Czech Technical University in Prague
Faculty of Information Technology
Department of Computer Systems

Master's thesis

Analysis and simulation of data prefetching algorithms for last-level cache memory

Bc. Carlos Escuín Blasco

Supervisor: prof. Ing. Pavel Tvrdík, CSc.
Co-supervisor: Assoc. Prof. Teresa Monreal Arnal (Universitat Politècnica de Catalunya)

June 25, 2018

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on June 25, 2018 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Escuín Blasco, Carlos. *Analysis and simulation of data prefetching algorithms for last-level cache memory.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstract

Memory latency is a major factor in limiting CPU performance and prefetching is a well-known mechanism to hide memory latency. Prefetchers operate trying to predict the memory accesses that are going to be requested in the future. It consists of fetching cachelines that have not been requested by the program yet.

The main objective of this thesis is to evaluate one of the latest hardware prefetching proposals, the Best Offset Prefetcher. Its author, Michaud [1], only evaluated the mechanism on a trace-driven simulator. Therefore, in this thesis we are going to evaluate the mechanism on a more detailed simulator, the Gem5 simulator from the University of Michigan [2].

Unfortunately, the Gem5 simulator, which is continuously under development, does not provide an infrastructure to simulate the impact of prefetchers in detail. Consequently, this infrastructure needed to be implemented. A work developed during the Ph.D of Martí et al. in the Universitat Politècnica de Catalunya [3] consisted of creating a prefetch framework able to handle prefetches in all cache levels over a MOESI directory-based protocol. Therefore, this framework is going to be integrated over the latest Gem5 release version in order to be able to implement the Best-Offset prefetcher upon it.

As a result of the different simulation environment we will observe that our results are not as optimistic as Michaud ones. We wanted to evaluate this prefetching mechanism on an environment as realisticly as possible. Consequently, our benchmarking results present less number of cache misses than the ones Michaud used, so the prefetcher activity is not so noticeable.

Keywords Prefetching, cache memory, memory hierarchy, data prefetching, computer architecture.

Contents

List of Figures

List of Tables

Introduction

An imbalance in technological improvements in the last years has led to an increasing gap between processor and memory performance. One way to mitigate this problem is to hide memory latency with techniques such as prefetching. This mechanism tries to predict the data the processor is going to request in the future with the aim of bring them to a closer cache hierarchy level.

Motivation and objectives

With the progress of the semiconductor process technology, the processor clock cycle time was significantly reduced [4]. However, the reduction of off-chip memory access time has been much less than that in the processor clock cycle time because the improvement in off-chip memory technology has primarily resulted in a large memory capacity. As a result, the off-chip main memory latency of modern processor chips has become more than hundred of processor clock cycles, and so is the pipeline stall time due to misses.

In addition, the constant increase in the amount of data used by the execution of new applications implies the need of more advanced content management policies. The bottleneck that the memory system generates is really crucial to be solved. Therefore, some aggressive techniques to hide the memory latency, as prefetching, which use the memory bandwidth that is not being used to increase the performance, have become more important.

The main objective of this thesis is to evaluate one of the latest hardware prefetching proposals, the Best Offset Prefetcher. Its author, Michaud [1], only evaluated the mechanism on a trace-driven simulator. Therefore, in this thesis we are going to evaluate the mechanism on a more detailed simulator.

Methodology

In order to validate new hypothesis and to quantify the improvements of new models in terms of cost and/or performance, it is necessary to use more complex simulators.

To properly evaluate a hardware prefetcher, besides of the program execution flow, it is specially important to know in detail how it interacts with the rest of the processor microarchitecture including the memory subsystem.

As far as we know, the Best-Offset Prefetcher has been evaluated on very specific trace-driven simulators as on the one provided in the DPC-2 contest[1], or on an in-house simulator based on the program instrumentation tool Pin [5]. In this thesis, we will use a cycle-accurate, execution-driven simulation platform based on the well known Gem5 full-system simulator [2] from the University of Michigan that allows to simulate real workloads cycle-by-cycle.

In addition, in order to conform the monocore and multicore simulation workloads this work will use the SPEC CPU 2006 benchmark suite [6]. It contains a set of benchmarks in order to measure the CPU, memory subsystem, and compiler performance.

Developed tasks

During the development of this thesis, some tasks have been carried out. Some of these tasks are the following ones:

1. In-depth study of the Gem5 simulator and the different alternatives for prefetching.

2. In-depth study of the literature about prefetching as well as the Best Offset prefetching mechanism.

3. Setting up the SPEC CPU 2006 environment. In addition, the different benchmarks were built and compiled for the x86 ISA.

4. Prefetch framework adaptation to provide the Gem5 simulator the infrastructure to be able to implement a prefetching mechanism.

5. Implementation of the Best Offset Prefetcher and the different optimizations.

6. Writing the thesis text.

[1] 2nd Data Prefetching Championship, 2015. `http://comparchconf.gatech.edu/dpc2/`

Thesis contents

This thesis is organized as follows: In Chapter 1, some prefetching background is introduced and the Best Offset prefetcher is explained. In Chapter 2, the changes to turn Gem5 into a prefetching-aware simulator are described in addition to the Best Offset prefetcher implementation. In Chapter 3, the environment parameters and the workloads used for evaluation are discussed. Finally, in Chapter 4 the results obtained from the simulator are analyzed and the Conclusion Chapter brings to an end the thesis.

Chapter 1

State of the Art

In this thesis we are going to focus on general-purpose microprocessor architecture. This architecture is usually composed by tens of cores that have associated one to two levels of private cache. Besides, there is a shared level of cache or the last level cache (LLC), main memory, and an interconnection network between the core units, the LLC, main memory, and the peripherals. For instance, the memory hierarchy diagram of one *Skylake*[2] microprocessor is provided in Fig. 1.1. In this figure we can see two cores, each one with two levels of private caches L1 and L2, L1 subdivided in instructions and data. A third level, L3, is shared between both cores, named in Fig. 1.1 as the shared LLC.

Cache memories are used to reduce the average memory access time. There is a trade-off between cache memory size and latency: The closer is the cache level to the core the lower is the access time. However, these cache levels cannot be as large as the upper levels ones because, in order to keep this low access time, the size has to remain low as well.

With the progress of the semiconductor process technology, the processor clock cycle time has been significantly reduced [4]. However, the decrease in off-chip memory access time has been much less than that in the processor clock cycle time because the improvement in off-chip memory technology has primarily resulted in a large memory capacity. As a result, the off-chip main memory latency of modern processor chips has become more than hundred of processor clock cycles, and so the pipeline stall time due to misses.

Memory latency is a major factor in limiting CPU performance and prefetching is a well-known mechanism to hide memory latency. Prefetchers operate trying to predict the memory accesses that are going to be requested in the future. It consists of fetching cachelines that have not been requested by the program yet.

[2]Name that receives a well known Intel microarchitecture. `https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf`

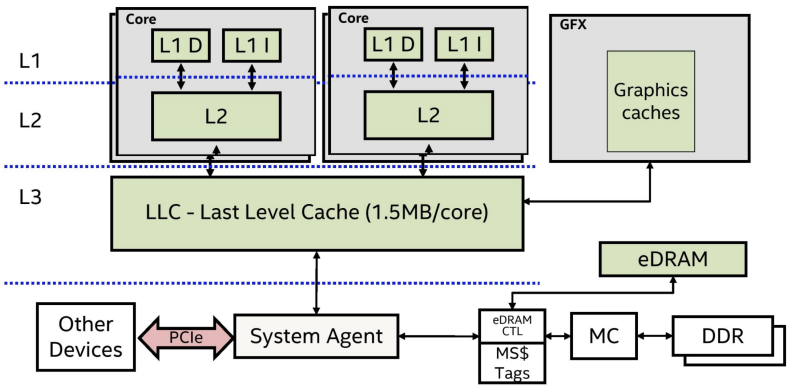


Figure 1.1: A block diagram of one microprocessor following Skylake microarchitecture. Image from `https://www.anandtech.com/show/10602/memory-frequency-scaling-on-skull-canyon`

1.1 Prefetching terminology

A few concepts are worthwhile to be introduced before talking about prefetching. Some of these are the following:

- A *prefetch hit*: A cacheline has been brought to the cache module by a previous prefetch request. An issued request from a core hits on this cacheline.
- *Prefetch distance*: It is the number of cacheline addresses that are between the prefetched cacheline address and the cacheline address that has generated the prefetch request of the core.
- The prefetcher *coverage* is the ratio between the number of misses without prefetching and the number of misses with the prefetching mechanism.
- *Accurate* or *useful* prefetches are the ones that eliminate original misses.
- *Harmful* are the prefetches that induce misses by replacing useful data.
- A *timely* prefetch is the one whose request is completed before the core ask for the data. However, a *late* prefetch is the one that is not issued because the cacheline has been already requested by the core.
- A *completed* prefetch is a prefetch request that has been successfully issued and completed.
- A *unuseful* prefetch is a completed prefetch that is evicted without being hit by a load operation.

- *Pollution* is a measure of the disturbance caused by prefetched data to the cache module.
- *Aggressiveness* sums up the number of prefetch requests that are issued by a prefetching mechanism at the same time.

1.2 Prefetching background

Prefetching can be approached in two main ways: software and hardware prefetching [7]. **Software prefetching** relies on the programmer or the compiler that are the ones in charge to fill the code with prefetching intrinsics. All the logic of the prefetching mechanism remains either in the application or the compiler. **Hardware prefetching** consists of having a dedicated hardware mechanism that watches the data access patterns of the memory requests and tries to predict the ones that are going to be accessed in the future. In this thesis we will focus on hardware prefetching.

Compared to instruction access patterns, data access patterns show less regularity, which makes data prefetching more challenging [8]. Instruction prefetching mechanisms might make sense for commercial workloads that produce misses in the L1 and L2 caches due to the large instruction working set size of these applications. However, for applications with a negligible I-cache miss rate (e.g., scientific), instruction prefetching is not required [8].

Depending on where the prefetching engine resides, we can classify the prefetching mechanisms as core-side or memory-side. In **core-side prefetching** the prefetching logic remains in the cache hierarchy and the prefetch requests are issued from there while in **memory-side prefetching** the prefetching engine resides in the main memory subsystem. Memory side prefetching can save precious chip space by storing metadata off-chip and can also perform optimizations at main memory side [9]. By comparison, core-side prefetching can avail more accurate knowledge of memory reference patterns and can perform cache level optimizations, such as avoiding cache pollution [10].

These memory access patterns might be regular or irregular. In order to be able to record irregular access patterns complex data structures would be needed but they cannot be afforded in hardware. Therefore, hardware prefetchers are intended for regular data access patterns and can be classified in two broad categories: immediate prefetches (agressives) and confirmation-based prefetchers (conservatives) [11].

Immediate prefetchers are the ones that issue a prefetch request as soon as they have an input address upon which the prefetch request can be built. The most common and simple example of this kind of prefetchers is the *Next-line prefetcher* [12]. Every time there is an access to the address X the prefetch request for the address $X + 1$ is issued.

Confirmation-based prefetchers are the ones that only issue a prefetch request if the prefetcher has built up previously some kind of confidence that

the prefetcher is going to be useful. A *stride prefetcher* [13] is a good example. For instance, for a stride D, when cacheline X is observed by the prefetcher mechanism for the first time, no prefetches are issued. However, the prefetching engine starts to record and wait for the access to cacheline $X + D$. Even when $X + D$ is accessed, still no prefetches are issued, only when $X + 2D$ is accessed the pattern will be confirmed and the prefetcher will issue the prefetch request for address $X + 3D$.

Immediate prefetchers have the disadvantage that they have a higher probability of issuing inaccurate prefetches. On the contrary, they can prefetch over some memory access patterns that the confirmation-based prefetchers cannot [11]. For example, consider a linked list of data structures that are exactly the size of two cachelines. A confirmation-based mechanism would consider the first cacheline as the beginning of a new pattern. However, the third cacheline access will never arrive because the linked list will jump somewhere else in memory. On the contrary, an immediate prefetcher as the Next-line one, will perfectly prefetch the second cacheline after the first access.

Hardware prefetching can be done at any cache level, in the private levels of cache or in the LLC. Prefetching at different levels leads to different possibilities and trade-offs. On the one hand, a L1 prefetcher can use some information that would be costly to propagate to the LLC. Besides, the L1 prefetcher is aware of the complete trace of the processor (loads and stores) while the LLC prefetcher is not. Another trade-off to take into account is the inaccurate prefetches, that is, the unuseful prefetches, the ones that are never requested by the application. Due to the limited capacity of the private levels of cache, the tolerance of these levels of cache to inaccurate prefetches is much less, while LLCs due to their greater capacity, tolerate it to a certain extent.

Finally, with the aim of achieving a greater coverage, that is, making the prefetch requests to include a greater number of future memory accesses, results in a greater number of total prefetch requests. Consequently, the on-chip interconnection network traffic increases, this may carry a network congestion as well as performance detriment [14].

1.2.1 Tagged Prefetcher

The offset prefetching is a generalization of the next-line prefetching. When a cacheline X is requested, the prefetcher issues the prefetch $X + O$ where O is the *prefetch offset*. In the case of the next-line prefetching $O = 1$.

One example of an offset prefetcher is the Tagged Prefetcher [12]. It is a hardware, data, core-side, and immediate prefetcher. The main idea of this prefetcher is that it requests the neighbouring cachelines from one cacheline when it is accessed. The Tagged prefetcher adds a *prefetch bit* to each cacheline to indicate if the cacheline has been prefetched. Whenever there is a miss or a prefetch hit a new prefetch request is issued. Prefetch requests are issued with a certain distance and aggressiveness. A certain distance to the original

cacheline is needed in order to avoid the request latency and a certain aggressiveness in order to gain more coverage. See Section 2.4 for more details of the implementation.

Algorithm *Tagged algorithm* **is**
  Data:
  Aggressiveness: Number of prefetches that are issued on every miss or prefetch hit on line X.
  Distance: Number of cachelines between the cacheline that suffers the miss or the prefetch hit and the first one to be prefetched.

  Function *LLC_Read_Request (LineAddress X, bool cache_hit, bool prefetch_hit)* **is**
    if ***not*** *cache_hit* ***or*** *prefetch_hit* **then**
      for i = *Distance* ***to*** *Distance+Aggressiveness-1* **do**
        $issuePrefetchRequest(X + i)$;
      end
  end
end

Algorithm 1: Pseudocode of the Tagged prefetching algorithm

1.2.2 Sandbox

Recently, Pugsley et al. introduced Sandbox Prefetching [11]. It represents another class of prefetchers. It combines ideas of global confirmation-based prefetchers and immediate ones in order to perform prefetches aggressively and safely. It enables the use of aggressive and immediate offset prefetchers in a safe and sandboxed environment in order to avoid their limitations, that is, neither the cache nor the memory bandwith are disturbed [11].

Sandbox prefetching tracks different offsets at run-time by adding the possibles prefetch addresses to a Bloom filter[3], rather than actually fetching the data into the cache. Subsequent cache accesses are tested against the contents of the Bloom filter to see whether the prefetcher under evaluation could have accurately prefetched data. Real prefetches are issued when the prefetcher under evaluation exceeds a threshold [11].

This is not a *stream* prefetcher, but what Pugsley et al. call an *offset prefetcher*. The main difference between an offset and a stream prefetcher is that offset prefetchers do not try to detect streams. Pugsley et al. showed that the Sandbox prefetcher matches or even outperforms the more complex AMPM prefetcher [15] that won the DPC-1 contest[4].

However, the Sandbox prefetcher does not take into account prefetch timeliness [1]. The Best Offset Prefetcher proposed by Michaud is an off-

[3]It is a probabilistic data structure used to test whether an element is a member of a set.

[4]1st Data Prefetching Championship, 2009. `https://www.jilp.org/dpc/`

set prefetcher that takes timeliness into account and will be explained below, since it is the object of this thesis.

1.3 Best Offset Prefetcher

Taking into account the previous assessments and classifications, the Best Offset prefetcher (BOP) [1], which is the target of this thesis, is a hardware, core-side, immediate prefetcher and it is intended to work as an LLC prefetcher. It is an offset prefetcher, it outperforms the Sandbox prefetcher with equal hardware [1].

The main idea of the mechanism is to continually adapting the prefetching offset to the needs of the application. It has a learning mechanism that decides which offset, among several candidates, that fits with the current application behavior. In the solution proposed by Michaud [1], a *recent request* (RR) table is built in order to keep track of the previous accesses that have generated completed prefetches. In addition, the BOP offers an *offset list (OL)* that is the set of predefined offsets that are being evaluated. Besides, there is a *score list (SL)* where each score is associated to each offset. Each *score* is a counter of hits in the RR table for the given offset, that is, how good is the current offset regarding to previous prefetches.

A schematic view of a BOP is shown in Figure 1.2. The symbol BO represents the current *prefetch offset* that is being used for prefetching. BO is a global variable and it is continuously being updated at the end of every *learning phase*. When a read request to the cacheline X arrives to the LLC cache, if it is a miss or a prefetch hit, a prefetch request to the cacheline $X + BO$ is issued.

1.3.1 Best offset learning

The current prefetch offset is set dynamically trying to be adapted to the application behavior. There is an algorithm, called "learning phase" that tries to find the best prefetch offset by evaluating some different offsets. The main idea is the following: an offset OL_i might be a good offset if there has been an access for the cacheline $X - OL_i$ when a cacheline X is accessed. This temporal locality has two main constraints. The first one is that the time between both accesses cannot be too long because the prefetch requested after the first one may be evicted before the second one is accessed. The second one is that the period of time between both accesses cannot be less than the latency of a prefetch to be completed (*late* prefetch).

The algorithm, see Algorithm 2, can be divided in two main functions, that is, the BOP logic is triggered when either one of the following events occurs: 1) The LLC is filled with a cacheline (P) and 2) the LLC receives a read request to a cacheline (X).

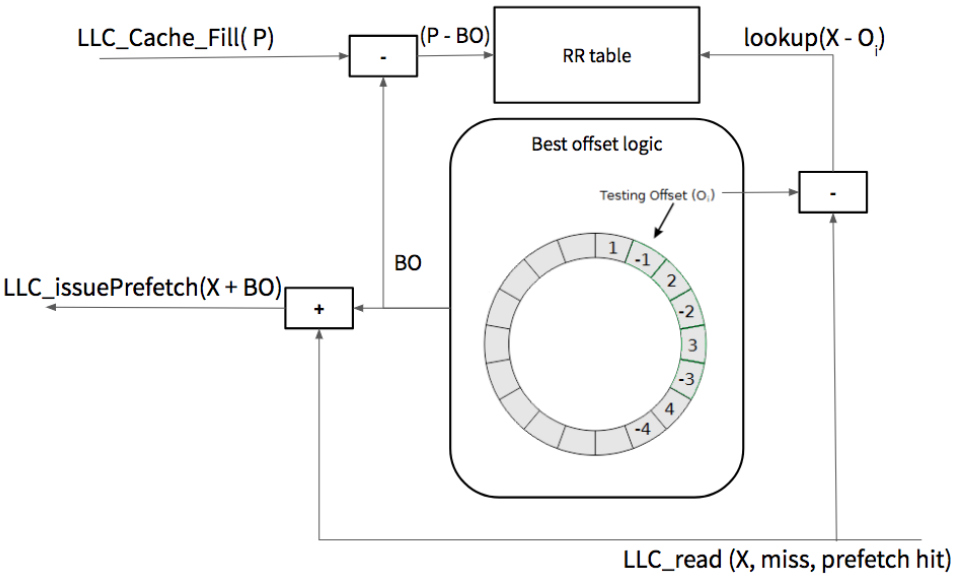


Figure 1.2: Schematic view of a BOP.

1. Every time a cacheline P is going to be inserted to the LLC, if it is a prefetch request being completed and arriving to the LLC, the address that generated that prefetch $(P - BO)$ is inserted into the RR table.

2. Every time there is a request to the LLC cache, if it is a miss or a prefetch hit, an offset OL_i is tested by looking up in the RR table. If the cacheline $X - OL_i$ is in the RR table, i. e., a prefetch for the cacheline $X - OL_i + BO$ has been completely prefetched. Besides, this also means that if the prefetch had been issued with OL_i instead of BO, it would have been a prefetch for the cacheline X.

```
Algorithm BOP algorithm is
    Data:
    BO: Golbal variable. It is the Best Offset, used to issue prefetches.
    It is updated every learning phase.
    RRTable: Recent Request table.
    OL: Set of offsets to be evaluated every learning phase.
    i: Index of the OL that indicates the offset being tested at the
    moment.
    SL: Set of scores associated to the offsets.
    prefetching: Boolean indicating whether the algorithm is issuing
    prefetch requests.
    round: Counter of the learning phase rounds.

    Function LLC_Cache_Fill (LineAddress P, bool prefetched) is
        if prefetched then RRTable.insert(P − BO) ;
        if not prefetching then RRTable.insert(P) ;
    end

    Function LLC_Read_Request (LineAddress X, bool cache_hit, bool
     prefetch_hit) is
        if not cache_hit or prefetch_hit then
            if RRTable.isPresent(X - OL_i) then SL_i + + ;
            i ← (i + 1)%NUMOFFSETS;
            if i = 0 then round + + ;

            //End of learning phase?
            if SL_i = SCOREMAX or round = ROUNDMAX then
                best_score ← SL_i;
                BO ← OL_i;
                for j = 0 to NUMOFFSETS do
                    if SL_j > best_score then
                        best_score ← SL_j;
                        BO ← OL_j;
                end
                SL.reset();
                prefetching ← True;
                if best_score ≤ BADSCORE then
                 prefetching ← False ;
                round ← 0;
            if prefetching then issuePrefetchRequest(X + BO) ;
    end
end
```

Algorithm 2: Pseudocode of the BOP algorithm

The BO, i. e., the offset that is currently being used for issuing the prefetch requests is updated every *learning phase*. A learning phase consists of several

rounds, in each round all the offsets are tested once, each one in different read requests to the LLC. At the start of each learning phase, all the scores are reset to 0. On every LLC access (miss or prefetch hit) one offset OL_i is tested; if $X - OL_i$ hits the RR table, the score SL_i associated to the tested offset (OL_i) is incremented. The current learning phase ends when either one of the following events happens first: one of the scores reaches the maximum value *SCOREMAX* or the number of rounds reaches *ROUNDMAX*. When the learning phase finishes, the offset whose associated score is the greatest becomes the new BO that is going to be used for issuing prefetches.

1.3.2 Prefetch throttling

According to the previously described behavior, one prefetch request is issued every time there is a LLC miss or prefetch hit. Hence, the BOP is a *degree-one* prefetcher. Generalizing this definition, we could say a degree-two BOP would be the one that issues two prefetches every time an access happens; for instance, one with the best offset and another with the second best offset. This might bring some extra performance for applications with irregular access patterns [1]. However, this would increase the number of prefetch requests, so that putting more pressure on memory bandwith.

However, BOP is still more agressive compared to other algorithms such as the confirmation-based ones, for instance, stream prefetching. Useless prefetches issued on irregular access patterns waste energy and memory bandwith. Consequently, we can observe that the *best score* at the end of each learning phase gives some information about prefetching accuracy. If this score is too low, this means that the prefetches with that offset will probably fail. Therefore, we will define a fixed threshold, *BADSCORE*; when the best score is not greater than this BADSCORE, the prefetcher is turned off. It is important to note that the best-offset learning will never cease. However, when the prefetcher is turned off, the insertion in the RR table changes: when a cacheline X is fetched it is inserted in the RR table instead of inserting $X - BO$ if it was a prefetch.

There are many implementation details that may differ between distinct approaches of the BOP. This first approach of the BOP algorithm, see Section 2.3.1, will be the first version implemented in this thesis, corresponding to the *BOP1* we will evaluate in Chapter 4. The implementation details will be described in Section 2.3.

Chapter 2

Implementation

As far as we know, the Best-Offset Prefetcher has been evaluated on very specific trace-driven simulators as on the one provided in the DPC-2 contest[5], or on an in-house simulator that is based on the program instrumentation tool Pin [5]. Consequently, this thesis will use a cycle-accurate, execution-driven simulation platform based on the well known Gem5 full-system simulator [2] from the University of Michigan that allows to simulate real workloads cycle-by-cycle.

To work over a simulator like Gem5, some previous study is needed to understand its infrastructure. Therefore, the first task has been to deeply study the Gem5 infrastructure and to understand the Ruby memory system and its support for prefetching.

2.1 Gem5 introduction

Gem5 [2] is a modular discret event driven computer system simulator platform. This means that the components of Gem5 can be rearranged, parameterized, extended, or replaced easily to suit your needs.

Gem5 is written primarly in C++ and Python. It can simulate a complete system with devices and an operating system in full system mode (FS mode), or user space only programs where system services are provided directly by the simulator in syscall emulation mode (SE mode).

Gem5 is mainly constituted of two components: the CPU and the memory subsystem. Each of them can be subdivided in many other blocks. This modelling, see Figure 2.1, allows the researcher to choose between different modules. These modules are basically ruled by a trade-off between simulation time and accuracy of the model. As we can see in Figure 2.1, on the top side, there is the CPU model divided in the ISA and the CPU. The CPU module, at the same time, is divided in submodules whose main difference remains in

[5] 2nd Data Prefetching Championship, 2015. `http://comparchconf.gatech.edu/dpc2/`

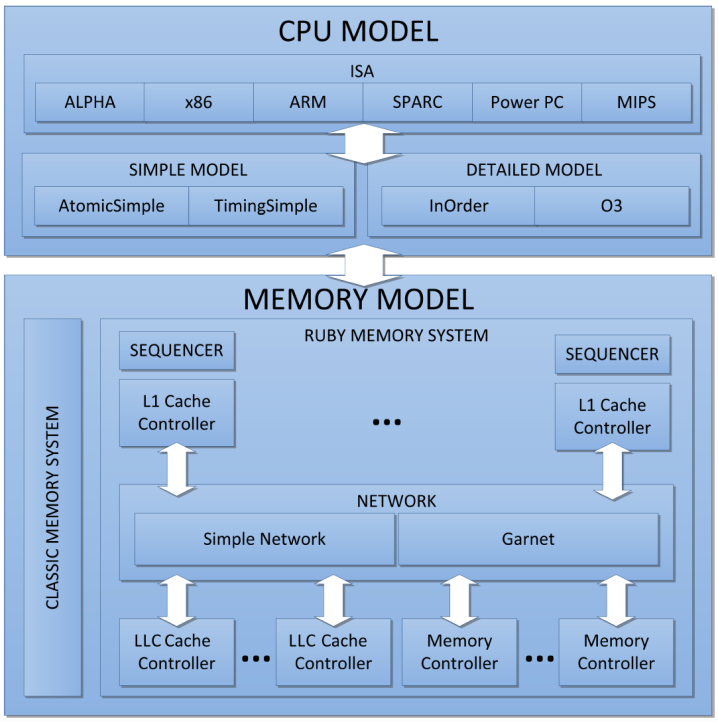


Figure 2.1: Schematic view of the Gem5 simulator. Image from [3]

the simulation time - accuracy trade-off. In the bottom, there is the memory model subdivided in the classic and the Ruby one.

One of the most important things about the simulator is that it is relatively new and it is continuously under development. Some of the features are not implemented yet and the documentation is very scarce.

2.1.1 Memory subsystem

The Gem5 simulator includes two different memory system models, Classic and Ruby. The Classic model provides a fast and easily configurable memory system, while the Ruby model provides a flexible infrastructure capable of accurately simulating a wide variety of cache coherent memory protocols.

Although the Ruby memory system takes more time to simulate, it has more detail and versatility. The classic memory system already has their own prefetch engines but can neither simulate the network on-chip (NoC) nor the coherent memory protocols. These two features are really important if we want to know the real implications of the prefetcher. Consequently, our thesis will use Ruby as the memory subsystem, because it provides more versatility and flexibility at the expense of an increase in the simulation time.

In addition, the last versions of the Gem5 Ruby subsystems, see Figure 2.2, include GARNET [16]. It is a detailed interconnection model inside Ruby and

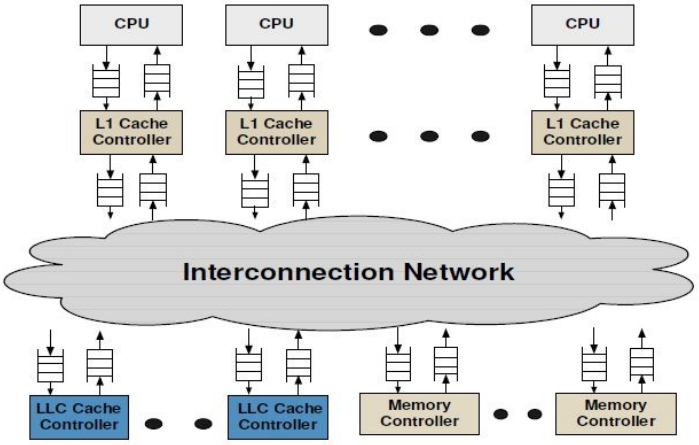


Figure 2.2: Schematic view of the memory hierarchy in Ruby. Image from `http://gem5.org/Ruby`

Gem5 that allows to simulate the NoC more in detail. Besides, it brings back important network utilization statistics really interesting in order to realize the prefetcher implications for the NoC congestion.

2.1.1.1 MOESI Protocol

The MOESI coherence protocol needs to be changed, so it is worthwhile to introduce the MOESI base protocol provided by Gem5. First of all, the MOESI protocol used by the framework is a two-level (L1 and LLC) and non-inclusive MOESI protocol. This means that the data that is present in the L1 caches cannot be present in the LLC.

It is important to note that the microarchitecture introduced in Figure 1.1 had two private levels of cache (L1 and L2) and a third shared one, that is the LLC. However, due to the fact that the MOESI protocol implemented only has one level of private cache, only one level of private cache is used. The first level is split in data and instructions and the second one corresponds to the shared LLC.

These protocols have well defined stable states which are the states with no on-going transitions, that is, the states where any request can be satisfied. Apart from these stable states, there are the transient states, the ones where the cache controller is waiting to receive an external event. For example, if a L1 cache with a cacheline in an *Invalid* stable state receives a read request from the processor, it will ask for the cacheline to the upper level. Besides, it will change the state to a transient one and will return back to a stable state when it receives the data requested for the processor.

The L1 stable states of cachelines in Gem5 are the following:

- **MM**: The cacheline is held exclusively by this node and is potentially modified.
- **O**: The cacheline is owned by this node. It has not been modified by this node. No other node holds this cacheline in exclusive mode, but sharers potentially exist.
- **M**: The cacheline is held in exclusive mode, but not written. No other node holds a copy of this cacheline. Stores will change the state to MM.
- **S**: The cacheline is held in shared state by 1 or more nodes. Stores will change the state to MM & I (for the others sharers).
- **I**: The cacheline is invalid.

2.2 Prefetch Framework Adaptation

Due to the fact that the BOP proposal is intended to work in the LLC, the coherence protocol relative to that level should have some kind of support to allow issuing prefetches. Unfortunately, the Gem5 coherence protocol implementations do not have this support. Consequently, this coherence protocol infrastructure had to be implemented. A work developed during the Ph.D of Martí et al. in the Universitat Politècnica de Catalunya [3] consisted of creating a prefetch framework that handles prefetches in all cache levels over a MOESI directory-based protocol.

This framework was developed over an out-of-date version of the Gem5 simulator, so the code was deprecated regarding to the latest one. There exists a big interest in the Gem5 community to have this framework included in the Gem5 official release. In this Section, all the changes done to the Gem5 simulator are explained, the Ruby memory subsystem and the coherence protocol in order to fully adapt the prefetch engine to the latest Gem5 version. One of the first tasks in this thesis consisted of turning the deprecated code into a useful framework inside the latest Gem5 official release version.

In order to convert Gem5 to a prefetching-aware simulator, some new modules must be added to the simulator, besides making some changes to the current code [3]. Fig. 2.3 gives a schematic representation of the modified (blue) and new (green) modules added to Ruby.

The prefetching module is included as a protocol-independent component. Therefore, a new simulation object (*SimObject* regarding to the Gem5 terminology) is added to act as a wrapper between the cache controller and the prefetcher. One of its main functions is to communicate the prefetch engine the main events that occur in the cache: read, write, and eviction. When one of these events happens, the wrapper must communicate the prefetch engine the corresponding information.

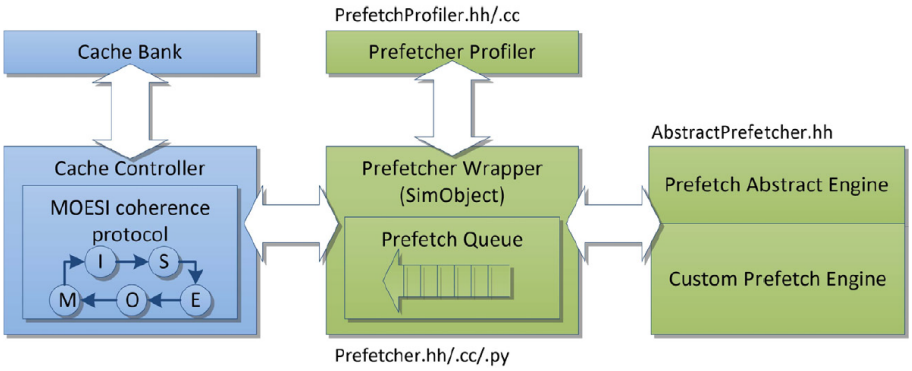


Figure 2.3: Schematic representation of the framework modules (green) and the existing Ruby infrastructure (blue). Image from [3]

In addition, we can see in Fig. 2.3 that the prefetch queue is allocated in this wrapper. This queue holds the prefetching requests generated by the prefetch engine before the coherence protocol checks if the requests make sense regarding the coherence information and issues them into the network on chip. It works as a simulation structure but it can also emulate real hardware constraints the prefetch engine may suffer. Whenever the prefetch engine generates a new prefetch request, it is stored in the prefetch queue and the cache controller is the responsible for picking up the requests and issuing them into the memory system.

Due to the fact that the objective is to make an adaptive prefetching module where any prefetch algorithm can be attached, an abstract class (*AbstractPrefetcher*) has been created in order to hold the prefetch engine activity. The custom prefetch engine should inherit this abstract class where all the functions the custom class must define are declared.

2.2.1 Official release version implications

In order to successfully turn the deprecated code into the official release version I had to carefully study the dependencies between the C++ classes in addition to identify them and the basic types. Moreover, it was also necessary to identify and modify some framework functionalities that were differently implemented in the new Gem5 version. Coming up next, some of the differences between both codes will be enumerated:

- The memory addresses were represented by a C++ class in the out-of-date version of the simulator. However, in the official release, they are defined as a basic type. Consequently, everything related the use of memory addresses had to be changed because they were implemented as an object instead as a predefined long integer.

- The cache controller was implemented in a different way. In the official release some of the functions are implemented in the *AbstractController* while this functionality was implemented in other different classes. This made the *prefetch wrapper* to interact in a different way with the cache controller.

- The statistics was implemented more in a hard-coded way in the old version. In the new one, a functionality called *RegStats* is included to ease the final print of them. The prefetch framework statistics were changed using the RegStats functionality.

- Some of the classes dependencies changed. Therefore, it has been necessary to study the inheritance and polymorphism of the different classes as well as the directory tree in order to adapt the framework classes to the up to date version.

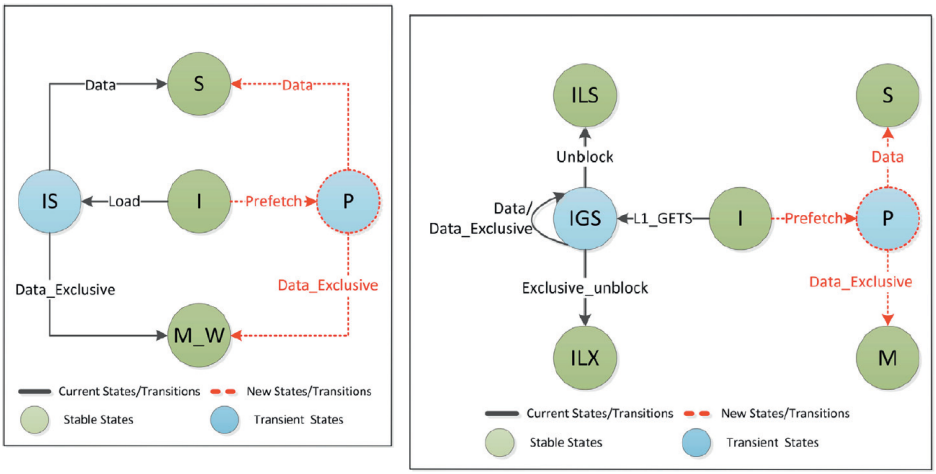
2.2.2 Changes to the cache controller

The cache controller is responsible for checking if there are pending prefetch requests in the queue. Besides, if necessary, it will arbitrate between these pending requests and the regular cache requests arriving to the current cache level. When there is no request from the processor pending to be processed, the controller will check if there are ready requests in the prefetch queue. If so, the oldest one will be issued unless the coherence state for that cacheline is valid. If this is the case, it will be discarded because it means that it is present in the current cache level, so there is no need to be prefetched. The request previously issued will trigger a *Prefetch* event in the coherence protocol and will bring the data from upper levels of cache or from another L1 private cache.

There is another important issue to take into account. Due to the fact that the LLC is shared among different cores, it could happen that the prefetch engine allocated in one core issues a prefetch whose owner is another one. Therefore, before issuing the request to main memory, it should be checked who is its owner. If it is a different one, the request should be forwarded to it before issuing it to main memory.

2.2.3 Changes to the MOESI protocol

Martí's framework was intended to provide prefetching support to the LLC as well as to the L1 cache. Therefore, the changes done to the used coherence protocol can be divided in two parts: 1) The changes related to the L1 cache coherence protocol due to prefetching that are illustrated in Fig. 2.4a and 2) the changes related to the LLC coherence protocol due to prefetching also exemplified in Fig. 2.4b.



(a) L1 coherence protocol with prefetching.

(b) LLC cocherence protocol with prefetching.

Figure 2.4: Changes in the MOESI protocol state machine due to prefetching. Image from [3].

For the L1 cache, when a load arrives, the controller will check if the target cacheline state is valid. If not, the request will be forwarded to the LLC and the state will be changed to a transient one until the data is received. When the data arrives, the state associated to the cacheline will be changed depending on if it is exclusive data or not and the core will be notified. As we can see in Fig. 2.4a, when a *Prefetch* event is triggered a new transient state (Prefetch, P) is reached waiting for the data. When it arrives, a stable state is also reached with the difference that the core is not notified this time.

Due to the fact that the LLC in the MOESI protocol implemented in Gem5 is shared and split among the different cores, every LLC fragment is only responsible for just a part of the memory space. Besides, it only stores the coherence information of the cachelines it is responsible for and the cachelines in the private caches below it. Therefore, as we can see in Fig 2.4b, when a load operation reaches the LLC and the state of the cacheline requested is invalid in the LLC and in the L1s, the request is propagated to main memory and the state is changed to a transient one. Since the MOESI protocol is non-inclusive, when the data arrives from main memory the data is not stored in the LLC but sent to the L1 that has requested it. The distinction between the LLC coherence protocol with prefetching and without prefetching is greater than the distinction between the L1 coherence protocol with prefetching and without prefetching.

2.2.4 Prefetching statistics

In order to pick up the statistics related to the prefetcher activity, the aim is to use the wrapper as a central point to collect the statistics. Any *SimObject*, as the wrapper is, may have one *Profiler* class associated. In order to runtime collect the statistics, the class *PrefetchProfiler*, see Fig. 2.3, is directly attached to the wrapper. The statistics allow to classify the prefetches in terms of their successes, as well as measuring the time the prefetch requests take in the memory hierarchy [3]. The initial set of statistics that are implemented in the prefetch framework can be found below:

- **Total prefetches**: The number of prefetch operations generated by the prefetching engine.
- **Completed prefetches**: The number of generated prefetch operations successfully issued and completed.
- **Useful prefetches**: The number of completed prefetches that were hit by a load operation before eviction.
- **Unuseful prefetches**: The number of completed prefetches that were evicted without being hit by a load operation.
- **Late prefetches**: The number of prefetch requests that are not issued because the cacheline is already requested by another load operation.
- **Canceled prefetches**: The number of prefetch requests canceled by the coherence protocol.
- **Overflowed prefetches**: The number of prefetch requests discarded because of overflow in the prefetch queue, see Figure 2.3.

2.3 Best Offset Implementation

The first step in order to add one prefetch engine is to create a new class that defines the behaviour of this new engine. This class must inherit the *AbstractPrefetcher* one and it must redefine the constructor and destructor methods according to the requirements of the Best Offset engine. Once the *BestOffsetEngine* class is defined, it is needed to be added to the *Sconscript*[6] file like this:

```
Source('BestOffsetEngine.cc');
```

The last step is to link the constructor of the new prefetch engine to the *SimObject*. If we would like to use this new prefetch engine in the executions we had to set the following execution flags in the command line: `--l2_prefetcher=BESTOFFSET`

[6]The Gem5 compiler uses these files to recognize all the C++ classes used in the implementation

2.3.1 The first approach. BOP1

The first approach tries to take one of the simplest implementations. This implementation corresponds to the one explained in Section 1.3. A quick summary of the values taken by the algorithmic parameters can be found in Table 2.1.

We are going to consider 4KB page boundaries and cacheline sizes of 64B. Therefore, it makes no sense to consider offsets greater than 64. The *OL* is hardcoded in the code and corresponds to the one submitted for the 2nd Data Prefetching Championsip by Michaud [17]. The offsets considered follow the form $2^i \times 3^j \times 5^k$ with $i, j, k \geq 0$.

The RR table is built as a direct-mapped 256-entry table indexed by a hash function. The impact of the number of entries from 32 to 512 has been studied by Michaud [1]. He claims that the performance gap can be noticed with the 256 entries. The hash function is a *xor-based* one. To obtain the index it xors the 8 least significant bits in the cacheline address with the next 8 ones.

In addition, Michaud studied the effect of the *BADSCORE* parameter. This parameter is a kind of threshold that indicates that the prefetcher is not going to issue accurate prefetch requests. Michaud concluded that this parameter must be less than 10 % of the *ROUNDMAX* parameter. Besides, he studied the effect of some values in this parameters. For a 4 KB page size, there is no effect in the majority of the SPEC CPU 2006 benchmarks and for the benchmarks he could see the effect, he claims that greater values hurt performance. Consequently, as a first approach, we are going to consider the value Michaud concluded for the rest of his simulations.

Number of offsets	46
OL	{1, -1, 2, -2, 3, -3, 4, -4, 5, -5, 6, -6, 8, -8, 9, -9, 10, -10, 12, -12, 15, -15, 16, -16, 18, -18, 20, -20, 24, -24, 25, -25, 27, -27, 30, -30, 32, -32, 36, -36, 40, -40, 45, -45, 48, -48}
RR table entries	256
SCOREMAX	31
ROUNDMAX	100
BADSCORE	1

Table 2.1: Values of the parameters for the first approach. See BOP1 in Chapter 4.

2.3.2 The second approach. BOP2

The second approach tries to take one of the ideas implemented and submitted for the 2nd Data Prefetching Championship [17] but not included in [1]. In Fig. 2.5 we can see the schematic view of the BOP with the main changes

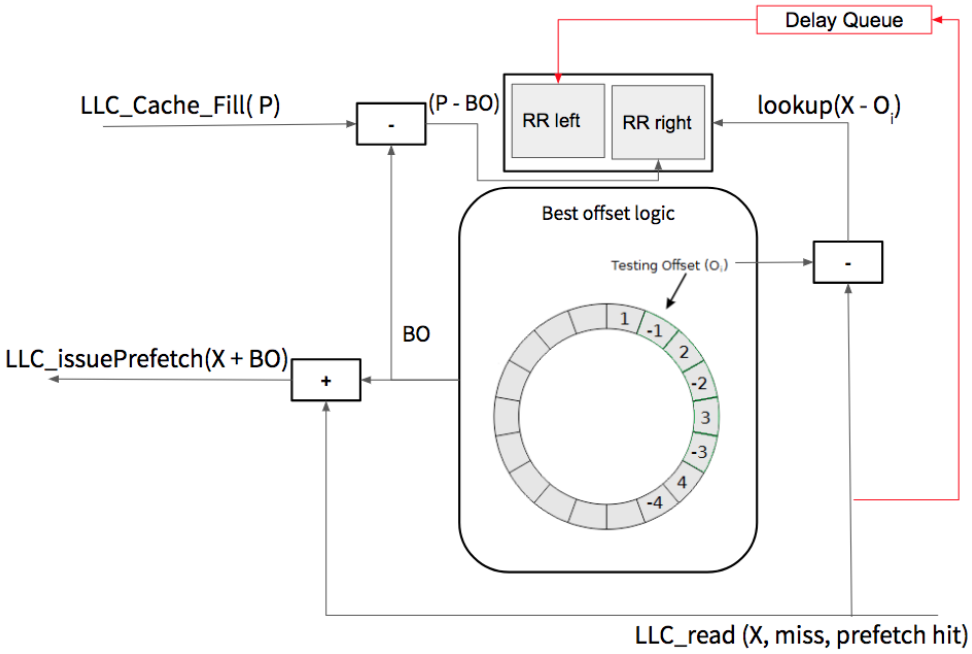


Figure 2.5: Schematic view of the optimized BOP. The changes are in red.

regarding the previous version highlighted in red. This is the BOP2 we will also evaluate in Chapter 4.

The RR table is built as a 2-skewed table, both tables indexed by different hash functions. Before, the RR table was only updated at the time a prefetched cacheline arrives to the LLC. In this second version, we will continue updating the right RR table every time a prefetched cacheline arrives to the LLC and we will update the left RR table at the time there is a request (miss or prefetch hit).

The rationale for updating the RR table at LLC fill time is to find an offset that yields timely prefetches whenever possible. However, Michaud later found that striving for timeliness is not always optimal [17]. There are cases where a small offset gives late prefetches but greater coverage and accuracy. Therefore, a simple solution is proposed: the *delay queue*. When there is a request to the LLC to the address X, this address will be inserted in the delay queue which will hold the address for some fixed time and then, it will be inserted into the left RR table.

The RR table activity of the first approach, BOP1, has been monitored and we could see that it was always full and being updated regularly. For this reason, and due to the fact that the update of the RR table is more aggressive in this second approach because of the delay queue, we have decided not to divide the RR table into left and right making each one of 128 entries but making each one of 256 entries.

Regarding the delay cycles, Michaud claims that the value of the delay in cycles should take value between the latency of a LLC hit and a LLC miss. In order to take a value according to our environment, we have monitored the access delay time to the LLC. We have observed that for a LLC hit, the L1 cache is served the data in less than 30 cycles while the LLC misses take more than 150 cycles in getting the data cacheline. Therefore, a value around $[50, 100]$ should be suitable, so we have taken 60 cycles since Michaud chose that value for it.

A quick summary of the values taken by the parameters can be found in Table 2.2.

Number of offsets	46
OL	{1, -1, 2, -2, 3, -3, 4, -4, 5, -5, 6, -6, 8, -8, 9, -9, 10, -10, 12, -12, 15, -15, 16, -16, 18, -18, 20, -20, 24, -24, 25, -25, 27, -27, 30, -30, 32, -32, 36, -36, 40, -40, 45, -45, 48, -48}
Right RR table entries	256
Left RR table entries	256
Delay Queue entries	15
SCOREMAX	31
ROUNDMAX	100
BADSCORE	1
Delay Queue delay	60 cycles

Table 2.2: Values of the parameters for the second approach. BOP2 in Chapter 4.

2.4 Tagged Prefetcher

When the framework described in Section 2.2 was developed by Martí et al., they wanted to implement some simple prefetch mechanisms to verify the correctness of the simulator. Therefore, the Tagged prefetcher implemented by them has been turned into useful code for the latest version of the Gem5 simulator. There are two main reasons to do so: 1) The importance to compare the BOP results with the original and simplest Tagged one and 2) Verify that the framework adaptation has been successful comparing the results obtained in the old version and in this thesis.

Martí et al. studied the effect of the aggressiveness parameter on this prefetcher [3]. They recorded the number of prefetch requests issued per kilo instructions and concluded that the Tagged prefetcher works in a simplistic way. The Tagged prefetcher with medium or high aggressiveness issues a lot of prefetch operations and the accuracy is low. Consequently, the cache becomes filled with a lot of useless data which evicts useful data (cache pollution).

This effect leads to an increase in the number of misses for this cache level. Therefore, in this thesis we are going to use a low aggressiveness since it seemed to work best in Martí's simulations [3].

More specifically, we are going to use value 1 for aggressiveness, that is, we are going to issue one prefetch request for each miss or prefetch hit. For the distance parameter, we are going to use value 2, that is, every time there is a miss or prefetch hit on cacheline X, we will issue one prefetch request for cacheline $X + 2$.

Chapter 3

Methodology

In order to validate new hypotheses and to quantify the improvements of new models in terms of cost and/or performance, it is necessary to use complex simulators. In addition, it is also needed to use a variety of workloads that are representative and return meaningful statistics about the proposal. Consequently, this work will use a cycle-accurate, execution-driven simulation platform based on the well known Gem5 full-system simulator [2], see Section 2.1 from the University of Michigan that allows to simulate real workloads cycle-by-cycle.

In addition, another task of this thesis has been setting up the SPEC CPU 2006 environment in order to build the benchmarks that are going to be used to get the results and to evaluate the Best Offset proposal.

3.1 Verification

One of the main problems of research and of the implementation of new models in simulators is the final ignorance about the correctness of the model implemented. Said otherwise, it is usually difficult to formally prove that the model implemented corresponds correctly to the conceptual model presented by the authors.

Some testing has been developed in order to prove that the implementation is correct and fits the conceptual model previously described. One extended technique is the use of *hints*, they consist of checking at run-time that incoherent events or situations regarding the conceptual model are not produced. Some of this checking techniques are going to be described and exemplified below:

- In-depth analysis of the code and the coherence protocol state diagram generated by the compiler.
  - Problem statement: When one prefetch request is issued in the LLC, the coherent state is changed to a transient one where the

LLC is waiting for the data. If a request arrives to the LLC at this moment, it is not a hit because the prefetch has not finished yet. However, in the implementation of the framework it is not counted as a miss nor as a late prefetch. This event should be called *partial miss* and should be accounted in the LLC misses.

  - Solution: I propose to identify this event and create a counter that is incremented every time it occurs. Therefore, at the end of the execution we will be able to adjust the statistics including this happenings in the LLC misses.

- Run-time checking that no prefetches are issued for cachelines that are valid in the cache hierarchy. An assert is used to finish the execution if this event occurs.

- Use of the statistics to check the coherence of the outcomes. Some of the checks we can do are the following ones:

  - It is a requirement that the number of useful prefetches is not greater than the number of completed prefetches.
  - It is a requirement that the number of total prefetches is the greater value and it can be decomposed in completed and cancelled.
  - Notice that the number of useful prefetches has correlation with the number of decreased LLC misses.
  - Notice that the network utilization is greater for multicore simulations than for monocore ones.

3.2 System parameters

We are going to perform both monocore and multicore simulations. The target Instruction Set Architecture (ISA) will be the x86. The memory model used is Ruby, see Subsection 2.1.1. It models inclusive/exclusive cache hierarchies with various replacement policies, coherence protocol implementations, interconnection networks, DMA and memory controllers, various sequencers that initiate memory requests and handle responses [2]. The parameters of the simulation can be seen in Table 3.1.

For both, monocore and multicore simulations we use two levels of cache, the L1 one split into data and instructions, 32 kB both. The LLC is shared between the different cores so 2 MB are allocated for each core. Therefore, we will use 2MB for the monocore simulations and 8 MB for the quad-core ones.

System Parameters	
Target ISA	x86
Coherence protocol	Two levels MOESI-Directory
L1d size	32kB
L1d associativity	8
L1i size	32kB
L1i associativity	8
LLC size	2MB/core
LLC associativity	16
Cache line size	64B
Page size	4kB
On-chip network	Crossbar

Table 3.1: Values of the simulated system parameters.

3.3 Workloads

The BOP has been evaluated with the SPEC CPU 2006 benchmark suite [6], these benchmarks are mainly programmed in C, C++, and Fortran. Another task developed in parallel with the implementation of the BOP has been to set up the SPEC environment. The final goal has been to build and compile all the SPEC CPU 2006 benchmarks under the x86 ISA in order to be able to be executed by the Gem5 simulator. In Table 3.2, the benchmark used for monocore and multicore simulations can be seen.

In order to face the representative part of every benchmark, *fast forwarding* is going to be used. It consists of executing the first instructions of each benchmark in a non-detailed way because it is assumed that the first bunch of instructions of every benchmark are intended for the initialization of the application. The non-detailed execution implies that the CPU is in the AtomicSimple mode and the memory subsystem is in the general one so the Ruby memory subsystem is not taking any statistics.

After that, *warm-up* of caches are applied during some instructions in order not to start the detailed simulation with an empty memory hierarchy. The simulation flow can be summed up to: 100 millions fast forward instructions, 50 millions warm-up instructions, and 500 millions of detailed simulation.

For the monocore simulations, we selected workloads that exhibit a non-trivial rate of LLC misses per instruction such as *lbm*, *soplex*, or *libquantum* [18]. These benchmarks are also amenable to regular prefetching. Some of these workloads do not work well with prefetching, such as *omnetpp* or *sjeng*, and are included in order to show that BOP does not hurt the performance of applications that are not prefetch-friendly.

Nonetheless, during this task of building the SPEC CPU we have found several problems. The main reason to discard some of the SPEC CPU bench-

marks was that some of the machine instructions resulting from compilation were not supported by the Gem5 x86 ISA. This is because Gem5 is not fully implemented, it is a simulator with some functionalities in development. In addition, some benchmarks presented Fortran runtime errors. Therefore, the simulations were done with the benchmarks that can be seen in Table 3.2.

Once the monocore simulations were launched, we could observe which benchmarks in our environment were generating the greatest network utilization. For the multicore simulations in four cores, each benchmark is being executed on each core. Therefore, the network traffic generated by the benchmarks are interacting with each other. Consequently, it would be interesting to choose four benchmarks to conform the *mixes* to be executed whose network utilization in the monocore simulations took importance.

Monocore benchmarks	
401.bzip2, 435.gromacs, 444.namd, 450.soplex, 454.calculix, 456.hmmer, 458.sjeng, 462.libquantum, 464.h264ref, 470.lbm, 471.omnetpp	
Quad-core mixes	
Mix 1	lbm, soplex, libquantum, gromacs
Mix 2	lbm, soplex, bzip2, hmmer
Mix 3	libquantum, gromacs, bzip2, hmmer
Simulation parameters	
Fast-forward instructions	100 M instructions
Warm-up instructions	50 M instructions
Simulated Instructions	500 M instructions

Table 3.2: Simulation benchmarks and workloads parameters.

3.4 Evaluated Models

The simulated, evaluated and compared models are the following ones:

- Base: Base system with no prefetching.
- BOP1: The first approach of the BOP mechanism, see Section 2.3.1.
- Tagged: Original and simplest Tagged prefetcher, see Section 2.4.
- BOP2: The optimizations of the BOP mechanism, see Section 2.3.2.

Chapter 4

Evaluation

In order to be able to quantify the improvements and benefits of new proposals, the statistics outcomes given by the Gem5 simulator can be used to get some performance, and behaviour metrics about the proposals. First of all, the metrics studied and evaluated are going to be introduced and then the monocore and multicore simulation results will be analyzed.

4.1 Metrics used

We need several metrics to help understand the results given by the simulator. These metrics will try to sum up the overall performance of the workloads, the impact of the prefetcher on the LLC demand misses and the network traffic and some prefetching metrics such as coverage and accuracy. The following metrics will be calculated and analyzed in all the executions:

- **Instructions per cycle (IPC)**: It is a well known metric in order to measure the performance improvement caused by a modification.

$$IPC = \frac{\#\ of\ Simulated\ Instructions}{\#\ of\ Simulated\ Cycles}$$

- **IPC speedup**: Ratio between the IPC of a pretching mechanism and the BASE system without prefetching.

$$IPC\ speedup = \frac{IPC_{PF}}{IPC_{BASE}}$$

- **LLC misses per kilo instructions (MPKI)**: It measures the number of LLC misses per every one thousand instructions.

$$MPKI = \frac{\#\ of\ LLC\ Misses \times 1000}{\#\ of\ Simulated\ Instructions}$$

- **MPKI reduction**: Ratio between the MPKI of a pretching mechanism and the BASE system without prefetching.

$$MPKI\ Reduction = \frac{MPKI_{BASE}}{MPKI_{PF}}$$

- **LLC mean access time**: It measures the mean access time, in cycles, that a L1 miss takes to be fulfilled.

$$LLC\ Access\ Time = \frac{\#\ of\ L1\ Miss\ Cycles}{\#\ of\ L1\ Misses}$$

- **LLC mean access time reduction**: Ratio between the mean access time of a prefetching mechanism and the BASE system without prefetching.

$$LLC\ Access\ Time\ Reduction = \frac{LLC\ Access\ Time_{BASE}}{LLC\ Access\ Time_{PF}}$$

- **Network utilization**: This metric is given by GARNET [16] trying to sum up the links and routers average utilization. The GARNET framework calculate this metric having some counters that are incremented every time any router or link is used. After that, they divide this accumulated value by the number of executed cycles. This metric is directly related with the MPKI one because when a miss on the LLC is produced, the demand has to be issued to main memory, which generates network traffic.

- **Coverage**: Fraction of original misses saved by prefetched cachelines.

$$Coverage = \frac{\#\ of\ Original\ misses - \#\ of\ Misses\ with\ prefetching}{\#\ of\ Original\ misses}$$

- **Accuracy**: Fraction of all launched prefetching operations that have proved to be useful, that is, that have eliminated some original misses.

$$Accuracy = \frac{\#\ of\ Useful\ prefetches}{\#\ of\ Total\ prefetches}$$

4.2 Monocore evaluation

Figure 4.1 represents the IPC speedup of the three prefetching versions related to the BASE, with no prefetching. The names of the benchmarks are on the X-axis and the speedup on the left Y-axis. In addition, in order to see the relationship with the MPKI metric, it is printed in the right Y-axis and represented with a line.

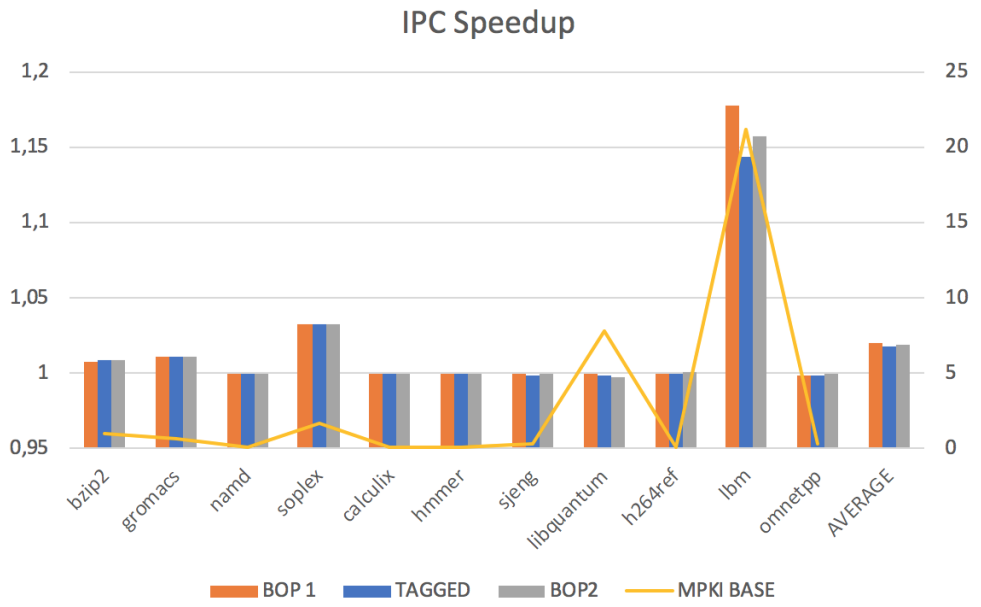


Figure 4.1: IPC Speedup (left) and BASE MPKI (right) in monocore simulations.

We can see that the IPC speedup is not noticeable in most of the benchmarks except for four of them (*bzip2*, *gromacs*, *soplex*, and *lbm*). For example, *lbm* benchmark is the one that provides the greatest speedup: 17,9 % for BOP1, 14,4 % for TAGGED, and 15,8 % for BOP2. Besides, we can notice that most of the benchmarks have a very low miss rate in LLC. For this reason, the prefetching activity related to them is very low so that they get no speedup. However, we cannot say that all the benchmarks that have a high miss rate in LLC are going to be improved in terms of performance by the prefetchers. For example, the *libquantum* benchmark has a high miss rate but it does not get any speedup from any of the prefetching mechanisms. Possibly, this is due to an irregularity in the *libquantum* memory access patterns.

If we look at the average values, we can see that the prefetching mechanism that brings at least some speedup is BOP1.

Figure 4.2 represents the mean LLC access time reduction of the three prefetching versions related to the BASE, with no prefetching. The names of the benchmarks are on the X-axis and the mean LLC access time reduction in the Y-axis.

The main observation we can do is that the mean access time to LLC gets reduced in the benchmarks that provide a certain speedup. Both metrics are strongly related because the performance speedup is taking advantage of the time reduction in the LLC accesses.

In addition, looking at the average values, the BOP1 gets the maximum

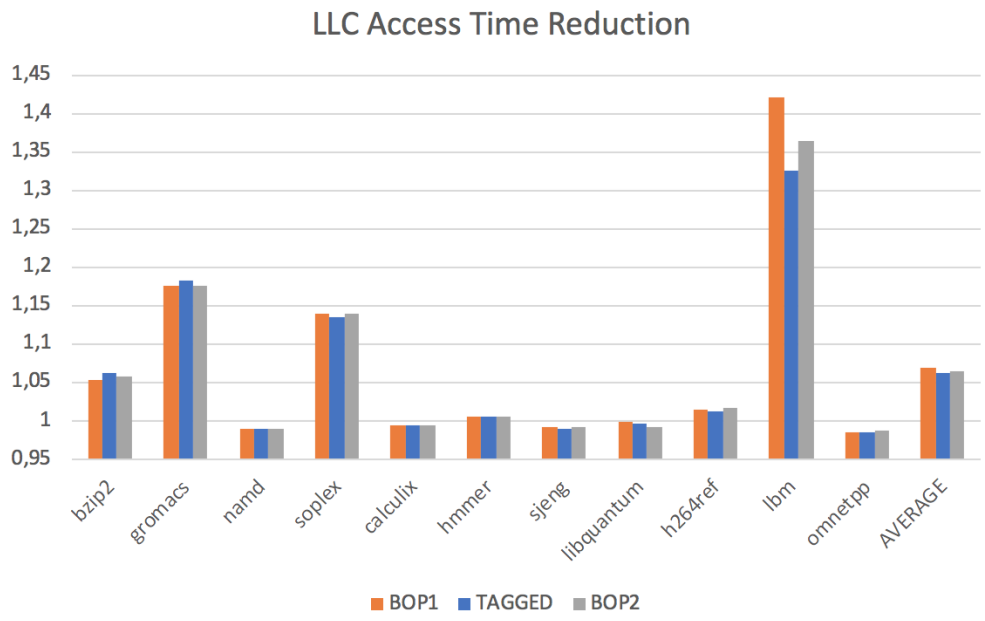


Figure 4.2: Mean LLC access time reduction in monocore simulations.

LLC access time reduction (7%) and the greatest speedup (2,1%) followed by the BOP2, 6,5% and 2%, respectively.

The network utilization is given in Figure 4.3. The names of the benchmarks are on the X-axis and the network utilization ratio can be seen in the Y-axis.

A correlation between the network utilization and the MPKI can be seen in the BASE version. The benchmarks that generate a greater MPKI also generate a greater network utilization. In addition, the prefetching mechanisms are not perfect, that is, not all the prefetch requests they issue are going to be useful. Therefore, we can see an increment in the network utilization due to these unuseful requests that were not issued in the BASE version.

As we have seen in the IPC analysis, the IPC was not noticeable for the benchmarks that have low MPKI. This relation is also visible in the network utilization increment. The increment is negligible for the benchmarks that present a low MPKI due to the low activity of the prefetcher. Moreover, the BOP1 overall network utilization, see *AVERAGE* in Fig. 4.3, is slightly greater than the BOP2 one.

The prefetching coverage and accuracy is given in Figure 4.4. The names of the benchmarks are on the X-axis and the coverage and accuracy ratios are on the Y-axis of figures 4.4a and 4.4b, respectively.

The coverage metric is directly correlated with the MPKI one because the coverage measures the fraction of the original misses, that is, the fraction of the misses of the *BASE* version, that are eliminated by the prefetching

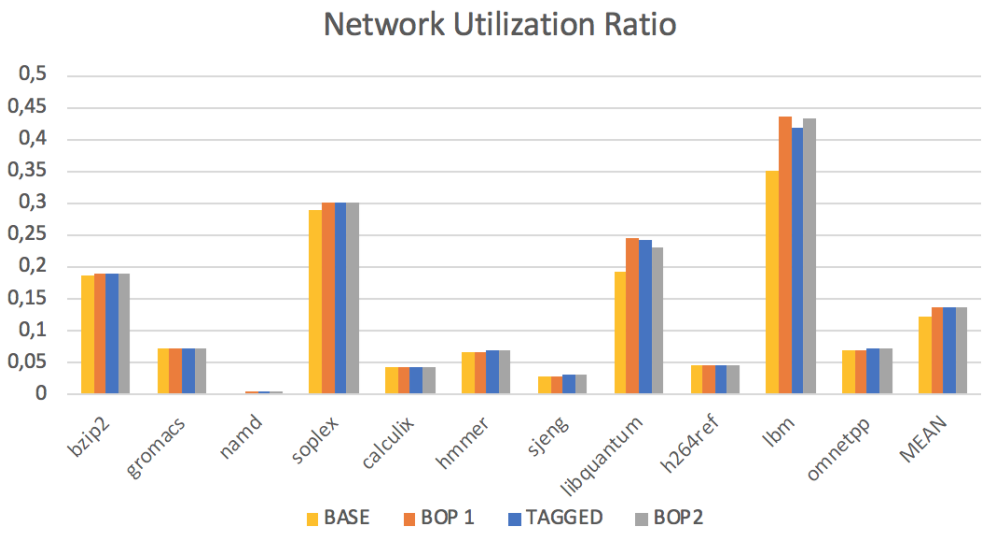


Figure 4.3: Network utilization ratio in monocore simulations.

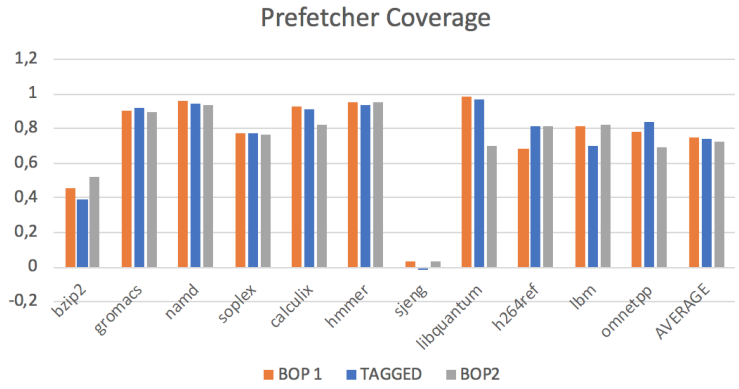
mechanisms. As it is visible in the mean, the *BOP1* and the *TAGGED* are the ones whose coverage is the greatest.

The accuracy metric analysis combined with the coverage one can be used to understand the behavior of the prefetching mechanism in each benchmark. For example, when the prefetcher has both high coverage and accuracy it means that the prefetching algorithm is recognizing the memory access patterns and is not issuing prefetch requests in excess. This can be noticed in benchmarks as *gromacs*, *namd*, *calculix*, *hmmer*, *libquantum* and *lbm*. Looking at the means we can see that BOP2 is the one that brings the greatest accuracy. Nevertheless, if the coverage ratio was high but the accuracy one was low that would mean that the prefetching algorithm is too aggressive because it is achieving coverage at the expense of issuing unuseful prefetch requests that may result in an increase of the network traffic.

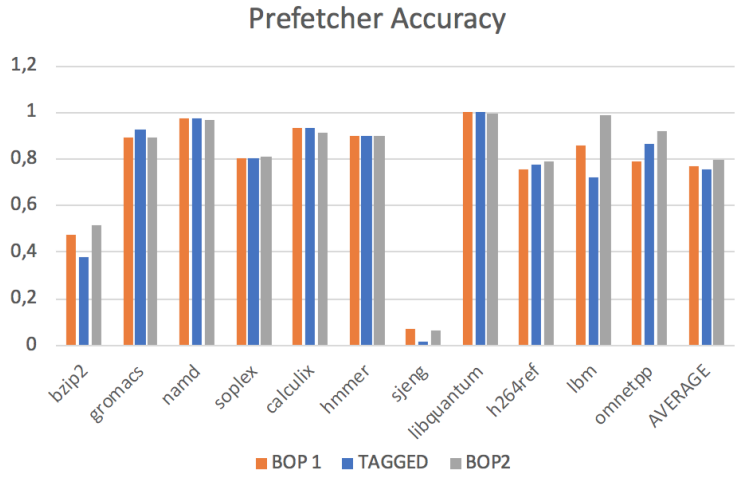
Trying to compare the prefetching algorithms between them, we can see that the BOP2 one is on average the best in terms of accuracy and BOP1 in terms of coverage. This makes sense because the TAGGED prefetcher is much simpler.

Another thing to be highlighted is the *sjeng* low accuracy (2%), and negative coverage (-1,8%) for the TAGGED prefetcher. That means that the prefetcher algorithm is not issuing useful prefetch requests and it is polluting the LLC because the miss ratio is being increased.

As we have seen in Fig. 4.3, the BOP1 overall network utilization (0,1371) is slightly greater than the BOP2 one (0,1359). This can be explained with the coverage and accuracy. We can see that BOP2 accuracy (79,5%) is greater than the BOP1 one (76,7%) but the opposite happens with the coverage



(a) Coverage



(b) Accuracy

Figure 4.4: Prefetching coverage and accuracy for the monocore simulations

(72,3% and 75,2%, respectively). This is because the BOP1 eliminates more misses (coverage) at the expense of issuing a greater number of prefetch request (less accurate), this is translated in an increase of the network utilization.

Figure 4.5 represents the MPKI reduction of the three prefetching versions related to the BASE, with no prefetching. The names of the benchmarks are on the X-axis and MPKI reduction in the Y-axis.

One thing that is important to note is that the benchmarks that experience a greater MPKI reduction do not have to be the ones with the greatest MPKI ratio. This can be seen in contrast to the yellow line in Figure 4.1.

The first observation in this figure is that the MPKI reduction is related to the coverage because both metrics sum up the the LLC misses that are removed by the prefetching mechanism. As we can see, the benchmarks with the greatest MPKI reduction are among the ones with the greatest coverage.

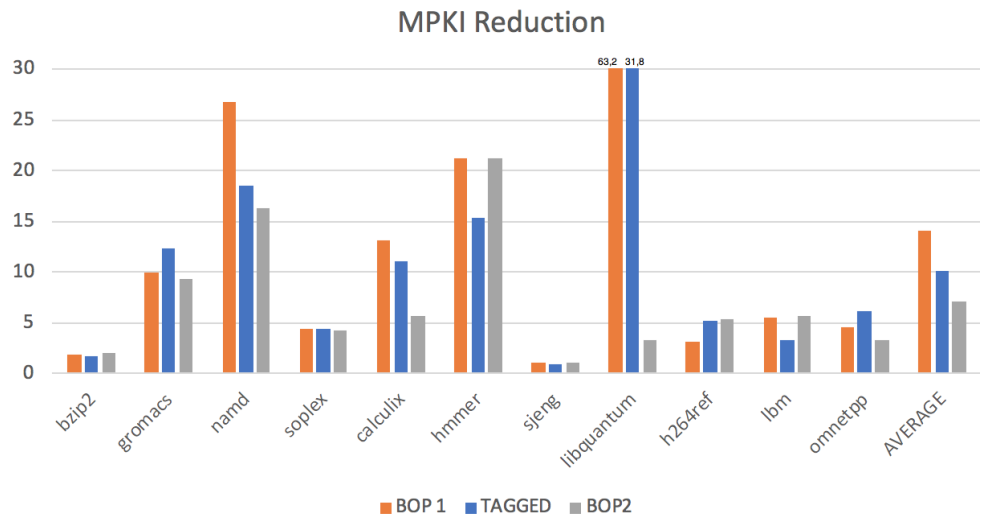


Figure 4.5: MPKI reduction.

Trying to compare the different prefetching mechnisms, looking at the average values, we can see that BOP1 gets the greatest MPKI reduction (14,1 times), followed by TAGGED (10,1 times) and BOP2 in the last position (7 times).

4.3 Multicore evaluation

Figure 4.6 represents the IPC speedup for the multicore execution in four cores. The names of the benchmarks and the mixes are on the X-axis and the speedup on the left Y-axis. The information is organized as follows: the first four groups of bars correspond to the *mix1*, the next four groups to the *mix2*, and the last four groups to the *mix3*.

One observation is that sometimes one prefetcher is better for one benchmark but it is sometimes another one depending on the mix where they are being executed. The reason for this is that the benchmarks memory access patterns to the LLC are interleaved in a different way depending the mix since different workloads are involved. For example, in the *lbm* benchmark, first group of bars for *mix1* and fifth group of bars for *mix2*. In *mix1*, we can see that the TAGGED and BOP2 prefetchers have the greatest IPC speedup (12,6% and 12,5%, respectively) while, in *mix2*, the BOP1 is the one with the greatest IPC speedup (13,7%).

In addition, some benchmarks get more penalized in terms of performance in the multicore simulations than in the monocore ones. For example, see *hmmer* IPC speedup in *mix2*: -2,2% for BOP1, -2,09% for TAGGED, and -2,1% for BOP2. The reason for this is that the memory access to the LLC

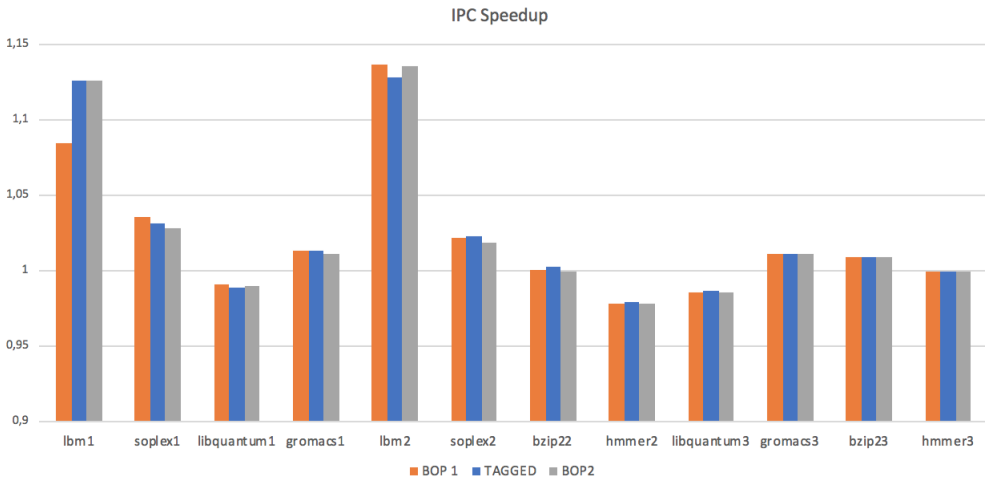


Figure 4.6: IPC Speedup in quad-core multicore simulations.

are interleaved within the mix so the prefetcher may choose one offset that is suitable for some benchmark but not for the others. Another reason for this is that the memory requests from all benchmarks are flowing together in the network. Therefore, some requests may be delayed and the mean LLC access time for some benchmarks might be increased.

Figure 4.7 represents the mean LLC access time for the multicore execution in four cores. The name of the benchmark and the mix can be observed in the X-axis and the reduction of time in the LLC accesses is plotted in the left Y-axis. The information is organized as follows: the first four groups of bars correspond to the *mix1* speedup, the next four groups to the *mix2* and the last four groups to the *mix3*.

As for the monocore execution, a direct link between the mean LLC access time reduction and the IPC speedup can be observed. We can see that *lbm* is the benchmark that gets the most LLC access time reduction. For example, as we can see in Figure 4.7, the LLC access time reduction for *lbm* is in *mix2* is 73,5% for BOP1, 66,2% for TAGGED and 73% for for BOP2. Said otherwise, without prefetching, *lbm* in *mix2* provides 143,5 LLC mean access time and it gets reduced to 82,7, 86,3, and 82,9 cycles for BOP1, TAGGED, and BOP2, respectively.

The MPKI and the network utilization for the multicore simulations is visible in Figure 4.8. The names of the benchmarks are on the X-axis and the MPKI and the network utilization ratio on the Y-axis of Figures 4.8a and 4.8b, respectively.

The first observation that must be noticed is the correlation with the monocore results, that is, in the case of MPKI, the mix with the greatest MPKI (*mix1*) is the one that is conformed with the benchmarks that present the greatest MPKI in monocore executions. The same phenomenom is visible

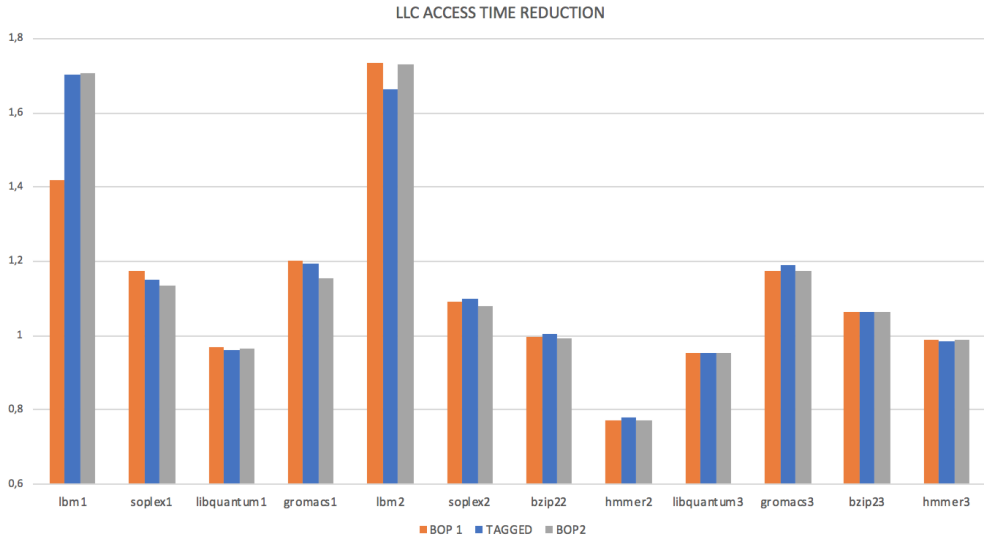


Figure 4.7: Mean LLC access time reduction in quad-core multicore simulations.

for the network utilization, *mix1* and *mix2* are the ones that are formed by the most network intensive benchmarks.

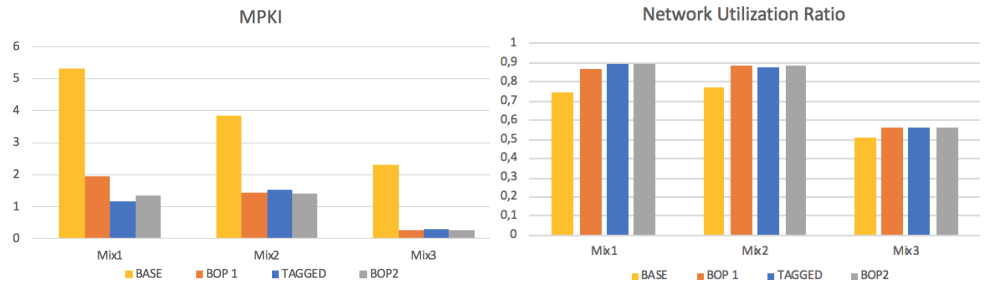
Another observation is the reduction of MPKI the prefetcher mechanisms achieve related to the non-prefetching version (yellow bar). We can see a significant MPKI reduction while the network utilization increase is not as remarkable.

Trying to compare the different mechanisms, we can see that BOP1 and BOP2 generate the lowest MPKI for *mix2* (1,44 and 1,41, respectively) and *mix3* (0,28 and 0,28, respectively) while the TAGGED prefetcher does it for *mix1* (1,16). Different mixes mean different memory access patterns to the LLC despite the fact that some mixes share some benchmarks. Therefore, some prefetchers can identify better some memory access patterns while other prefetchers can do it with another memory access patterns.

Due to the fact that in the multicore executions all the LLC requests from all the benchmarks flow together in the on-chip network, a significant increase in the network utilization can be noticed with respect to the monocore executions. In the monocore simulations, the greatest network utilization ratio is 0,45 for the *lbm* benchmark, see Fig. 4.3, while all network utilization factors in the multicore simulations for the same benchmarks are greater than 0,5.

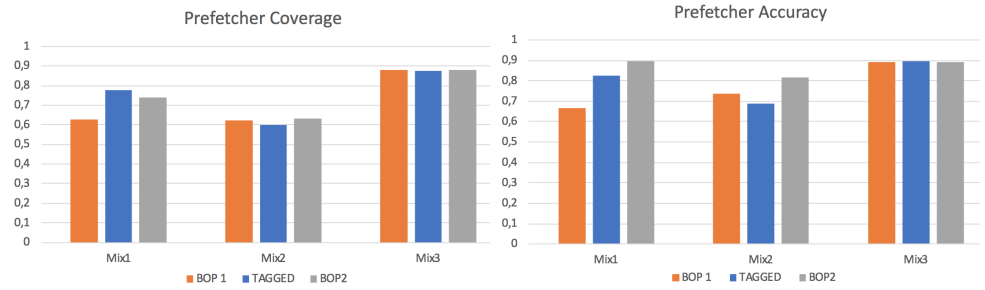
The prefetching coverage and accuracy is given in Figure 4.9. The names of the benchmarks are on the X-axis and the coverage and accuracy ratio on the Y-axis of Figures 4.9a and 4.9b, respectively.

Due to the fact that in the multicore executions all the LLC requests from all the benchmarks flow together in the on-chip network, it can be seen a



(a) MPKI (b) Network Utilization

Figure 4.8: MPKI and network utilization ratio for quad-cores multicore simulations.



(a) Coverage (b) Accuracy

Figure 4.9: Prefetching coverage and accuracy for the four-cores multicore simulation.

decrease in the coverage and accuracy of the prefetching mechanisms. This decrease must be compared with the coverage and accuracy of the benchmarks that conform each mix in the monocore executions. The reason for this is that the prefetching mechanism is observing the memory access patterns from all benchmarks at the same time. Therefore, it is very difficult, in the case of the BOP, that the learning algorithm can find an offset so that all the benchmarks take advantage of it.

On the contrary, we can see that BOP2 provides the best accuracy for all the mixes and the best coverage for *mix2* and *mix3*. Therefore, we could say that it seems to be more scalable. However, we know that for BOP2 is more complex in terms of hardware. It includes the delay queue and the RR table is doubled. Therefore, we should take into account whether this added complexity is worthwhile.

Conclusions

One of the conclusions that we can bring out from this thesis is that the hardware prefetching mechanisms interact with all the memory hierarchy components in a very complex way. In addition, it is sometimes really difficult to identify the advantages and the drawbacks of these mechanisms. The framework developed by Marti et al. [3] has shown its importance because it provides some features and metrics in order to ease the way to pick up prefetching statistics.

One of the main contributions of this thesis is the upgrade of the previously described prefetching framework. Due to the lack of support of the Ruby memory subsystem to prefetching, this framework is a really powerful alternative to model and to evaluate different prefetching mechanisms. Furthermore, it is prepared to easily implement additional prefetching policies so that Gem5 users can test their own prefetching proposals.

Regarding the performance results, we have observed that the improvements in terms of performance of the BOP are not as optimistic as Michaud results [1]. Michaud experiments are done on a trace-driven simulator, based on Pin [5]. Every workload Michaud used to feed this simulator is conformed with twenty samples of the same benchmarks stitched together. Each sample is got from different regions of the benchmark where there is a great LLC miss rate. The result of this stitched samples is a workload with a high LLC miss rate so that the activity of the prefetcher and its improvements can be noticed. However, we think that these traces are not realistic at all because real workloads will never present these kind of execution flow. Therefore, the improvements of the prefetching mechanisms could not have been observed in such a optimistic way.

The difference between BOP1 and BOP2 approaches is the the delay queue and the split RR table. We cannot say that the BOP2 improves the BOP1 to justify all the hardware complexity the BOP2 includes. However, we can notice that the BOP2 seems to work better in the multicore simulations than in the monocore ones. Therefore, we could think that the BOP2 is more

scalable and it could be interesting to simulate in an environment with 8 - 16 cores. We think that this second approach, the one that Michaud presented in the DPC-2 contest[7] [17], is taking the advantage of the concrete simulator of the contest. Therefore, Michaud decided not to include these optimizations in the Best Offset Prefetching paper because he did not find the advantages in a more generic environment.

Future work

One of the main limitations of this work is the short time we had for its development. A master thesis could be divided in three stages: the state of the art study, the implementation of the idea and the obtaining of results. For this last one, it could be useful to have more time to be able to simulate the effect of the different implementation parameters in the results outcomes. However, the time is limited and due to the size of the simulator, it has been impossible to be sure about the correctness of the parameters values for this environment. As Michaud did with its implementation and his simulator, a smaller and less detailed one than Gem5, one of the future work lines could be the study of the effect of the parameters values in different environments.

The Gem5 community had a lot of interest to have this framework working for the official release version of the simulator. Therefore, one of the possible lines of future work could be to work together with Martí et al. and the Gem5 community to include this framework in the official release version.

As we have seen in the BOP multicore behaviour, all the accesses from different cores to the LLC are interleaved in the network. Besides, the BOP learning phase makes no distinction on which core has generated each access. Therefore, we think that it could be interesting to identify the different execution flows and distinguish between them at the learning phase. The reason for that is trying to adapt the offset used for prefetching to each execution flow and study the possibility to have different offsets for each core or execution flow.

Personal Assessment

In order to conclude this report, I am going to exhibit a personal opinion about the project.

First of all, I feel satisfied with the knowledge obtained because of the development of this Master Thesis. It has allowed me to widen my knowledge about cache memories, coherence protocols, on-chip networks more than what I learned in my Bacherlor and Master subjects. Moreover, the use of the Gem5 simulator and the methodology explained will be very useful for me if I decide to continue my research career in the computer architecture field.

[7] 2nd Data Prefetching Championship, 2015. `http://comparchconf.gatech.edu/dpc2/`

During the development of this thesis we have encountered several setbacks. One of the main limitations has been that the Gem5 simulator is a relatively new simulator and it is continuously under development. Therefore, there are some features that are not as far implemented as we previously thought, for instance the Gem5 prefetching support. Another thing to be enhanced would be to enlarge the workloads, that is, to make more SPEC CPU 2006 benchmarks work under the Gem5 environment.

Bibliography

[1] Michaud, P. Best-offset hardware prefetching. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016, pp. 469–480, doi:10.1109/HPCA.2016.7446087.

[2] Binkert, N.; Beckmann, B.; Black, G.; et al. The Gem5 Simulator. *SIGARCH Comput. Archit. News*, volume 39, no. 2, Aug. 2011: pp. 1–7, ISSN 0163-5964, doi:10.1145/2024716.2024718. Available from: `http://doi.acm.org/10.1145/2024716.2024718`

[3] Martí Torrents, R. M.; Molina, C. Network aware performance evaluation of prefetching techniques in CMPs. *Simulation Modelling Practice and Theory*, volume 45, 2014: pp. 1 – 17, ISSN 1569-190X, doi:https://doi.org/10.1016/j.simpat.2014.03.005. Available from: `http://www.sciencedirect.com/science/article/pii/S1569190X14000434`

[4] Ishii, Y.; Inaba, M.; Hiraki, K. Access map pattern matching for high performance data cache prefetch. *Journal of Instruction-Level Parallelism*, volume 13, 2011: pp. 1–24.

[5] Luk, C.-K.; Cohn, R.; Muth, R.; et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, New York, NY, USA: ACM, 2005, ISBN 1-59593-056-6, pp. 190–200, doi:10.1145/1065010.1065034. Available from: `http://doi.acm.org/10.1145/1065010.1065034`

[6] Henning, J. L. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News*, volume 34, no. 4, Sept. 2006: pp. 1–17, ISSN 0163-5964, doi:10.1145/1186736.1186737. Available from: `http://doi.acm.org/10.1145/1186736.1186737`

[7] Solihin, Y. *Fundamentals of Parallel Multicore Architecture*. CRC Press, 2015.

[8] Mittal, S. A survey of recent prefetching techniques for processor caches. *ACM Computing Surveys (CSUR)*, volume 49, no. 2, 2016: p. 35.

[9] Yedlapalli, P.; Kotra, J.; Kultursay, E.; et al. Meeting midway: Improving CMP performance with memory-side prefetching. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, IEEE Press, 2013, pp. 289–298.

[10] Srinath, S.; Mutlu, O.; Kim, H.; et al. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, IEEE, 2007, pp. 63–74.

[11] Pugsley, S. H.; Chishti, Z.; Wilkerson, C.; et al. Sandbox Prefetching: Safe run-time evaluation of aggressive prefetchers. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014, ISSN 1530-0897, pp. 626–637, doi:10.1109/HPCA.2014.6835971.

[12] Smith, A. J. Cache Memories. *ACM Comput. Surv.*, volume 14, no. 3, Sept. 1982: pp. 473–530, ISSN 0360-0300, doi:10.1145/356887.356892. Available from: `http://doi.acm.org/10.1145/356887.356892`

[13] Chen, T.-F.; Baer, J.-L. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, volume 44, no. 5, May 1995: pp. 609–623, ISSN 0018-9340, doi:10.1109/12.381947.

[14] Wong, W. A.; Baer, J.-L. The Impact of timeliness for hardware-based prefetching from main memory. 2002.

[15] Ishii, Y.; Inaba, M.; Hiraki, K. Access Map Pattern Matching for Data Cache Prefetch. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, New York, NY, USA: ACM, 2009, ISBN 978-1-60558-498-0, pp. 499–500, doi:10.1145/1542275.1542349. Available from: `http://doi.acm.org/10.1145/1542275.1542349`

[16] Agarwal, N.; Krishna, T.; Peh, L.-S.; et al. GARNET: A detailed on-chip network model inside a full-system simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, IEEE, 2009, pp. 33–42.

[17] Michaud, P. A Best-Offset Prefetcher. *2nd Data Prefetching Championship.*, 06 2015. Available from: `http://comparchconf.gatech.edu/dpc2/`

[18] Jaleel, A. Memory characterization of workloads using instrumentation-driven simulation.