the requests will wait until the cache controller picks them up from the queue to issue the requests to the memory hierarchy.

3. **Evaluation phase**: This phase typically takes place when the memory block that a prefetch request has brought to the cache is evicted (making the actual end of a request life cycle). At that moment statistics are taken dynamically for each request. This profiling information is used in the analysis phase by some dynamic techniques to modify the variables that the prefetcher is working with.

In the following, we explain the challenges we have detected. Notice that each challenge directly affects one of the previously described phases of the prefetcher. We can therefore say that our study covers all the phases of any prefetching mechanism.

6.2.1 Pattern detection

This challenge appears in the **analysis phase**. Many prefetchers usually save the history of memory accesses in internal structures in order to analyze the stream of accesses and, with

this information, generate the prefetching requests. The problem is that the prefetcher is distributed in the same way as the memory. This means that each prefetcher can only be aware of a certain part of the stream of accesses. Figure 6.3 shows an example of how a stream may be distributed. Notice that prefetchers that work in unified memory are able to analyze the full memory pattern. However, as we can see, in DSM each prefetcher can only be aware of a certain part of the pattern. This distribution does not allow the prefetcher to make accurate analyzes, so the prefetcher may not work properly (it may launch fewer requests and/or be more inaccurate).

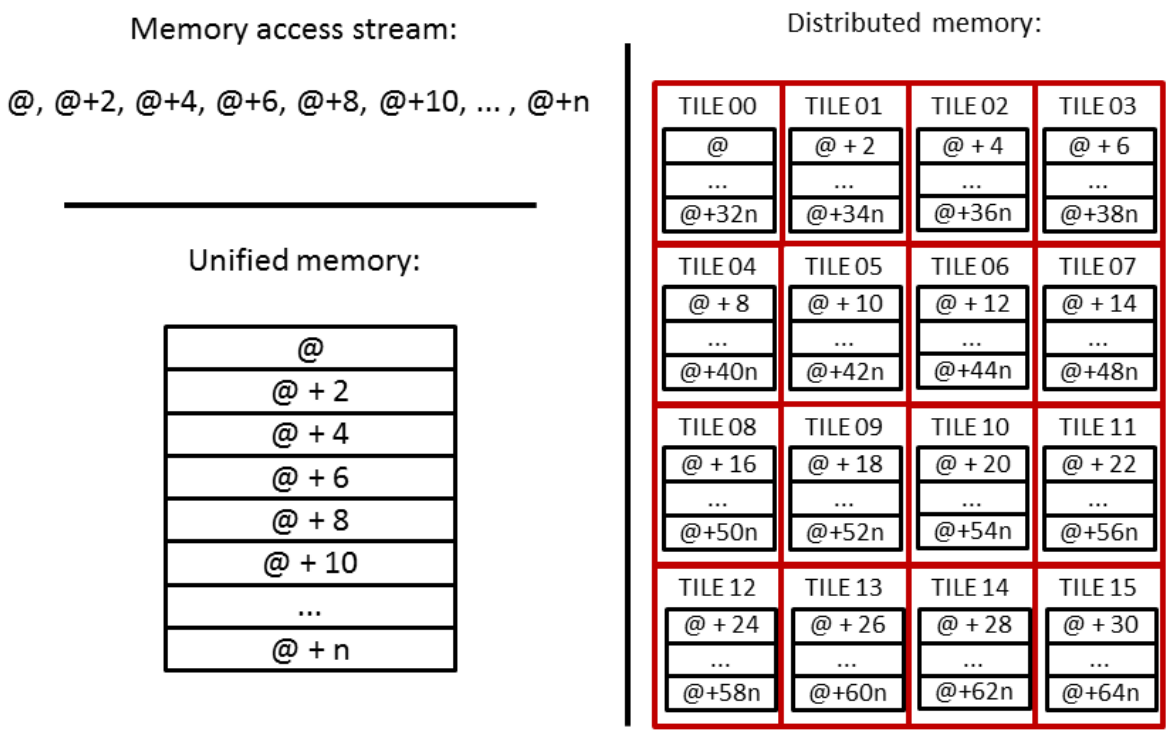


Fig. 6.3 A possible distribution of a memory accesses stream: on the left in a unified memory and on the right in a distributed and shared memory.

6.2.2 Prefetching queue filtering

The prefetching queue filtering challenge appears in the **request generation phase**. In a unified memory system, there would be only one prefetch queue in which all the prefetch requests can be queued. The merging operation implemented on this queue may filter many recently repeated prefetch operations. However, if these operations are not filtered, they will be sent to the tile where they are addressed, which will waste power and increase the network contention. Moreover, once they are in the tile they will most probably realize that the data

is already in the memory. In the DSM, instead of only one prefetching queue, there is a prefetching queue in each tile. This means that repeated prefetch requests may exist in several prefetching queues. However, as there is no communication between the prefetching queues, these requests will not be filtered. This wastes power and increases network contention.

6.2.3 Dynamic profiling

This challenge appears in the **evaluation phase** of the prefetcher. When the prefetcher attached to a tile starts generating requests, these requests may be addressed to any other tile in the system depending on the address. This means that the statistics related to a prefetch request are not collected in the same tile where this prefetch request is generated. For this reason, the statistics for accuracy, lateness, or pollution that a tile is recording are not related to the prefetcher allocated in this tile. These statistics are in fact a collection from the prefetch requests received by this tile from the prefetchers in all of the tiles. This means that if the prefetcher or some management technique needs to use these dynamic statistics and collects the statistics in its tile, they will most probably be inaccurate and the decisions taken based on them may be wrong. Note that, this challenge have a direct impact on the collection of all the metrics. For this reason, it is sufficient to analyze only one of them. In our case, we have focused on accuracy which we believe that is the most important one.

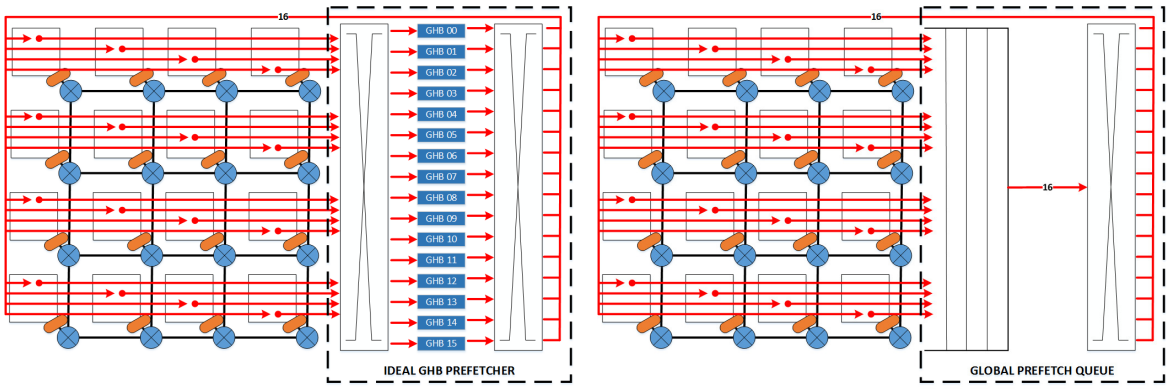
6.3 Challenge evaluation methodology

The specific details are described in the next section. As in the other chapters, the prefetchers that we have used are some of the most representative ones found on the literature, the Tagged prefetcher [24] and the GHB [59]. In the following subsections, we explain the modifications performed to the simulator in order to quantify each challenge.

6.3.1 Pattern detection

To quantify this challenge, we developed an ideal prefetcher. This ideal prefetcher is centralized, gathering information from all the caches in a single set of shared structures and trying to recover the memory correlation from each processor. This prefetching system monitors the memory accesses in any cache module at the time they are generated and without generating any congestion in the network. To guarantee this, each tile has a direct connection with the prefetcher as shown in Figure 6.4a. Note that, the correlation and the patterns that the prefetchers try to predict are in the order that the core is executing the memory instructions. However, the DSM is receiving requests from all the cores at the same

time without any order. For this reason, the centralized prefetcher would not be able to recover the correlation from the memory access stream. To solve this problem, the ideal prefetcher classifies each memory request, according to its execution core. As it is shown in Figure 6.4a, there is one prefetcher per each core, thus when a memory operation is detected, it is assigned to the prefetcher related to its execution core. When a prefetch request is issued, it is queued in the prefetch queue of the tile where is has to be placed, so no extra operations are needed. When a controller is able to issue a prefetch operation, it will check if there is any operation to issue in its prefetching queue. If so, the prefetch operation will be issued. Note that, as the prefetcher is ideal, the costs in terms of time or power of all these connections and switches are not taken into account. Moreover, the prefetch mechanism we used to quantify this challenge is the GHB prefetcher. Note that the tagged prefetcher does not use the stream of memory accesses and is therefore not affected by this challenge.



(a) Diagram of the ideal prefetcher.

(b) Diagram of the global queue.

Fig. 6.4 Diagram of the evaluation infrastructure of the pattern detection and the queue filtering challenges.

6.3.2 Prefetching queue filtering

To quantify this challenge, we implemented in the simulator a global buffer that holds the list of pending requests in each of the prefetching queues. With this centralized information, when queuing a request, the prefetch queue can detect whether this request is in any other tile and merge it. Obviously, this buffer is used only to quantify this challenge. For this reason, the communication between the buffer and the tiles is done instantaneously and without congesting the network. Figure 6.4b shows how the requests generated by the prefetchers in the cores are sent to the global prefetching queue. The mechanism to issue the prefetch operations is the same as in the previous implementation. To quantify this challenge, we count the number of requests that are not filtered due to the dispersion of the prefetch queues.

We assume the tagged prefetcher because it is the one that is unaffected by the pattern detection challenge.

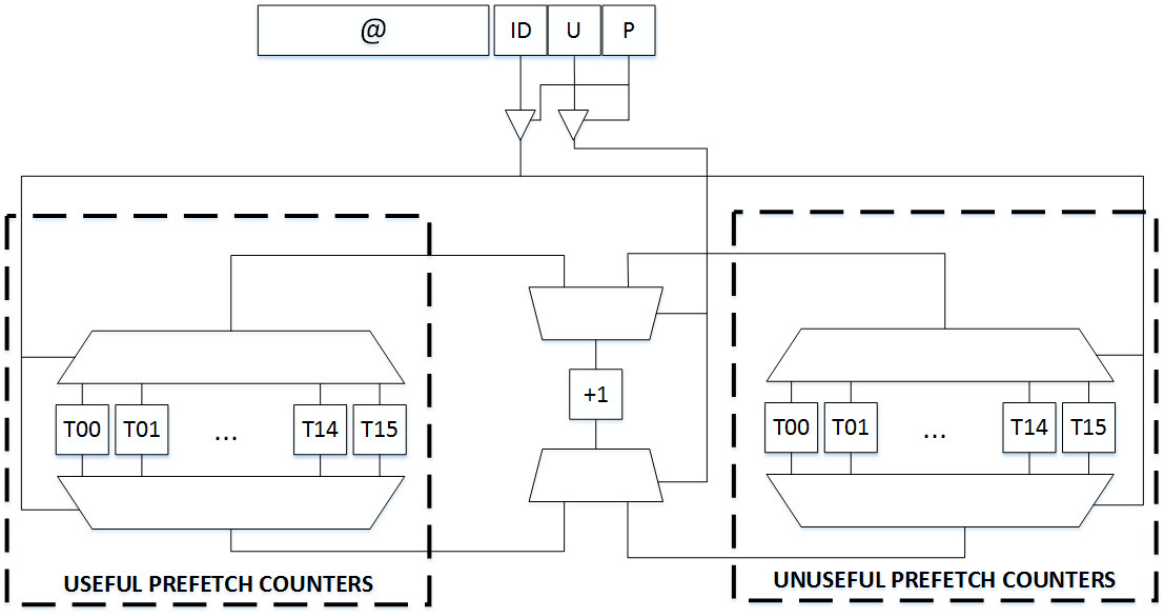


Fig. 6.5 Implementation of the useful prefetch. ID: bits to track the issuer of the prefetch. U: bit to track if the prefetch is useful or not. P: bit to track if this has been a prefetch line or not. T00 to TN: accumulators to track the useful or unuseful prefetch from each tile.

6.3.3 Dynamic profiling

To quantify this challenge, we included two new arrays of statistics for each prefetch engine in the simulator. This set of statistics counts, for each tile, the useful and non-useful prefetches issued by all the prefetchers to this tile and preserves the information of the tile that has issued the prefetch request. Figure 6.5 shows the implementation of these counters in each tile. Note that when a cache line is evicted from the cache, its prefetching associated information is checked. If the line was a prefetched line (P bit), it will become an useful or unuseful (U bit) prefetch. Then, according to the issuer of the operation (ID bits, where the number of bits depends on the number of cores) the information is going to be aggregated in its corresponding accumulator (T00 to TN, where N is the number of cores). By aggregating the information from all the tiles, we can obtain dynamically accurate information about the behavior of each specific prefetch engine. To quantify the absolute error of the accuracy, we have compared two values in each tile.

- The first one is the one that we have already commented, which is the realistic value of the accuracy in each tile.

- The second value is the result of aggregating all the values from T00 to TN in the same tile. This would be the value obtained without doing any modification in the system. However, as stated before, this second statistic accounts for requests from different prefetches and is not representative of any one in particular. For this reason this is not an accurate value.

We use the tagged prefetcher not the GHB prefetcher because, due to the pattern detection challenge, the latter is irrelevant.

Hardware specifications	**Values**
ISA	x86
CPU model	TimingSimple
Number of tiles	64
L1 Data cache size	16KB per tile
L1 Instruction cache size	16KB per tile
L2 Unified cache size	16MB
Network	Garnet
Topology	Mesh
Prefetcher cache level	L2
Simulated cycles	350 millions
Tagged prefetch aggr/dist	2/2
GHB depth/width	4/4

Table 6.1 Simulator specifications.

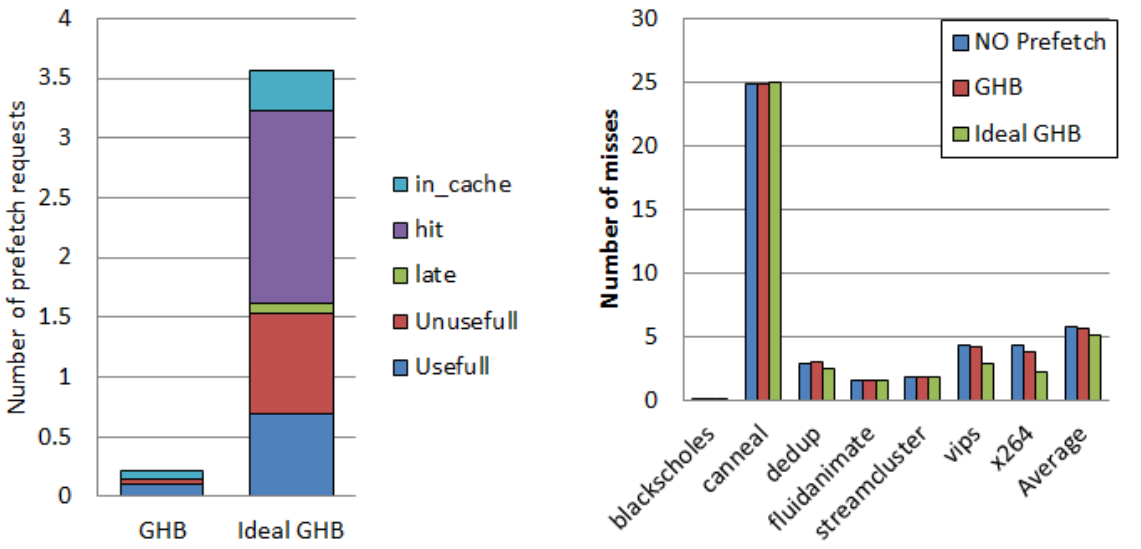
6.4 Challenge analysis and quantification

The framework consists of a mesh composed by 64 tiles with private first level data and instruction caches, and a shared L2 cache. The L2 cache comprises several banks and each of these banks is associated with a local tile. Each memory address is deterministically associated with a given L2 location. This means that there is no replication in the L2 cache, though a given block can be in several L1 caches. L2 and L1 are not inclusive. The coherence protocol employed is the MOESI CMP directory. The prefetchers used to quantify the challenges are the tagged prefetcher [24] and the GHB [59]. The hardware specifications are shown in Table 6.1. The simulator used for the analysis was gem5 [7] and the benchmark suite in this performance study was a subset of the PARSEC 2.1 benchmark suite [6].

With this simulation infrastructure, we have analyzed and quantified the error of each challenge following the methodology indicated in the previous section. Results are shown in the next subsections.

6.4.1 Pattern detection

As we stated earlier, we used the GHB prefetcher to analyze this challenge. This prefetcher records the miss stream of the memory it is working with, analyzes it, and attempts to find a correlation between the last accesses. If successful, it can trigger up to 16 prefetching requests (depending on the aggressiveness) per memory miss. Figure 6.6a shows the evaluation of all the requests issued. The total value of the bars represents the number of generated requests every 1k instructions. Note that a distributed GHB can only be aware of certain parts of the pattern. For this reason, it is unable to find correlation in the miss stream and is therefore unable to generate prefetch requests. The unified and ideal GHB therefore launches up to 6.5 times more useful prefetches than the distributed one.



(a) Evaluation of the generated requests by GHB and the ideal GHB for the pattern detection challenge analysis.

(b) MPKI in L2 without prefetching, GHB, and ideal GHB for the pattern detection challenge analysis.

Fig. 6.6 Pattern detection challenge analysis.

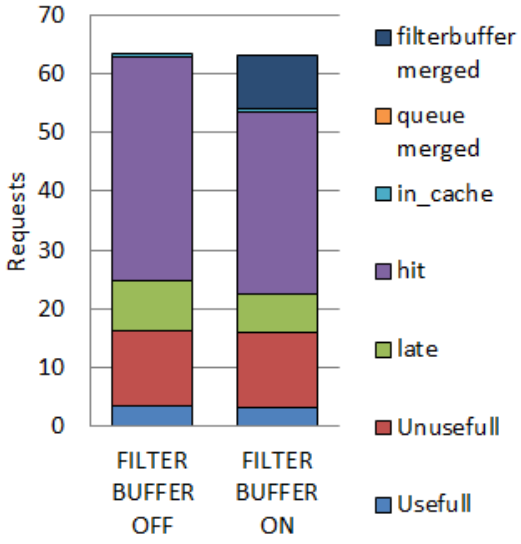
Figure 6.6b shows the misses every 1k instructions. As a consequence of a prefetch engine that is not working properly, the MPKI is not reduced as much as it should be. We can see that in almost all the workloads the ideal GHB reduces the MPKI to a higher degree than the distributed GHB does. However, in the canneal benchmark, the MPKI of the ideal GHB is higher than the distributed GHB. The reason for this is that the GHB engine is not smart enough to predict the data required by this benchmark. Increasing the activity of the prefetcher therefore means increasing the pollution and, consequently, the MPKI.

Nevertheless, on average, the ideal GHB manages to decrease the MPKI to a higher degree than the distributed GHB does.

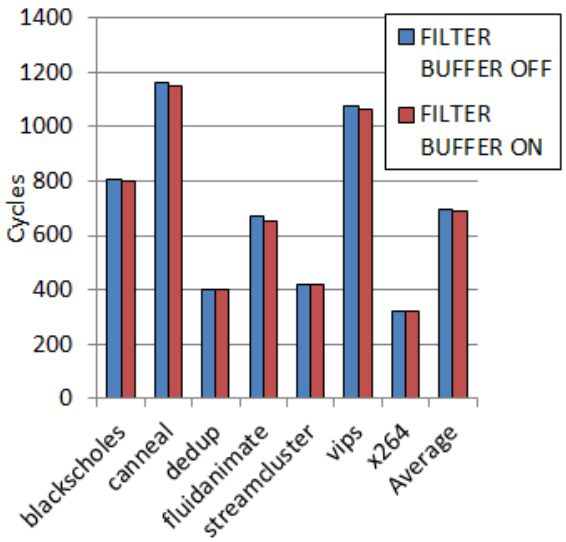
6.4.2 Prefetching queue filtering

To analyze this challenge we used a global filtering buffer. This buffer is aware of all the prefetch requests in all the prefetch queues. When queuing new requests, if the request is in any other queue of the system, it is merged. Figure 6.7a shows not only the issued requests but also the generated ones. The stacked bar with the filter buffer merge value represents the number of requests that the distributed prefetcher queue is not able to filter. For this reason, without the filtering buffer, the prefetcher will inject up to 30% more traffic into the network.

Figure 6.7b shows the average latency of a miss in L1. We can see that the filtering has a direct effect on performance. This is because the filtering effect reduces congestion in the network, which reduces the L1 miss latency.



(a) Average number of generated requests by the Tagged prefetcher for the prefetching queue filtering.

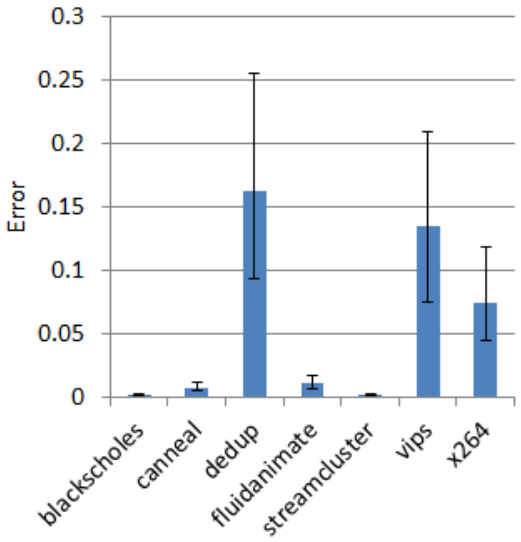


(b) Average miss latency in L1 for all the benchmarks for the prefetching queue filtering.

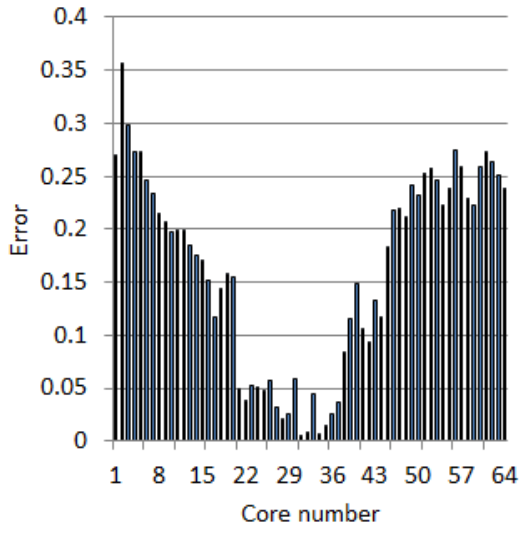
Fig. 6.7 Queue filtering challenge analysis.

6.4.3 Dynamic profiling

To quantify this challenge, we measured the accuracy of the prefetcher in two ways: (1) the real accuracy for each prefetch in each tile and (2) the accuracy measured blindly in each tile



(a) Absolute error in accuracy for the Tagged prefetcher for the dynamic profiling analysis.



(b) Absolute error for all the tiles with the dedup benchmark for the dynamic profiling analysis.

Fig. 6.8 Dynamic profiling challenge analysis.

for the aggregated prefetching requests issued in it. We calculated the absolute error between these two values and the result is shown in Figure 6.8a and Figure 6.8b. In Figure 6.8a, we can see two elements: the bars, whose value represents the average error among all the tiles and the segment bars, which represent the maximum and minimum errors among all the tiles in the same chip. We can see that the error for some benchmarks is relatively small. This happens because, depending on the behavior of the tagged prefetch, the requests issued from one tile are almost always addressed to the same tile and this effect balances out the statistic. However, the error becomes significant when it is analyzed dynamically.

To see the error in the various tiles in greater detail, Figure 6.8b shows the absolute errors for all the tiles in the dedup benchmark, which is the one with the greatest errors. We can see that the error can sometimes reach differences of over 35%. Note that dynamic mechanisms, which base their decisions on these statistics, may make wrong decisions.

6.5 Facing the challenges

We propose three possible strategies that may guide researchers into finding solutions to the challenges: (1) **replication**, (2) **centralization**, or (3) **distribution**, scheme that mixes the centralized and replicated techniques. All these strategies can solve the three challenges that the baseline faces. Note that these proposals are only three possible examples of how to

deal with these challenges. Moreover, each of these modifications add new drawbacks to the system.

6.5.1 Replication

The idea behind this strategy is to replicate global prefetching information in each tile. To explain this technique, we will describe the behavior of the prefetcher following this approach in each of the three phases introduced in Section 6.2. As in Section 6.2, Figure 6.9 shows four tiles that represent a subset of the entire tiled system. In this section, we have also tagged the arrows with a number that represents the behavior in each phase.

1. **Analysis phase** :When the shared memory is accessed, a new notification message is generated with the information of the memory access (address, core that generated the demand request, if it has been a hit, miss, or useful prefetch, etc.) and broadcast to all the tiles in the system. The prefetcher information is replicated in each tile, which means that all the prefetchers read the request generated by the accessed memory module and update their status, keeping separate information for the different cores. This isolation can be achieved statically (using separate tables) or dynamically (expanding the access index with the core identifier).

2. **Request generation phase**: At the end of the analysis phase, several requests may be generated. Note that these requests may or may not be addressed to the same tile as the one from which they have been generated. As all the prefetching engines have the same access information, all of them are going to generate the same requests. This allows the requests that are not addressed to the same tile as the one from which they are generated to be filtered before being queued. This is done in order to prevent extra traffic and too many hit prefetches.

3. **Evaluation phase**: The prefetching profiling information will be gathered as usual in the shared memory piece by marking as a useful prefetch, for example, a prefetched cache line that is actually accessed by a demand request.

When this strategy is applied the challenges are faced in the following way:

- **Pattern detection:** The patterns are not distributed anymore because all the modules keep track of all the memory accesses in all the tiles. In this way, each prefetcher keeps a replica of the access memory information. For this reason, each replica is able to detect the patterns without any problems. The isolation of the information at a core level, avoids the distortion of the memory patern that may be introduced when several cores or applications are accesing the same cache.

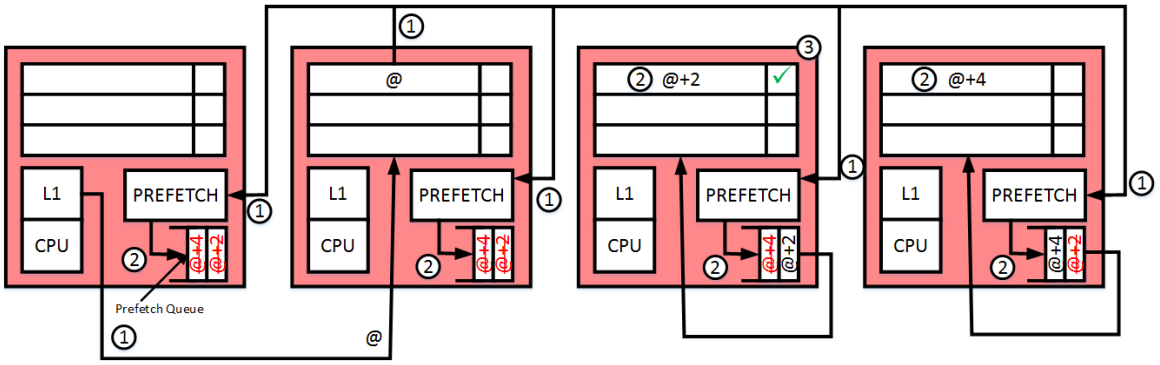


Fig. 6.9 Phases of the Replication technique: (1) analysis, (2) request generation, and (3) evaluation.

- **Prefetching queue filtering:** The requests are queued to the same tile from which they are generated or they are filtered. For this reason, the same request cannot be located in two different tiles. If there is a replicated request in the same tile, the replicated request will be filtered when queued in the prefetch queue.

- **Dynamic profiling:** Due to the filtering mechanism performed in the request generation phase, a distributed memory piece will only receive prefetching requests from a single prefetching engine, that is, the one in the same tile. Therefore, all the prefetched blocks in the memory will belong to that prefetching engine and all the profiling information regarding the prefetch activity on that piece of the memory will be correct and related with the prefetcher in that tile.

As has been shown, the replication strategy deals with all the challenges presented in this study. However, each of the techniques proposed in this study has some drawbacks compared to the baseline (which cannot deal with the challenges). Table 6.2 shows the main drawbacks of the techniques proposed for solving the challenges and compares them to the baseline. In the following points, we define these drawbacks and show how the replication is affected by them.

- The **total prefetcher size** depends on the implementation of each prefetch engine. As it is a highly variable value, we have decided to put a reference value that represents the size of a prefetcher in a tile (M) in the baseline. When there are replications, each replica of the prefetcher must have enough available space to store information about the whole access pattern of all the cores. Therefore, the size of the data structures must be scaled appropriately. The maximum size required for a replica is the size of the prefetcher in a tile from the baseline multiplied by the number of tiles. For this reason, the total size will be the size of a replica multiplied by the number of replicas.

If there is a replica in each tile, the total size will be the size of a replica multiplied by the number of tiles. However, the size of a replica can be optimized if the prefetcher uses dynamic tables instead of static ones. For this reason, the size of the prefetcher would be M*K*N instead of M*N*N where K is a variable number between 1 and N. Nevertheless, the performance of the prefetcher will decrease with small values of K.

- The **messages per access** are the maximum number of notifications that a distributed memory module can send to one or several prefetchers per access. In the case of replication, for every cache access, a notification message is sent to every tile in the system (a broadcast message). Note that in the baseline, the communication between the cache module and the prefetcher only takes place inside the same tile. There are techniques that help to reduce the cost of sending these broadcast messages, and studies are being carried out into solutions to make this feasible using, for example a graphene-enabled wireless broadcast [1].

- The **prefetcher throughput** ratio refers to the maximum number of requests that a prefetch module has to process per cycle. In replication, the notification messages must be processed by the prefetcher and each prefetcher can be the target of a notification message per core and per cycle. For this reason, the contention of the network is not the only problem given that the prefetcher also needs to be available to attend the requests. On the other hand, although the requests that are not addressed to the prefetcher tile are filtered, they still need to be generated and each access memory notification may trigger the generation of prefetching requests. Note that the GHB prefetcher (with the configuration used in this study) can generate up to 16 requests per miss in the distributed memory. This means that the throughput requirements of the prefetching engines will probably need to be quite high in order to achieve a good efficiency.

- The **NoC prefetcher requests** are the maximum number of requests that a prefetcher module can inject into the network per cycle. However, in the replication, when the prefetch requests are generated, they do not need to be sent into the network (as in the baseline), because the request is always resolved in the same tile.

- The **total traffic increment** refers to the total amount of traffic that can be injected into the network per cycle by all the tiles in the system, either through the notifications from the distributed memory module or through the requests injected by the prefetcher. When there is a replication, the messages generated per access significantly increase. However, there is a reduction in the number of requests that are generated by the prefetcher. For example, in a 64 tile chip, each access to the distributed memory would

generate 64 new control messages. In the worst of cases, if all the tiles generated a memory access at the same time, there would be 4096 messages generated during the same cycle, which could easily congest the network if not properly managed, whereas in the baseline, in a 64 tile chip, if all the prefetchers generate a request at the same time there would be a peak of 64 messages in the same cycle.

- The **extra bits per cache block** refers to the extra bits needed to store the information for the profiling in each cache block. In the replication, this is not a problem, because the storage requirements will be the same as the baseline.

	Baseline	**Replicated**	**Centralized**	**Distributed**
Total prefetcher size	M*N	M*N*N	M*N	M*N
Pref throughput	1	N	N	$N \rightarrow 1$
Extra bits per cache block	1	1	1	$log_2 N$
Total traffic increment	N (uni)	N (br)	2N (uni)	N (uni) + N(br)
Messages per access	0	1 (br)	1 (uni)	1 (uni)
NoC pref requests	1 (uni)	0	1 (uni)	1 (br)

Table 6.2 Consumption of resources by the proposed techniques. N: Number of tiles, uni: Uni-cast message , br: Broadcast message, M: Size of a prefetcher in a tile from the baseline.

6.5.2 Centralization

A different approach from the previous strategy is centralization. The main idea behind centralization is to have a new independent module in the system with the prefetcher, instead of having one prefetcher in each tile. The way this proposal behaves is represented in Figure 6.10. In the following points, we explain step by step how the centralized strategy would work.

1. **Analysis phase**: When a distributed memory access occurs, the accessed module sends a notification message to the prefetch module with the information of the access, the same as in the replicated approach.

2. **Request generation phase**: The prefetcher will generate a certain number of requests according to its preferences. These requests will be queued in the prefetch queue (which is stored in the prefetch module). The prefetching requests in the queue will be sent in turn to the shared memory module responsible for the memory request through the interconnection module.

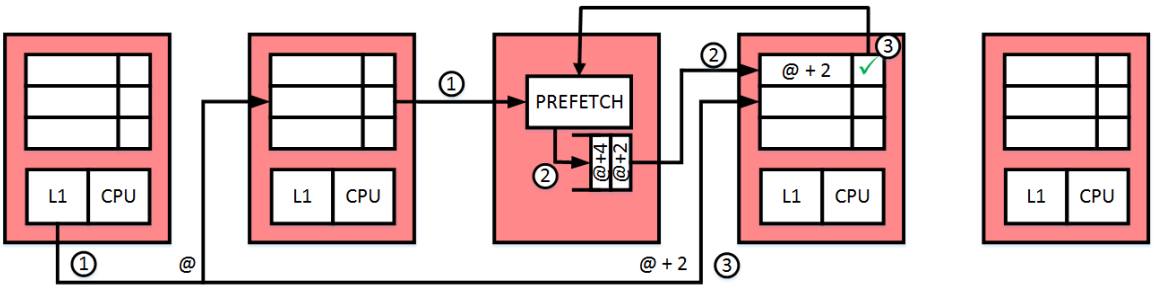


Fig. 6.10 Phases from the prefetcher in a distributed memory system: (1) analysis, (2) request generation, and (3) evaluation.

3. **Evaluation phase**: The profiling information can be collected from the individual pieces of the distributed shared memory or sent to the centralized prefetcher every time the cache module is accessed. For example, each cache block could take note with a bit that the block has been brought to the cache by a prefetch request and send useful/unuseful prefetching information when is accesed by a demand request or replaced by one without being used.

When this strategy is applied the challenges are dealt with in the following way:

- **Pattern detection:** The information about the access patterns is centralized. For this reason, the prefetcher has no problem successfully recognizing the patterns.
- **Prefetching queue filtering:** There is only one prefetching queue in the whole system; that is, the queue is filtered in a centralized mode, which means there are no redundant requests in different prefetching queues.
- **Dynamic profiling:** There is only one prefetching engine. Therefore, all the profiling information collected is related to that engine. The profiling information can be periodically collected from the different tiles or sent to the prefetcher with the request that is generated when the distributed memory is accessed, thus always keeping an updated profiling image in the centralized module.

Centralization solves some of the drawbacks of the replication technique. However there are still some drawbacks that affect this technique.

- **Total prefetcher size:** The prefetcher would be the same size as one of the replicas in the replication technique. For this reason we set the size at M*N (the same as the baseline). Moreover, the same optimization of the dynamic tables could be applied in this technique, thus reducing the size of the prefetcher from M*N to M*K where

K is a number between 1 and N. Nevertheless, as in the replication technique, the performance of the prefetcher will decrease with small values of K.

- **Messages per access:** Instead of sending each notification to all the tiles, centralization sends them to only one (the central prefetcher module). So instead of sending a broadcast message, it is enough to send an uni-cast message to the prefetch module. Nevertheless, sending an uni-cast message through the network is more than what the baseline needs to work.

- **Prefetcher throughput:** The centralized approach does not solve the problem of the prefetcher throughput requirements. With 64 tiles, the prefetch module can receive up to 64 requests per cycle and has to generate possible prefetching requests in response to each of them. The prefetching queue has to be bigger than in the replication approach as it has to store requests for all the tiles in the system.

- **NoC prefetcher requests:** In order to generate the same performance as in the baseline, the centralized approach has to be able to inject N requests into the network per cycle. In the other approaches, each prefetch module only has to inject one request every cycle. However, as this option is centralized, there is only one prefetch queue, which means that it has to generate requests for all the tiles.

- **Total traffic increment:** Centralization reduces the network traffic because the messages are unicast instead of broadcast. However, the number of messages in the network is still greater than in the baseline. The reason for this is that in the baseline the request is always analyzed by the prefetcher in the same module from which the distributed memory is accessed. In the centralized approach each distributed memory access represents a new notification message that traverses the network. Moreover, in contrast with the replicated prefetcher, all the prefetching requests must traverse the network prior to arriving at the appropriate tile.

- **Extra bits per cache block:** As in the replication strategy, this is not a problem, because the storage requirements are the same as the baseline.

6.5.3 Distribution

The distributed strategy makes use of the same prefetching structure as the baseline. However, the allocation of the data is somewhat different. As stated before, the idea behind the prefetch techniques is to try to correlate the memory accesses that a certain core is following to execute a program in order to predict the next accesses. The idea behind this strategy is to

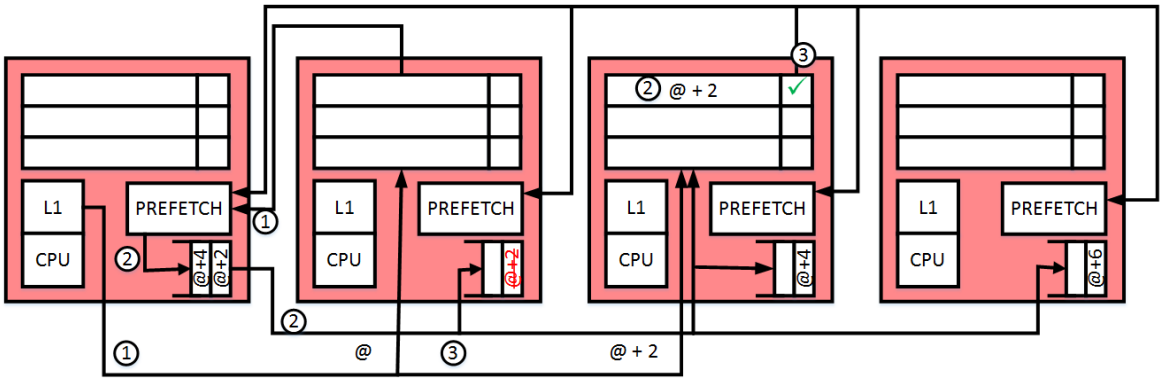


Fig. 6.11 Phases from the prefetcher in a distributed memory system: (1) analysis, (2) request generation, and (3) evaluation.

recover the correlation of each core access after it has been dispersed in the shared memory. To do this, each prefetch module monitors the requests generated by the core in its same tile. Figure 6.11 shows the behavior of this technique.

1. **Analysis phase**: As in the previous strategies, when a distributed memory is accessed, a new notification message will be generated. However, this request will be sent only to the tile that contains the core from where this access originated. Once the request is in this tile, the prefetcher allocated in the tile analyzes it and generates the appropriate number of prefetching requests. Each prefetcher will only gather information on the access pattern of its associated core.

2. **Request generation phase**: In the request generation phase, the requests are generated and queued in the prefetching queue of each prefetcher. This does not solve the problem of replicated requests in different queues of the baseline. Therefore, to solve this problem, each time a prefetch request is processed from the prefetch queue, a broadcast message is sent to all the tiles with the address of the prefetch request. All the tiles check if they have this address in their prefetch queue and, if so, they erase this request from the prefetch queue. Moreover, the tile that has to resolve the memory access keeps the address and processes it.

3. **Evaluation phase**: Each tile will store basic profiling information indicating the associated tile that generated the prefetching request. To avoid storing information per prefetching engine in each of the tiles, the information can be sent to the home tile in each of the notification messages as suggested for the previous strategy.

When this strategy is applied the challenges are dealt with in the following way:

- **Pattern detection:** With the new organization of the patterns, the prefetcher is able to detect the whole access pattern of the processor it is monitoring without interference from other cores.
- **Prefetching queue filtering:** The broadcasting mechanism for the prefetching requests allows the queues from all the tiles to filter the replicated copies of each request.
- **Dynamic profiling:** The per tile information associated with the prefetching profiling information stored in each piece of the distributed shared memory allows profiling statistics to be properly associated with each of the prefetching engines.

The distributed technique solves some of the problems of the centralized and replicated techniques. However it still has some drawbacks:

- **Total prefetcher size:** The total prefetcher size is the same as the baseline.
- **Messages per access:** As in the centralization technique, each distributed memory module can issue a notification uni-cast message per cycle. This is higher than in the baseline where the communication takes place within the same tile and no notification messages need to be issued.
- **Prefetcher throughput:** In the distributed approach, instead of sending all the information to a centralized module, the requests are distributed among all the tiles in the system. For this reason, a worst-case scenario in a given cycle may mean that the number of notification messages addressed to a given prefetcher is equal to the number of cores, although this would be uncommon. Therefore, the throughput requirements will be lower than in the two previous strategies.
- **NoC prefetcher requests:** Note that with this strategy the prefetcher must send a notification to all the tiles in the system. Although the number of prefetcher requests is usually lower than the number of accesses in the memory module, a broadcast message has a significant cost. For this reason the broadcast message sent by the prefetcher is not as critical as the broadcast message sent in the replication technique, but it is still a big drawback.
- **Total traffic increment:** As in the centralization strategy, the distribution strategy generates the same number of messages for the memory accesses. However, the messages injected into the network by the prefetcher are broadcast instead of uni-cast. This means that more traffic is injected in the distributed approach; however, there is less than in the replication strategy because, as stated before, in the replication strategy

a broadcast message is generated for each memory access. In the distributed approach, this broadcast message is only generated when a prefetch request is injected into the network.

- **Extra bits per cache block:** The distribution strategy requires more storage than the baseline to keep track of per tile prefetching requests. Specifically, $log_2 N$ bits per cache block will be required to keep track of the tile that issues the prefetch requests.

6.6 Conclusions

In this chapter, we have shown the challenges when trying to prefetch in a distributed and shared memory system. We have described, quantified, and faced these challenges: (1) pattern detection, (2) prefetch queue filtering, and (3) dynamic profiling. We have also shown that the memory access pattern in the private cache, which is more or less regular and predictable, becomes totally irregular in the distributed memory. For this reason, the techniques used by the prefetcher to guess the future misses in the unified memories do not properly apply when these prefetch engines are distributed. Moreover, the dispersion of the prefetching queues reduces the potential of the prefetchers. Finally, the techniques that use prefetch efficiency statistics from the caches to dynamically modify the behavior of the prefetching mechanism are also challenged. We believe that the experimental results provided in this work are important for the community, as we make strong evidence that when targeting this kind of architectures new approaches for prefetching are needed. Furthermore, we have proposed some hints on how these challenges can be solved. For this reason, as future work, this analysis opens the door to several works that wish to address these challenges and solve them or make feasible and efficient the techniques proposed on this study.

Chapter 7

Conclusions and future work

> To succeed, jump as quickly at opportunities as you do at conclusions.
>
> *Benjamin Franklin*

7.1 Conclusions

Although we already have included a conclusion section in the main chapters of this document, we want to make a final remark of all our work during this thesis. For this reason, in this chapter we will look over all the studies that we have presented. Moreover, we will evaluate the objectives that we presented in the Introduction chapter.

Prefetching is one of the oldest memory latency hiding techniques. However, prefetching is implemented in almost all the commercial processors, what means that has been a technique that at its moment shake the industry up. There is a lot of literature about prefetching and techniques to improve its performance. Along these years prefetching has evolved with the emerging technologies that has changed the way it worked at the very beginning. New technologies such as out-of-order execution or multiprocessor designs have increased the complexity of prefetching. With each of these new generation of technology new prefetching schemes have been studied. This has been the case of our thesis which covers some new topics that has not been treated and extends some others that were already studied before. Actually, we have taken a look on the challenges of prefetching in the distributed and shared memory systems what is something that nobody studied before. Moreover, we study techniques from the state of the art to dynamically manage prefetching such as filtering and prioritization. We improve their performance by using novel heuristics.

In the next points of this chapter, we will evaluate the objectives of the thesis and will give some directions in how this work could be continued.

7.2 Objective evaluation

In this section, we will address all the objectives that we presented at the beginning of the thesis and we will evaluate if we have been able to fulfill them or not.

- **Characterize the NoC behavior of prefetching in multicore tiled platforms**

  This objective have been addressed in Chapter 3 by modifying a state of the art simulator that simulates the CPU and the NoC. The implemented modifications allow us to inject prefetch traffic in the interconnect of the processor, to distinguish between the prefetching traffic in the interconnect and the traffic generated by the processor, and finally, to evaluate the congestion that the prefetcher injected in the interconnection network. After the analysis carried out with this modified simulator, we can conclude that the extra traffic injected by the prefetcher has important consequences in the whole system if it is not properlly taken into account. This work has been published in [81] and [79].

- **Enhance the profiling of prefetching statistics used in dynamic management techniques to explore techniques to better handle prefetching traffic**

  In order to enhance the profiling techniques for prefetch requests, in Chapter 4, we have proposed several techniques that enhance the collection of profiling information of prefetch requests. Moreover, the proposed techniques evaluate the profiled data and predicts a level of confidence for each potential prefetch request to be generated In our study, we have focused in the accuracy to predict the impact on the system performance. Nevertheless, other statistics could be used to evaluate this impact. Furthermore, we have focused on the more successful technique and we have proposed a feasible hardware implementation for it. This work has been published in [82], [86] and [55].

- **Propose novel management techniques to improve the performance of prefetching techniques**

  In Chapter 5, we have studied techniques that improve the management of the prefetching requests in the whole system. We have developed a new technique based on a combination of two existing ones (filtering and prioritization). Moreover, we have

compared this technique with the ones from the state of the art. With this work, we have also shown that our confidence predictor can be used conjunction with state of the art techniques that dynamically manage the prefetching requests in the system and improve them. This work has been published in [85].

- **Identify the challenges for prefetching in shared distributed memory systems and propose directions to address them**

  This objective is addressed in Chapter 6, where, we have focused on why there are not studies about prefetching in the distributed and shared caches. We have pointed out the challenges that appears when the prefetch engines designed for private caches work in shared and distributed caches. Moreover, we have quantified these challenges.

  Moreover, we have given directions in how these challenges could be addressed. Although the provided proposals are mostly theoretical, we consider that they can be good starting points for future researchers looking for solutions for them. This work has been published in [84] and [83].

7.3 Future work

Along this thesis, we have expanded some existing works and we have started new ones. However, any of them can be used as the starting points of new works. In the following points, we propose several ideas that can be the seed for new projects.

- **Extend the implementation of the prefetcher in gem5 for the other memory coherence protocols**: In chapter 3, we explain how do we have implemented the prefetcher in the ruby memory system of gem5. However, we only have modified the MOESI_CMP_directory memory coherence protocol. This has been enough for preparing the simulation infrastructure of this thesis. Nevertheless, the online version of gem5 has several protocols already implemented to work with. The idea of this future work is to modify these other coherence protocols of gem5 to let them work with the prefetching infrastructure developed in this thesis.

- **Use the confidence predictor for other techniques and statistics**: The mechanism presented in Chapter 4 is a generic heuristic that, as a proof of concept, we have applied to enhance prioritization and filtering in Chapter 5, but could be used in conjunction with other dynamic prefetch management techniques like, for example, throttling. Moreover, we could use the proposed confidence predictor not only for accuracy but for other prefetching related statistics that are sometimes used by dynamic management

techniques, like lateness or pollution. The idea behind this future work would be to analyze which mechanisms would take advantage of a better prefetch profiling and improve them with this novel scheme.

- **Use the confidence predictor in other environments** An interesting area to explore would be the combination of our confidence predictor for prefetching with other techniques that could use code region level statistics to improve the performance of the system. Furthermore, the same concepts that has lead the design of this mechanism could be applied in other environments such as branch predictors or other kind of predictions.

- **Try the techniques proposed to face the challenges in the shared and distributed cache**: Maybe the chapter that have a more obvious future work is Chapter 6. In this work, we let everything ready to do the next step: Implement the proposals suggested in the chapter and compare them. In order to give a more detailed starting point for the future work done in this research, we think that, although none of the strategies (replicated, centralized, and distributed) proposed in this chapter are perfect (they have advantages and disadvantages), the distributed mechanism seems to be the strategy which would face in a better way most of the challenges. This approach comes up as the combination of the previous two (replication and centralization), thus it solve most of the problems of distributed pattern and dynamic profiling efficiently. Furthermore, as it is distributed, the solution may be scalable to architectures with higher number of tiles. Nevertheless, although this technique increments the congestion of the network in a lower degree than the others strategies, it still can be a problem for architectures with limited network bandwidth. Moreover, it still has a big drawback to solve: how the prefetching queue filtering challenge is solved feasibly. Furthermore, in case of not being feasible, it should be proposed some feasible mechanism to implement these techniques in real systems.

7.4 Thesis contributions

In this section, we list the publications achieved by this thesis as well as the publications in which we are pending to contribute.

7.4.1 Prefetching evaluation in multi-core platforms

- M.Torrents, R.Martínez, C.Molina, **An Accurate and Detailed Prefetching Simulation Framework for gem5** , The Second gem5 User Workshop (GEM5'15), held in conjuction with the 42nd International Symposium on Computer Architecture (ISCA'15), Portland, (USA), June 2015.
- M.Torrents, R.Martínez, C.Molina, **Network Aware Performance Evaluation of Prefetching Techniques in CMPs**, Journal of Simulation Modelling Practice and Theory, Volume 45, June 2014.
- M. Torrents, R. Martínez, P. Lopez, J. M. Codina, A. Gonzalez, **Comparative Study of Prefetching Mechanisms**, In XXI Jornadas de Paralelismo, 2010.

7.4.2 Confidence predictor mechanisms for prefetching in CMPs

- M.Torrents, R.Martínez, C.Molina, **A novel confidence predictor to improve the prefetching performance in CMPs**, Tech. Report UPC-DAC-RR-ARCO-2016-3, July 2016.
- M.Torrents, R.Martínez, C.Molina, **Improving the Prefetching Performance Through Code Region Profiling**, In Proceedings of the 2nd International BSC Doctoral Symposium (BSC'15), Barcelona, (Spain), May 2015.
- R. Martínez, E. Gibert, P. Lopez, M. Torrents, et. al **Profiling asynchronous events resulting from the execution of software at code region granularity**.US Patent App. 13/993,054-2011.

7.4.3 Improving the Prioritization and Filtering of Prefetch Requests

- M.Torrents, R.Martínez, C.Molina, **Improving the prioritization and filtering of prefetching request**, Tech. Report UPC-DAC-RR-2016-5, July 2016.

7.4.4 Prefetching Challenges in Distributed Shared Memories for CMPs

- M.Torrents, R.Martínez, C.Molina, **Facing Prefetching Challenges in Distributed Shared Memories**, The Journal of Supercomputing (JSC'16), February 2016.
- M.Torrents, R.Martínez, C.Molina, **Prefetching Challenges in Distributed Memories for CMPs**, In Proceedings of the International Conference on Computational Science (ICCS'15), Reykjavík, (Iceland), June 2015.

References

[1] Abadal, S., Cabellos-Aparicio, A., Lemme, M. C., Nemirovsky, M., et al. (2013). Graphene-enabled wireless communication for massive multicore architectures. *Communications Magazine, IEEE*, 51(11):137–143.

[2] ATI (2006). Atistream technology. http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx. [Online].

[3] Baer, J.-L. and Chen, T.-F. (1991). An effective on-chip preloading scheme to reduce data access penalty. In *Supercomputing, 1991. Supercomputing'91. Proceedings of the 1991 ACM/IEEE Conference on*, pages 176–186. IEEE.

[4] Balfour, J. and Dally, W. J. (2006). Design tradeoffs for tiled cmp on-chip networks. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 187–198. ACM.

[5] Bell, S., Edwards, B., Amann, J., Conlin, R., Joyce, K., Leung, V., MacKay, J., Reif, M., Bao, L., Brown, J., Mattina, M., Miao, C. C., Ramey, C., Wentzlaff, D., Anderson, W., Berger, E., Fairbanks, N., Khan, D., Montenegro, F., Stickney, J., and Zook, J. (2008). Tile64 - processor: A 64-core soc with mesh interconnect. In *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, pages 88–598.

[6] Bienia, C., Kumar, S., Singh, J. P., and Li, K. (2008). The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM.

[7] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A. (2011). The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7.

[8] Bjerregaard, T. and Mahadevan, S. (2006). A survey of research and practices of Network-on-Chip. In *ACM Comput. Surv.*, volume 38, New York, NY, USA. ACM.

[9] Boggs, D., Baktha, A., Hawkins, J., Marr, D. T., Miller, J. A., Roussel, P., Singhal, R., Toll, B., and Venkatraman, K. (2004). The microarchitecture of the intel pentium 4 processor on 90nm technology. *Intel Technology Journal*, 8(1).

[10] Byna, S., Chen, Y., and Sun, X.-H. (2009). Taxonomy of data prefetching for multicore processors. *Journal of Computer Science and Technology*, 24(3):405–417.

[11] Chen, T.-F. and Baer, J.-L. (1995). Effective hardware-based data prefetching for high-performance processors. *IEEE transactions on computers*, 44(5):609–623.

[12] Cheng, L., Muralimanohar, N., Ramani, K., Balasubramonian, R., and Carter, J. B. (2006). Interconnect-aware coherence protocols for chip multiprocessors. In *ACM SIGARCH Computer Architecture News*, volume 34, pages 339–351. IEEE Computer Society.

[13] Chidambaram Nachiappan, N., Mishra, A. K., Kademir, M., Sivasubramaniam, A., Mutlu, O., and Das, C. R. (2012). Application-aware prefetch prioritization in on-chip networks. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 441–442. ACM.

[14] Chou, C.-L. and Marculescu, R. (2010). Run-time task allocation considering user behavior in embedded multiprocessor networks-on-chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(1):78–91.

[15] Chou, C.-L., Ogras, U. Y., and Marculescu, R. (2008). Energy-and performance-aware incremental mapping for networks on chip with multiple voltage levels. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1866–1879.

[16] Das, R., Mutlu, O., Moscibroda, T., and Das, C. R. (2009). Application-aware prioritization mechanisms for on-chip networks. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 280–291. ACM.

[17] Das, R., Mutlu, O., Moscibroda, T., and Das, C. R. (2010). Aérgia: exploiting packet latency slack in on-chip networks. In *ACM SIGARCH computer architecture news*, volume 38, pages 106–116. ACM.

[18] Dundas, J. and Mudge, T. (1997). Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 11th international conference on Supercomputing*, pages 68–75. ACM.

[19] Ebrahimi, E., Mutlu, O., Lee, C. J., and Patt, Y. N. (2009). Coordinated control of multiple prefetchers in multi-core systems. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 316–326, New York, NY, USA. ACM.

[20] Flores, A., Acacio, M. E., and Aragón, J. L. (2010a). Exploiting address compression and heterogeneous interconnects for efficient message management in tiled cmps. *Journal of Systems Architecture*, 56(9):429–441.

[21] Flores, A., Aragón, J. L., and Acacio, M. E. (2010b). Energy-efficient hardware prefetching for CMPs using heterogeneous interconnects. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 147–154. IEEE.

[22] Flores, A., Aragon, J. L., and Acacio, M. E. (2010c). Heterogeneous interconnects for energy-efficient message management in CMPs. *IEEE Transactions on Computers*, 59(1):16–28.

[23] Fu, J. W., Patel, J. H., and Janssens, B. L. (1992a). Stride directed prefetching in scalar processors. *ACM SIGMICRO Newsletter*, 23(1-2):102–110.

[24] Fu, J. W. C., Patel, J. H., and Janssens, B. L. (1992b). Stride directed prefetching in scalar processors. *SIGMICRO Newsl.*, 23(1-2):102–110.

[25] gem5 (2011a). gem5 cache coherence protocols. http://www.m5sim.org/Cache_Coherence_Protocols/. [Online].

[26] gem5 (2011b). gem5 cache coherence protocols. http://www.m5sim.org/MOESI_CMP_directory. [Online].

[27] gem5 (2012). gem5 status matrix. http://www.m5sim.org/Status_Matrix/. [Online].

[28] Hennessy, J. L. and Patterson, D. A. (2011). *Computer architecture: a quantitative approach*. Elsevier.

[29] Hinton, G., Sager, D., Upton, M., Boggs, D., et al. (2001). The microarchitecture of the pentium® 4 processor. In *Intel Technology Journal*. Citeseer.

[30] Ho, R., Mai, K. W., and Horowitz, M. A. (2001). The future of wires. *Proceedings of the IEEE*, 89(4):490–504.

[31] Hu, J. and Marculescu, R. (2004). Application-specific buffer space allocation for networks-on-chip router design. In *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 354–361. IEEE Computer Society.

[32] Jeffers, J. and Reinders, J. (2013). *Intel Xeon Phi coprocessor high-performance programming*. Newnes.

[33] Jiménez, V., Gioiosa, R., Cazorla, F. J., Buyuktosunoglu, A., Bose, P., and O'Connell, F. P. (2012). Making data prefetch smarter: adaptive prefetching on power7. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 137–146. ACM.

[34] Joseph, D. and Grunwald, D. (1997). Prefetching using markov predictors. *SIGARCH Comput. Archit. News*, 25(2):252–263.

[35] Kamruzzaman, M., Swanson, S., and Tullsen, D. M. (2011). Inter-core prefetching for multicore processors using migrating helper threads. *ACM SIGPLAN Notices*, 46(3):393–404.

[36] Kandiraju, G. B. and Sivasubramaniam, A. (2002). *Going the distance for TLB prefetching: an application-driven study*, volume 30. IEEE Computer Society.

[37] Khan, M., Laurenzanoy, M. A., Marsy, J., Hagersten, E., and Black-Schaffer, D. (2015). Arep: Adaptive resource efficient prefetching for maximizing multicore performance. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 367–378. IEEE.

[38] Khan, M., Sandberg, A., and Hagersten, E. (2014). A case for resource efficient prefetching in multicores. In *2014 43rd International Conference on Parallel Processing*, pages 101–110. IEEE.

[39] Kim, S. and Veidenbaum, A. V. (1997). Stride-directed prefetching for secondary caches. In *Parallel Processing, 1997., Proceedings of the 1997 International Conference on*, pages 314–321. IEEE.

[40] Kroft, D. (1998). Lockup-free instruction fetch/prefetch cache organization. In *25 years of the international symposia on Computer architecture (selected papers)*, pages 195–201. ACM.

[41] Laurenzano, M. A., Zhang, Y., Tang, L., and Mars, J. (2014). Protean code: Achieving near-free online code transformations for warehouse scale computers. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 558–570. IEEE Computer Society.

[42] Lee, C. J., Mutlu, O., Narasiman, V., and Patt, Y. N. (2008a). Prefetch-aware DRAM controllers. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 200–209. IEEE Computer Society.

[43] Lee, C. J., Narasiman, V., Mutlu, O., and Patt, Y. N. (2009). Improving memory bank-level parallelism in the presence of prefetching. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 327–336, New York, NY, USA. ACM.

[44] Lee, J. et al. (2008b). Globally-synchronized frames for guaranteed quality-of-service in on-chip networks, 35th ieee. In *ACM International Symposium on Computer Architecture (ISCA)*, volume 12.

[45] Lee, J., Kim, H., Shin, M., Kim, J.-H., and Huh, J. (2014). Mutually aware prefetcher and on-chip network designs for multi-cores. *Computers, IEEE Transactions on*, 63(9):2316–2329.

[46] Lee, J., Kim, H., and Vuduc, R. (2012). When prefetching works, when it doesn't, and why. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(1):2.

[47] Lee, R. L., Yew, P.-C., and Lawrie, D. H. (1987). Data prefetching in shared memory multiprocessors. Technical report, Illinois Univ., Urbana (USA). Center for Supercomputing Research and Development.

[48] Levinthal, D. (2009). Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. *Intel Performance Analysis Guide*, 30:18.

[49] Lin, W.-F., Reinhardt, S. K., and Burger, D. (2001). Reducing DRAM latencies with an integrated memory hierarchy design. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 301–312. IEEE.

[50] Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. (2008). Nvidia tesla: A unified graphics and computing architecture. *IEEE micro*, 28(2):39–55.

[51] Liu, F. and Solihin, Y. (2011). Studying the impact of hardware prefetching and bandwidth partitioning in chip-multiprocessors. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 37–48. ACM.

[52] Liu, G. (2010). *Accurate, timely data prefetching for regular stream, linked data structure, and correlated miss pattern*. PhD thesis, University of Florida.

[53] Ma, S., Jerger, N. E., and Wang, Z. (2011). Dbar: an efficient routing algorithm to support multiple concurrent applications in networks-on-chip. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 413–424. IEEE.

[54] Martin, M. M. K., Sorin, D. J., Beckmann, B. M., Marty, M. R., Xu, M., Alameldeen, A. R., Moore, K. E., Hill, M. D., and Wood, D. A. (2005). Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33:2005.

[55] Martinez, R., Codina, E., Lopez, P., Torrents, M., et al. (2011). Profiling asynchronous events resulting from the execution of software at code region granularity. US Patent App. 13/993,054.

[56] Mowry, T. C. (1994). *Tolerating latency through software-controlled data prefetching*. PhD thesis, Citeseer.

[57] Mowry, T. C. (1998). Tolerating latency in multiprocessors through compiler-inserted prefetching. *ACM Transactions on Computer Systems (TOCS)*, 16(1):55–92.

[58] Mowry, T. C., Lam, M. S., and Gupta, A. (1992). Design and evaluation of a compiler algorithm for prefetching. In *ACM Sigplan Notices*, volume 27, pages 62–73. ACM.

[59] Nesbit, K. and Smith, J. (2004). Data cache prefetching using a global history buffer. In *Software, IEEE Proceedings-*, pages 96–96.

[60] Ogras, U. Y., Hu, J., and Marculescu, R. (2005). Key research problems in noc design: a holistic perspective. In *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 69–74. ACM.

[61] Owens, J. D., Dally, W. J., Ho, R., Jayasimha, D., Keckler, S. W., Peh, L.-S., et al. (2007). Research challenges for on-chip interconnection networks. *IEEE micro*, 27(5):96.

[62] Patel, A., Afram, F., Chen, S., and Ghose, K. (2011). MARSSx86: A full system simulator for x86 CPUs. In *Design Automation Conference (DAC'11)*.

[63] Peh, L.-S. and Dally, W. J. (2001). A delay model and speculative architecture for pipelined routers. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 255–266. IEEE.

[64] Perez, D. G., Mouchard, G., Temam, O., and Network, H. (2004). Microlib: A case for the quantitative comparison of micro-architecture mechanisms. In *In Proceedings of the 3rd Annual Workshop on Duplicating, Deconstructing, and Debunking*, pages 43–54.

[65] Przybylski, S. (1990). The performance impact of block sizes and fetch strategies. *SIGARCH Comput. Archit. News*, 18(2SI):160–169.

[66] Pugsley, S. H., Chishti, Z., Wilkerson, C., Chuang, P.-f., Scott, R. L., Jaleel, A., Lu, S.-L., Chow, K., and Balasubramonian, R. (2014). Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 626–637. IEEE.

[67] Qu, W., Fan, X., Liu, Y., Yang, H., and Chen, L. (2010). Memory system prefetching for multi-core and multi-threading architecture. In *2010 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE)*, volume 1, pages V1–526. IEEE.

[68] S., B., C., Y., and S., X.-H. (2009). Taxonomy of data prefetching for multicore processors. *Journal of Computer Science and Technology*, 24:405–417.

[69] Salminen, E., Kulmala, A., and Hamalainen, T. D. (2008a). Survey of network-on-chip proposals. *white paper, OCP-IP*, 1:13.

[70] Salminen, E., Kulmala, A., and Hamalainen, T. D. (2008b). Survey of Network-on-Chip proposals. *White paper. (2008)*.

[71] Sandberg, A., Eklöv, D., and Hagersten, E. (2010). Reducing cache pollution through detection and elimination of non-temporal memory accesses. In *Proceedings of the 2010 ACM/IEEE international conference for high performance computing, networking, storage and analysis*, pages 1–11. IEEE Computer Society.

[72] Sinharoy, B., Kalla, R. N., Tendler, J. M., Eickemeyer, R. J., and Joyner, J. B. (2005). Power5 system microarchitecture. *IBM Journal Research and Development*, 49(4/5):505–521.

[73] Smith, A. J. (1982). Cache memories. *ACM Comput. Surv.*, 14(3):473–530.

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

[75] Srinath, S., Mutlu, O., Kim, H., and Patt, Y. N. (2007b). Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *High-Performance Computer Architecture*, pages 63–74.

[76] Tendler, J. M., Dodson, J. S., Fields, J. S., Le, H., and Sinharoy, B. (2002). Power4 system microarchitecture. *IBM Journal Research and Development*, 46(1):5–25.

[77] Tien-Fu, C. and Baer, J. L. (1995). Effective hardware-based data prefetching for high-performance processors. *Computers, IEEE Transactions on*, 44:609–623.

[78] Tilera (2014). Tile-gx processor family webpage. http://www.tilera.com/products/processors/TILE-Gx_Family/. [Online].

[79] Torrents, M., Martinez, R., López, P., and González, A. (2012). Comparative study of prefetching mechanisms. *CEDI*.

[80] Torrents, M., Martinez, R., and Molina, C. (2013). Prefetching module for the gem5 cmp simulator. *Technical Report, UPC-DAC-RR-2013-3*.

[81] Torrents, M., Martinez, R., and Molina, C. (2014). Network aware performance evaluation of prefetching techniques in {CMPs}. *Simulation Modelling Practice and Theory*, 45:1 – 17.

[82] Torrents, M., Martinez, R., and Molina, C. (2015a). Improving the prefetching performance through code region profiling. In *BSC Doctoral Symposium (2nd: 2015: Barcelona)*, pages 90–90. Barcelona Supercomputing Center.

[83] Torrents, M., Martinez, R., and Molina, C. (2015b). Prefetching challenges in distributed memories for cmps. *Procedia Computer Science*, 51:1463–1472.

[84] Torrents, M., Martinez, R., and Molina, C. (2016a). Facing prefetching challenges in distributed shared memories for cmps. *The Journal of Supercomputing*, 72(4):1453–1476.

[85] Torrents, M., Martinez, R., and Molina, C. (2016b). Improving the prioritization and filtering of prefetch requests. Technical report, UPC-DAC.

[86] Torrents, M., Martinez, R., and Molina, C. (2016c). A novel confidence predictor to improve the prefetching performance in CMPs. Technical report, UPC-DAC.

[87] Vanderwiel, S., Vanderwiel, S., and Lilja, D. J. (1996). A survey of data prefetching techniques. Technical report, University of Minesota.

[88] Wang, H., Peh, L.-S., and Malik, S. (2003). Power-driven design of router microarchitectures in on-chip networks. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 105. IEEE Computer Society.

[89] Zhang, M. and Asanović, K. (2005). Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *International Symposium on Computer Architecture*, pages 336–345. IEEE Computer Society.

[90] Zhuang, X. and Lee, H.-H. (2003). A hardware-based cache pollution filtering mechanism for aggressive prefetches. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pages 286–293. IEEE.

[91] Zhuang, X. and Lee, H.-H. S. (2007). Reducing cache pollution via dynamic data prefetch filtering. *IEEE Trans. Comput.*, 56(1):18–31.