Hardware Data Prefetchers for Shakti I-Class Processor

A Thesis

Submitted by

MANSI CHOUDHARY

in partial fulfilment of the requirements

for the award of the degree

Of

BACHELOR OF TECHNOLOGY

&

MASTER OF TECHNOLOGY

in

ELECTRICAL ENGINEERING

June 2022

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY MADRAS

CHENNAI – 600036

THESIS CERTIFICATE

This is to undertake that the Thesis titled **HARDWARE DATA PREFETCHERS FOR SHAKTI I-CLASS PROCESSOR**, submitted by me to the Indian Institute of Technology Madras, for the award of **Bachelor of Technology & Master of Technology in Electrical Engineering**, is a bona fide record of the research work done by me under the supervision of **Dr. V. Kamakoti**. The contents of this Thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Chennai 600036 **Mansi Choudhary**

Date: June 2022

Dr. V. Kamakoti
Research advisor
Professor
Department of Computer Science and Engineering
IIT Madras

Dr. Nitin Chandrachoodan
Research co-advisor
Professor
Department of Electrical Engineering
IIT Madras

ACKNOWLEDGEMENTS

Firstly, I would like to take this opportunity to express my gratitude to my project guide, Prof. V. Kamakoti for giving me the opportunity to work on this project. His extensive knowledge and experience in this field has been a great source of inspiration. I would also like to acknowledge Prof. Nitin Chandrachoodan for agreeing to co-guide this project and for extending his support to me everytime I needed it. Most importantly, I would like to thank my project mentor, Dr. Nitya Ranganathan for her constant guidance, feedback and immense support throughout the course of my project. It is because of her valuable mentorship that I have successfully completed this project.

I would like to thank my parents and sister for their encouragement and unwavering support throughout my life. Finally, I would also like to thank my friends for helping me and cheering me on at all times.

ABSTRACT

KEYWORDS Hardware prefetcher; I-Class; Core-side; Cache-side; Stride; Linestride; BRAM; Offset; Microbenchmark tests; Benchmark tests

Hardware prefetching is a microarchitectural feature to anticipate addresses that a processor may likely reference in future based on past access patterns and speculatively fetch data or instructions from slow lower-level memories into faster upper-level memories before the core requests them. Examples of this could be fetching from main memory into last-level cache, fetching from unified L2 Cache into instruction cache etc. The objective of this optimisation technique is to reduce the average memory access time (AMAT). From the "memory wall" point of view, Hardware Prefetching is a latency hiding technique that hides the off-chip memory access latency.

The aim of this project is to modify and optimise the existing core-side L1 data cache prefetchers and also design new core-side and cache-side L1 data cache prefetchers, for the out-of-order RISC-V processor, the Shakti I-Class core, in order to achieve speedup and higher cache hits. The cache-side prefetchers could be adapted to work as the L2 data cache and instruction cache prefetchers. The implementation of the prefetchers has been done in the Bluespec System Verilog (BSV) language.

The prefetchers are tested with microbenchmarks (assembly, C) and a subset of SPEC 2017 benchmarks. The performance of the processor with and without prefetchers is analysed and the comparison of the different prefetchers in different configurations is highlighted. Our results show that carefully selecting and tuning prefetchers can show good performance improvement for L1 data prefetching. The cache-side offset prefetchers may work better with the L2 cache.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

AGU	Address Generation Unit.
AMAT	Average Memory Access Time.
BRAM	Block Random Access Memory.
BSV	Bluespec SystemVerilog.
DTLB	Data Translation Lookaside Buffer.
FESVR	Front-end Server.
I-Cache	Instruction Cache.
L1 D-Cache	Level 1 Data Cache.
L2 Cache	Level 2 Cache.
LSU	Load Store Unit.
MSHR	Miss Status Holding Register.
PC	Program Counter.
PK	Proxy Kernel.
PRQ	Prefetch Request Queue.
RR	Recent Requests.
SAXPY	Single-Precision A·X Plus Y.
SHR	Stride History Register.
SPT	Stride Prefetch Table.

CHAPTER 1

INTRODUCTION

The gap between processor and memory performance has been increasing over the years. The penalty of each memory access ranges from a few cycles to hundreds of processor cycles, and thus, memory access becomes a crucial performance bottleneck for the system. Prefetching is one of the techniques to break this “memory wall”. In this project, we have designed, implemented, optimised and tested various hardware prefetchers for the Shakti I-Class core. Bluespec SystemVerilog was used to implement the prefetchers. The functional testing and performance analysis for the prefetchers was done using BSV testbenches, directed assembly tests, microbenchmark and benchmark tests with RTL simulation using PK and FESVR. The performance with the Shakti I-Class core of the various prefetchers is analysed using parameters like speedup, number of prefetches, cache hits etc.

1.1 HARDWARE PREFETCHERS

Processors use prefetching as a technique to enhance their execution performance. It involves fetching instructions or data from where they are originally stored (usually main memory) to local memory (like caches or external buffers) before they are actually requested by the processor. From the memory wall point of view, it is a latency hiding technique that hides the off-chip memory access latency. For hardware prefetching, the idea is that specialised hardware observes load/store access patterns and prefetches data based on past access behaviour.[5]

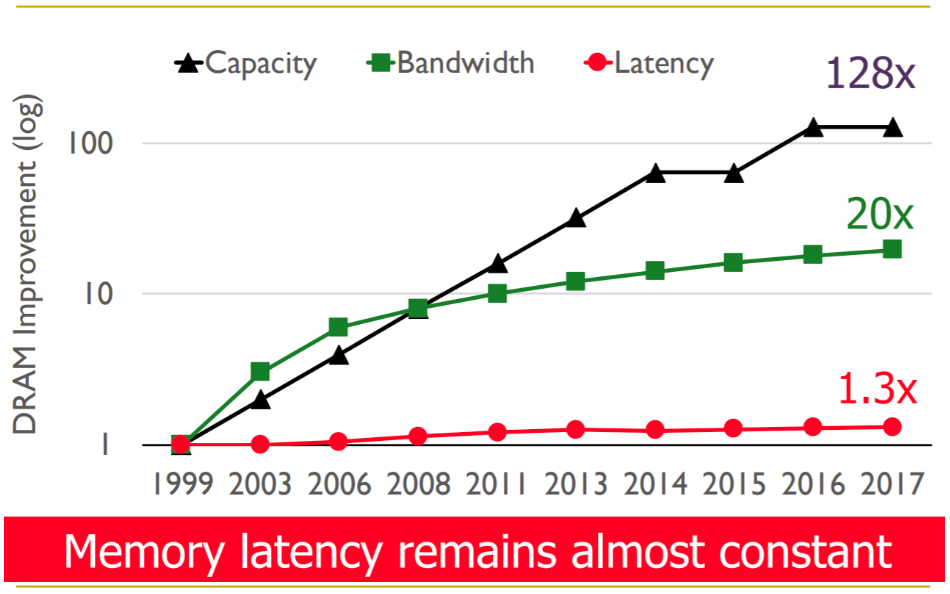

Figure 1.1: The Memory Wall Problem

Source: Mutlu (2020)

1.2 ABOUT SHAKTI I-CLASS PROCESSOR

The Shakti I-Class core[1] for which prefetchers have been implemented in this project, is a super-scalar multi-wide out-of-order processor aimed at the compute, mobile, storage and networking segments. It has potential applications in general-purpose computing and high-end embedded markets with a target operating frequency range of 1.5-2.5 GHz.

The I-Class core can fetch/ dispatch/ commit 4 instructions per cycle. It can issue 7 instructions- up to 5 integer and 2 floating-point instructions in a cycle. It implements RV64IMAFDC: multiplication and division, atomic, single and double-precision floating-point, compressed instructions extensions of the RISC-V 64-bit base integer instruction set. It is equipped with performance-oriented features like aggressive branch prediction, deep pipeline stages, register renaming (which remove false dependencies along with checkpointing), a reorder buffer that stores instruction metadata for all instructions in flight, operand bypass, a memory dependence predictor and non-blocking caches (both instruction and data caches) supporting multiple outstanding misses with Miss

Status Holding Registers (MSHRs). The I-Class core is under active development. It is implemented in the BSV language.

1.3 WHY BLUESPEC VERILOG?

All the prefetchers in this project have been implemented in Bluespec SystemVerilog (BSV) language[4]. BSV is a high-level functional hardware description programming language to handle chip design which comes with a SystemVerilog front-end. Concurrency can be realised reliably in BSV since it is a rule-based language where hardware is described as object-oriented modules. With all its features, BSV can greatly raise a hardware designer's productivity and is hence preferred for this project. The BSC (Bluespec Compiler) is used to generate Verilog from the BSV code. After the Verilog code is generated, it can be taken through the usual FPGA (Field Programmable Gate Array) or ASIC (Application Specific Integrated Chip) development flows.

1.4 TESTING METHODOLOGY

BSV testbenches were used to debug and test the functional correctness of the prefetchers at the module-level. After integrating the prefetchers with the processor, directed assembly tests, microbenchmarks, and benchmarks were used to debug and analyse the performance. All testing was done with simulation on Verilator[3].

1.5 THESIS CONTRIBUTIONS

This project aims to modify and optimise the existing core-side L1 D-Cache prefetchers (Stride, Linestride, Stride BRAM), implement new core-side (Linestride BRAM, Complex Stride BRAM) and L1 data cache-side prefetchers (Nextline Offset, Best Offset) for the out-of-order RISC-V processor, the Shakti I-Class core. Thorough testing and performance analysis has been done for the modified as well as new prefetchers using testbenches, assembly, microbenchmark and benchmark tests.

1.6 ORGANISATION OF THE THESIS

The outline of the rest of the thesis is as follows. Chapter 2 dives deeper into the motivation behind hardware prefetching, explains the design and key terms and parameters of a basic hardware prefetcher and discusses the types of prefetchers we have implemented. Chapter 3 is an elaborate description of the design and functionality of the existing and newly implemented core side prefetchers and their enhancements or optimisations. Chapter 4 describes the design, functionality and optimisation of the cache side prefetchers which we have implemented. Next, in Chapter 5, the various tests written to check the functional correctness and evaluate the performance of each prefetcher are discussed. The experimental setup and the statistics and results from the performance analysis of the prefetchers are also shown. Finally, towards the end, Chapter 6 concludes the thesis with the future scope of this project and further work that can be done.

CHAPTER 2

HARDWARE PREFETCHERS

This chapter discusses the motivation behind hardware prefetching and explains the basic idea of hardware prefetching. It also provides an overview of the major requirements of a hardware prefetcher and some key parameters.

2.1 MOTIVATION

Cache prefetching is an optimization technique to anticipate addresses that a processor may likely reference in the future based on past access patterns and speculatively fetch data or instructions from slow lower-level memories into faster upper-level memories before the core requests them.

Main memory latency is high compared to cache access latency. The average time taken to access data from the memory is referred to as Average Memory Access Time (AMAT). AMAT has three parameters: hit latency, miss rate, and miss penalty. Hit latency (H) is the time to hit in the cache. Miss Rate (MR) is the frequency of cache misses, while Average Miss Penalty (AMP) is the cost of a cache miss in terms of time. AMAT can be defined as follows[7]:

$$AMAT = H + MR \cdot AMP$$

The objective of prefetching is to reduce the AMAT. Prefetching is a latency-hiding technique as prefetching data into faster memories like caches and then accessing it from there is usually many orders of magnitude faster in comparison to directly accessing it from the main memory. It helps to reduce the cache miss rate (MR) or miss penalty (AMP) via parallelism. Hence, timely and accurate prefetching of data can result in an improvement in performance.

2.2 HARDWARE PREFETCHERS

Hardware-based prefetching is typically accomplished by having a dedicated hardware mechanism in the processor that watches the stream of instructions or data being requested by the executing program, predicts the next few elements that the program might need based on this stream and prefetches them into the processor's cache.

Basic idea of Hardware Prefetching:

- Hardware monitors processor accesses
- Memorizes or finds patterns/strides
- Generates prefetch addresses automatically

Tradeoffs in Hardware Prefetching:

- + Can be tuned to system implementation
- + Does not waste instruction execution bandwidth
- - Area/power for the prefetching logic
- - Software can be more efficient when looking for more complex data access patterns

2.3 PREFETCHING METRICS AND REQUIREMENTS

Accuracy

Accuracy is the fraction of total prefetches that were useful. Quantitatively, accuracy is the ratio of the number of correct prefetches to the total number of prefetches, i.e.

$$\text{Prefetch Accuracy} = \frac{\text{Cache Misses eliminated by prefetching}}{(\text{Useless Cache Prefetches}) + (\text{Cache Misses eliminated by prefetching})}$$

Misprediction in prefetching does not affect correctness as the prefetched data at a "mispredicted" address is simply not used. There is no need for state recovery, in contrast to branch misprediction or value misprediction.

However, when the prefetchers become too aggressive it can cause an increase in traffic on the memory interconnection network which can lead to an extra miss penalty for genuine demand requests from the core. Apart from memory bandwidth, useless data also wastes resources like cache or prefetch buffer space and energy consumption. If the speculatively prefetched blocks evict or replace useful demand data or cache lines, it can cause pollution of cache that could result in a net increase in the cache miss rate. In such cases, prefetchers can degrade a processor's performance. So, accurate prediction of addresses to prefetch is important.

Coverage

Coverage is the fraction of total misses that are eliminated because of prefetching. It is the ratio of the number of useful prefetches to the total number of cache misses, i.e.

$$\text{Coverage} = \frac{\text{Cache Misses eliminated by Prefetching}}{\text{Total Cache Misses}},$$

where, Total Cache Misses = Cache misses eliminated by prefetching + Cache misses not eliminated by prefetching

Timeliness

The qualitative definition of timeliness is how early a block is prefetched versus when it is actually referenced. Prefetching too early might cause the prefetched data to be evicted from the storage before it is used. On the other hand, prefetching too late is also futile. the prefetcher can be made more timely by making it more aggressive: try to stay far ahead of the processor's access stream.

Access Pattern

A hardware prefetcher predicts what to prefetch based on past access patterns. These patterns could be spatial or temporal, core-side load PC based or cache-side offset based, and the observed pattern could be of cache hits and misses or of misses only (either in L1, or L2 D-cache or I-cache).

Location

Prefetching can be done in various levels of caches such as memory to L2, memory to L1 and L2 to L1. It can be used for data as well as instruction caches.

2.4 PREFETCH PARAMETERS

Some of the key parameters in the design of prefetchers are:

- **Prefetch Degree:** Number of prefetch requests to issue at a given time.

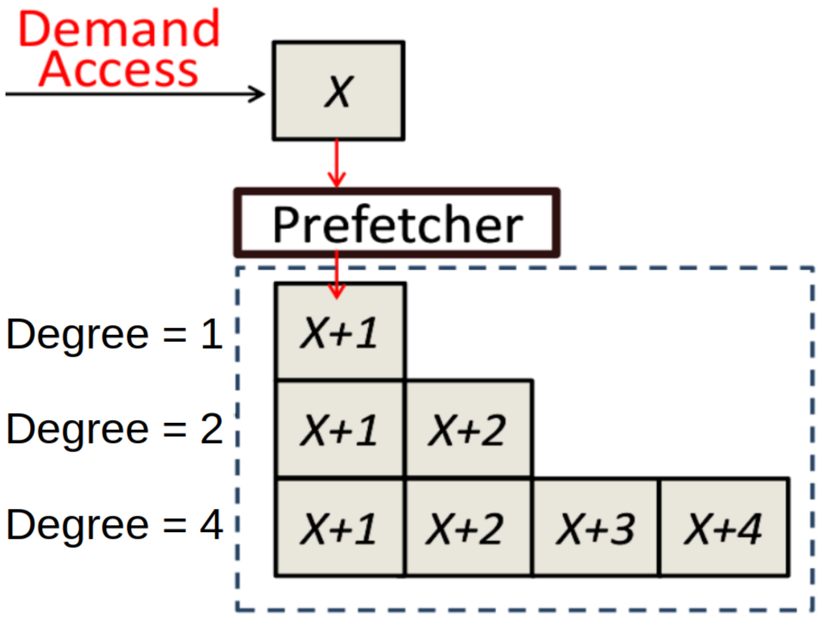

Figure 2.1: Prefetch Degree

- **Prefetch Distance:** Determines how far ahead the estimated prefetch address are from the prefetch requests issued. It is usually preferred to keep this value as a power of 2.

- **Stride (or Line stride) Width:** Number of bits used to capture the stride (or line stride) value. It is chosen based on the expected maximum stride (or line stride) value that can be seen in most situations.

- **Maximum Confidence:** Maximum possible confidence value. The number of bits used to capture the value of confidence in the stride (or line stride) pattern depends on this value.

- **Threshold Confidence:** If the confidence value in the stride (or line stride) pattern

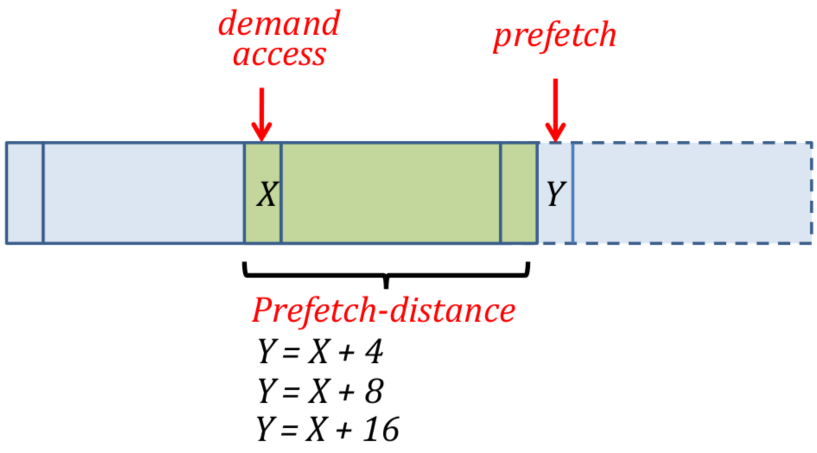

Figure 2.2: Prefetch Distance

seen is greater than or equal to this threshold value, prefetch requests can be generated.

- **Line Offset:** $log_2(cache\_line\_size)$. It is equal to the log of the number of bytes in a cache line. It is used to calculate line address in line stride prefetcher.

- **N (as in N-way BRAM):** The number of ways in the set-associative BRAM.

- **Prefetch throttling:** For all runs, prefetch throttling has been enabled. This ensures that 2 MSHRs are always reserved for demand requests. This drops some prefetch requests but helps avoid the situation of cache signalling busy due to MSHRs getting full with the prefetch requests.

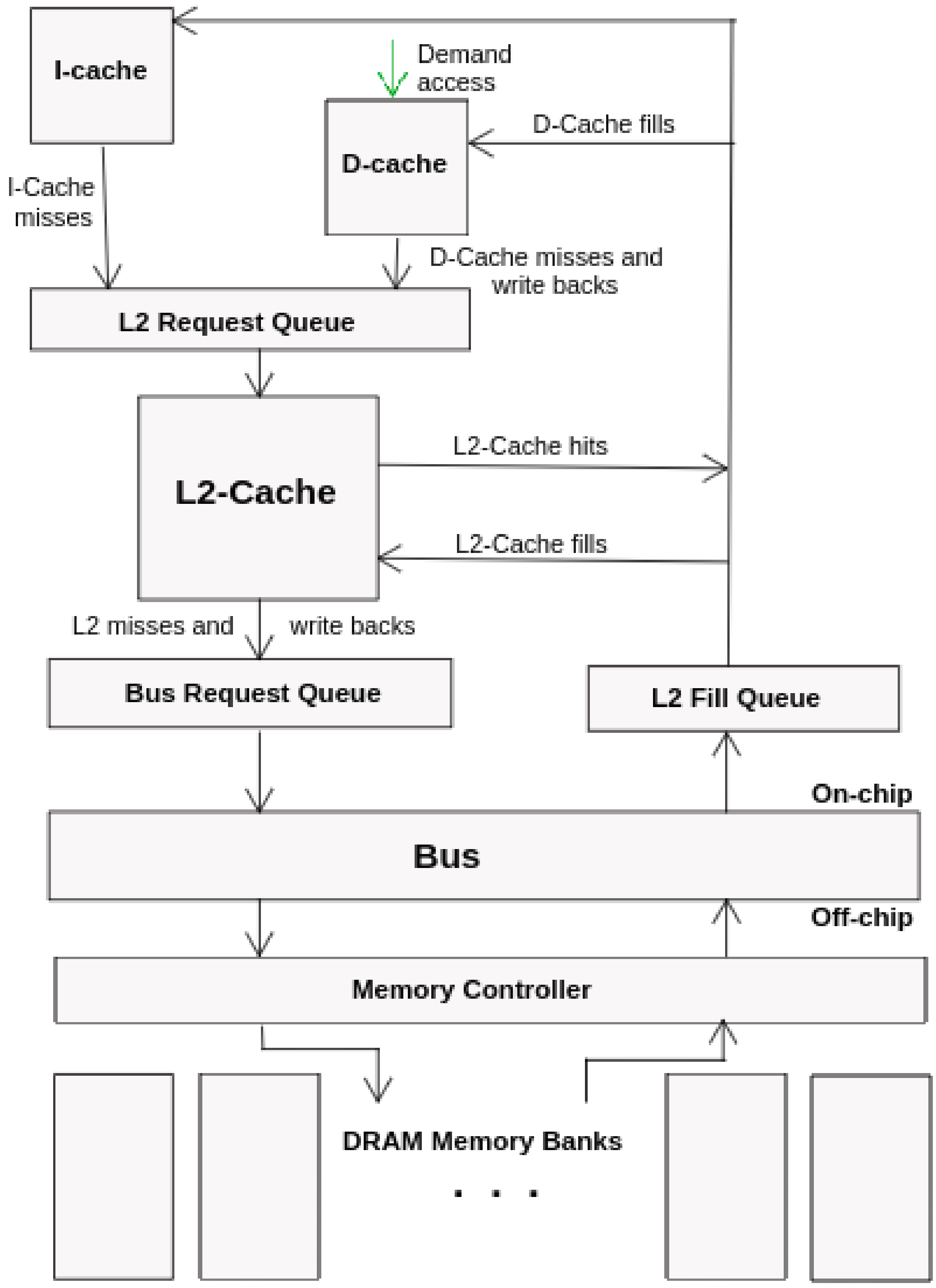

Figure 2.3: Block diagram of a memory system without prefetcher

Source: Mutlu (2020)

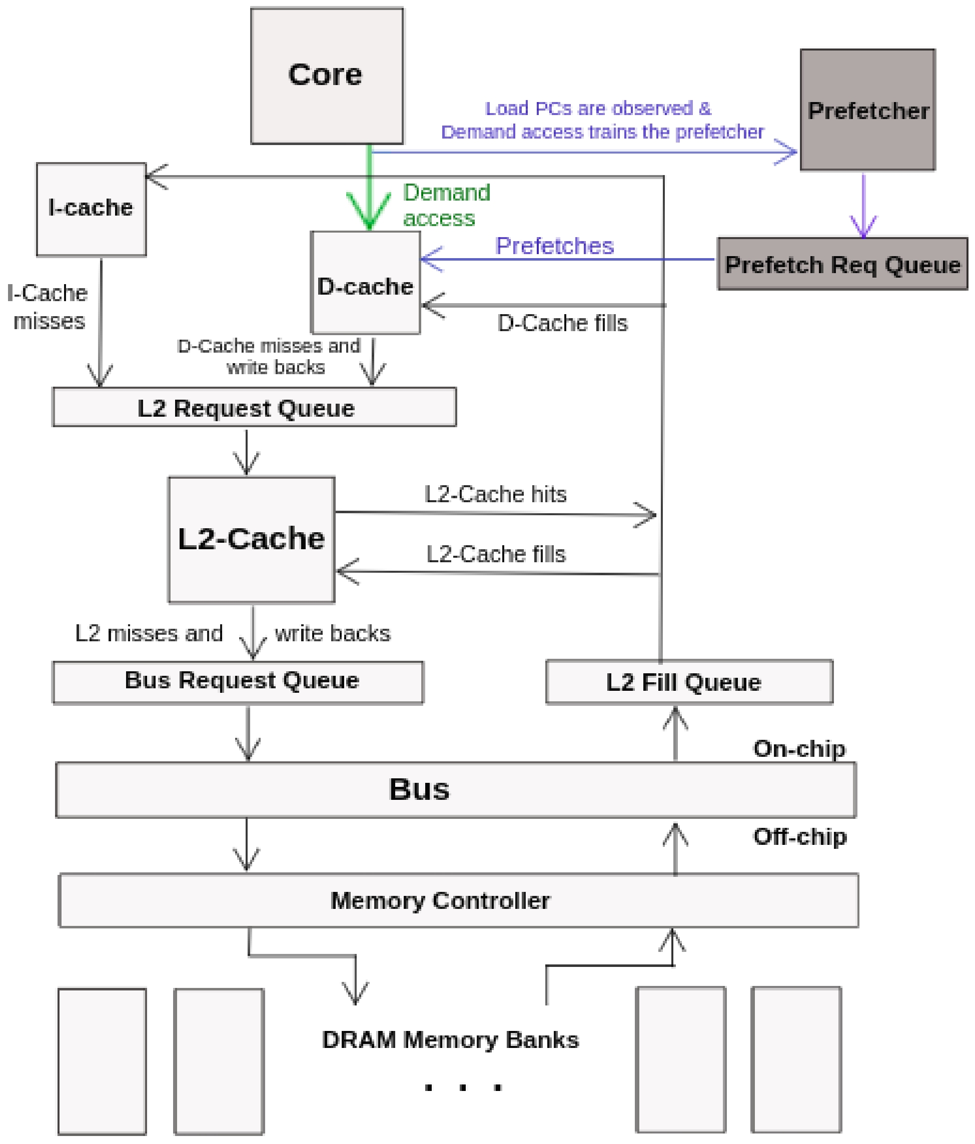

Figure 2.4: Block diagram of a memory system with a core side prefetcher

Source: Mutlu (2020)

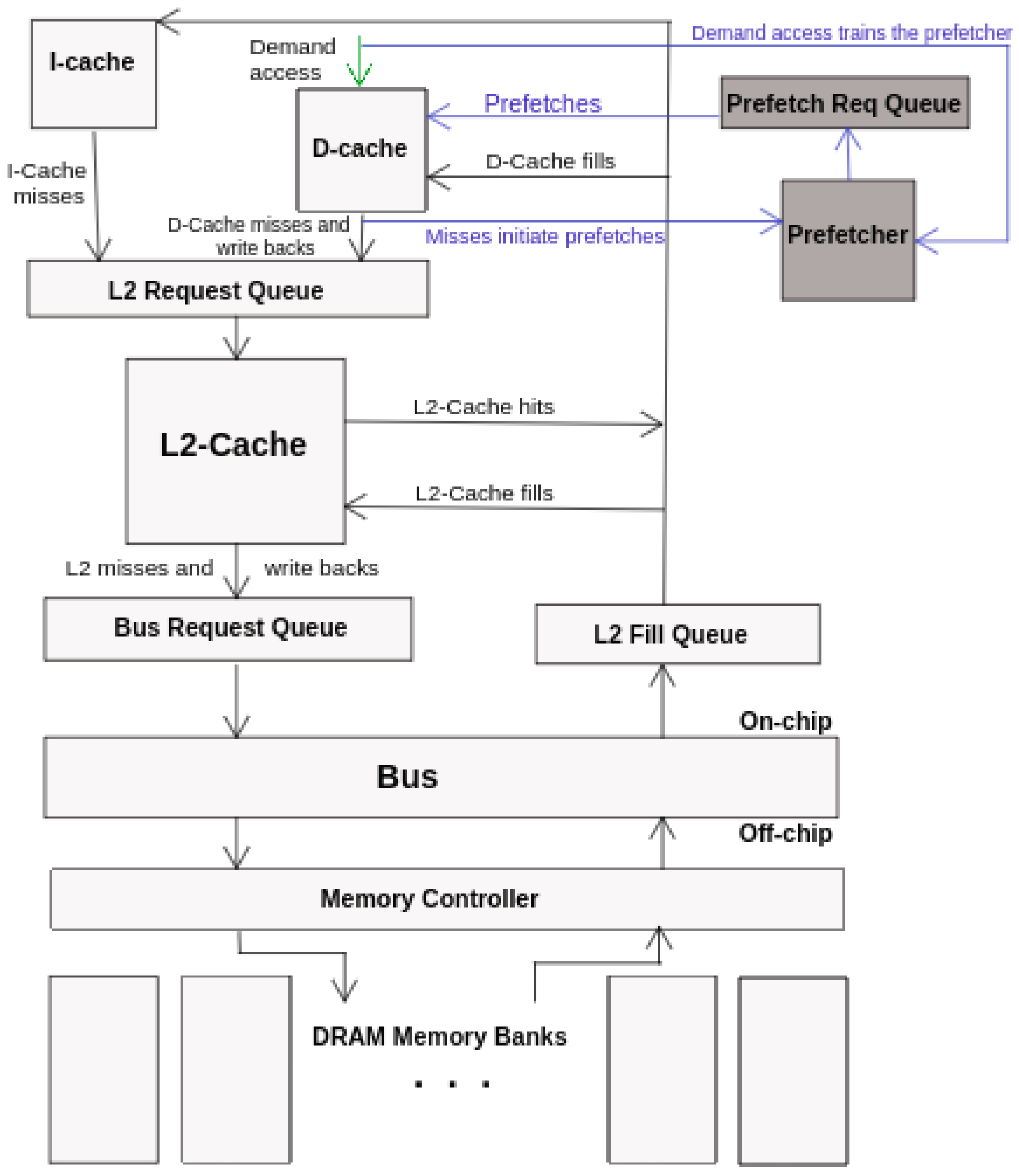

Figure 2.5: Block diagram of a memory system with a L1 data cache-side prefetcher

Source: Mutlu (2020)

CHAPTER 3

CORE-SIDE PREFETCHERS

In this chapter, we look at the design of various existing and newly implemented core-side prefetchers.

Core-side prefetching refers to prefetching that is initiated by the core. The load PCs and demand accesses observed by the core are used to train the prefetcher (see figure 2.4). The prefetch request is sent to a cache, in our case the Level 1 Data Cache. In this project, we have optimised, tested and analysed some of the existing core-side prefetchers as well as designed some new ones for the Shakti I-Class processor.

NEXTLINE PREFETCHERS

The nextline prefetcher is a very basic load address-based core-side prefetcher which simply prefetches the next cache line on a demand access. This existing prefetcher is studied but not tested in our project. This prefetcher has been discussed in Somisetty (2021). The nextline/previous-line prefetcher that was implemented for the I-Class core works as follows for prefetch degree equal to ‘n’:

Case (i): If the current demand access address is greater than the previous address, the prefetch requests will be issued for the next ‘n’ cache lines.

Case (ii): If the current address is lesser than the previous address, prefetch requests will be issued for the previous ‘n’ lines.

3.1 STRIDE PREFETCHERS

Typically in programs with simple loops, addresses which are at a constant distance or offset are referenced continuously in consecutive iterations. For example, when accessing the elements of an array: if the addresses X, X+4, X+8, X+12 ... are referenced, the difference between the addresses is a constant, 4. This difference is called stride. The address pattern can be predicted and the next address to be fetched may be at a distance of 4 from the previous address. Hence, we can prefetch the predicted address. This is the idea behind a PC-based stride prefetcher. The distance between the memory addresses referenced by a load instruction (i.e. stride of the load) as well as the last address referenced by the load is observed and recorded. If the same stride is detected multiple times for that load instruction, the confidence in that stride pattern increases and when the confidence is above a certain chosen threshold value, the address [last address + stride] is prefetched (accordingly changed if prefetch degree or distance are not equal to 1).[6]

3.1.1 Stride Prefetcher with register array SPT

The first stride prefetcher implemented for the I-Class core uses a register array for the SPT (Stride Prefetcher Table). The prefetcher observes the PC of the load instruction and load addresses. The SPT keeps track of observed addresses and their stride patterns. It is a PC-indexed direct-mapped table. The lower-order bits of PC become the index used to locate a particular SPT entry and the higher-order bits of the PC are used for the tag. Each entry of the SPT has the following components: tag, previous address, stride and confidence counter. Each component/column of the SPT is represented by a vector with Num_Entries number of entries. All the counters and widths are parameterisable (Num_Entries, stride width, maximum confidence, confidence threshold, prefetch degree and prefetch distance). Index width is $log_2(Num\_Entries)$ and confidence counter width is $log_2(Maximum\_confidence)$. Stride is calculated in bytes.

(Current stride) = (Current address) - (Previous address)

Index	Tag	Previous address	Stride	Confidence counter
0	...	...	...	...
1	..	..	..	..
.	.	.	.	.
.	.	.	.	.
Num_Entries-1	..	..	..	..
Num_Entries	...	...	...	...

Table 3.1: Stride Prefetch Table

There are two phases in the stride prefetcher: the training phase and the generation phase.

The training segment is pipelined into two stages, one for reading the table and second for updating the SPT. When the load address is received by the AGU, the prefetcher training is triggered. In the second stage, the SPT is updated.

In the training phase, whenever a load instruction is seen, the SPT is read and one of the following operations takes place:

1. Load PC hits in the SPT i.e. tag of the current load instruction matches with the tag present in the SPT entry at that index. If current stride matches stride previously-stored in SPT entry, increment confidence counter in SPT if (confidence counter) < (maximum confidence) and update previous address in SPT.

2. Load PC hits in the SPT. If current stride does not match stride previously stored in SPT entry and (confidence counter) > 0, then decrement confidence counter.

3. Load PC hits in the SPT. If current stride does not match stride previously stored in SPT entry and (confidence counter) = 0, then update the stride field in SPT with the current stride and update previous address.

4. Load PC misses in the SPT (tag mismatch). Check whether the confidence counter = 0. If it is, it can be replaced by the information from the current load (tag and previous address) and the stride field can be reset to the default value. If not, decrement the confidence counter.

In the generation phase, if (confidence counter) >= (threshold confidence), prefetch degree number of prefetch addresses are sent to the Prefetch Request Queue (PRQ).

The prefetch addresses generated depend on the prefetch distance value as:

(Prefetch address) = (Current address) + (Stride)*(Prefetch distance)

A prefetch request will be generated only if the prefetch address falls in the same page as that of the current load address. Also, a prefetch request will be issued only if the prefetch address doesn't fall in the same cache line as the demand address line or the previous prefetch request generated.

Default values of parameters used:

- Maximum value of confidence = 15
- Confidence_threshold = 7
- Prefetch_degree = 1
- Prefetch_distance = 2
- Number of entries in SPT (Num_Entries) = 256
- Stride_width = 12

3.1.2 Stride Prefetcher with BRAM SPT

For this version of the prefetchers, the SPT is designed as a PC-indexed n-way set-associative mapped dual-ported Block Random Access Memory (BRAM). The implementation uses the BRAMCore package of BSV. In BRAM, the read operation has one cycle of extra latency (i.e. data is available one cycle after the read request is initiated).

The components of the BRAM SPT entry are the same as the register array SPT. Each entry in the BRAM contains a tag, a previous address (last seen address), a stride, and a confidence counter. The Num_Entries in this case is equal to the number of entries in a single set of the BRAM. The index width is still equal to $log_2(Num\_Entries)$ and confidence counter width is same as the register array SPT, $log_2(Maximum\_confidence)$. Since the BRAM SPT is n-way set-associative, the total number of entries in the SPT are now n*(Num_Entries). We will be referring to this prefetcher as the Stride BRAM prefetcher henceforth.

Idx	Tag0	Prv_addr0	Stride0	Conf0	Tag1	Prv_addr1	Stride1	Conf1
0	...	...	...	...	...	...	...	...
1	..	..	..	..	..	..	..	..
.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.
N-1	..	..	..	..	..	..	..	..
N	...	...	...	...	...	...	...	...

Table 3.2: A BRAM based 2-way set-associative SPT

There are two phases in the stride BRAM prefetcher, the training phase and the generation phase, same as in the stride prefetcher.

The training segment in the stride BRAM prefetcher is pipelined into three stages instead of two.

In the first pipeline stage, the read index is latched onto the read port of the BRAM SPT when the PC and address of a load generated in the AGU are received.

In the next cycle, the BRAM SPT entry data is available to be read and all "n" entries of the index are stored in registers. For the case where the same load PC trains in back-to-back cycles, bypassing is used in the second stage.

Bypassing : Bypassing is an optimization that is introduced to deal with the case where the same load PC comes in to train the prefetcher in back-to-back cycles. In this case, there is a PC that is writing the updated SPT entry to BRAM in stage 3 and in the same cycle, the same PC from another load instance is trying to train again using the old values read from SPT in stage 2. This problem is solved using bypassing. In stage 2, it is checked if the stage 3 valid register is true in that cycle and if the PCs match, the updated values computed by the previous load are used and if not, the SPT entry values read from the BRAM are used.

The training of the prefetcher is then performed in the same cycle and one of the following operations takes place:

1. Load PC hits in the SPT i.e. the tag of the current load instruction matches with one of the tags present in the SPT entries at that index. If current stride matches stride previously stored in the same SPT set entry, increment confidence counter in set entry if (confidence counter) < (maximum confidence) and update previous address in set entry.

2. Load PC hits in the SPT (same as PC hit in previous case), if current stride does not match stride previously stored in SPT set entry and (confidence counter) > 0, then decrement confidence counter.

3. Load PC hits in the SPT (same as PC hit in previous case), if current stride does not match stride previously stored in SPT set entry and (confidence counter) = 0, then update the stride field in SPT set entry with the current stride and update previous address.

4. Load PC misses in the SPT (tag mismatch), i.e, none of the tags present in the entries of the current index match the tag of the load PC. Here, an entry replacement policy is necessary. Compare all the confidences in the entry (in all the sets at the given index level) and select the entry with the least confidence as the target. If the confidence of this entry is 0, replace it by the information from the current load (tag and previous address) and the stride field can be reset to a default value (= 1). If not, decrement the confidence counter of the entry.

The values from stage 2 are written in pipeline registers. In the third pipeline stage, if there is a need to update the table entry, the new entry is written in the BRAM SPT using the write port. It is also in this stage that the computation and generation of prefetch requests happen if the generate conditions are satisfied. The generation phase is the same as that in the stride prefetcher. The prefetch addresses generated depend on prefetch distance value as:

$$(\text{Prefetch address}) = (\text{Current address}) + (\text{Stride})*(\text{Prefetch distance})$$

A prefetch request will be generated only if the prefetch address falls in the same page as that of the current load address. Also, a prefetch request will be issued only if the prefetch address doesn't fall in the same cache line as the demand address line or the previous prefetch request generated.

Default values of parameters used:

- Maximum value of confidence = 7
- Confidence_threshold = 3
- Prefetch_degree = 1
- Prefetch_distance = 2
- Number of entries in SPT (Num_Entries) = 64
- Number of sets in BRAM (n) = 4
- Stride_width = 12

The advantage of a BRAM-based implementation of the SPT is that it uses Block RAMs instead of on-chip registers. On-chip registers are limited and Block RAMs are a more compact form of memory on a chip, making BRAMs better suited for bigger data storage. Also, the BRAMs are dual-ported which allow us to perform 2 operations (1R,1W in our case) on it in the same cycle. The n-way set associativity of the SPT ensures that same indexed addresses are not as frequently replaced as in a direct-mapped table.

3.2 LINESTRIDE PREFETCHERS

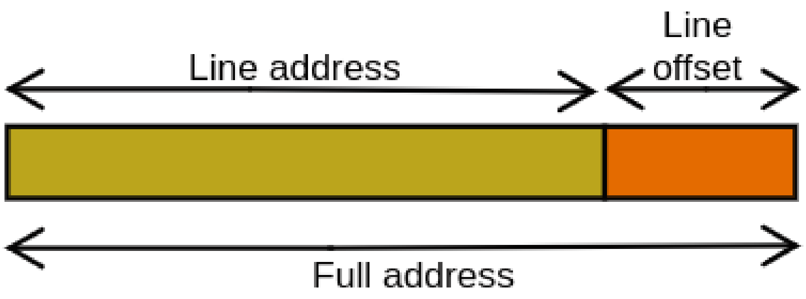

Figure 3.1: Extraction of line address from the full address.

The linestride prefetcher is a modification of the stride prefetcher. Here, the stride is captured in multiples of cache lines instead of bytes. Though this may comparatively be less accurate, it saves a lot of space in the SPT. Also, instead of storing the entire previous address, only the previous line address is stored i.e. only the higher-order bits of the address are stored. In the stride prefetcher, it takes more space to store the stride value and the complete previous address. Here, the space for storing stride and the previous address reduces. If a cache line size is k (64 for this processor) bytes, then the 'line_offset' parameter is $log_2 k$ (=6 here) and the line address then becomes (load address » line offset) i.e. ignoring the 'line_offset' number of lower-order bits. Now, the stride is computed as the difference between the previous line address and the current line address, and this stride is called as linestride henceforth in the thesis.

(Current linestride) = (Current line address) - (Previous line address)

(Prefetch line address) = (Current line address) + (Linestride)*(Prefetch distance)

3.2.1 Linestride Prefetcher with register array SPT

The SPT is implemented as a PC-indexed direct-mapped table of registers. Each entry contains a tag, a previous line address (last seen line address), a stride, and a confidence counter. The working of the register array-based line stride prefetcher is the same as that of the stride prefetcher except that the line stride replaces the stride in all computations and the line addresses replace the full addresses.

Index	Tag	Prev line addr	Linestride	Confidence cntr
0	...	...	...	...
1	..	..	..	..
.	.	.	.	.
.	.	.	.	.
Num_Entries-1	..	..	..	..
Num_Entries	...	...	...	...

Table 3.3: Stride Prefetch Table for Linestride Prefetcher

Default values of parameters used:

- Maximum value of confidence = 7

- Confidence_threshold = 3
- Prefetch_degree = 1
- Prefetch_distance = 2
- Number of entries in SPT (Num_entries) = 256
- Stride_width = 5
- Line_offset = 6

3.2.2 Linestride Prefetcher with BRAM SPT

The SPT is implemented as a PC-indexed 4-way set-associative mapped dual-ported BRAM. Each entry contains a tag, a previous line address (last seen line address), a stride, and a confidence counter. The working of BRAM-based linestride prefetcher is the same as that of the BRAM-based stride prefetcher except that the linestride replaces the stride in all computations and the line addresses replace the full addresses. This prefetcher also employs bypassing, has the 4 stages in the training stage and same prefetch generate conditions as the other stride prefetchers.

Idx	Tag0	Prv_line0	Linestride0	Conf0	Tag1	Prv_line1	Linestride1	Conf1
0	...	...	...	...	...	...	...	...
1	..	..	..	..	..	..	..	..
.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.
N-1	..	..	..	..	..	..	..	..
N	...	...	...	...	...	...	...	...

Table 3.4: A BRAM based 2-way set-associative SPT for Linestride Prefetcher

Default values of parameters used:

- Maximum value of confidence = 7
- Confidence_threshold = 3
- Prefetch_degree = 1

- Prefetch_distance = 2
- Number of entries in SPT (Num_entries) = 64
- Number of sets in BRAM (n) = 4
- Stride_width = 5
- Line_offset = 6

3.3 COMPLEX STRIDE PREFETCHER

Complex Stride Prefetcher is a PC-based variable stride prefetcher. For this prefetcher, we implement a PC-indexed stride prefetch table in a dual-ported BRAM (alternatively in an array of registers), that can track up to N (=2, default) consecutive strides. The strides are tracked at the byte-level in a stride history register (SHR) which is a component of every entry in the table. Each stride is captured using M (=10, default) bits. There is a status bit added to the SHR to track the last tracked stride. Hence, we use a 21-bit SHR to capture the last two consecutive strides.

The indexing and learning are similar to the regular stride prefetcher. The new stride that is computed on a load resolution is shifted into the SHR and the last seen address in the table entry is also updated as in the original stride algorithm. The stride pattern that is observed is *a,b,a,b,a,b....*

Index	Tag	Prev addr	Stride1	Stride2	Status	Conf cntr
0	...	...	...	...	...	...
1	..	..	..	..	..	..
.	.	.	.	.	.	.
.	.	.	.	.	.	.
Num_Entries-1	..	..	..	..	..	..
Num_Entries	...	...	...	...	...	...

Table 3.5: BRAM based SPT for Complex Stride Prefetcher with Stride History

When the confidence is high and a particular stride is seen, the next address to prefetch is computed based on the distance and the sum of the two strides.

(Prefetch address) = (Current address) + (Stride1 + Stride2)*(Prefetch distance)

Default values of parameters used:

- Maximum value of confidence = 7
- Confidence_threshold = 3
- Prefetch_degree = 1
- Prefetch_distance = 2
- Number of entries in SPT (Num_Entries) = 64
- Number of sets in BRAM (n) = 4
- Stride_width = 10
- SHR_width = 21

CHAPTER 4

CACHE SIDE PREFETCHERS

In this chapter, we look at the design of the newly implemented cache-side offset prefetchers for the L1 D-Cache.

For cache-side prefetchers the training is based on cache misses at a cache level. In our case, the demand misses and fills from the Level 1 Data Cache train the prefetcher. The misses at this level initiate prefetching from the Level 2 Cache (see figure 2.5) or in the current version of I-Class, the main memory. In this project, we have implemented two offset prefetchers in the L1 data cache, the Nextline Offset prefetcher and the Best Offset prefetcher. These prefetchers can be further used at the L2 level as well as for the instruction cache. At the L2 level, the prefetchers can be more aggressive for better coverage and timeliness without worrying about cache pollution, as the L2 cache sizes are significantly higher.

Our prefetchers are adapted from the prefetcher proposed in the paper Michaud (2016). In the paper, the prefetcher was designed for the L2 cache, but since the present I-Class core does not have an L2 cache, we are implementing it on the L1 cache. The first prefetcher, the Nextline Offset prefetcher, is a very basic version with no offset training. The RR table (see section 4.2) is filled only with the L1 access lines and not the fill lines. The next prefetcher, the Best Offset prefetcher, is a simplified version of the prefetcher implemented in the paper, with access lines not filling the RR table (and hence no delay queue). Also, the RR table is not skewed-associative in our case.

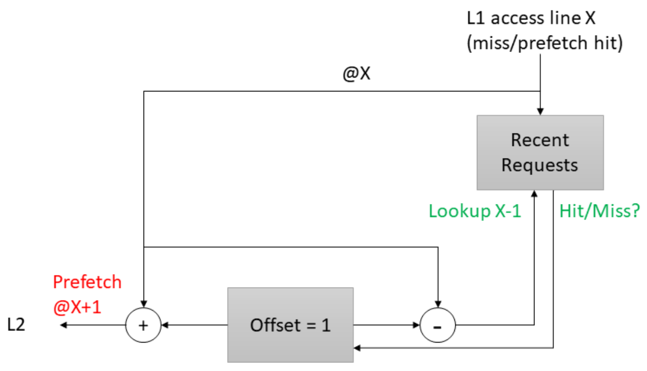

Figure 4.1: Block diagram of Next Line Offset Prefetcher implemented in the project

4.1 NEXTLINE OFFSET PREFETCHERS

The Nextline Offset prefetcher prefetches one line (the next line) into the L1 cache on every cache miss or hit on a prefetched line. The prefetch line address is generated by adding one to the demand access address line. It is a simplified version of the Best Offset (BO) prefetcher discussed in section 4.2. We have chosen a single fixed offset, 1, because it is a common offset value for many applications.

In the training phase, the demand access lines (misses or prefetched hits) from L1 are stored in the RR (recent requests) table. If there is a hit in the RR table, the confidence of the prefetcher is incremented. Whenever there is a new demand access line (X), the prefetch address line X+1 is sent out to the L2 cache, if the confidence of the prefetcher is above the threshold.

4.2 BEST OFFSET PREFETCHERS

The Best Offset (BO) prefetcher prefetches one line into the L1 cache on every cache miss or hit on a prefetched line. The prefetch line address is generated by adding an offset to the demand access address. The BO prefetcher tries to automatically find an offset value that yields timely prefetches with the highest possible coverage and accuracy. It evaluates an offset value by maintaining a table of recent requests addresses (RR table) and by searching these addresses to determine whether the line currently requested would have been prefetched in time with that offset.

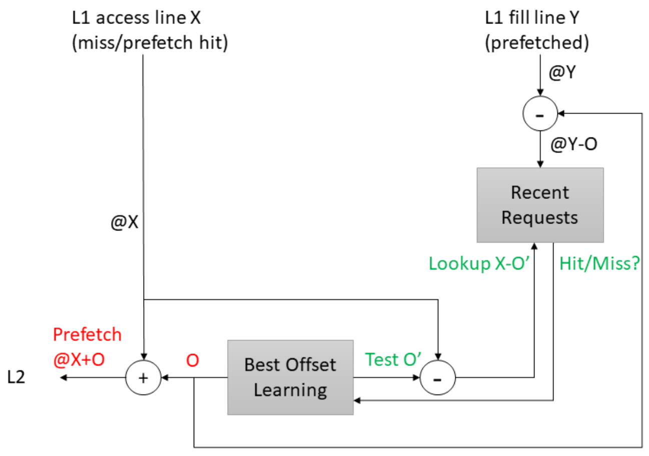

Figure 4.2: Block diagram of Best Offset Prefetcher implemented in the project

Source: Michaud (2016)

We store in a Recent Requests (RR) table the base address of each recent prefetch, that is, the address of the demand access that generated the prefetch. Whenever a prefetched line Y is entered into the L1 cache, we write the address $Y - O$ into the RR table, where O is the prefetch offset, i.e., the offset currently used for prefetching. We consider a list of possible offsets. This list is fixed at design time. In our case, we have chosen a list of 16 five-bit offsets. We associate a score with every offset in the list. On every L2 miss or prefetched hit for a line address X, we test the n^{th} offset O_n from the list by searching if the line address $X - O_n$ is in the RR table. If address $X - O_n$ is in the RR table, this means that line X would likely have been prefetched successfully with offset O_n, and the score for offset O_n is incremented. The next L2 access tests offset O_{n+1} and so on. A round corresponds to a number of accesses equal to the offset list size. Immediately after a round is finished (i.e., all the offsets have been tested once), n is reset and a new round starts. The prefetcher counts the number of rounds. In our project, we constantly train the prefetcher. The offsets are continuously evaluated, with the prefetch offset tracking the program behaviour.

Default values of parameters used:

- Confidence threshold = 4
- RR Width = 10
- RR Number of Entries = 64
- RR Ways = 2
- Offset Width = 5
- Offset Number Entries = 16
- Offset Index Width = 4

CHAPTER 5

TESTING AND PERFORMANCE ANALYSIS

This chapter contains the testing configurations, methodologies and results, as well as an analysis of the results and insights. There are some tables and graphs to better visualize the results.

5.1 I-CLASS CONFIGURATION

5.1.1 I-Class Default Core Configuration

The Shakti I-Class core is highly parameterizable but the default configuration is being used for the experiments for performance analysis of the prefetchers. The I-Class core is continuously under development, hence the different tags for different tests throughout the timeline of the project. The values of some of the important parameters of the default configuration of the core are as follows:

Parameter	Value
Fetch stage width (instructions per cycle)	4
Decode stage width (instructions per cycle)	4
Issue stage width (instructions per cycle)	7
Commit stage width (instructions per cycle)	4
Load_buffer_size in LSU	16
Store_buffer_size in LSU	16
Number of integer registers in physical register file	128
Number of floating point registers in physical register file	64
Number of entries in Re-order buffer	120

Table 5.1: Values of parameters in the default configuration of I-Class core.

5.1.2 I-Class L1 Data Cache Configuration

Details about the data cache and some of the important cache parameters are as follows:

- Cache type: 4-way set associative, Virtually Indexed Physically Tagged, non-blocking cache
- Cache size: 16 Kilobytes

- Cache line size: 64 bytes
- Maximum number of requests accepted by cache per cycle: 1
- Maximum number of responses sent by the cache to the core per cycle: 1
- DMSHR size: 12
- Additional MSHR secondary entries: 3 per line (1 primary, 2 secondary)
- DTLB size: 16

5.1.3 Parameters Varied for Testing

To understand the effects of various parameters and evaluate the performance of prefetchers better, directed tests were run with I-Class default core and cache while varying the following parameters:

- **Additional memory latency cycles:** In the current implementation of I-Class, an L2 cache is not yet present. A load hit in the L1 data cache would return the data in 2 cycles. A miss will go to the DRAM whose latency is typically over 100-200 cycles in a real-time system. To realise this long latency of a miss in simulation too, we can enforce additional memory latency by increasing the value of that parameter. A prefetcher primarily helps in hiding the memory latency and hence the impact of a prefetcher in improving the performance is expected to be higher in these long latency simulations. Tests were run for 3 values of additional memory latency = 0 and 100 cycles to observe the effects.

- **Prefetch distance:** Timeliness of the prefetch requests can be observed by varying the prefetch distance parameter. A higher prefetch distance value can result in more timely prefetches. However, if the value is too large, the prefetch addresses generated may be too far and the program is less likely to require the data at those addresses which may result in lower accuracy of the prefetches. Values 2, 4 and 8 have been considered for the experiments for prefetch distance in this project.

- **Prefetch degree:** The value of prefetch degree was set to 1 for all the runs. Higher prefetch degree could worsen performance as cache accepts only one request per cycle and more number of prefetch requests would lead to the MSHR getting full and the cache getting busy.

- **Prefetch throttling:** For all runs, prefetch throttling has been enabled. This ensures that 2 MSHRs are always reserved for demand requests. This drops some prefetch requests but helps avoid the situation of cache signalling busy due to MSHRs getting full with the prefetch requests.

5.2 BSV TESTBENCHES

For debugging and functional testing of the prefetcher codes, BSV testbenches were used. Using these testbenches, module-level testing of the prefetchers is done by calling input interface methods and sampling the outputs.

5.3 DIRECTED ASSEMBLY TESTS

Directed tests in our context, are small assembly language programs written to test the functionality and performance of a particular feature that we are interested in. They help in realising if the feature works in the way that we expect it to i.e. perform well in certain expected scenarios and show degraded performance in certain other scenarios. They can also be used to tune the various parameters involved in the design. These tests also serve as a basis for the performance analysis. Throughout the project, these tests were run on the prefetchers with different versions of I-Class. Below are results with selected tags of I-Class (mentioned in the captions). Some of the tests used have been explained in Chapter 3 and Appendix A of Somisetty (2021).

Results and Analysis

Core-side: We observe that in the SAXPY Line test, there is approximately a 1.25x speedup with all the four stride prefetchers. This speedup is almost 2x in the higher latency case. In the SAXPY Word test, the Stride reg and Stride BRAM prefetchers show a slight speedup in the high memory latency scenario.

The other stride and linestride tests either show a very slight speedup or a slight slow down. This can be due to less cycles in the tests or lack of stride patterns.

Cache-side: In the SAXPY test statistics tabulated in tables 5.6 and 5.7, some speedup is observed, although it is much lesser than PC-based core-side prefetchers. Best offset prefetcher produces higher prefetcher than nextline offset.

Table 5.2: Saxpy Word test results with I-Class 5.8.4 - mem latency =10, prefetch dist =2

Test_saxpy_word	No prefetcher	Stride reg	Stride BRAM	Linestride reg	Linestride BRAM
Total instr	45095	45095	45095	45095	45095
Core Cycles	84200	70054	70216	84202	84200
Speedup	1	1.2019	1.1992	1	1
Prefetch requests generated	0	5038	4808	1	0
Prefetch requests enqueued	0	5038	4808	1	0
DCache_prefetches_mshr_allocated	0	128	129	0	0
DCache_load_hits	8070	8196	8197	8070	8070

Table 5.3: Saxpy Line test results with I-Class 5.8.4 - mem latency =10, prefetch dist =2

Test_saxpy_line	No prefetcher	Stride reg	Stride BRAM	Linestride reg	Linestride BRAM
Total instr	45094	45094	45094	45094	45094
Core Cycles	383471	196981	189279	195545	189050
Speedup	1	1.9467	2.026	1.961	2.0284
Prefetch requests generated	0	4823	3603	4822	3623
Prefetch requests enqueued	0	4823	3603	4822	3623
DCache_prefetches_mshr_allocated	0	4791	3586	4786	3596
DCache_load_hits	7	2200	1905	2147	1907

Table 5.4: Saxpy Word test results with I-Class 5.8.4 - mem latency =100, prefetch dist =4

Test_saxpy_word	No prefetcher	Stride reg	Stride BRAM	Linestride reg	Linestride BRAM
Total instr	45095	45095	45095	45095	45095
Core Cycles	84200	70295	70152	84202	84200
Speedup	1	1.1978	1.2003	1	1
Prefetch requests generated	0	4929	4762	1	0
Prefetch requests enqueued	0	4929	4762	1	0
DCache_prefetches_mshr_allocated	0	128	128	0	0
DCache_load_hits	8070	8196	8197	8070	8070

Table 5.5: Saxpy Line test results with I-Class 5.8.4 - mem latency =100, prefetch dist =4

Test_saxpy_line	No prefetcher	Stride reg	Stride BRAM	Linestride reg	Linestride BRAM
Total instr	45094	45094	45094	45094	45094
Core Cycles	383471	212964	184119	216262	184341
Speedup	1	1.8006	2.0827	1.7732	2.0802
Prefetch requests generated	0	3811	3798	4189	3779
Prefetch requests enqueued	0	3811	3798	4189	3779
DCache_prefetches_mshr_allocated	0	3771	3778	4151	3784
DCache_load_hits	7	2094	1969	2151	1972

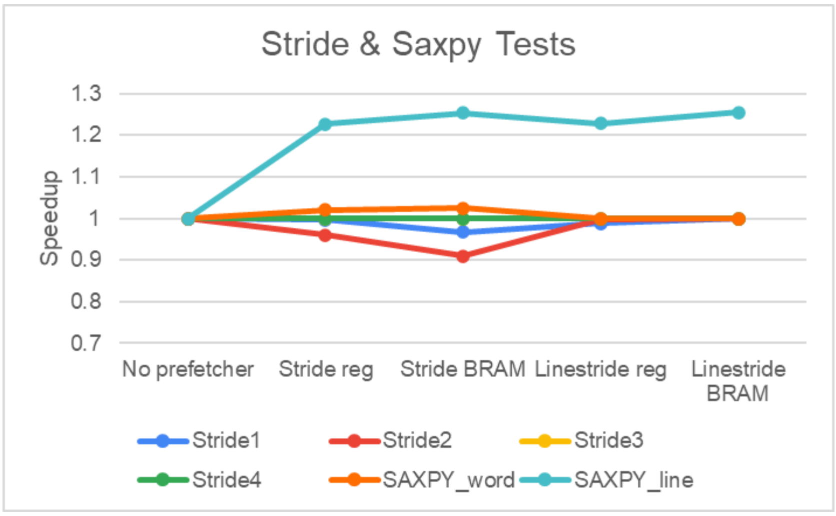

Figure 5.1: Stride and SAXPY tests speedups with I-Class tag 5.8.4

Table 5.6: Saxpy Word test results with Cache-side prefetchers - I-Class 6.1.3, mem latency =10, prefetch dist =2

Test_saxpy_word2	No prefetcher	Nextline offset	Best offset
Total instr	180263	180263	180263
Core Cycles	222964	221144	220098
Speedup	1	1.0082	1.013
Prefetch requests generated	0	181	1936
Prefetch requests enqueued	0	181	1936
DCache_prefetches_mshr_allocated	0	181	617
DCache_load_hits	27928	28170	29645

Table 5.7: Saxpy Line test results with Cache-side prefetchers - I-Class 6.1.3, mem latency =10, prefetch dist =2

Test_saxpy_line2	No prefetcher	Nextline offset	Best offset
Total instr	180262	180262	180262
Core Cycles	456625	434610	422423
Speedup	1	1.0507	1.081
Prefetch requests generated	0	9811	11072
Prefetch requests enqueued	0	9811	11072
DCache_prefetches_mshr_allocated	0	2759	7857
DCache_load_hits	3	2732	10397

5.4 FESVR TESTS

We have run C microbenchmark tests on the I-Class RTL simulation with RISCV-PK and FESVR and have used Spike to verify the results. The microbenchmark tests include programs like linked list manipulation, matrix add and matrix multiplication.

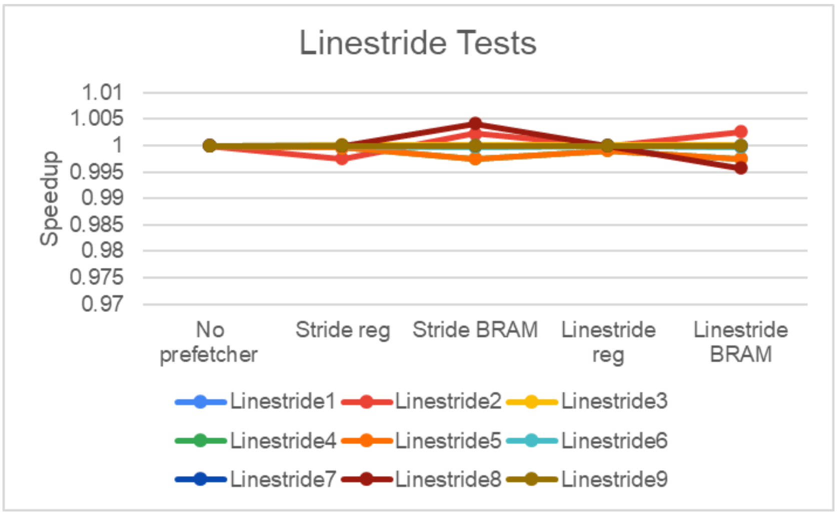

Figure 5.2: Linestride tests speedups with I-Class 5.8.4

Proxy Kernel

Proxy Kernel (PK) is an abstraction of a kernel that provides system services through FESVR running on the host. Together with FESVR, it provides an application binary interface (ABI) and an application execution environment (AEE).

Front-end Server

The front-end server facilitates communication between a host machine and a RISC-V target. It loads the ELF and emulates the peripheral device over the Host Target Interface (HTIF).

Spike

Spike[2], a RISC-V ISA Simulator, implements a functional model of one or more RISC-V harts. It is a Golden Reference Functional ISA Simulator. It could be used for full system emulation or proxied emulation with HTIF.

Results and Analysis

Core-side: In tables 5.8 and 5.9, we can see a significant number of prefetches generated and enqueued, but a very small fraction of this translates to allocated MSHRs. The speedup we see is almost negligible.

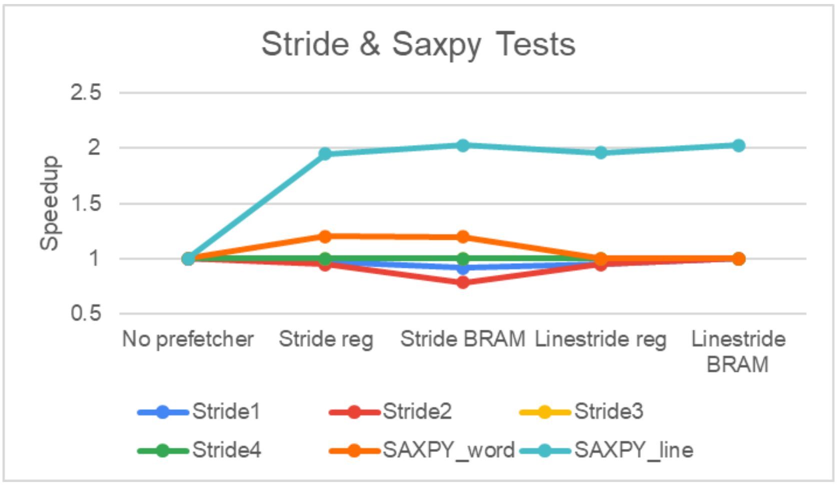

Figure 5.3: Stride and SAXPY tests speedups with I-Class 5.8.4 - mem latency = 100, prefetch dist = 4

Cache-side: The obtained results of testing show that cache-side prefetchers show very less or no speedups for the C microbenchmark tests.

Table 5.8: Statistics of Matrix Multiplication microbenchmark test on I-Class 5.9.9 FESVR with input matrices of size 100x100, memory latency =10, prefetch dist =2

Test Statistics/Prefetcher	No prefetcher	Stride BRAM	Complex Stride BRAM
Matrix Mult			
Core Cycles (Total)	562348867	562311271	562347255
Total instr	729160351	729210400	729315010
Core Cycles	562036736	562036736	562036736
User Cycles	548861699	548869823	548775289
Speedup	1	1.0001	1
Prefetch requests generated	0	278392	309372
Prefetch requests enqueued	0	278392	309364
DCache_requests_total	207366948	207667329	207934556
DCache_prefetches_mshr_allocated	0	1556	2295
DCache_load_hits	115438228	115460640	f3f3f3115709117
DCache_store_hits	91041235	91047809	f3f3f391042466
DCache_fill_requests	105395	105782	f3f3f3106120

Table 5.9: Statistics of Matrix Multiplication microbenchmark test on I-Class 5.9.9 FESVR with input matrices of size 100x100, memory latency = 100, prefetch dist = 8

Test Statistics/Prefetcher	No prefetcher	Stride BRAM	Complex Stride BRAM
Matrix Multiplication			
Core Cycles (Total)	573369119	573375447	572929159
Total instr	728082606	728089971	728371999
Core Cycles	572522496	572522496	572522496
User Cycles	557608900	557805907	557893282
Speedup	1	1	1.0008
Prefetch requests generated	0	794192	1065465
Prefetch requests enqueued	0	794192	1065312
DCache_requests_total	206882983	207648628	207961519
DCache_prefetches_mshr_allocated	0	3727	4132
DCache_load_hits	115071893	115081870	115052522
DCache_store_hits	90949594	90951028	91025169
DCache_fill_requests	105555	106545	107359

Table 5.10: Statistics of Matrix Multiplication microbenchmark test with Cache-side prefetchers on I-Class 6.1.3 FESVR with input matrices of size 100x100, memory latency = 100, prefetch dist = 8

Test Stats/Prefetcher	No prefetcher	Nextline offset - fixed	Best offset
Core Cycles (Total)	574033179	572316835	574826559
Total instr	728350686	727675999	728887291
Core Cycles	573571072	571473920	574619648
User Cycles	558820749	556923464	560200978
Speedup	1	1.003	0.9986
Speedup - user cycles	1	1.0034	0.9975
Prefetch requests generated	0	41989	78436
Prefetch requests enqueued	0	41138	75974
DCache_requests_total	206823456	207073312	207133586
DCache_prefetches_mshr_allocated	0	37845	68268
DCache_load_hits	114986591	115297011	115195140
DCache_store_hits	91015371	90959980	91079582
DCache_fill_requests	106943	124228	149337

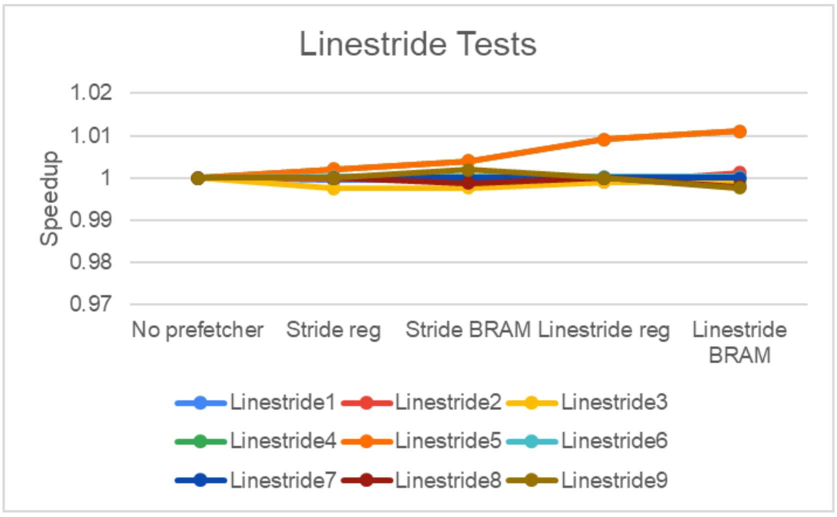

Figure 5.4: Linestride tests speedups with I-Class tag 5.8.4 - mem latency = 100, prefetch dist = 4

5.5 SPEC BENCHMARK TESTS

For further analysis, we have run SPEC benchmark tests on I-Class RTL simulation with RISCV-PK and FESVR. In this project, we have run the following benchmark tests: Imagick, Xalan, Perl and Povray. These particular tests have been chosen considering the simulation times, which range from a few days (for Imagick) to a couple of weeks (for Povray). All of these tests were run with test inputs only but to completion.

Results and Analysis

Core-side: For Imagick benchmark tests with an additional memory latency 100, a speedup of around 2% is observed for stride BRAM and complex stride prefetchers, with a significant number of prefetches generated and enqueued. For the Povray benchmarks tests, the number of prefetchers generated is significantly lesser, and the speedup is minimal.

Cache-side: The obtained results of testing show that cache-side prefetchers show a good speedup for the Imagick benchmark test in the additional memory latency equal to 100 case. The nextline offset prefetcher shows a speedup of 4.8% and the best offset prefetcher shows a speedup of 8.75%.

Table 5.11: Statistics of Imagick benchmark test on I-Class 5.9.9 FESVR - memory latency =10, prefetch dist =2

Test Statistics/Prefetcher	No prefetcher	Stride BRAM	Complex Stride BRAM
Imagick			
Core Cycles (Total)	207993799	207341319	207312531
Total instr	248864144	248254078	248974203
Core Cycles	207618048	206569472	206569472
User Cycles	107868001	107302254	107136697
Speedup	1	1.0031	1.0033
Prefetch requests generated	0	124521	136831
Prefetch requests enqueued	0	124512	136808
DCache_requests_total	93478700	93345255	93730238
DCache_prefetches_mshr_allocated	0	70443	74782
DCache_load_hits	79032120	78970113	79361609
DCache_store_hits	8406829	8405939	8409461
DCache_fill_requests	1523587	1523629	1526431

Table 5.12: Statistics of Imagick benchmark test on I-Class 5.9.9 FESVR - memory latency = 100, prefetch dist = 8

Test Statistics/Prefetcher	No prefetcher	Stride BRAM	Complex Stride BRAM
Imagick			
Core Cycles (Total)	319780095	312900579	313746307
Total instr	240807462	243377429	242144055
Core Cycles	318767104	312475648	313524224
User Cycles	203678312	197323359	198083782
Speedup	1	1.022	1.0192
Prefetch requests generated	0	273881	260474
Prefetch requests enqueued	0	272780	255630
DCache_requests_total	89364478	90948824	90312046
DCache_prefetches_mshr_allocated	0	111087	95666
DCache_load_hits	74896078	76468953	75815073
DCache_store_hits	8175222	8178532	8177994
DCache_fill_requests	1524894	1531691	1537733

Table 5.13: Statistics of Povray benchmark test on I-Class 5.9.9 FESVR - memory latency = 100, prefetch dist = 8

Test Statistics/Prefetcher	No prefetcher	Stride BRAM	Complex Stride BRAM
Povray			
Core Cycles (Total)	8787517339	8782018615	8785119331
Total instr	2301634498	2302600819	2302396958
Core Cycles	8787066880	8781824000	8784969728
User Cycles	8676799994	8671219925	8674504337
Speedup	1	1.0006	1.0003
Prefetch requests generated	0	67135	26770
Prefetch requests enqueued	0	67076	26707
DCache_requests_total	1242747033	1242498771	1243153941
DCache_prefetches_mshr_allocated	0	32360	17066
DCache_load_hits	699578378	699356715	699926605
DCache_store_hits	307365844	307392274	307380387
DCache_fill_requests	73272722	73262966	73268745

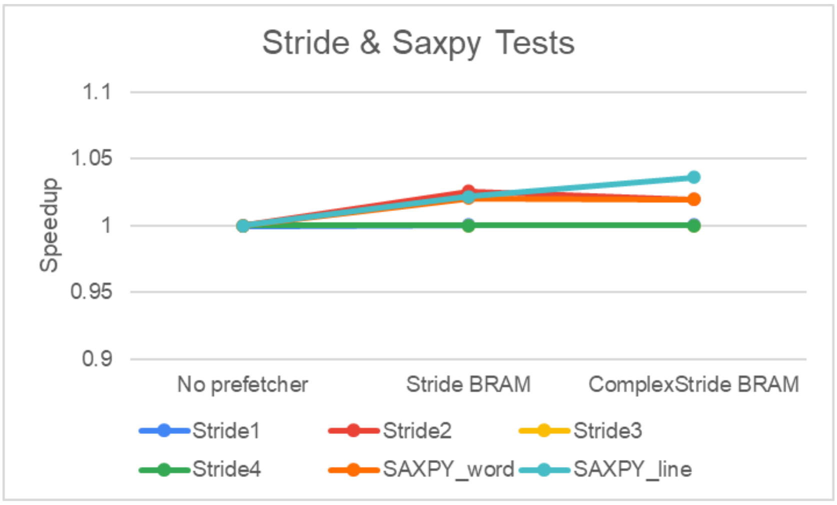

Figure 5.5: Stride & SAXPY tests speedups with I-Class 5.9.9 - mem latency =10, prefetch dist =2

Table 5.14: Statistics of Imagick benchmark test with Cache-side prefetchers on I-Class 6.1.3 FESVR - memory latency=100, prefetch dist=8

Test Stats/Prefetcher	No prefetcher	Nextline offset	Best offset
Imagick			
Core Cycles (Total)	319855059	305206267	294122935
Total instr	242627375	241347265	243869903
Core Cycles	319815680	305135616	293601280
User Cycles	203844965	190816927	183423383
Speedup	1	1.048	1.0875
Speedup - user cycles	1	1.0683	1.1113
Prefetch requests generated	0	583094	1032825
Prefetch requests enqueued	0	565481	982392
DCache_requests_total	90261178	90215012	91913436
DCache_prefetches_mshr_allocated	0	533395	946352
DCache_load_hits	75766094	75472683	76949321
DCache_store_hits	8202595	8537734	8867244
DCache_fill_requests	1526342	1789905	2023392

5.6 INSIGHTS

The BRAM versions of the Stride prefetchers perform as well or even better than the register versions. They are also better to implement on hardware, and have a set-associative SPT. Overall, the Stride BRAM has the best performance in the tests, with Linestride BRAM also performing almost as well.

The data cache-side offset prefetchers do not improve the performance significantly for the shorter tests. This could be because at the L1 level, the prefetcher could potentially

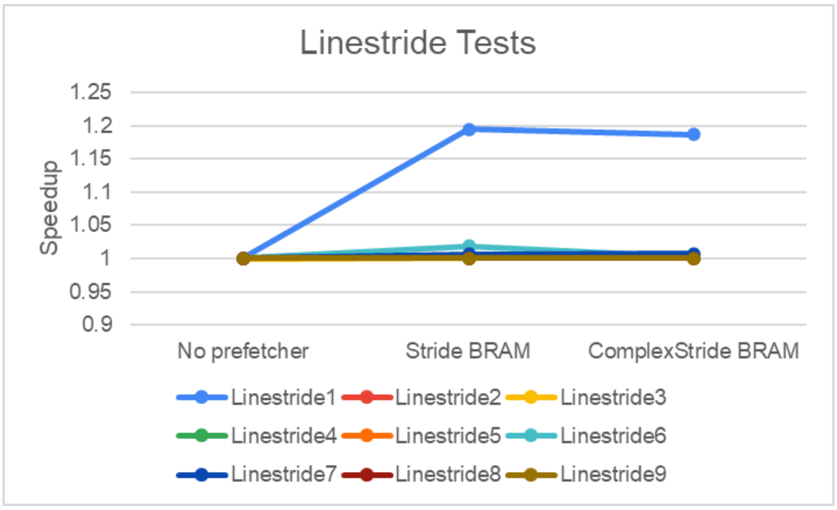

Figure 5.6: Linestride tests speedups with I-Class 5.9.9 - mem latency =10, prefetch dist =2

be too aggressive, and might not have enough training period in the short tests performed. But for a longer test like the imagick benchmark, the speedups are high showing that a proper training period could train the offset prefetcher well.

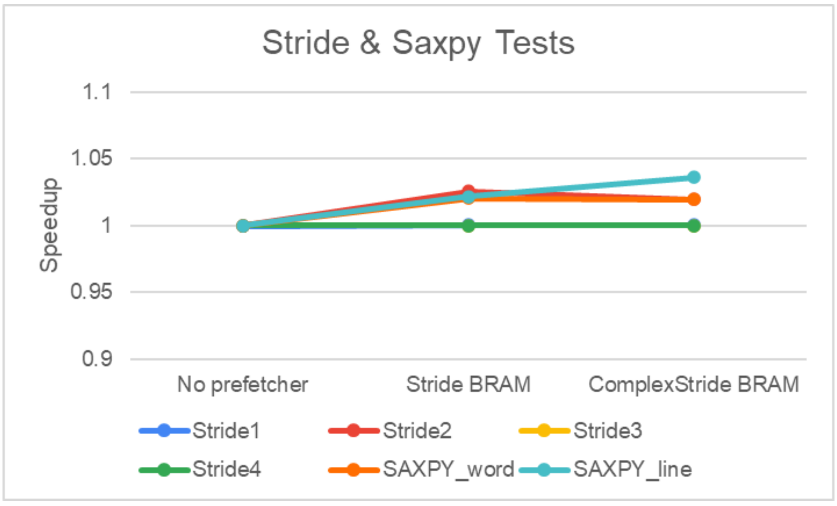

Figure 5.7: Stride & SAXPY tests speedups with I-Class 5.9.9 - mem latency =100, prefetch dist =4

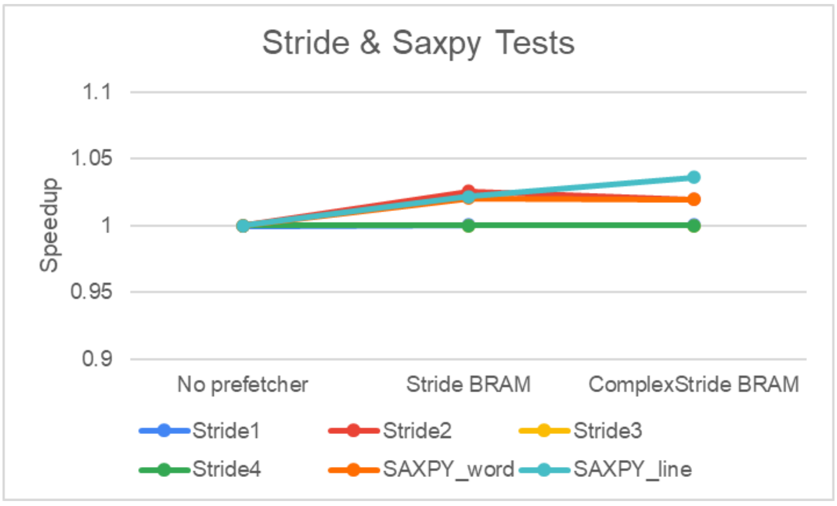

Figure 5.8: Linestride tests speedups with I-Class 5.9.9 - mem latency =100, prefetch dist =4

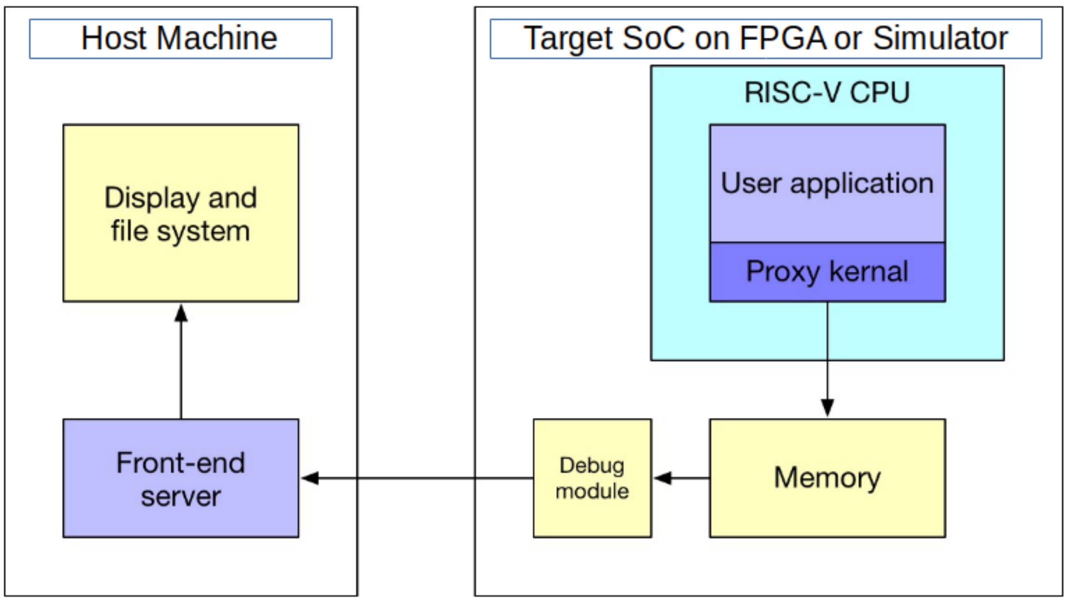

Figure 5.9: PK (proxy kernel) & FESVR (front-end server) placement in the system. PK runs on target CPU, while FESVR runs on host computer. In our setup, Verilator simulation of the benchmark is done on top of PK and FESVR all on the host machine. For system call emulation, PK communicates with the FESVR thread which then runs the system calls on the host machine and returns the results.

Source: https://phdbreak99.github.io/riscv-training/12-demo.syscall/

CHAPTER 6

CONCLUSION

In this project, we have experimented with various prefetchers, including core side and cache side prefetchers. We have tested them using various assembly and benchmark tests. We have learnt that prefetching, when working accurately and timely, can improve the performance of the processor. The advantage of the hardware prefetchers we have implemented for the Shakti I-Class processors is that the designs are simple enough to be implemented with a few bytes of storage and there is no degradation to the processor's performance.

We have seen that different prefetchers (nextline, stride, complex stride, offset etc) perform well in different scenarios. Based on the requirements of the application that the processor is used for, the most suitable prefetcher can be implemented. In the cases where the memory latencies are high, the prefetchers did help bring down the cycle counts more often than not. However, several factors affect the performance of the prefetchers, including smaller training period in tests, low MSHR counts etc.

6.1 FUTURE SCOPE

In the future, further testing could be performed with longer tests to train the prefetchers, to observe higher speedups. Further modifications or optimisations could be done to the prefetchers implemented. The complex stride prefetcher could further be modified to detect more complex stride patterns. In the future, a "hybrid prefetcher" could also be designed such that an assortment of any of the prefetchers required for an application could be chosen at once.

We have currently implemented the cache side prefetchers on the L1 level. This could be extended to the L2 Cache as well as the Instruction cache of the processor. Additionally, other types of hardware prefetchers can also be explored, for example prefetchers that can adapt to the application using machine learning, and post synthesis programmable prefetchers.

REFERENCES

1. *Gitlab I-Class repository.* URL `https://gitlab.com/shaktiproject/cores/i-class.git`.

2. *Spike.* URL `https://github.com/riscv-software-src/riscv-isa-sim`.

3. *Verilator.* URL `https://www.veripool.org/verilator/`.

4. *Bluespec SystemVerilog Reference Guide, Bluespec Inc.*. Revision 2014.

5. **Chen, T.-F.** and **J.-L. Baer** (1995). Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, **44**(5), 609–623.

6. **Fu, J. W. C.**, **J. H. Patel**, and **B. L. Janssens**, Stride directed prefetching in scalar processors. *In Proceedings of the 25th Annual International Symposium on Microarchitecture*, MICRO 25. IEEE Computer Society Press, Washington, DC, USA, 1992. ISBN 0818631759.

7. **Hennessy, J. L.** and **D. Patterson**, *Computer Architecture: A Quantitative Approach*. 2011, 5^{th} edition.

8. **Michaud, P.**, Best-offset hardware prefetching. *In 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2016.

9. **Mutlu, O.** (2020). Lecture 18: Prefetching. *Computer Architecture Notes from ETH Zürich, Fall 2020.*

10. **Pakalapati, S.** and **B. Panda**, Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching. *In 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 2020.

11. **Panda, B.** (n.d.). Lecture 12: Hardware prefetching. *Modern Memory Systems Notes from IIT Kanpur*.

12. **SHAKTI**, *Shakti Processors.* URL `https://shakti.org.in/`.

13. **Somisetty, M.** (2021). Performance analysis and enhancement of hardware prefetchers for shakti i-class processor. *IIT Madras*.

14. **Wikipedia**, *Cache Prefetching.* URL `https://en.wikipedia.org/wiki/Cache_prefetching`.