System level cache prefetching algorithms for complex GPU workloads

FABIAN EKLUND SIGLEIFS
ERIK HÄGGSTRÖM
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

System level cache prefetching algorithms for complex GPU workloads

Lunds Tekniska högskola
June 2024

Fabian Eklund Sigleifs	fabian.eklund.sigleifs@gmail.com
Erik Häggström	erikhaggstrom05@gmail.com

Master's thesis carried out at ARM

Supervisors:

Henrik Wall	ARM	Henrik.Wall@arm.com
Mathias Palmqvist	ARM	Mathias.Palmqvist@arm.com
Michael Doggett	Lund University	michael.doggett@cs.lth.se

Examiner:

Pietro Andreani	Lund University	pietro.andreani@eit.lth.se

Abstract

Prefetching is a well known concept for CPUs but for GPUs it is fairly unexplored. The memory management of a GPU plays a crucial role in its performance, and cache prefetching has the potential to lower the overall latency. This thesis compares different types of prefetching methods for GPUs and remaking some CPU prefetchers to fit the GPU architecture. All these prefetchers were then put inside the system level cache (SLC), between the GPU and external memory.

Five different methods were tested on framework based on ARM's GPU model. The thesis was mainly based upon prefetching techniques discussed in the following papers: *Adaptive Stream Detection*[11], *Best-offset* [18], *Many-thread aware* [17], *APOGEE* [26], and *Last-level collective cache prefetcher* [19]. The prefetchers produced in this thesis were either heavily inspired by or implemented as closely as possible to the designs in the papers.

The thesis concludes that for graphics workloads the best prefetchers implemented can achieve 0.51-1.29% decrease in GPU cycles on average, depending on the chosen GPU configurations, while also lowering the estimated energy usage.

DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY | LUNDS TEKNISKA HÖGSKOLA | 2024-05-24

MASTER'S THESIS System level cache prefetching algorithms for complex GPU workloads
STUDENT Fabian Eklund Sigleifs, Erik Häggström
SUPERVISORS Michael Doggett (LTH), Henrik Wall (ARM), Mathias Palmqvist (ARM)
EXAMINER Pietro Andreani (LTH)

Improving GPU performance using Cache Prefetching

POPULAR SCIENCE SUMMARY **Fabian Eklund Sigleifs, Erik Häggström**

Prefetching is the process of guessing what data a system wants, before it asks for it. This idea has been used for Central Processing Units (CPUs) for a long time but is rather unexplored for Graphic Processing Units (GPUs). This thesis will look at five different prefetchers and evaluate their performance.

Accessing the main memory can sometimes slow things down because, even though the processor is fast, it's held up waiting for the memory to catch up. This is due to the fact that accessing the main memory takes a rather long time. The solution is using something called cache. Cache consists of small memory units that are much faster than the main memory. These units are placed in multiple levels between the processor and the main memory. When the processor needs to access data, it goes through all the levels of cache until it finds the data it's looking for in the nearest level. Then, it retrieves that data and sends it back to the processor through all the levels again. When new data arrives at a level where it isn't already stored in the cache, it gets placed there so that it can be found faster the next time.

Normally a GPU consists of two levels of cache but sometimes a third one called System Level Cache (SLC) is added. In this thesis, different prefetching approaches are evaluated for this third level of cache for a GPU that is performing complex tasks, in this case demanding mobile graphics.

The main tasks for a prefetcher is to decide on three main topics. What data to prefetch, how much to prefetch, and also when to prefetch. When deciding what to prefetch the simple approaches often look at the difference between consecutive cache requests. If this difference is regular for several requests in a row, prefetch requests are sent to the main memory. And if the differences are not in any way regular, no prefetch requests are sent. So the challenge is how do we make sure that we only prefetch when a pattern is present and do nothing when there is no pattern present.

In the thesis five different prefetching methods were tested. Three used more simple difference calculation, one used a scoring system to find the pattern, and another used probability theory. It was found that in most of the cases tested there was a performance increase of 1.3% at best. It was also seen that the energy consumption went down. It can then be concluded that the introduction of prefetching has a positive effect on the performance on the GPU model used.

Contents

1 Introduction **1**
1.1 Context . . . 1
1.2 Project goal . . . 1

2 Background **2**
2.1 Basics of computer graphics . . . 2
2.2 GPU architecture . . . 2
2.3 Cache hierarchy . . . 3
2.4 Cache structure . . . 4
2.5 Challenges for memory accesses . . . 5
2.5.1 Prefetching design aspects . . . 6
2.6 Prefetching models . . . 7
2.6.1 Stride Prefetcher . . . 7
2.6.2 Stream Prefetcher . . . 8
2.6.3 Global History Buffer . . . 8
2.6.4 Adaptive Stream Detection . . . 9
2.6.5 Best-offset prefetcher . . . 10
2.6.6 Many-thread aware prefetcher . . . 11
2.6.7 APOGEE prefetcher . . . 11
2.6.8 Last-level collective prefetcher . . . 12

3 Method **14**
3.1 GPU model . . . 14
3.2 Workflow . . . 14
3.3 Implemented prefetching models . . . 15
3.3.1 Decision of implemented prefetchers . . . 15
3.3.2 Naive Stride Prefetcher, NSP . . . 15
3.3.3 Adaptive Degree Prefetcher, ADP . . . 18
3.3.4 Adaptive Stream Detective Prefetcher, ASDP . . . 19
3.3.5 Modified Best-Offset Prefetcher, MBOP . . . 20
3.3.6 Last-level collective prefetcher . . . 20

4 Results **21**
4.1 Simulation parameters . . . 21
4.1.1 Prefetching model parameters . . . 21
4.2 Cycle count . . . 23
4.3 Hit rate . . . 24
4.4 Read requests . . . 25
4.5 Power estimation . . . 26
4.6 Energy estimation . . . 27
4.7 Prefetch accuracy . . . 28
4.8 Averages . . . 29
4.9 Summary . . . 29
4.9.1 Cycle count . . . 29
4.9.2 Hit rate . . . 29
4.9.3 Read requests . . . 29
4.9.4 Power and energy estimation . . . 29
4.9.5 Prefetch Accuracy . . . 30

4.9.6 Status on LLC prefetcher 30
4.10 Other GPU configurations 30

5 Discussion **33**
5.1 Evaluation of performance 33
5.2 Further exploration 34

6 Conclusion **36**
6.1 Summary 36
6.2 Future research 36

A Results for other GPU configurations **40**
A.1 1 core and 1 slice 41
A.1.1 Cycle count 41
A.1.2 Hit rate 42
A.1.3 Read requests 43
A.1.4 Power estimation 44
A.1.5 Energy estimation 45
A.1.6 Prefetch accuracy 46
A.2 4 cores and 2 slices 47
A.2.1 Cycle count 47
A.2.2 Hit rate 48
A.2.3 Read requests 49
A.2.4 Power estimation 50
A.2.5 Energy estimation 51
A.2.6 Prefetch accuracy 52
A.3 14 cores and 8 slices 53
A.3.1 Cycle count 53
A.3.2 Hit rate 54
A.3.3 Read requests 55
A.3.4 Power estimation 56
A.3.5 Energy estimation 57
A.3.6 Prefetch accuracy 58

1 Introduction

In this chapter the overall topic of the thesis will be presented as well as the project goal.

1.1 Context

System Level Cache (SLC) is an optional extension of the cache levels between a Graphics Processing Unit (GPU) and the DRAM which can play a pivotal role in increasing the performance of the system. The workloads for GPUs are increasing in complexity with developments in, for example, graphics rendering, scientific simulations, and machine learning. These complex tasks are heavily dependent on the GPUs memory management scheme, which includes the algorithms used for deciding which data to be available at each cache level. Prefetching is a method for loading the cache with data that is anticipated to be used at a later time. The objective of the method is to reduce data access latency by ensuring relevant data is pre-loaded in the cache. The method can exist in both hardware and software, however only hardware solutions will be discussed in this thesis.

Prefetching is widely used for Central Processing Units (CPUs), where it has been thoroughly studied [20]. Multiple algorithms have been developed including the common stride, stream and the global history buffer. Prefetching for GPUs is not as widely studied but in recent years more studies have emerged testing both modified CPU prefetchers and exploring new prefetching methods[17], [19], [23].

1.2 Project goal

The goal of this project is to investigate different prefetching algorithms that can be implemented for the SLC of a GPU and decide upon which of these explored algorithms would have the highest performance increase and are able to be implemented on a GPU model provided by ARM. The next step is to implement and modify the chosen algorithms to function on the GPU model. The objective is to be able to measure the performance gain between the different algorithms compared to the GPU model without prefetching, for complex workloads. In the model provided by ARM the SLC is only connected to the GPU, so having it shared with the CPU will not be considered in this thesis.

2 Background

In this chapter the theory used throughout the work will be presented starting with broad topics of GPU architecture and ending with recently developed methods for GPU prefetching.

2.1 Basics of computer graphics

The visuals on a computer screen is made up of pixels. The goal of generating or rendering computer graphics is to paint every single pixel in a certain color according to the 2D image that is to be portrayed on the screen [4].

The visuals in graphics is called a scene which contains different models or objects, each model or object has different properties such as surface and lighting. The models in the scene are in turn made up of render primitives. Render primitives are simple geometrical shapes like triangles. Afterwards the view of the scene is determined and this is called a camera. This determines the angle and location of the view. To make all the objects fit the screen we need to move them from their coordinate system to one that can be represented in the 2D space, so move from 3D to 2D. These steps of rendering are called the rendering pipeline and consists of three steps, application, geometry, and rasterizer[1].

2.2 GPU architecture

The Graphics Processing Unit (GPU) is a specialized hardware component used to accelerate computer graphics. More recently GPUs have also been used for image processing and training of neural networks. This is due to its parallel structure. The parallel structure means the hardware is split into several cores where each core contains several threads where a group of threads is called a warp.

A GPU is a pipelined structure with specialized hardware consisting of Input-assembler stage, Vertex shader, rasterizer and pixel shader[29]. The pipelined structure as well as the fact that each stage is very wide, in terms of being able to execute a lot of instructions simultaneously, enables its high parallelism.

> The **input-assembler stage** reads primitive data (points, lines and/or triangles) from user-filled buffers and assembles the data into primitives that will be used by the rest of the pipeline. The input-assembler can assemble vertices into several different primitive types[25].
>
> The **vertex shader** is the programmable shader stage in the rendering pipeline that handles the processing of individual vertices. This stage must always be active for the pipeline to execute. The vertex shaders usually perform transformations to the post-projection space [28].
>
> **Texture mapping** is the process of putting textures such as images and patterns on top of existing triangles in a frame. This texture data has to be obtained from the main memory, and to reduce latency, texture caches are introduced to the system. These texture caches have a high hit rate since heavily reuse data of neighbouring pixels [8].

A **rasterizer** is the pipeline stage that converts vector information (composed of shapes or primitives) into a raster image (composed of pixels) to display real-time 3D graphics. During this process, each primitive is converted into pixels, while interpolating per-vertex values across each primitive. Rasterization includes clipping vertices to the field of view of a perspective virtual camera system called view furstum. It does this by performing divide by z which is used to convert the 3D coordinates to a 2D screen[25].

A **pixel shader** is the part of the pipeline that enables shading. A pixel shader is a program that combines constant variables, texture data, interpolated per-vertex values, and other data to produce per-pixel outputs. The rasterizer stage invokes a pixel shader once for each pixel covered by a primitive[25].

The architecture used in the GPU model provided by ARM uses a tile based architecture and the configurations used for this model are intended for high-end mobile devices.

2.3 Cache hierarchy

The reason of introducing caches inside a system is to reduce the overall latency of memory requests by exploiting two factors: temporal and spatial locality. Temporal locality refers to the fact that recently used data will probably be used again, while spatial locality refers to the presumption that data that exists nearby recently used data will be needed in the future. Caches are normally divided into two or three layers named L1, L2 and L3 [9]. SLC refers to an additional cache level outside of the GPU that can also be shared with the CPU, display controller, and ISP. The division of cache follows a structure where the cache closest to the computational unit, the L1 cache, is the smallest one, and the further away the bigger the caches get. Systems often contain multiple L1 caches that belong to their own computational units, while L2 caches are shared between them all.

GPUs generally contain two layers of caches, L1 and L2, as can be seen in the simple model in figure 1. The L1 cache is the smallest and fastest cache and is supposed to hold the data that is most frequently accessed. The L2 cache is a bigger cache than L1 that is shared between more computational units. The SLC is an extension of this model which places a third even bigger cache between L2 and the off-chip dynamic random access memory (DRAM) to further reduce the latency of the memory requests. In this thesis the SLC is only connected to the GPU and is not shared with the CPU.

Managing caches was originally done by the software but nowadays it is hardware managed. Managing the cache inside a GPU can be very challenging because of the large amount of threads and low cache sizes. Lal, S., Varma, B.S. and Juurlink discusses this subject in the paper A Quantitative Study of Locality in GPU Caches for Memory-Divergent Workloads [16]. They emphasize the problem by comparing the size of CPU caches per thread with GPU caches per thread and state that CPU caches per thread are normally in the sizes of 16 KB, 128KB and 1MB for L1, L2 and SLC respectively. For GPUs on the other hand the sizes per thread range between 32-102 B for L1 and L2 [16].

This clearly shows the difficulty of management, especially for irregular GPU applications that further complicates the problem.

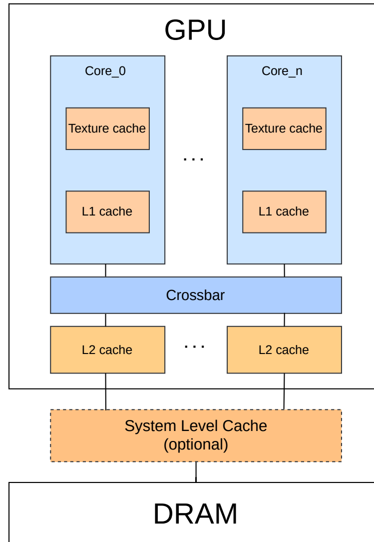

Figure 1: A simple overview of a GPU cache hierarchy

Inside a GPU the threads are grouped in warps, usually containing 32 threads. When discussing locality inside the cache architecture it can help to divide it into intra-warp locality and inter-warp locality. Intra-warp locality is the cause of a warp requesting an address and later requesting it again while inter-warp locality means that a warp requests an address that is later requested by another warp.

Lal, S., Varma, B.S. and Juurlink analyzed the cache line usage of a NVIDIA GPU and observed that on average 57% of the cache lines were never re-used, however a large number of them got evicted too early. The study also tried the system using infinite cache sizes and noted that this number then declines to 30%, which proves that with better cache management there is a lot of performance to be gained. This could for example be done by introducing a SLC to the GPU.

2.4 Cache structure

A cache memory is divided into multiple lines, called cache lines or blocks. These cache lines are the smallest possible data entries that can be loaded into the memory. Every data entry in the main memory is mapped to a line in the cache, and this can be done in multiple different ways. The simplest mapping method is known as direct mapped, wherein each main memory address is assigned to a specific cache line. However, since the size of the cache is much smaller, many addresses will be mapped to the same cache location. Consequently, frequent data replacements within cache lines may occur, leading to a phenomenon known

as thrashing, where cache entries are replaced repeatedly without being utilized. [5]

One solution to this problem is to use a set associative cache. To use this method the cache memory is divided into a number of sets, each set containing multiple cache lines. Each main memory address is now mapped to a line in all of these sets, but it can only exist in one of them. For example a 4 way associative cache implies that each data entry in the main memory could exist in 4 different locations in the cache.

There are more decisions to be taken when designing a cache and some of them are connected to a subject called cache policies. Cache policies decide on rules for how the cache will act when receiving read and write requests. If a cache miss takes place, meaning that the requested data does not exist in the cache memory, it has to take a decision to fetch the data to the cache memory or not. The chosen allocation policy will decide this. If a read allocate policy is used, the cache will only allocate its lines to read requests, while a write allocate policy will allocate both read and write requests in the cache [5].

When allocating a cache line you also need a replacement policy to decide upon which line to allocate and how to handle data that already is allocated in that position. If a cache line contains dirty data, meaning data that has not been updated to the main memory, it has to write this data to the main memory before replacing the cache line. This phenomena is called eviction and the cache line selected for replacement is called victim. [13]

Choosing a victim can be done in multiple ways and some of the most common ones are pseudo-random, least recently used, most recently used, and first in first out. These different policies will all have impact on how the cache memory will function and could increase the hit rate if chosen wisely.

A cache also contains write policies for deciding how to handle write requests. There are two main write policies, one is called write-through and decides that writes are performed to both the cache and the main memory simultaneously, meaning that they contain the same data. The other one is called write-back, and using this method will cause the cache to contain updated data while the main memory contains old data, when an eviction takes place it is therefore very important that this updated data is written to the main memory [5].

2.5 Challenges for memory accesses

The main challenge of managing the memory of a GPU is to minimize the latency for each memory request. There are a lot of factors that will impact this latency, including bandwidth limitations, contention, data movement and memory hierarchy. [15]

The bandwidth determines how much data can be transferred per unit of time. There are two ways of increasing this, either by increasing the size of the link, meaning being able to send larger memory requests, or by parallelisation which enables many memory accesses to be sent during the same time period. When a memory is shared between many units contention can occur, meaning that conflicts take place when the units are fighting over the same memory. Contention issues are solved by inferring rules on the link for example a queue system [6]. Solving contention issues can be crucial for increasing the latency of a system since long delays can occur when the link is highly utilized.

The challenge of managing data movement involves deciding how and which data should be moved between the different cache levels and decisions regarding when to evict, bypass, promote or prefetch cache lines need to be taken. This has to be done carefully to not cause too much strain on the link while still maintaining performance. Cache pollution, which refers to when data is loaded into the cache without being used, has to be reduced. This is a big challenge in the case of prefetching since it is about speculating on what requests will appear in the future, and if these speculations are wrong, cache pollution will occur. A prefetching unit that can adapt to different workloads may therefore be required to achieve desired behaviour.

Another challenge is how the memory hierarchy is built. This includes how many levels of caches are used, number of slices and their interconnections. This has to be chosen with respect to the balance between the required performance, cost and power consumption of the GPU.

An important factor when managing cache memory is timeliness. When prefetching it is preferred to have as many timely prefetches as possible, a timely prefetch refers to a fetch that is not too early nor too late. In figure 2 the timeliness of prefetching can be seen, if a prefetch is performed too late you will lose some of the speedup that could have been gained and if done too early it will be a burden to the system without any performance gain at all. The goal of a prefetcher is to make the prefetches in the timely period as often as possible. However, since multiple requests are being handled simultaneously, having a late prefetch may in some cases impact the performance less since the system can process something else meanwhile [17]. This is especially true for GPU's since they are more parallelized than CPU's.

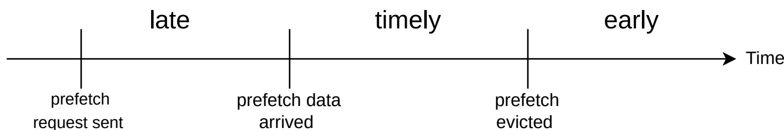

Figure 2: Figure describing the concept of timeliness for a prefetch request. For example: If a request for an address arrives during the first period, the prefetch request was too late, and if a request arrives in the latest period the prefetch request was sent too early, meaning that it was already evicted.

2.5.1 Prefetching design aspects

The main tasks for a prefetcher is to decide on three main topics. What data to prefetch (next line or to follow a specific pattern), how much to prefetch (how many cachelines to fill) and also when to prefetch (eg., after consecutive cache misses, high confidence or as often as possible). [23]

When deciding what to prefetch the simple approaches often look at the delta between consecutive cache requests, called stride. If this stride is considered to be the distance to the next line in the memory it is instead refereed to as a stream. Prefetchers based on stride and stream patterns also need the workload to exhibit such behaviours in its cache accesses for the prefetcher to function.

This is a big challenge in the design since you would like to utilize when the read pattern is very regular and predictable but also for the system to not slow down during periods of irregularity. The best way would be to have a prefetcher that is able to predict both regular and irregular patterns, but this is a very difficult challenge. An issue for hardware based approaches of prefetching is to observe and act upon short strides and streams (less than 5 requests). This is because it will take at least 2 requests for the prefetcher to detect a stream or stride and then the prefetcher will struggle to make an impact since the stride is already or soon to be finished [11].

Another relevant term is the distance of the prefetch request. This value represents how far in the future it should prefetch the data, this value should be chosen to maximize the amount of timely requests as mentioned in figure 2. This is a hard task since the optimal distance may vary throughout the applications runtime.

How much to prefetch is often referred to as the degree of the prefetcher and it is simply the number of lines it fetches each time a request is sent from the prefetcher. This number could also vary depending on how confident the system currently is about the patterns observed.

When to prefetch can be decided in a lot of different ways, a common method is to calculate a confidence value that represents how likely the prefetches will be useful for the cache, meaning that they are going to lead to a cache hit. If this confidence value is above a certain threshold the prefetcher would then issue a prefetch request.

If the stride for the memory requests of an application is constant, this approach will provide good results. But if the application has a irregular access pattern, methods using stride values will have more difficulties finding good prefetch address candidates. This also applies for stream prefetchers were the next cachelines are used instead of calculating the deltas.

When evaluating a prefetcher the accuracy of the prefetcher can be calculated according to equation 1

$$Accuracy = \frac{Prefetch\ hits}{Total\ number\ of\ prefetches} \tag{1}$$

This value will tell how large percentage of the issued prefetches that leads to a hit. Keeping this value as high as possible will be crucial for not polluting the system without any performance gain, however there may be cases were it is worth it to have lower accuracy if the system can handle the extra load and it leads to a larger number of total prefetch hits.

2.6 Prefetching models

In this section simple prefetching methods will be discussed. These were originally developed for CPUs but the main concepts can be applied when designing prefetchers for GPUs.

2.6.1 Stride Prefetcher

Stride prefetchers function by calculating the delta between recent memory accesses. These delta values are then used to predict the next memory access

and prefetch this data into memory beforehand [10]. If the following for loop is being executed:

```
for(i = 0, i < 100, i++){
    C[i] = A[2*i] * B[3*i]
}
```

the stride prefetcher would be able to calculate the stride of the memory accesses in A and B by subtracting the memory addresses of the accesses. If $a(k)$ and $b(k)$ are the memory addresses of the variables $A[k]$ and $B[k]$ you could calculate the stride as $strideA = a(k) - a(k \cdot 2)$ and $strideB = b(k) - b(k \cdot 3)$ for different k values. If we focus on the load request when k=2 for A, the stride would be calculated to $a(2) - a(4)$. The prefetcher would then issue a prefetch depending on its degree D, that decides how many data should be requested. The general formula for the prefetches addresses would then be $a(k+stride \cdot i)$ where i ranges from 1 to D. If $D = 3$ and $k = 2$ it would issue a prefetch request for addresses $a(4 + 2 \cdot 1)$, $a(4 + 2 \cdot 2)$ and $a(4 + 2 \cdot 3)$ since the stride is equal to two. The prefetcher manages this by saving information about every load request in a stride prediction table (SPT) containing stride prediction entries (SPEs), which contain the stride, base address, degree as well as other information like warp or core identification [19].

It is also common for the SPE to contain a confidence value. This value increases if the stride for the newly loaded data is the same as the previous stride, and decreases or is set to zero if they are different. If this value is below a certain threshold (CONFTHRESH) the SPE will not issue prefetches for the entry.

2.6.2 Stream Prefetcher

Stream prefetchers utilize cache misses to detect patterns in the cache structure. It identifies this in k consecutive accesses [11], for example, three accesses. After the first miss is detected at line X, the stream is started. Then when the second miss is detected at line $X + Y$ a pattern is guessed. And at the third miss at line $X + Z$ (where $Z > Y$) confirms the pattern. Then the next line $X + D$ (where D is the next line following the pattern) is prefetched. So the choice of k consecutive accesses decides on how accurate vs. how many prefetches are taken. All this information is stored in a stream table by the stream prefetcher. It is filled upon confirmation of the direction of the stream. This table contains multiple entries per page and stores prefetching-metadata for several pages at once. A miss can only trigger prefetching if such an entry exists for the same page [24].

2.6.3 Global History Buffer

The Global History Buffer(GHB) is a way to improve advanced prefetchers that store history data. GHB is a way to make sure that there is less stale data stored both reducing area and increasing speed. It is implemented as a FIFO queue with n entries that holds the n most recent cache misses. Each entry in the FIFO queue is an Index Table (IT) that is accessed with a key as in conventional prefetch tables. The entries in the IT contain pointers into the GHB. This is not a prefetching method in itself but an extra tool to increase

performance in for example stride prefetching by clearing out stale data making the look up of previously prefetched data faster [22].

2.6.4 Adaptive Stream Detection

Hur and Lin presents an improvement on the standard stream prefetcher in the paper *Memory Prefetching Using Adaptive Stream Detection* [11]. Their solution of deciding when to prefetch is statistically based using a histogram called Stream Length Histogram (SLH). They define time periods called epochs defined by a certain number of memory accesses, and during each epoch all the stream lengths are recorded into the SLH. A decision on which addresses to prefetch is then taken based upon the stream lengths recorded in the SLH.

The method uses a function called $lht(i)$ which represents the number of read requests that are part of a stream with length i or longer. The i value can be $1 < i < fs$ where fs is a variable representing the maximum stream length to be tracked. It then uses this lht function to calculate the probability of the current stream to be of a certain length k. They present a simple way of deciding if to prefetch or not based on the lht values by calculating the following inequality:

$$lht(i) < 2 \cdot lht(i+1)$$

To be able to track this information the authors divided the process into two parts called Stream Filter and Likelihood Tables. In the Stream filter the current streams are tracked. Each entry saves the last accessed address, the length of the stream, the streams direction and its lifetime. The direction is simply comparing if the addresses are increasing or decreasing and the lifetime indicates when the entry should be evicted.

Every time an entry is updated, meaning that a new address belonging to that stream is found, the length is updated as well as the lifetime being increased with the predetermined value, $LIFETIMEEXT$. This causes the system to not evict the entry since a stream was found. At every cycle it also decreases the lifetime value of all the entries and evicts those that became zero.

The Likelihood Tables are divided into LHT_curr and LHT_next the first one containing SLH information about the current epoch and the second is used for saving information to be used for the next epoch. These tables contain lht values for all the tracked stream lengths for both positive and negative direction.

Every time a stream of length k is evicted from the Stream Filter the LHT_curr is decremented by k for all i where $1 \leq i \leq k$ and LHT_next is instead incremented by the same values. At the end of each epoch all the streams in the Stream Filter are evicted and the values are used to modify the LHT_next. Then the values from LHT_next is copied to LHT_curr and LHT_next is cleared.

The method can then use the LHT table to obtain the inequality as described before by checking if $LHT_curr(i) < 2 \cdot LHT_curr(i+1)$ is true and a prefetch should be issued. Where i is the currently observed stream length.

In the follow up paper *Feedback Mechanisms for Improving Probabilistic Memory Prefetching* Hur and Lin presents further improvements on the Adaptive Stream Detection prefetcher [12]. This improvements consisted of three main features: Length-based stream detection, Adaptive epoch length and Variable length prefetching. The length-based stream detection is a mechanism that

controls the lifetime of streams in a way that favors short streams by reducing the $LIFETIMEEXT$ value for longer streams, thus freeing up more space for shorter streams in the Stream Filter. The adaptive epoch length automatically adjusts the epoch length based on a calculated similarity score at the end of each epoch. The similarity score is simply the average difference of all the current and previous LHT values, if this score is above a certain threshold the tables are classified as dissimilar. When a dissimilar table is observed the system will try to reduce the epoch length by a factor of 2 or increase it with a factor of 2 until it achieves similarity.

The last improvement called variable-length prefetching is the mechanism of prefetching multiple lines at the same time depending on the inequality:

$$lht(i) < 2 \cdot lht(i + k)$$

where the k value is the number of consecutive lines to be prefetched. This is simply an extension of the inequality presented in the earlier paper, allowing for more requests to be sent simultaneously. The paper also describes a throttle mechanism that hinders prefetches of multiple lines when the internal buffers are more than half full. The authors states that this reduces the risk of putting too much strain on the system [12].

2.6.5 Best-offset prefetcher

In the paper *Best-offset hardware prefetching* P. Michaud presents a prefetcher that is based on a score system to select the best possible offset to use for prefetching [18]. The term offset is similar to the earlier described stride but in this case it refers to a fixed stride being used during a longer period. The method uses epochs like the case of the Adaptive Stream Detection prefetcher with a fixed length (LIFETIME). During these epochs a fixed number of the latest request are saved in a Recent Requests (RR) table which is later used to find the best offset. The method includes a list of possible offsets to evaluate and each epoch is divided into a number of rounds equal to the number of offsets to be evaluated. During each round the incoming request addresses are subtracted with the value of the current offset. If the result can be found in the RR table the method assumes that if a prefetch requests would have been sent, it would have lead to a hit. It therefore increases the score of this offset by one. When the epoch is finished it compares the scores of all the offsets and selects the best one to be used for prefetching during the next epoch. If the score obtained during the last epoch is above a certain threshold the prefetcher will issue prefetches for every cache miss and prefetch hit depending on a PREFETCHITS parameter. The prefetcher will use the best offset from the latest epoch during the current epoch. The prefetcher is of degree one but it is still considered rather aggressive according to the authors since it sends out a lot of requests in total, however adjusting the threshold for the score values can change this behaviour.

Another prefetcher called the Aggregate Stride Prefetcher is also closely related in function to the best-offset prefetcher [3], however it trains the different offsets simultaneously, requiring more hardware resources for the selection, but potentially also acquiring a better performing offset. The prefetcher also ignores some of the most recent prefetch requests to make the offsets more timely, since the method makes its decision based on older data. There is also another similar prefetcher presented in the paper *Multi-Lookahead Offset Prefetching* [27]

that is based on similar ideas as the Aggregate stride prefetcher and the Best-offset prefetcher. The authors refer to lookaheads as different time horizons where different amount of recent accesses are ignored, which differs from the Aggregate Stride Prefetcher which has uses a fixed number. The Lookahead Offset Prefetcher simultaneously calculates scores for 16 different lookaheads. These 16 scores correspond to ignoring none to ignoring all of the last 15 recent memory accesses.

2.6.6 Many-thread aware prefetcher

Lee et al. presents a algorithm for GPU prefetching in the paper *Many-Thread Aware Prefetching Mechanisms for GPGPU Applications* [17]. The paper is focused around both software and hardware based solutions. The hardware based solution includes two parts, one for per warp stride and one for inter-thread prefetching. Inter-thread prefetching refers to a thread prefetching data to another thread than itself.

The *Scalabale Hardware Prefetcher Training* used further develops on the concepts of standard stride training to enable it to save stride information per warp in a *per warp stride* (PWS) table. The paper also proposes a method called *Stride promotion* which enables the model to share stride information between these PWS tables. The model assumes that when a few warps have the same stride, all warps will likely have the same stride. They set a threshold of three PWS tables having the same stride for this stride promotion to take place. When this happens the stride information is moved from the PWS table to a *global stride* (GS) table.

In parallel to this mechanism another mechanism is looking at inter-thread stride using an inter-thread prefetching (IP) table which is trained with software. Similar to the case of the per warp prefetching this algorithm waits for three different warps having the same stride for the same PC to issue a prefetch request. The model also features a throttling mechanism to minimize the risk of the prefetching causing any negative impacts on performance. This throttling mechanism manages how many prefetches are being executed based on the number of early evicted prefetches and intra-thread merges. These merges occur when a prefetch request is late and a demand request is coming at the same time. The throttling behaviour can be observed in table 1.

Early eviction rate	Merge	Action
high	-	no prefetch
medium	-	fewer prefetch
low	high	high prefetch
low	low	no prefetch

Table 1: Table of throttling mechanism for Many-thread aware prefetcher [17]

2.6.7 APOGEE prefetcher

The APOGEE prefetcher utilizes interactions with the memory system and looks for consistent memory access patterns between the threads of a warp. It

contains two prefetching parts called *Fixed Offset Address* (FOA) and another called *Thread Invariant Access* (TIA).

General for APOGEE, is if a consistent pattern is detected in the addresses then information about these addresses are stored in a table.

The APOGEE prefetcher looks at the memory access pattern and if a consistent pattern is detected, it is stored in a table. The information stored, see table 2, is Program Count, the address accessed by one thread, its offset with the adjacent thread and how consistent the offset is across the adjacent thread called confidence. Once the confidence reaches a certain level it prefetches data based ib the offset and what is called distance. The distance is how far ahead of the last access the data should be fetched. Tid is the thread index and references all indices in the GPU which in turn are then split into warps.

Load PC	Address	Offset	Confidence	Tid	Distance	PF Type	PF State
0x253ad	0xfcface	8	2	3	2	0	00

Table 2: Example of a prefetch table used by APOGEE [26].

For the FOA prefetching it looks for the same offset between addresses with the same PC. If the offset is the same as previously the confidence is increased, and if not, the confidence is set to zero and a new offset is calculated. It also takes into account the difference in thread indices. And for the TIA the *PF Type* is flagged and the offset is zero, since what is happening is that all the threads want to access the same address at different times. The issue with this is to have the SPT keep the value and fetch in a timely manner. To make sure that the prefetch is not too late or early the *PF State* is used with three states: 00, 01 and 10. Once a load of the corresponding entry happens the state is set to 00. After the computation of its future address, if a prefetch request is sent, the state is changed to 01. Transition from state 01 to 10 occurs when the data comes back from memory to the cache. Whenever a new load of the same entry occurs, the state is reset to 00. So if an entry is in the 01 state when a request for the same entry arrives it means that the prefetching is too slow and the method needs to increase the prefetching distance.

2.6.8 Last-level collective prefetcher

Michelogiannakis et al. proposes a stride based prefetcher for CMP's. A CMP (cellular multiprocessor) is a multiprocessing architecture for Intel CPUs from Unisys providing up to 32 processors that are crossbar connected to a memory and several PCI cards [2]. But it is also stated in this paper that this architecture also could be interesting for GPUs. The method further develops on the strided prefetcher by also implementing groups of SPEs. The SPEs are put into a reference preditiction table (RPT). The entries in this table are based on the following hash function:

$$\left(\frac{PC_{request}\%NumLines_{RPT}}{4} + Core_{ID}\right)\%4 \tag{2}$$

With this hash function, all SPEs with the same PC value have to be in one of four RPT lines. Hence the modulo four in the hash function. When an SPE gets a confidence level over *CONFTHRESH* the SPE is placed in a group with

other SPEs from the same PC. The SPE with the lowest base address is placed in a group table connected to the PC and the rest are put in a doubly-linked list with this first SPE. This results in that whenever one of the SPEs are triggered the rest of the group is also fetched. The order of the list matters to utilize the order of the fetches, see equation 3.

$$\begin{aligned} Base_1, ...Base_N, ..., \\ (Base_1 + S_1), ..., (Base_N + S_N), ..., \\ (Base_1 + S_1 \cdot D_1), ...(Base_N + S_N \cdot D_N) \end{aligned} \tag{3}$$

3 Method

In this chapter, the work's methodology will be presented, as well as general information about the GPU model which was used to obtain performance results for different implemented prefetching algorithms.

3.1 GPU model

A GPU framework developed by ARM based on their hardware was used for implementation and testing of the prefetching methods. The model is a software model of ARM's GPU hardware and tries to mimic the hardware as close as possible in terms of functionality and performance. The model also allows for changing the specifications of the GPU meaning that we can obtain results for the prefetcher implementation for different cache sizes and number of cores.

The model has a naive approach regarding the connection to the DRAM since it assumes that it takes a fixed number of cycles for every memory request to come back. Which is not a completely accurate representation of the system. The simulation also only takes into account the GPU in the simulation so the DRAM is not shared with any other components. If running system tests with requirements on more accurate representation of other components the simulator gem5 can be used. Gem5 is a system level simulation tool used to make more accurate system simulations for hardware. However due to lack of time this was not considered in this thesis.

The baseline model used for the system level cache does not contain any prefetching mechanism. It does however always fetch the whole cache line when receiving a memory request. The data available from requests to the system level cache is the following:

ID: containing information about which L2 cache slice it came from (unit ID), its sequential number, transaction type (read or write) and which core sent the request (core ID).

Request reason: which GPU block sent the request and why.

Address: address to be fetched.

Cache flags: containing information about if the data is cacheable and bufferable.

3.2 Workflow

The first step of our process was to understand the current SLC implementation. Logs were printed for different signals inside the system to gain understanding. Benchmarks were run on the current implementation to provide a baseline result for further comparison.

The SLC logs were exported to python for data exploration purposes. A simple fixed stride prefetcher was built outside of the GPU model for early testing. During this testing different values for degree and cache sizes were tested and plotted. The next step was to bring this fixed stride prefetcher into the GPU model and to compile the code and perform benchmarks. Parameters for the method were implemented as global variables that could be changed with flags during runtime, this increased the efficiency of testing different configurations

of the prefetch models. The same process was later used for the rest of the prefetching algorithms explored.

The benchmarks used during testing of the models were gaming benchmarks provided by ARM. The benchmarks contain frames from different mobile games and 28 of these were selected to use for comparison of the prefetching models. The result numbers focused on were cycle count, SLC hit rate, external read requests, power estimate, energy estimate and prefetch accuracy.

3.3 Implemented prefetching models

In this chapter the implemented prefetchers will be presented. First, basic models like the Naive Stride Prefetcher, and later more complex models like the Adaptive Stream Detective Prefetcher and Modified Best-offset Prefetcher.

3.3.1 Decision of implemented prefetchers

All of the prefetchers described in the background section were analyzed in terms of being able to be implemented or not. Some of the prefetchers use information for their prefetches that is not available at the SLC of our model. In these cases the model has been skipped like the case of the Many Thread Aware Prefetcher and APOGEE. However the Naive Stride Prefetcher and Adaptive Degree Prefetcher implemented are both using some of the concepts provided in the APOGEE and Many Thread Aware Prefetcher papers. The Adaptive Stream Detective Prefetcher was able to be implemented in a very similar way as described in the paper and the Best-Offset Prefetcher implemented is slightly changed, thus called modified, but still has the same main functionality as the original.

3.3.2 Naive Stride Prefetcher, NSP

The first prefetcher implemented was a fixed stride prefetcher inspired by the APOGEE paper. It puts all memory requests in a Stride Prediction Table (SPT) containing Stride Prediction Entries, (SPEs) containing five values:

Address: previous memory request address

Confidence: number that represents the number of consecutive memory requests having constant stride

Recently_prefetched: boolean that represents if the current SPE has been prefetched recently without getting updated afterwards.

Lifetime: number that represents the remaining read access cycles until the SPE gets evicted from the table.

Positive: if the prefetcher uses positive or negative fixed stride

At every read request the SPT is updated according to following:

1. Two address are calculated for, one for positive and one for negative fixed stride. This value is simply the address subtracted by fixed stride and the address added with the fixed stride.

2. If one of these addresses are found in an SPE in the SPT, the confidence value for that SPE increases by 1 and the lifetime with LIFETIMEEXT. Recently_prefetched is set to zero for that SPE. If no matching SPT is found and there is free spaces in the table, a new SPE will be created with initial lifetime equal to the LIFETIME parameter.

3. All the lifetime values get deducted by 1.

Depending on the model parameter PREFETCHHITS the prefetcher will prefetch on only cache misses or both cache misses and hits. During a prefetch, a function is called that returns a queue of addresses to be fetched from the DRAM. In the code example in listing 1 it can be seen how the prefetcher inserts the prefetch addresses in the prefetching queue. The prefetching is heavily dependent on the fixed variables DEGREE, CONFTHRESH and DISTANCE. These variables were implemented as global runtime configuration parameters, enabling easy testing of different configurations during testing and when running benchmarks. It should be noted that the prefetcher is only using the confidence value to pass the threshold and is not adapting in any way to how confident it is. Having a more adaptive model will be tested in later models.

For keeping the values in the SPT recent, a lifetime method similar to the one in the Adaptive Stream Detection Prefetcher was used, meaning that if a SPE is not used for a fixed amount of read accesses it will be evicted from the SPT.

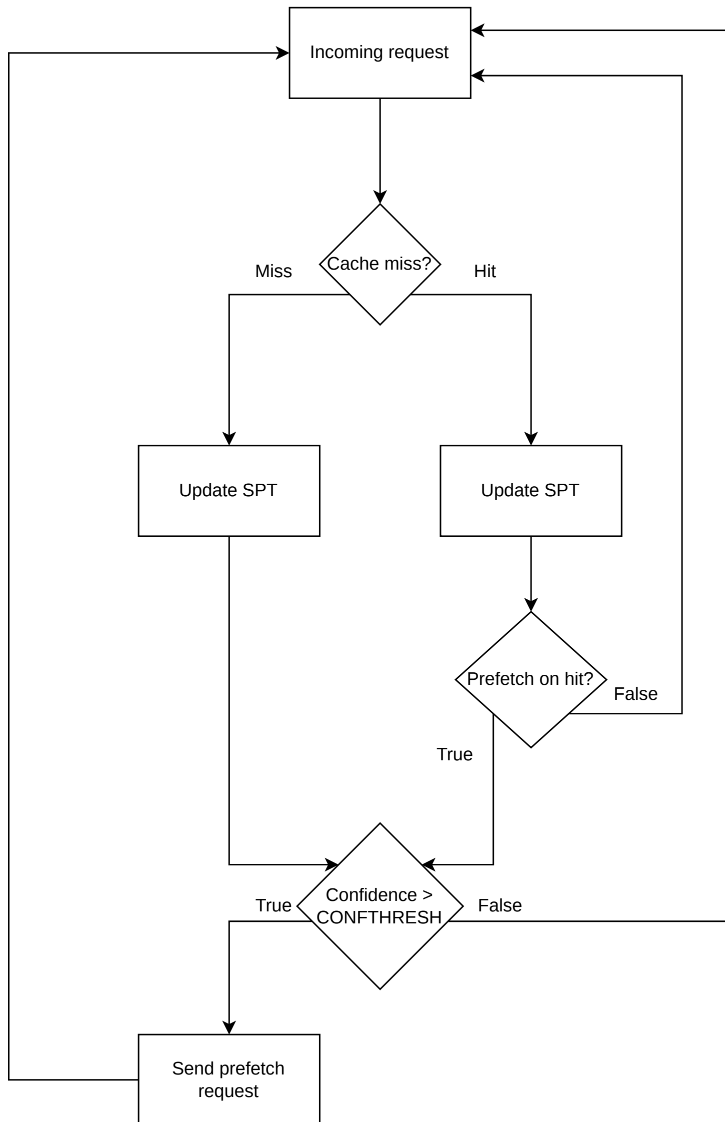

Figure 3: Flow diagram for the basic idea of prefetching

Listing 1: Code of the getPrefetch() function

```
for (SPE spe: SPT) {

    if(spe->confidence >= CONFTHRESH && !spe>recently_prefetched){

        for (int i = 1+DISTANCE; i < 1+DISTANCE+DEGREE; i++){

            prefetch_queue.push(spe->address + i*spe->stride);
        }
        spe->recently_prefetched = true;
    }

}
return prefetch_queue;
```

3.3.3 Adaptive Degree Prefetcher, ADP

The two main differences between ADP and NSP is that the stride is not fixed and that the confidence level decides the degree of the prefetch. ADP also uses reason to categorize the strides, thus only requiring to compare one stride in the SPT. The stride is decided such that when a reason that is not logged in the SPT a new SPE is logged in the SPT with an initial stride of zero. At the next occurrence of the request reason incoming it takes the new address and subtracts the logged address and save it as the new stride. Now for each time this stride is found in a row the confidence in the SPE is increased by one, see figure 4.

When the confidence has reached the CONFTHRESH a prefetch is started just like NSP. Now since ADP has, as in the name, adaptive degree this means that the higher the confidence the higher the degree. So for a SPE with high confidence will prefetch more lines than an SPE with low confidence based on the paper from Jaekyu Lee et. al [17]. But instead of using WARPs as they do in the paper the ADP uses request reasons as a way to make the information less random for the pattern detector.

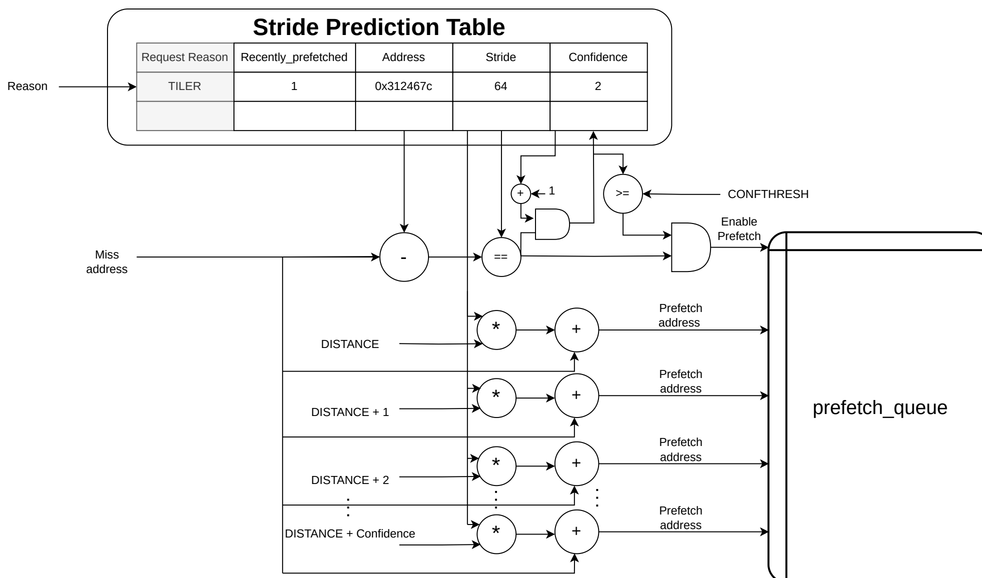

Figure 4: A graph of how prefetches are put into the prefetch queue for the ADP

3.3.4 Adaptive Stream Detective Prefetcher, ASDP

The ASDP implemented is close in function to the one described in the paper *Memory Prefetching Using Adaptive Stream Detection* [11] mentioned in the background. The method includes four configuration parameters that can be varied and the authors does not provide any recommendation for the values of these. The parameters are:

EPOCHLENGTH: the number of cycles of an epoch, meaning before LHT_curr is updated.

LIFETIME: the number of cycles before an entry is evicted from the Stream filter

LIFETIMEEXT: the number of lifetime that is added to entries each time they are observed.

DEGREE: the maximum number of prefetch requests sent out at a time, if they fulfill the inequality.

Multiple iterations of tests were performed to decide these parameters and the best ones were used during comparison to the other prefetchers. In the paper they only issued prefetch request at a time of two consecutive cache misses. In our implementation prefetch requests could be issued from cache hits as well, depending on the input flag PREFETCHHITS. Also, the improvements length-based stream detection, adaptive epoch length and variable length prefetching from the follow-up paper were implemented [12]. However, the throttle mechanism used in the paper was not used since it did not lead to improved performance.

3.3.5 Modified Best-Offset Prefetcher, MBOP

The best-offset inspired prefetcher implements a scoring table like in the original paper [18]. However it does not divide each epoch into rounds and instead calculates the offset to all the previous 16 accesses, and increases these offsets scores by one. This behaviour more resembles the Aggregate Stride Prefetcher [3]. There were also trials for increasing the degree by considerering multiple lookaheads like for the multi-lookahead prefetcher [27], but this implementation was not able to provide any further performance increases and is therefore not included in the thesis.

3.3.6 Last-level collective prefetcher

The prefetcher that was implemented based on this model used the same design with RPT and group table. There were only two things that differed from the original paper [19] which was that the hash function was changed and that when an SPE did not match the correct stride with the same hash function it was removed instead of lowering the confidence. The new hash function was done by concatinating Core ID and Reason. This gives each reason from each core its own entry to be able to find the best possible stride. The reason for not using PC was because it was not sent to the SLC and was therefore not available. And the reason for instantly removing the SPE when a stride did not match was because it gave an increase in performance.

4 Results

In this chapter benchmark results for the different prefetching models will be presented. The results will be focused around one GPU configuration while remaining configurations will be presented in Appendix A.

4.1 Simulation parameters

The simulation parameters seen in table 3 were used for the GPU model provided by ARM. We will refer to this configuration as the standard configuration. The standard configuration was the one being used during the creation of the prefetchers while the other configurations were used later during testing.

Parameter	**Value**
Number of cores	4
L2 cache slice size	256 KiB
Number of L2 slices	1
L3 cache slice size	1024 KiB
Number of L3 slices	1
L3 associativity	4 way

Table 3: Simulation parameters used for the GPU model

4.1.1 Prefetching model parameters

The different models have parameters that change the behaviour of the said model. Various different parameters were evaluated for each one of the models and the best observed parameters were further used for the final testing. The best model was regarded as the one providing the largest average decrease of cycle count on the chosen 28 gaming applications.

Parameter	**Value**
DEGREE	3
DISTANCE	0
CONFTHRESH.	0
LIFETIME	80
LIFETIMEEXT	40
TABLESIZE	8
PREFETCHHITS	false

Table 4: Parameters used for the NSP model

Parameter	**Value**
DEGREE	2
DISTANCE	0
CONFTHRESH	0
EPOCHSIMTHRESH	100
EPOCHLENGTH	adaptive (min:256, max:8000)
LIFETIME	80
LIFETIMEEXT	80
TABLESIZE	16
PREFETCHHITS	true

Table 5: Parameters used for the ASDP model

Parameter	**Value**
DEGREE	1
SCORETHRESH	50
EPOCHLENGTH	300
MAXOFFSET	32
NRRECENT	16
PREFETCHHITS	false

Table 6: Parameters used for the MBOP model

Parameter	**Value**
DISTANCE	1
CONFTHRESH	0
TABLESIZE	Nr. of AXI Reasons (48)

Table 7: Parameters used for the ADP model

Parameter	**Value**
DEGREE	2
CONFTHRESH	0
RPTTABLESLIZE	Nr. of cores · Nr. of AXI Reasons (192)
GTTABLESIZE	Nr. of AXI Reasons (48)

Table 8: Parameters used for the LLC model

4.2 Cycle count

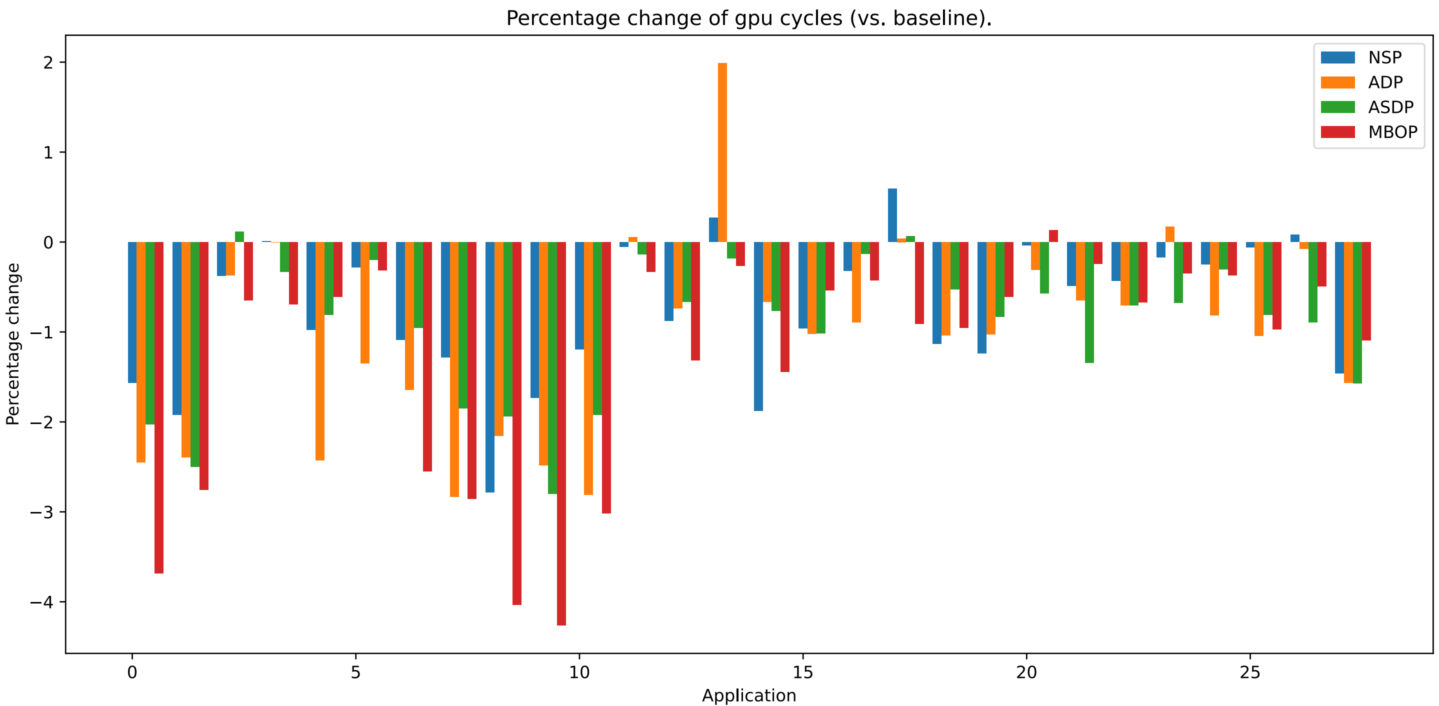

Figure 5: Percentage change of GPU cycles against baseline for all prefetchers. Negative values are better.

4.3 Hit rate

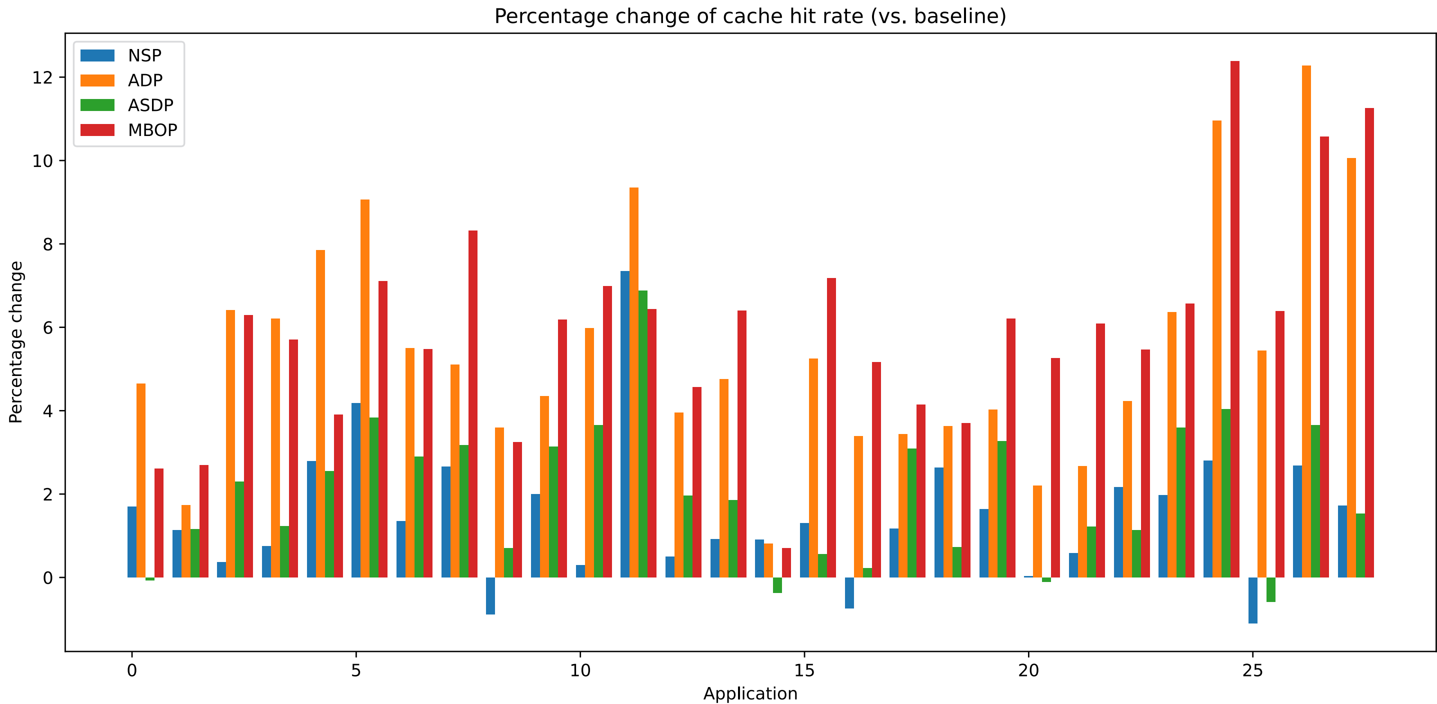

Figure 6: Percentage change of cache hit rate against baseline for all the different prefetchers.

4.4 Read requests

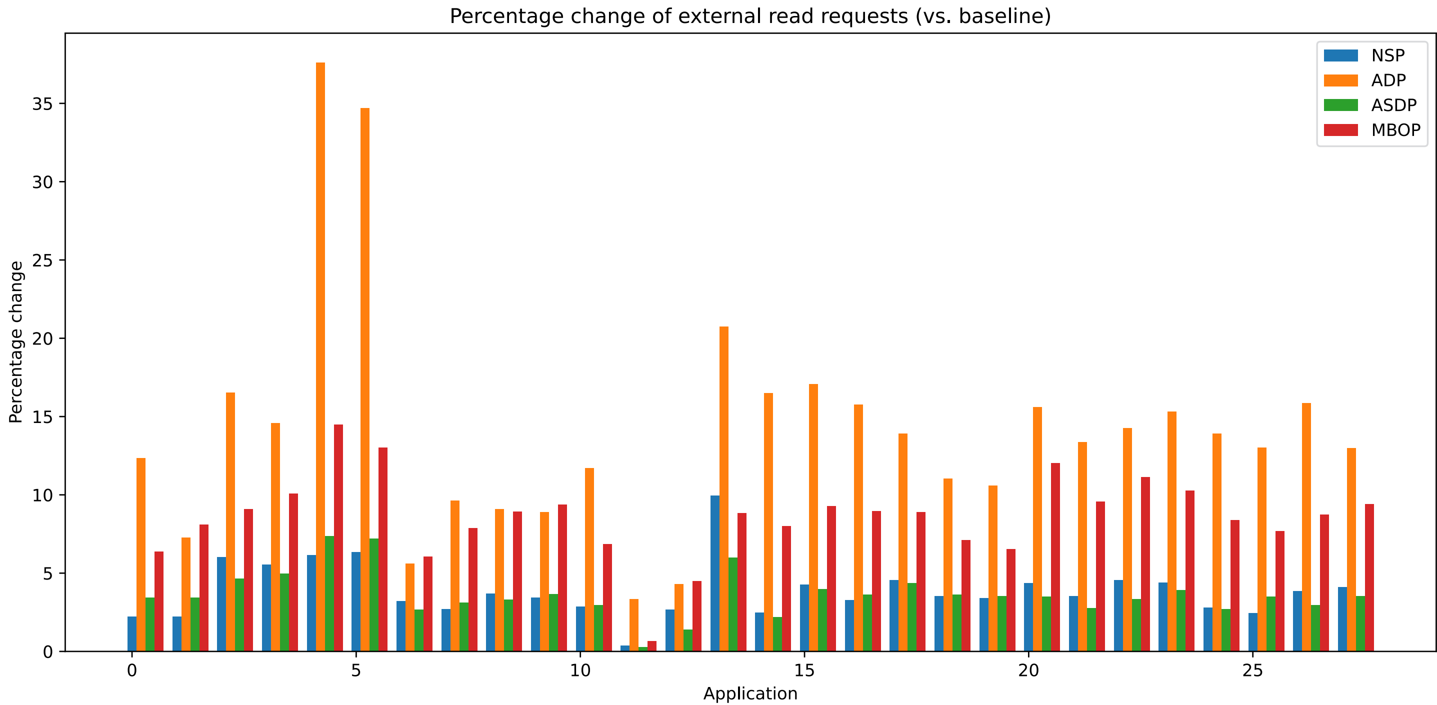

Figure 7: Percentage change of read requests sent from the SLC to the ext. memory.

4.5 Power estimation

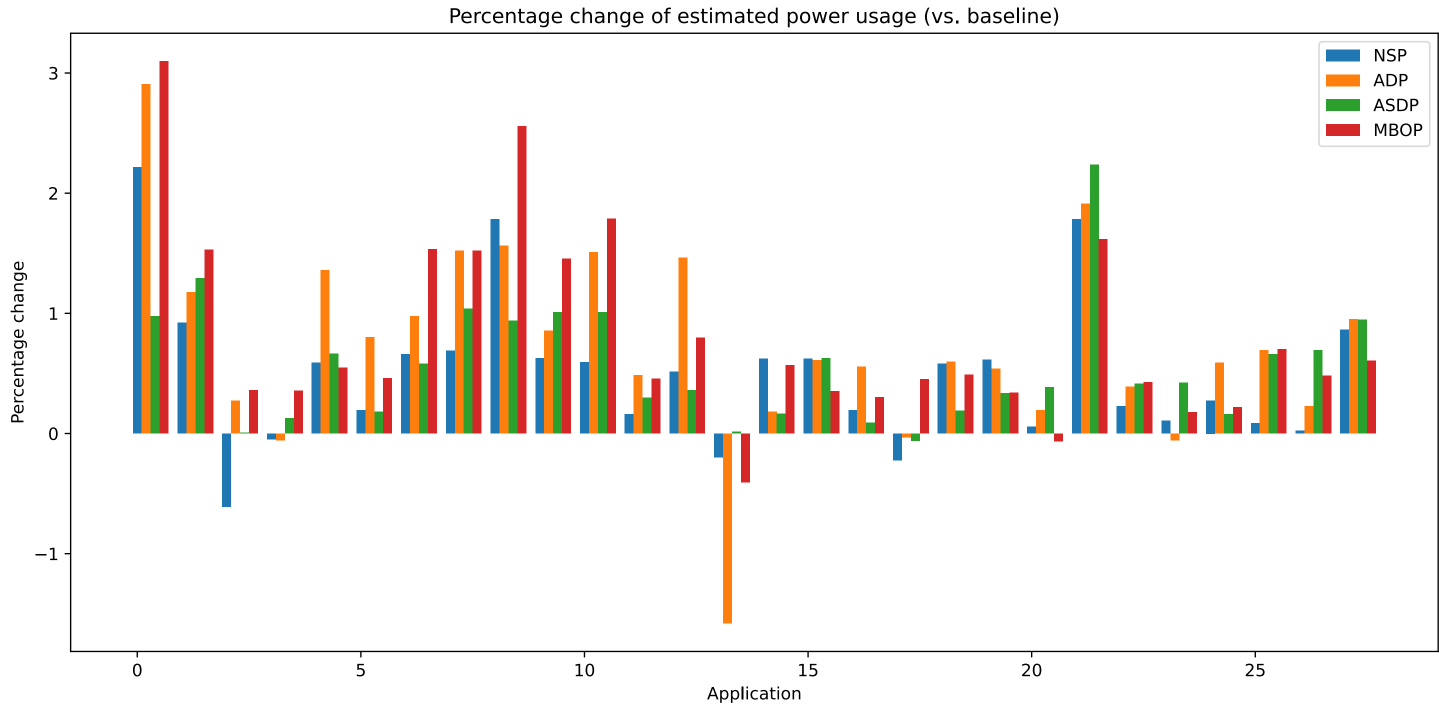

Figure 8: Percentage change of estimated power for all the different prefetchers.

4.6 Energy estimation

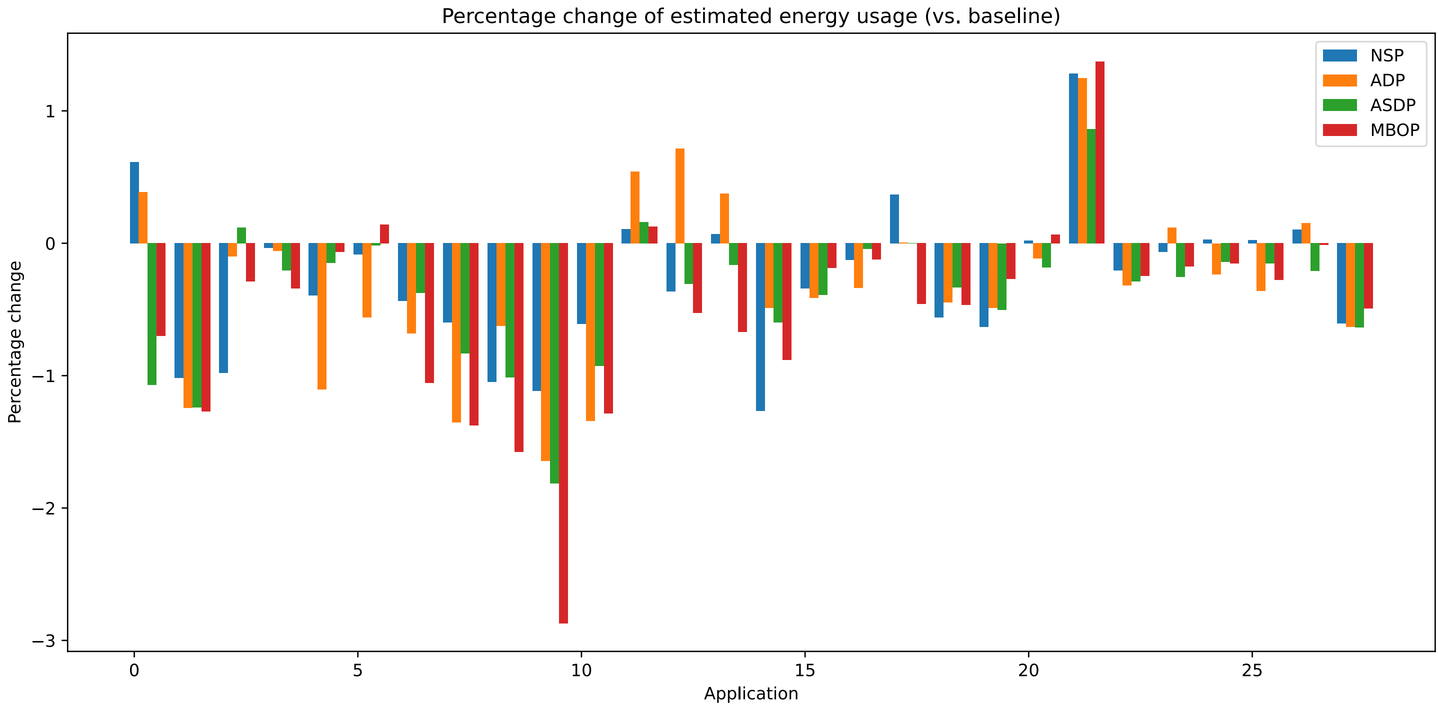

Figure 9: Percentage change of estimated energy against baseline for all the different prefetchers. Negative values are better.

4.7 Prefetch accuracy

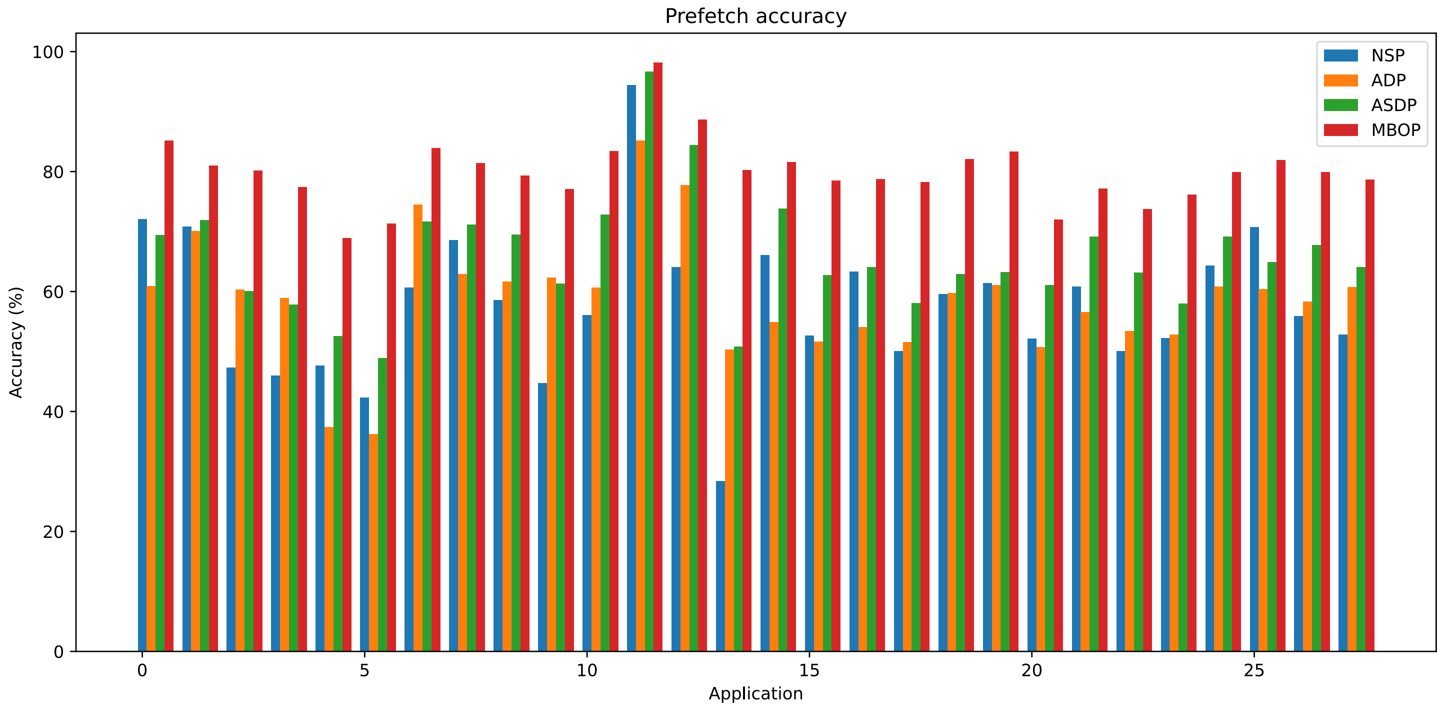

Figure 10: Prefetch accuracy of the implemented prefetchers. Higher values are better.

4.8 Averages

Measurement	NSP	ADP	ASDP	MBOP
GPU cycles [%]	−0.77	−1.04	−0.94	−1.29
Cache hit rate [%]	+1.5	+5.5	+2.0	+6.0
External read requests [%]	+3.9	+14.1	+3.6	+8.6
Estimated energy [%]	−0.28	−0.32	−0.38	−0.50
Estimated power [%]	+0.50	+0.74	+0.56	+0.81
Prefetch Accuracy [%]	58	59	66	80

Table 9: Table containing the average percentage differences compared to baseline and average prefetch accuracy of all the applications for the GPU configuration 4 cores and 1 slice.

4.9 Summary

4.9.1 Cycle count

Looking at figure 5 it is noticeable that for most of the tests all of the prefetchers have a positive impact on the cycle count. However, in application 13 an increase of around 2% can be observed for the ADP which represents a decrease in performance. In the best case scenarios a cycle count decrease of around 4.1% can be seen for the MBOP and in general this method is the best performing one in regards to the average in table 9. It should however be noted that the performance of the ADP is rather similar to the one of MBOP and for some applications it exceeds the MBOP in cycle reduction.

4.9.2 Hit rate

The ADP and MBOP always have a positive impact on hit rate, see figure 6. The other two prefetchers have mostly a positive impact with some smaller negative changes in hit rate. The average cache hit rate change is positive for all of them, see table 9. It can be observed that there is not a clear correlation between cache hit rate and cycle count as the largest hit rate increases do not correspond to the largest decreases of cycle count.

4.9.3 Read requests

In figure 7 the increase of external read request count can be seen for each prefetcher. For all of the prefetchers except for ADP the increase of read requests is rather uniform across the tests. It can be observed that NSP and ASDP have the lowest increase in general, see table 9. The ADP exhibits the most irregular behaviour of the prefetchers, and it has values between 37% and 3% which means that its behavior is very application dependent.

4.9.4 Power and energy estimation

Power and energy estimations can be seen in figures 8 and 9. In the power estimation, figure 8, it can be observed that in almost all cases the prefetchers lead to an increase in power consumption. Looking at energy estimation in figure 9

we can instead see that for most cases there is a change in the negative direction. Comparing 8 and 5 with the energy, one can observe that the applications with the largest decreases of cycle count have the largest increase in power and decrease in energy. It should be noted that both the power and energy numbers are rather naive estimates, since the DRAM is not properly modelled and also since the operations inside of the SLC is not taken into account for the estimates. The estimates should therefore not be regarded as accurate but more as an indication.

4.9.5 Prefetch Accuracy

Figure 10 shows the accuracy of the prefetchers for each application. It can be seen that MBOP performs the best out of the four. It should however be noted that this number is highly correlated to the number of external read requests. If a model issues more read requests it is expected to have a lower accuracy, as is the case for the ADP. The NSP and ASDP do have lower accuracy and lower cache hit rate which means that both prefetchers make worse prefetch predictions in general compared to MBOP. The ADP and NSP which are both based on the strided prefetcher share similar accuracy which is expected, but it can be seen that the ADP is more aggressive issuing a larger number of external read requests, thus increasing the cache hit rate.

4.9.6 Status on LLC prefetcher

The LLC prefetcher implemented ended up being very restrictive, in other words the increase in requests is too low to have a clear impact on performance. This leads to it having very similar performance to the baseline or worse and it was thus not included in the results.

4.10 Other GPU configurations

The simulations were performed for multiple other GPU configurations, see Appendix A. The results differ on a large scale compared to the results presented in this chapter. It should be noted that none of the prefetchers implemented had any specific way of handling a multiple slice setup which may lead to sub optimal performance since some memory access patterns may get lost when requests are sent to different slices. The time spent on parameter search for these models was also lower than the case of the standard configuration. These results are still included to point out some of the difficulties of designing a prefetcher that is usable for a broad range of GPU configurations. Additional GPU configuration experimentation in terms of SLC cache size, cache associativity and buffer sizes were also performed, however due to lack of time this was not properly analyzed and is therefore not included in the thesis.

The 1 core and 1 slice configuration favors the ADP instead of the MBOP in terms of cycle count, see figure 11 in appendix A.1. The cycle count reduction is however smaller for all of the prefetchers meaning that they perform worse compared to the standard configuration. It can especially be seen for NSP and ASDP which have most of their results close to 0.25% decrease. Looking at the 4 core and 2 slice cycle count results in figure 17 it can be noted that the MBOP

and ADP are performing rather similar but having more applications where it leads to increases in cycle count compared to the 4 core 1 slice configuration.

For the case of the 14 core and 8 slice configuration the cycle count values tend to behave more irregularly than in the case of the other configurations, see figure 23. For this configuration the NSP and ASDP lead to a decrease in performance for a majority of the applications. The ADP is the only prefetcher showing a somewhat regular performance increase for this configuration.

The number of external read requests vary a lot on the different configurations for the ADP. For NSP and ASDP it varies less and for MBOP it is rather stable. This is easily readable in the avarages in tables 10, 11 and 12. It can also be noted that the avarage accuracy is unstable between the different configurations, except for the MBOP. By looking at the accuracy and external read request number simultaneously you can see that they act according to expectations in regards to the cache hit rate, meaning that having low accuracy can still lead to high cache rate if enough read requests are being sent and vice versa.

The values in the following tables are taken from the figures in appendix A.

Measurement	**NSP**	**ADP**	**ASDP**	**MBOP**
GPU cycles [%]	−0.18	−0.80	−0.17	−0.52
Cache hit rate [%]	+3.3	+23.2	+3.8	+19.2
External read requests[%]	+3.5	+30.0	+4.1	+12.5
Estimated energy [%]	+0.11	−0.22	−0.10	−0.24
Estimated power [%]	+0.29	+0.58	+0.07	+0.29
Prefetch Accuracy [%]	64	59	70	77

Table 10: Table showing the average percentage differences compared to baseline and average prefetch accuracy for each measurement for the GPU configuration with 1 core and 1 slice.

Measurement	**NSP**	**ADP**	**ASDP**	**MBOP**
GPU cycles [%]	−0.35	−0.51	−0.28	−0.49
Cache hit rate [%]	+0.27	+1.61	−0.01	+3.88
External read requests [%]	+9.7	+43.7	+1.9	+12.9
Estimated energy [%]	−0.22	−0.29	−0.14	−0.24
Estimated power [%]	+0.13	+0.23	+0.14	+0.26
Prefetch Accuracy [%]	38	32	56	72

Table 11: Table showing the average percentage differences compared to baseline for each measurement and average prefetch accuracy for the GPU configuration with 4 core and 2 slices.

Measurement	**NSP**	**ADP**	**ASDP**	**MBOP**
GPU cycles [%]	−0.15	−1.18	+0.15	−0.27
Cache hit rate [%]	+1.10	+1.05	+0.95	+2.73
External read requests [%]	+4.05	+95.70	+3.58	+11.98
Estimated energy [%]	−0.09	−0.50	+0.08	−0.09
Estimated power [%]	+0.10	+0.72	−0.05	+0.21
Prefetch Accuracy [%]	37	20	43	72

Table 12: Table showing the average percentage differences compared to baseline for each measurement and average prefetch accuracy for the GPU configuration with 14 core and 8 slices.

5 Discussion

In this chapter the acquired results will be discussed to compare the performance of the models in terms of GPU cycles, hit rate, number of requests, accuracy and power estimation numbers.

5.1 Evaluation of performance

MBOP is the prefetcher being able to lower the cycle count the most using the standard GPU configuration. It is also the most energy efficient prefetcher which makes it the best overall prefetcher for this configuration. The explanation for this performance increase over the remaining prefetching models is the models ability to observe and act upon complex address patterns using the models scoring system, and since it is a degree one prefetcher it keeps the read requests on a stable level. The ADP which offers the second best performance on the standard configuration does suffer from its high prefetch degree, leading to very high number of read requests in most of the applications. This should in return impact the energy and power numbers negatively, however in the estimates acquired this does not seem to be the case as ADP is still able to decrease the overall energy consumption. To find out whether this is accurate the tests would need to be performed using a model which includes a more representative DRAM, like running it on gem5. The same argument can be said for the other GPU configurations, in all of them the ADP heavily impacts the external read requests while still achieving promising energy estimations.

For the other GPU configurations it should however be noted that the ADP is in favor in terms of the GPU cycles, looking at the averages in table 10, 11, 12 one can observe that it performs the best on all of these configurations. This performance increase looks to be coming from the prefetches of the method being more timely, since the cache hit rate is actually not the highest of the three models in the 4 core and 2 slice and 14 core and 8 slice configurations. The reason for the better timeliness of the method is likely due to its high degree of fetching, meaning it fetches a long distance in the future. This does come at a cost however, since the accuracy is low for the model, which means that a lot of prefetches are not being used and is only polluting the system. To improve this, further work could be done on the adaptive part of the model to try and minimize the number of useless fetches. Adaptive distance was never implemented for the model, and it could be a usable way of reducing the number of read requests (by lowering degree) while still maintaining the good timeliness of the model.

The slice architecture makes it so that continuous strides can be hard to detect. Since there is a separate prefetcher in each slice of the SLC our prefetchers are unable to look at all incoming read requests from all L2C. This makes it harder to find long patterns sent from the L2C since it can easily be broken off by the following request being sent from a different core connected to a different L2C resulting in being received by a different slice in the SLC and, in extension, a different prefetcher. This is one of the most limiting factors to the prefetchers since it inhibits their basic idea of finding long patterns of address requests.

Looking at the average decrease of the cycle count may not always be the best way of evaluating the performance of the prefetchers. The GPU manufacturers may have different prioritization for the different applications, meaning that

improving some of them would be more beneficial than others. This should in that case also be considered when deciding on a prefetcher. There may also be restrictions regarding the case of performance decrease because of the prefetcher. If a prefetcher that never causes increases in cycle count is required, one would need to redesign the implemented prefetchers taking this into account, probably leading to them being more restrictive in their fetching, leading to lower, but more safe performance gains.

Another thing that should be taken into account is the area cost for each prefetcher. In this thesis the prefetchers were never implemented in hardware descriptive language and estimates for these area costs are therefore hard to obtain. Multiple optimizations of the models could have been tried to reduce the estimated area, for example, saving only the last bits when calculating the strides for some of the prefetchers. This would lead to smaller area for both the memory and computational units used, since bit length would be lower. However, in general, the NSP model would be the model with the overall lowest area cost because of its simple algorithm and since it has the smallest table size of the prefetch models evaluated. For the remaining models, its hard to conclude much since they include more complicated logics. In regards of table size ASDP and MBOP have smaller tables of 16, compared to ADPs 48, but in regards of computational logic it would be the opposite, ASDP and MBOP is estimated to require more computational hardware than ADP. In the end, to draw conclusions about area costs, the models should be optimized and implemented in hardware descriptive language for better estimations.

The gem5 model which is a system simulator was not used in this master thesis since there were no time to get it working. We assume that more accurate test results would have been generated. But the results that is generated from the model is accurate enough to make predictions on how good the different prefetchers are compared to baseline and against each other but might not give us very precise performance numbers.

5.2 Further exploration

An improvement possibility to take into account would be ways of combining the different prefetchers to try and achieve improved performance. Implementing a confidence based system like the ADP for the MBOP prefetcher could lead to the MBOP prefetcher fetching more data with hopefully the same high accuracy, leading to better overall performance. One could also do experiments trying to incorporate the ASDP's histogram into one of the other prefetchers to see if this would lead to any improvements for short streams. It should however be noted that the more advanced the prefetchers become more area and power will be used by the prefetchers, which in the end, might lead to overall worse performance.

When running the GPU model with configurations of 2 or more cache slices the prefetchers are struggling finding as good access patterns, since the accesses are divided between the slices and there is no current way of communicating between these. Having a prefetching control unit that is communicating and snooping on the addresses arriving at all the cache slices could therefore lead to a better solution. This controller would in that case be able to see the same patterns as the case for 1 slice, thus enabling the system to achieve similar performance gains to the standard configuration. This was not implemented

during this thesis due to lack of time.

6 Conclusion

6.1 Summary

In the end, it can be concluded that using a prefetcher in the SLC can lead to performance increases for gaming graphic workloads. Different prefetchers can have vastly different results and for the standard configuration used in this thesis the MBOP performed the best overall. However for the other GPU configuration used the ADP seem to have a slight edge. It can also be concluded that the performance changes heavily depending on the gaming application used and that for some of them all prefetchers tested are having difficulties improving the performance.

6.2 Future research

In extension to the suggested explorative work described earlier it should also be noted that there exists a lot of prefetch algorithms that were not implemented in this thesis. Some interesting ones are the Long short term memory based hardware prefetcher which implements a neural network to predict the future accesses [30], Snake: a variable-length chain-based prefetcher which uses chains of variable strides [21], WaSP: warp scheduling to mimiic prefetching in graphics workloads, which exploits localities in graphic workloads using warp information[14] and the Treelet prefetching for ray tracing, which uses a tree data structure to manage its prefetches [7].

References

[1] Computer science. [Online; accessed 27-March-2024]. URL: https://cs.lth.se/edaf80/lectures/.

[2] Pc mag. encyclopedia - cmp. [Online; accessed 27-March-2024]. URL: https://www.pcmag.com/encyclopedia/term/cmp?_gl=1%2Av0o6vx%2A_up%2AMQ..%2A_ga%2AMjA4NzY3NTkzNi4xNzEyODI2NzU5%2A_ga_2Y85WP1X8R%2AMTcxMjgyNjc1OS4xLjAuMTcxMjgyNjc1OS4wLjAuMA..

[3] Exploiting long-term behavior for improved memory system performance. 2016. URL: https://api.semanticscholar.org/CorpusID:64381716.

[4] Edward Angel and Dave Shreiner. *Interactive Computer Graphics*. Pearson, 2020.

[5] ARM. Cache architecture. [Online; accessed 6-March-2024]. URL: https://developer.arm.com/documentation/den0013/d/Caches/Cache-architecture.

[6] Kyoshin Choo. Reducing cache contention on gpus. In *Electronic Theses and Dissertations. 454*. eGrove, 2016. URL: https://egrove.olemiss.edu/etd/454.

[7] Yuan Hsi Chou, Tyler Nowicki, and Tor M. Aamodt. Treelet prefetching for ray tracing. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 742–755, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3613424.3614288.

[8] Michael Doggett. Texture caches. *IEEE Micro*, 32(3):136–141, 2012. doi:10.1109/MM.2012.44.

[9] Val Eze, Martin Eze, Enerst Edozie, and Esther Eze. Design and development of effective multi-level cache memory model. *International Journal of Recent Technology and Applied Science (IJORTAS)*, 5:54–64, 09 2023. doi:10.36079/lamintang.ijortas-0502.515.

[10] J.W.C. Fu, J.H. Patel, and B.L. Janssens. Stride directed prefetching in scalar processors. In *[1992] Proceedings the 25th Annual International Symposium on Microarchitecture MICRO 25*, pages 102–110, 1992. doi:10.1109/MICRO.1992.697004.

[11] Ibrahim Hur and Calvin Lin. Memory prefetching using adaptive stream detection. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 397–408, 2006. doi:10.1109/MICRO.2006.32.

[12] Ibrahim Hur and Calvin Lin. Feedback mechanisms for improving probabilistic memory prefetching. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 443–454, 2009. doi:10.1109/HPCA.2009.4798282.

[13] Akanksha Jain and Calvin Lin. Cache replacement policies. [Online; accessed 6-March-2024]. URL: `https://par.nsf.gov/servlets/purl/10113803`.

[14] Diya Joseph, Juan Luis Aragón, Joan-Manuel Parcerisa, and Antonio Gonzalez. Wasp: Warp scheduling to mimic prefetching in graphics workloads, 2024. `arXiv:2404.06156`.

[15] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. Accel-sim: An extensible simulation framework for validated gpu modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, May 2020. URL: `http://dx.doi.org/10.1109/ISCA45697.2020.00047`, `doi:10.1109/isca45697.2020.00047`.

[16] Varma B.S. Lal, S. and Juurlink. A quantitative study of locality in gpu caches for memory-divergent workloads. In *Int J Parallel Prog 50*, page 189–216, 2022. `doi:10.1007/s10766-022-00729-2`.

[17] Jaekyu Lee, Nagesh B. Lakshminarayana, Hyesoon Kim, and Richard Vuduc. Many-thread aware prefetching mechanisms for gpgpu applications. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 213–224, 2010. `doi:10.1109/MICRO.2010.44`.

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

[19] George Michelogiannakis and John Shalf. Last level collective hardware prefetching for data-parallel applications. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 72–83, 2017. `doi:10.1109/HiPC.2017.00018`.

[20] Sparsh Mittal. A survey of recent prefetching techniques for processor caches. *ACM Comput. Surv.*, 49(2), aug 2016. `doi:10.1145/2907071`.

[21] Saba Mostofi, Hajar Falahati, Negin Mahani, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Snake: A variable-length chain-based prefetching for gpus. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 728–741, New York, NY, USA, 2023. Association for Computing Machinery. `doi:10.1145/3613424.3623782`.

[22] K.J. Nesbit and J.E. Smith. Data cache prefetching using a global history buffer. In *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, pages 96–96, 2004. `doi:10.1109/HPCA.2004.10030`.

[23] Reena Panda, Yasuko Eckert, Nuwan Jayasena, Onur Kayiran, Michael Boyer, and Lizy Kurian John. Prefetching techniques for near-memory throughput processors. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, New York, NY, USA, 2016. Association for Computing Machinery. `doi:10.1145/2925426.2926282`.

[24] A. Rohan, B. Panda, and P. Agarwal. Reverse engineering the stream prefetcher for profit. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroSamp;PW)*, pages 682–687, Los Alamitos, CA, USA, sep 2020. IEEE Computer Society. URL: `https://doi.ieeecomputersociety.org/10.1109/EuroSPW51379.2020.00098`, `doi:10.1109/EuroSPW51379.2020.00098`.

[25] Michael Satran. Graphics pipeline - win32 apps, Feb 2022. URL: `https://learn.microsoft.com/en-us/windows/win32/direct3d11/overviews-direct3d-11-graphics-pipeline`.

[26] Ankit Sethia, Ganesh Dasika, Mehrzad Samadi, and Scott Mahlke. Apogee: Adaptive prefetching on gpus for energy efficiency. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 73–82, 2013. `doi:10.1109/PACT.2013.6618805`.

[27] Mehran Shakerinava, Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Multi-lookahead offset prefetching. 2019. URL: `https://api.semanticscholar.org/CorpusID:199570386`.

[28] OpenGL Wiki. Vertex shader — opengl wiki,, 2017. [Online; accessed 6-March-2024]. URL: `http://www.khronos.org/opengl/wiki_opengl/index.php?title=Vertex_Shader&oldid=14097`.

[29] Cyril Zeller, Randy Fernando, Matthias Wloka, and Mark Harris. Programming graphics hardware. In Daniel Cohen-Or and Sabine Coquillart, editors, *25th Annual Conference of the European Association for Computer Graphics, Eurographics 2004 - Tutorials, Grenoble, France, August 30 - September 3, 2004*. Eurographics Association, 2004. URL: `https://doi.org/10.2312/egt.20041034`, `doi:10.2312/EGT.20041034`.

[30] Yuan Zeng and Xiaochen Guo. Long short term memory based hardware prefetcher: a case study. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '17, page 305–311, New York, NY, USA, 2017. Association for Computing Machinery. `doi:10.1145/3132402.3132405`.

A Results for other GPU configurations

In thus appendix the gathered measurements for the other gpu configurations are provided. The number of slices are the same for L2 cache and SLC.

A.1 1 core and 1 slice

A.1.1 Cycle count

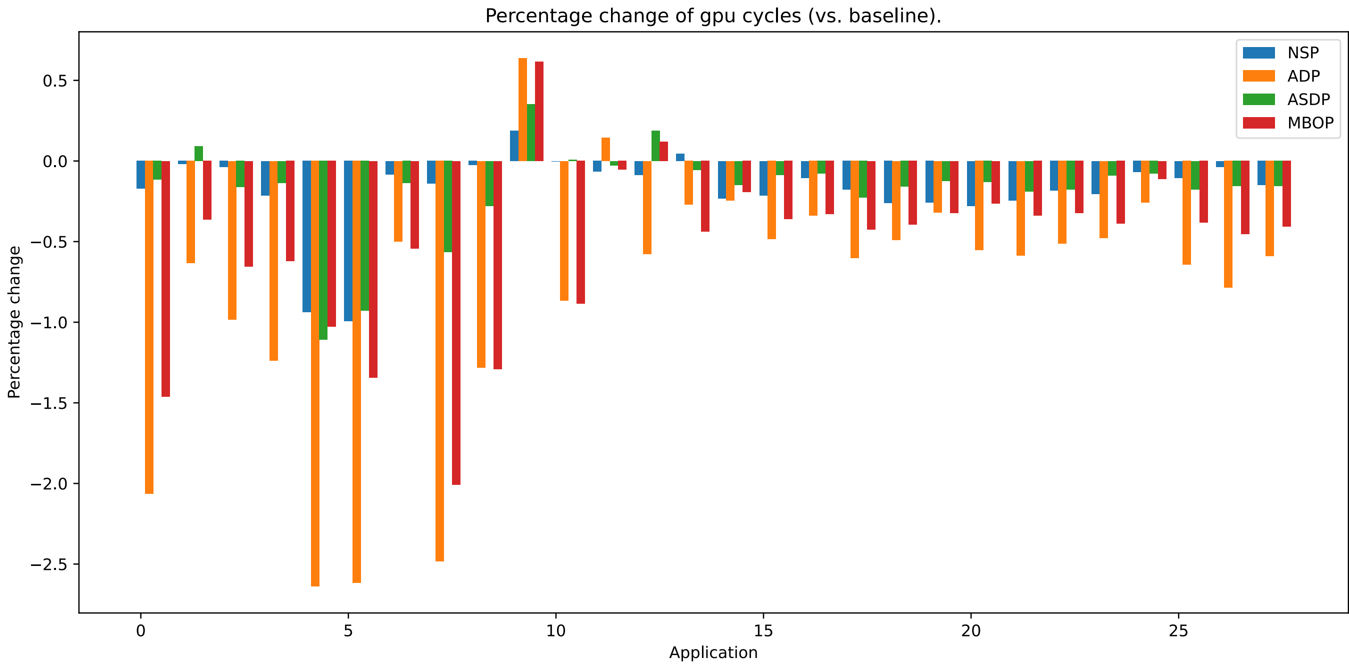

Figure 11: Percentage change of GPU cycles against baseline for all prefetchers. Negative values are better.

A.1.2 Hit rate

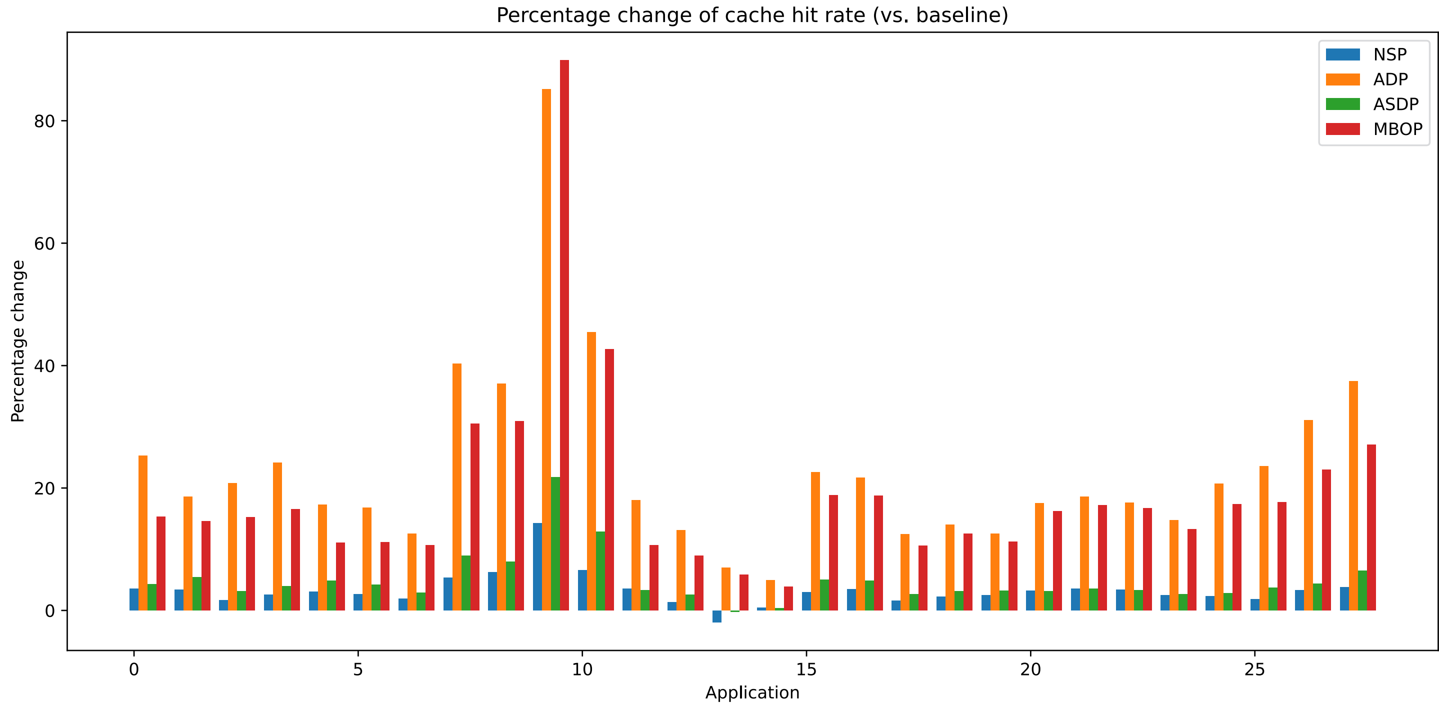

Figure 12: Percentage change of cache hit rate against baseline for all the different prefetchers.

A.1.3 Read requests

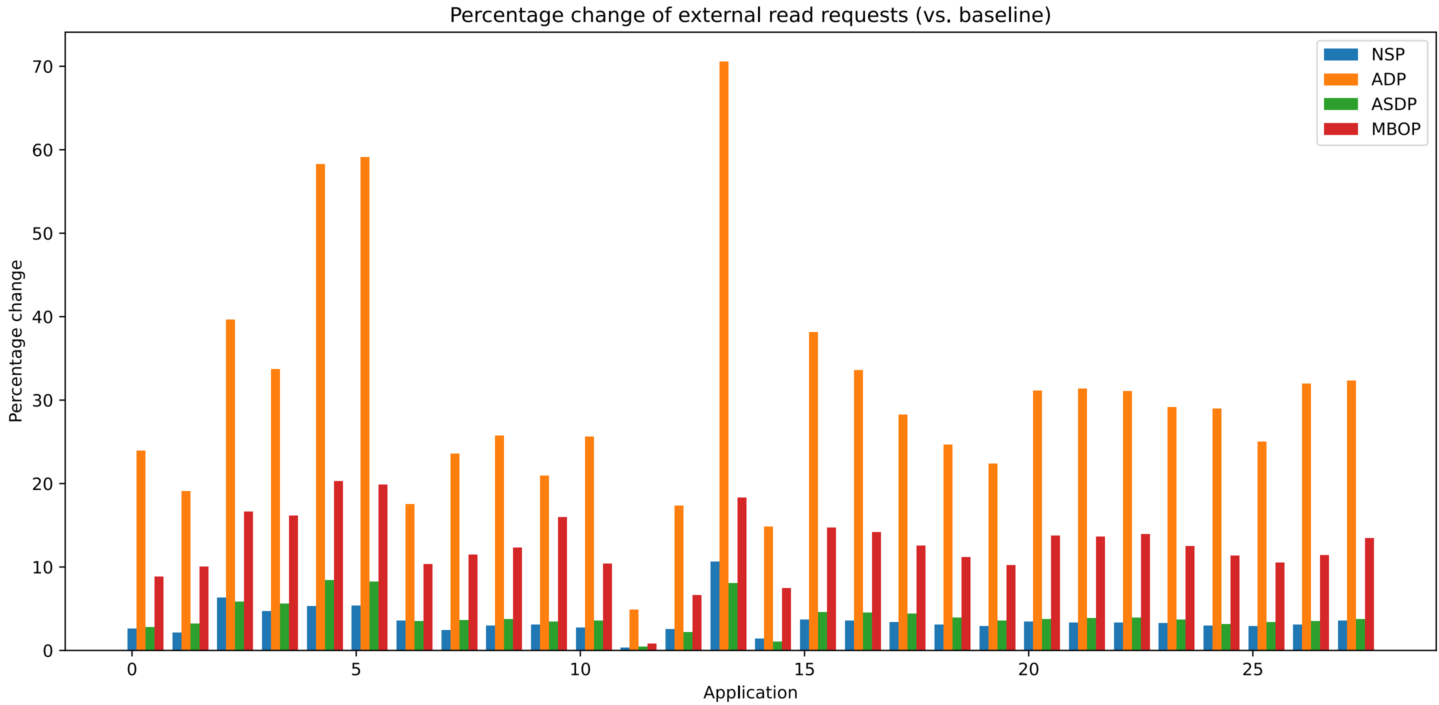

Figure 13: Percentage change of read requests sent from the SLC to the ext. memory.

A.1.4 Power estimation

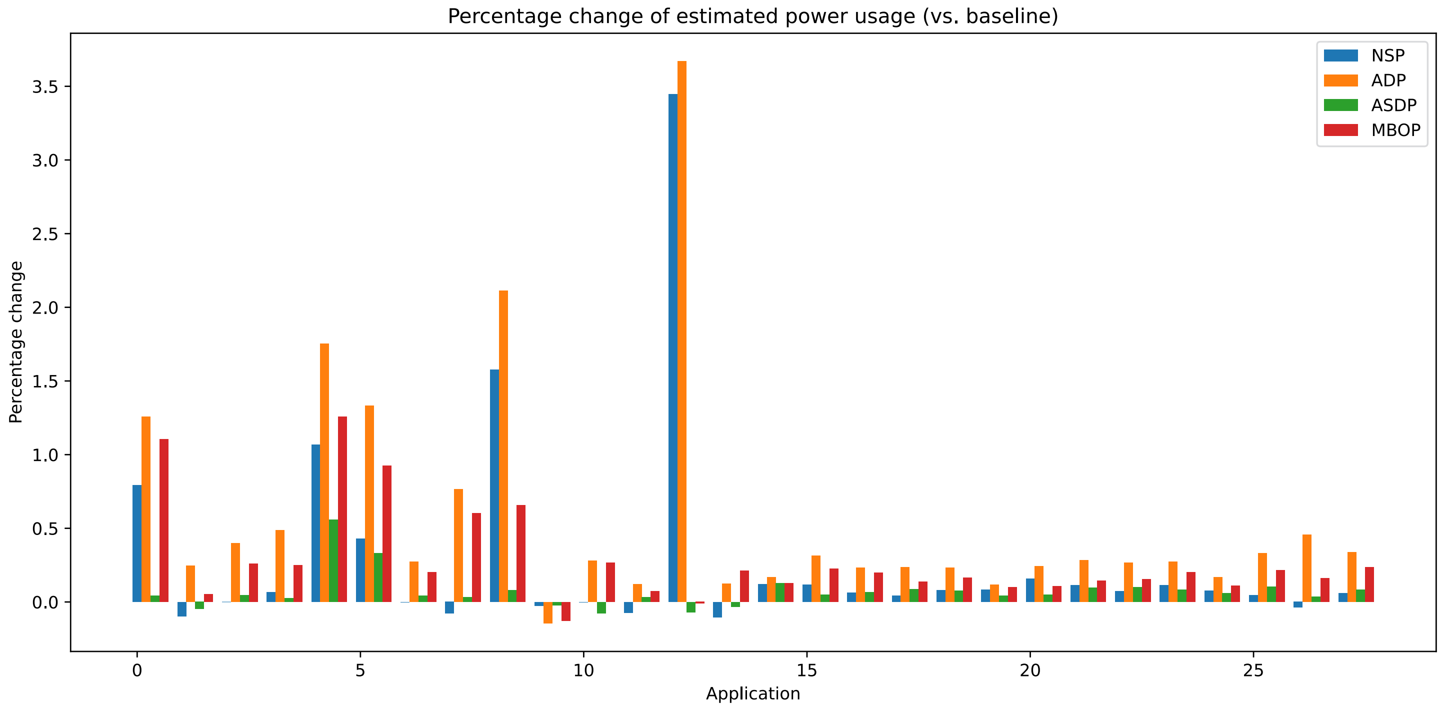

Figure 14: Percentage change of power for all the different prefetchers.

A.1.5 Energy estimation

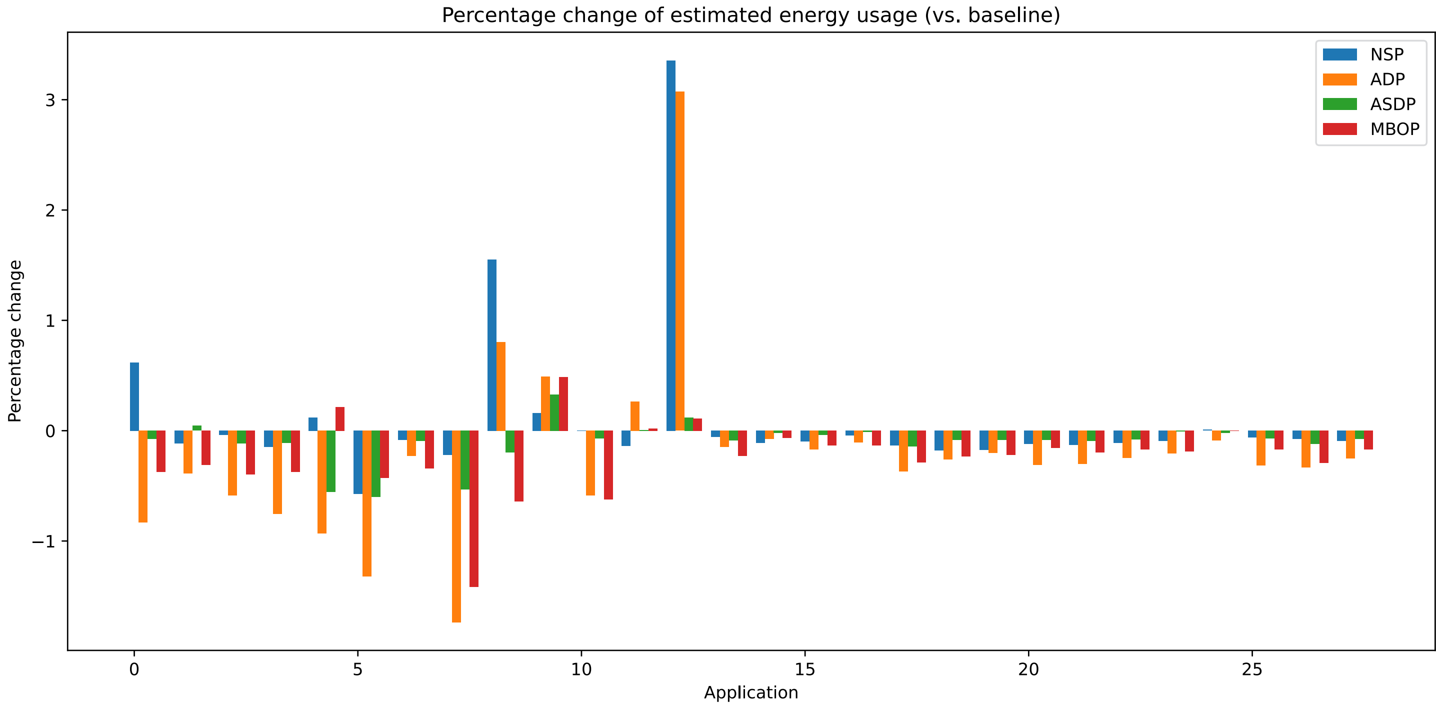

Figure 15: Percentage change of energy against baseline for all the different prefetchers. Negative values are better.

A.1.6 Prefetch accuracy

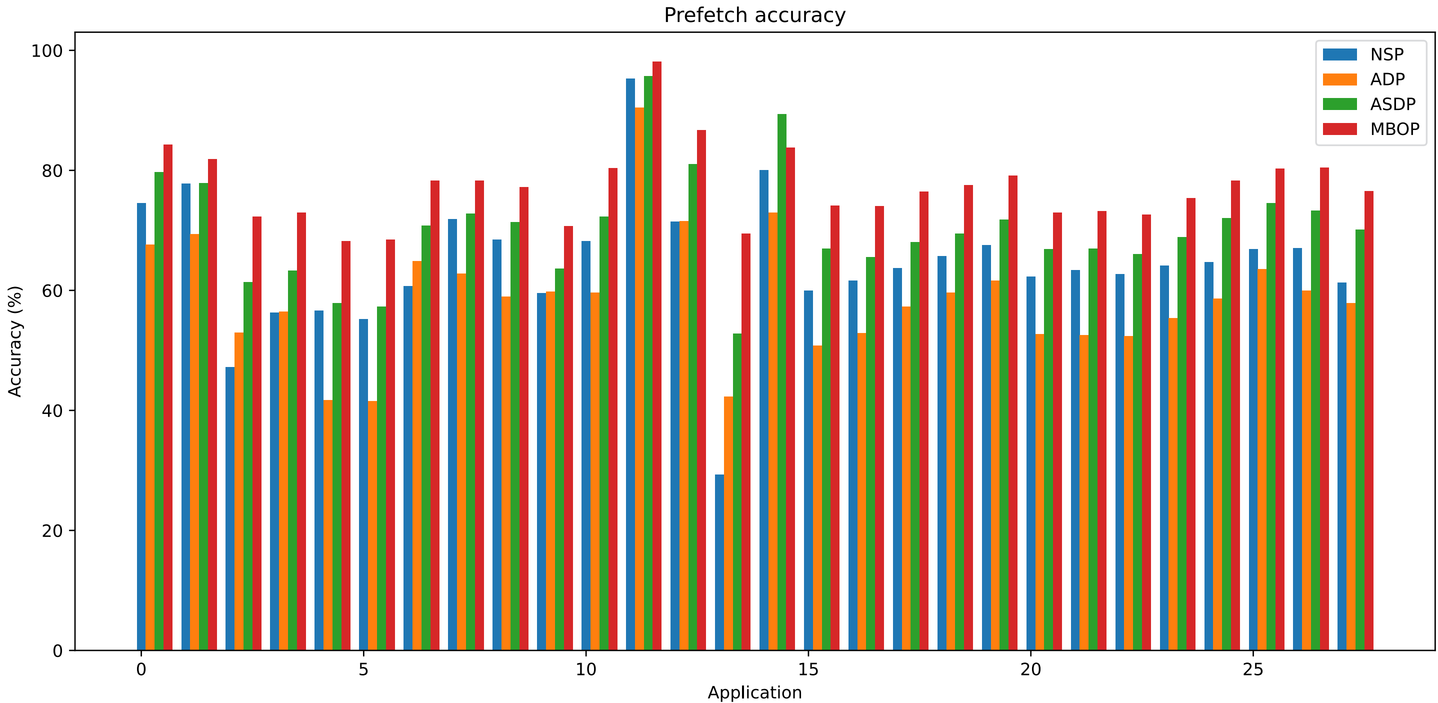

Figure 16: Prefetch accuracy of the implemented prefetchers. Higher values are better.

A.2 4 cores and 2 slices

A.2.1 Cycle count

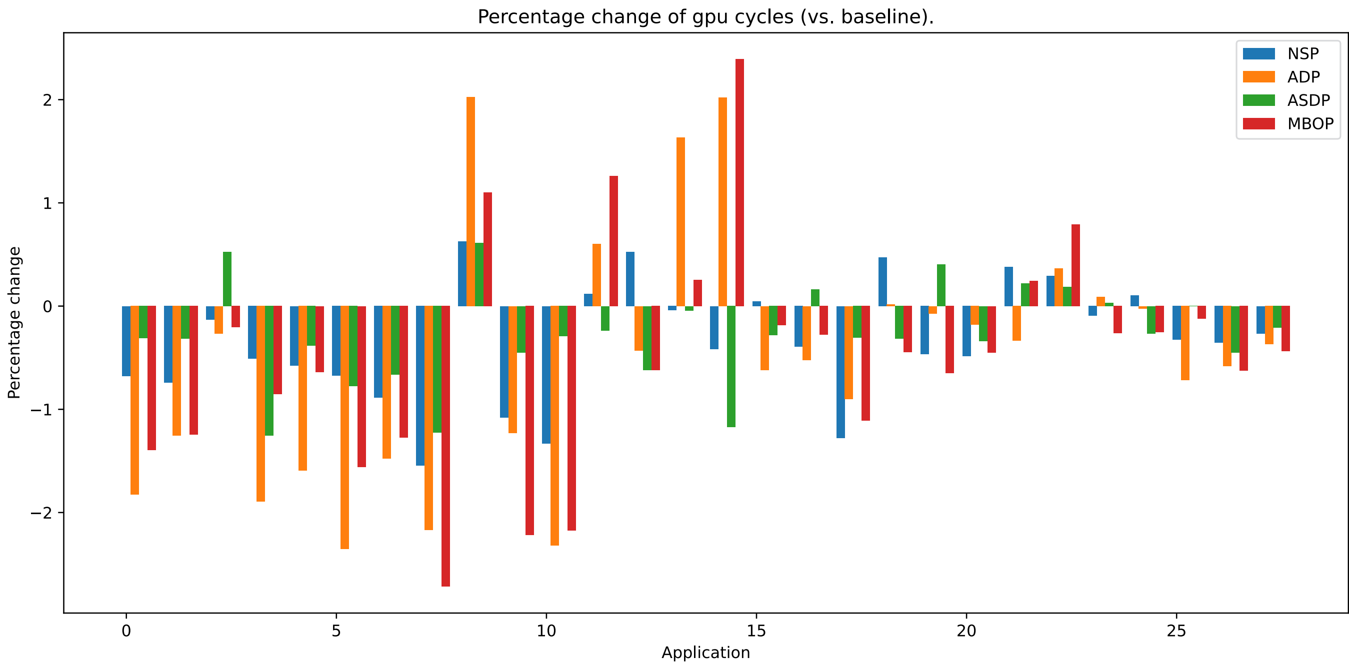

Figure 17: Percentage change of GPU cycles against baseline for all prefetchers. Negative values are better.

A.2.2 Hit rate

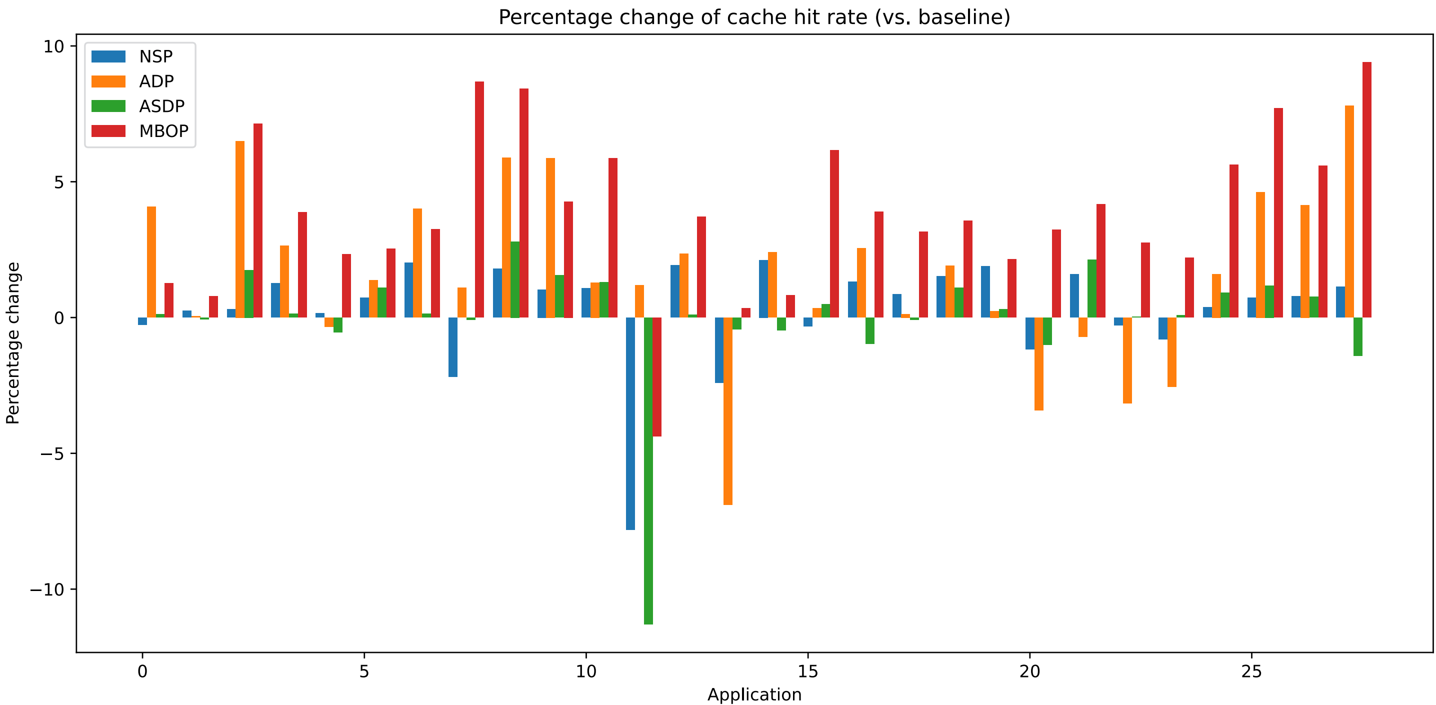

Figure 18: Percentage change of cache hit rate against baseline for all the different prefetchers.

A.2.3 Read requests

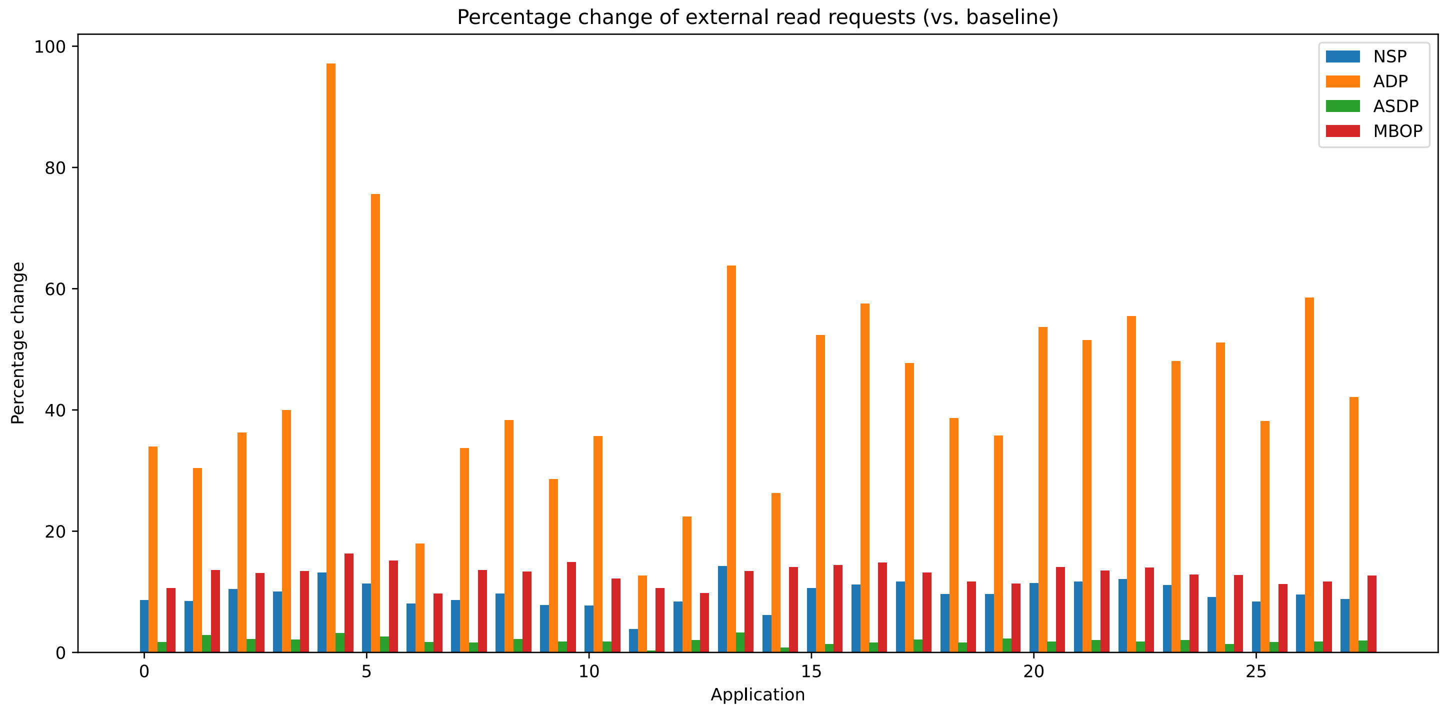

Figure 19: Percentage change of read requests sent from the SLC to the ext. memory.

A.2.4 Power estimation

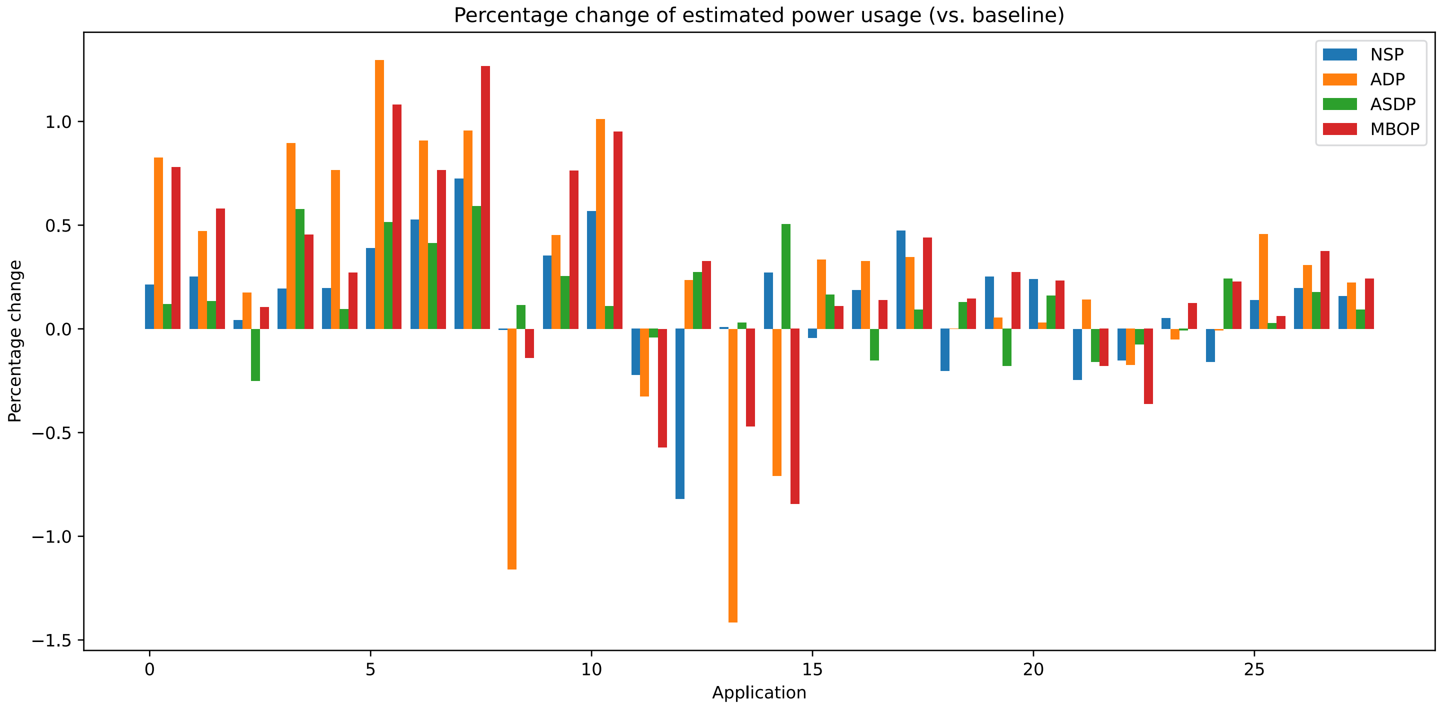

Figure 20: Percentage change of power for all the different prefetchers.

A.2.5 Energy estimation

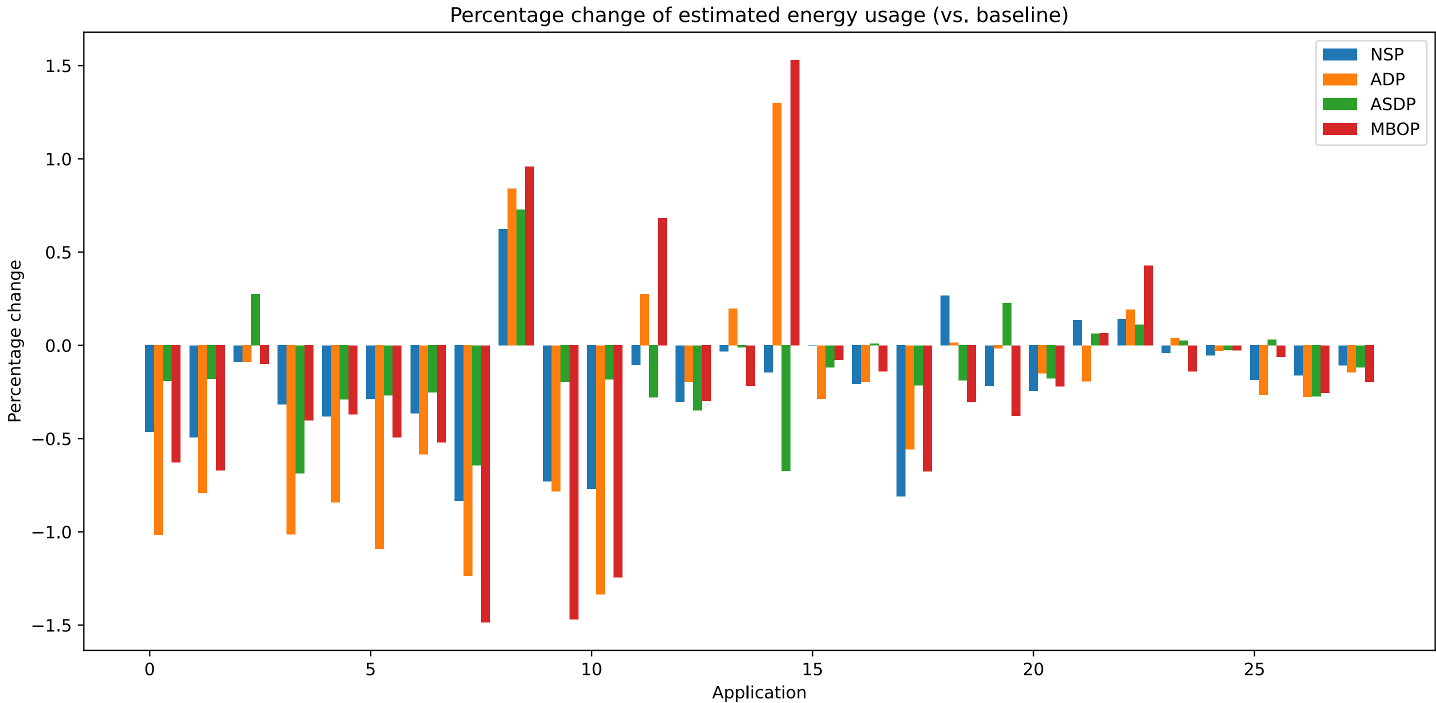

Figure 21: Percentage change of energy against baseline for all the different prefetchers. Negative values are better.

A.2.6 Prefetch accuracy

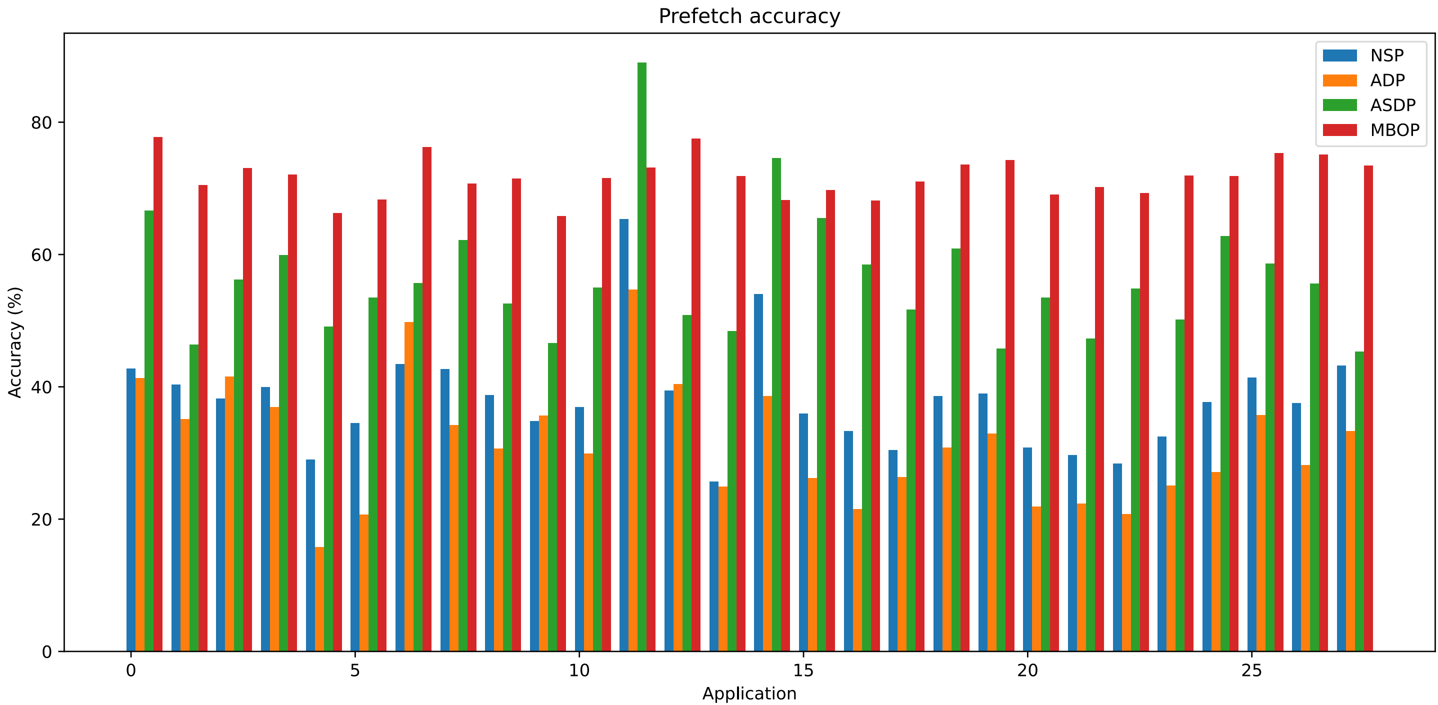

Figure 22: Prefetch accuracy of the implemented prefetchers. Higher values are better.

A.3 14 cores and 8 slices

A.3.1 Cycle count

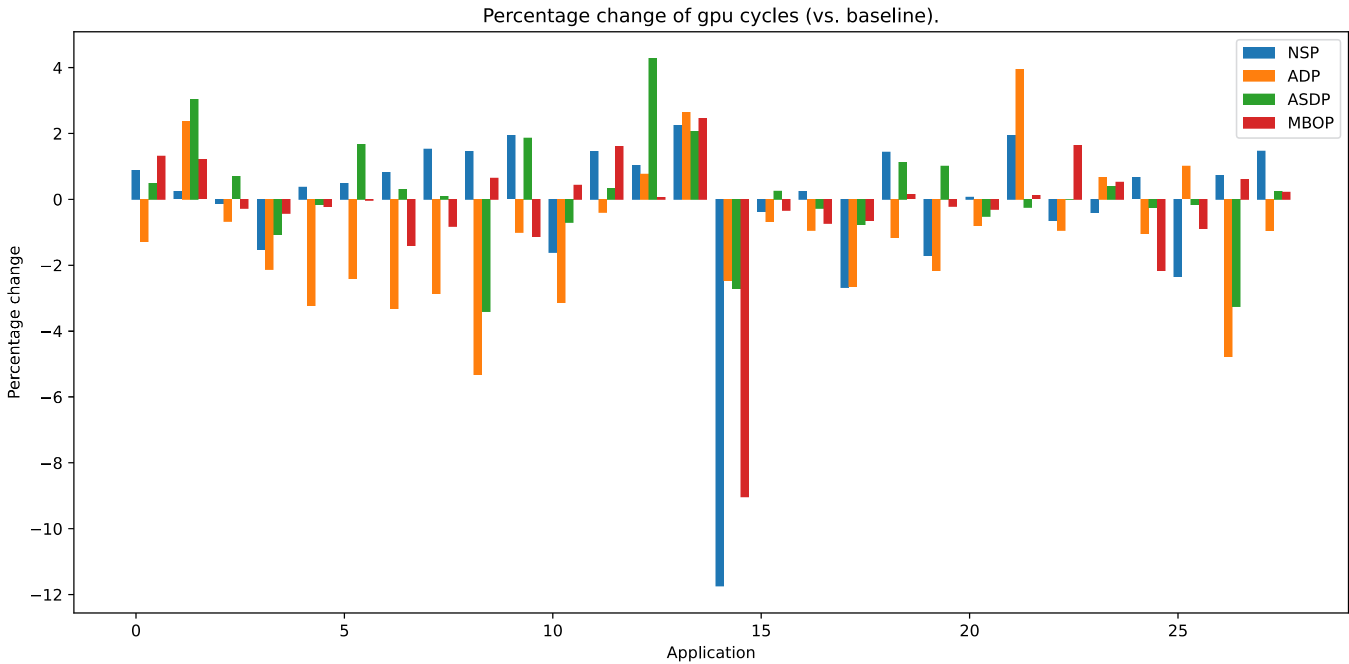

Figure 23: Percentage change of GPU cycles against baseline for all prefetchers. Negative values are better.

A.3.2 Hit rate

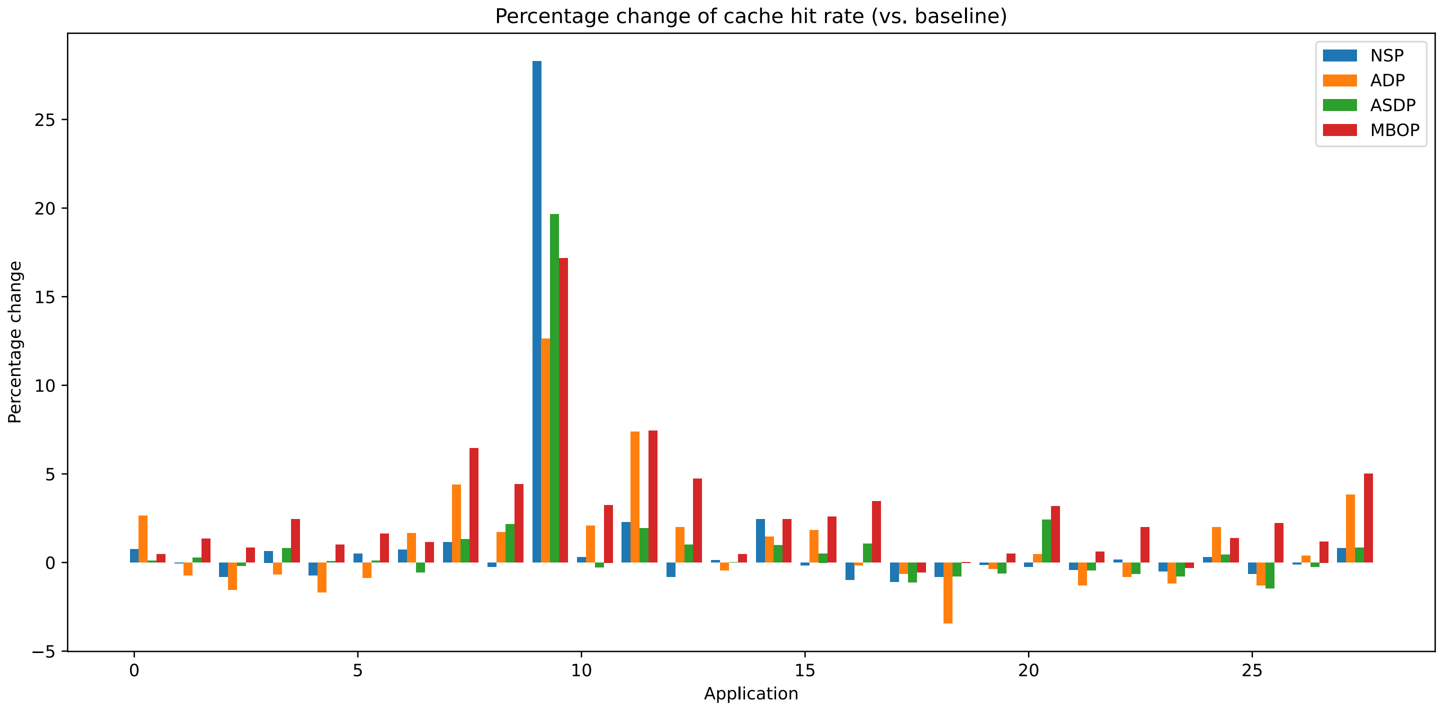

Figure 24: Percentage change of cache hit rate against baseline for all the different prefetchers.

A.3.3 Read requests

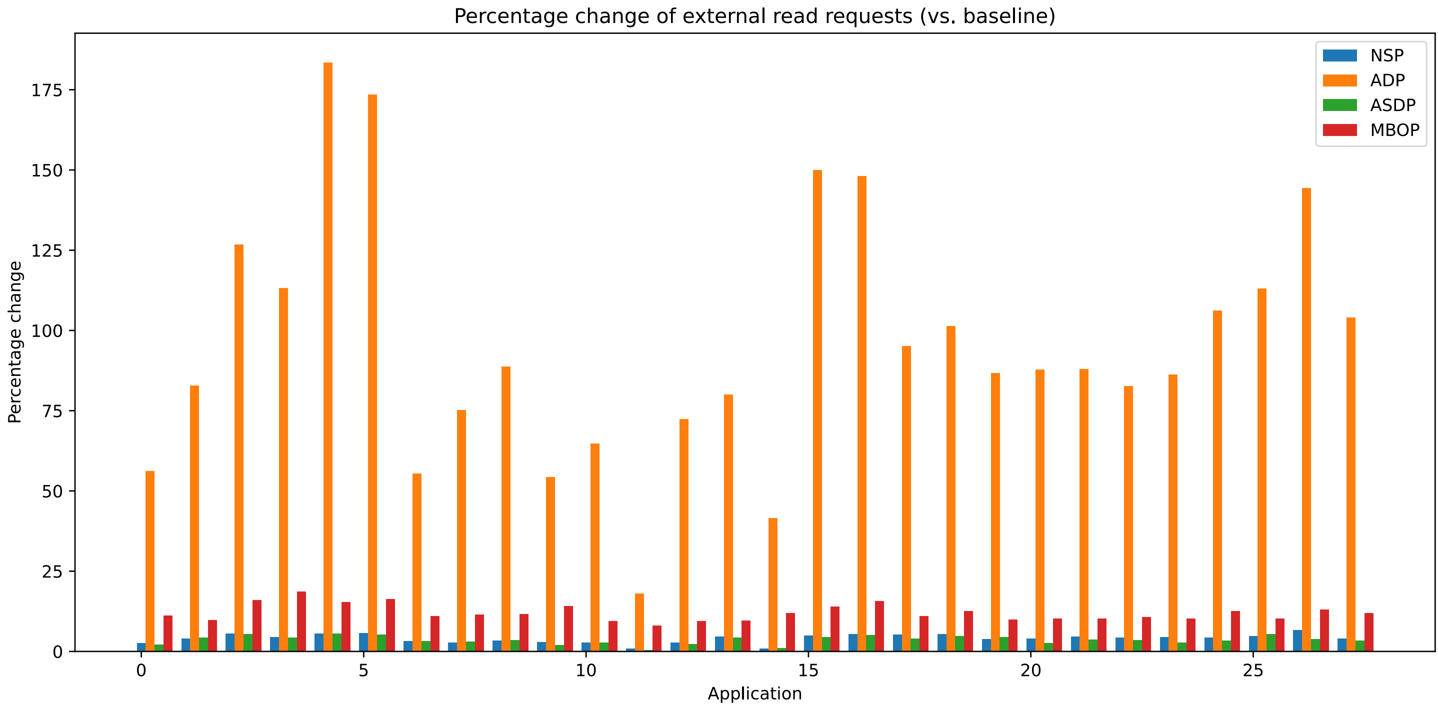

Figure 25: Percentage change of read requests sent from the SLC to the ext. memory.

A.3.4 Power estimation

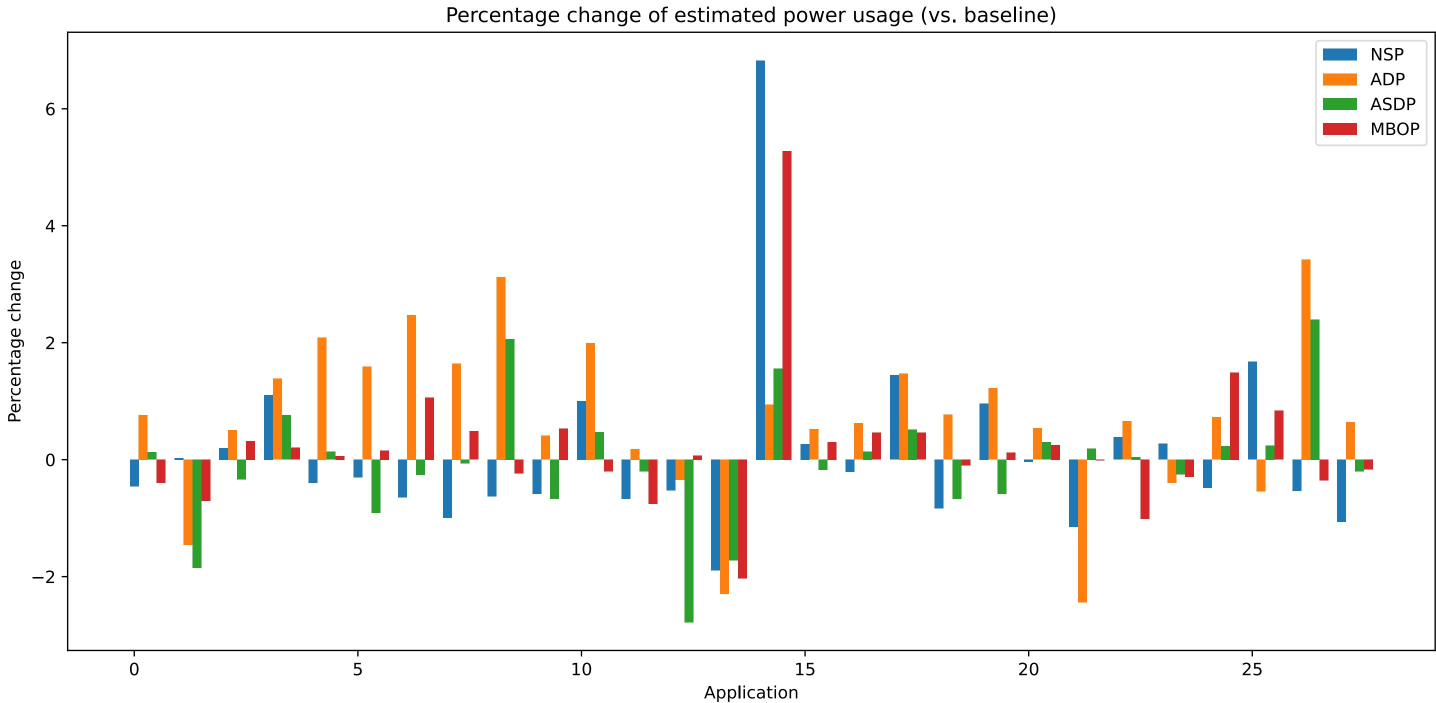

Figure 26: Percentage change of power for all the different prefetchers.

A.3.5 Energy estimation

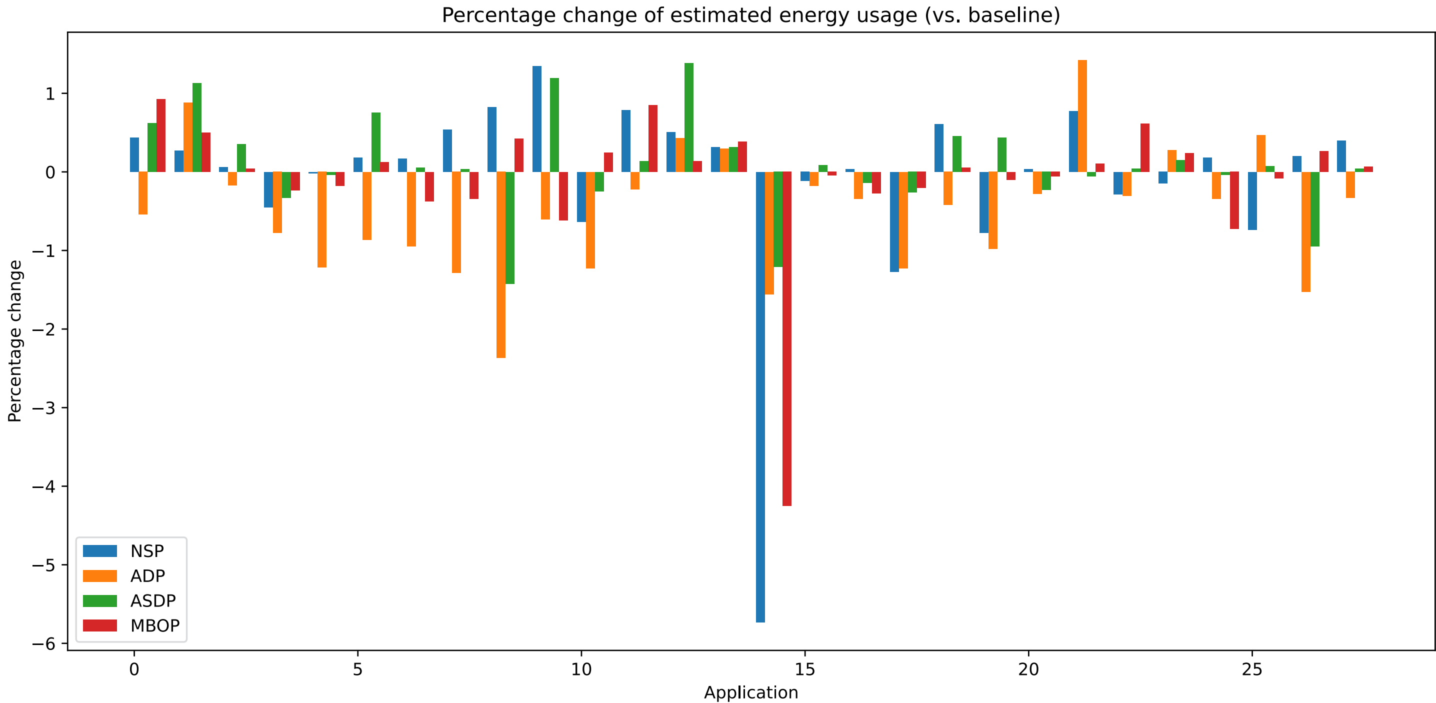

Figure 27: Percentage change of energy against baseline for all the different prefetchers. Negative values are better.

A.3.6 Prefetch accuracy

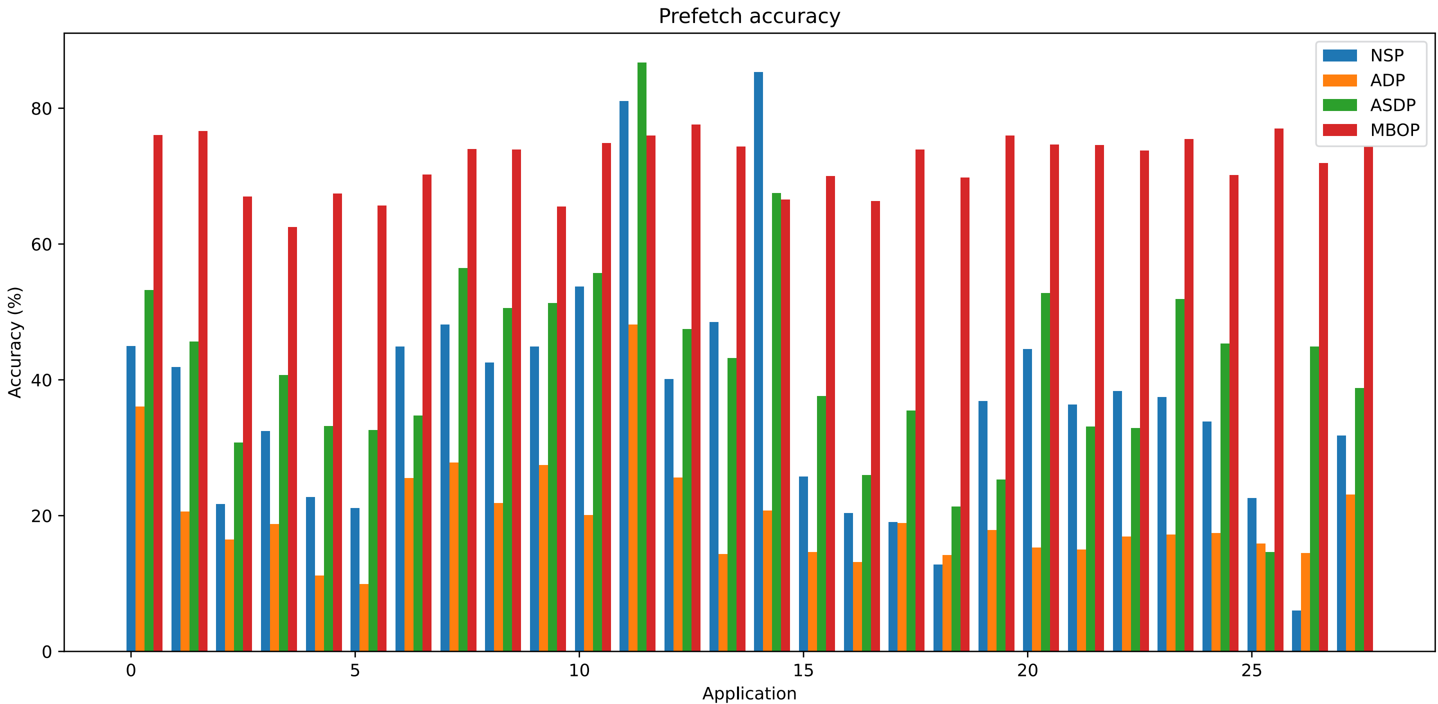

Figure 28: Prefetch accuracy of the implemented prefetchers. Higher values are better.

Printed by Tryckeriet i E-huset, Lund 2024

LUND
UNIVERSITY

Series of Master´s theses
Department of Electrical and Information Technology
LU/LTH-EIT 2024-971
http://www.eit.lth.se