Performance analysis and enhancement of hardware prefetchers for Shakti I-Class processor

A Project Report

submitted by

MEENAKSHI SOMISETTY

in partial fulfilment of the requirements

for the award of the degree of

BACHELOR OF TECHNOLOGY

&

MASTER OF TECHNOLOGY

in

ELECTRICAL ENGINEERING

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY MADRAS

June 2021

THESIS CERTIFICATE

This is to certify that the thesis titled **Performance analysis and enhancement of hardware prefetchers for Shakti I-Class processor**, submitted by **Meenakshi Somisetty**, to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelor of Technology & Master of Technology in Electrical Engineering**, is a bona fide record of the research work done by her under our supervision. 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.

Prof. V. Kamakoti	**Prof. Harishankar Ramachandran**
Project Guide	Project Co-Guide
Professor	Professor
Dept. of Computer Science & Engineering	Dept. of Electrical Engineering
IIT-Madras, 600 036	IIT-Madras, 600 036

Place: Chennai

Date: 28th June 2021

ACKNOWLEDGEMENTS

I would like to express my gratitude towards 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. Harishankar Ramachandran as the co-guide of this project and I am grateful for his support. I would also like to express my deepest gratitude to my project mentor, Dr. Nitya Ranganathan for her immense guidance and constant support throughout the course of my 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 extending their help and support at all times.

ABSTRACT

KEYWORDS: Prefetcher; Stride; I-Class; Bluespec SystemVerilog.

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. The objective of this optimization technique is to reduce the average memory access time. It helps to reduce the cache miss rate or miss penalty via parallelism. Hardware prefetchers can be of two types: data prefetchers and instruction prefetchers. The aim of this project is to modify or enhance the existing L1 data cache prefetchers for the Shakti I-Class core and also design new prefetchers for this out-of-order RISC-V processor in order to achieve higher instructions per cycle or frequency. The design of the prefetchers has been done in the Bluespec SystemVerilog (BSV) language. The implementation of the most common next-line prefetcher and different program counter based prefetchers is discussed in this thesis.

The various directed tests that were written for the performance check of the prefetchers are also mentioned in this thesis. The performance of the processor with and without prefetchers is analysed and the comparison of the different prefetchers in different configurations is highlighted in further sections of the thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

BSV	Bluespec SystemVerilog
PC	Program Counter
SPT	Stride Prefetch Table
PRQ	Prefetch Request Queue
BRAM	Blocked Random Access Memory
SAXPY	Single-Precision A·X Plus Y
MSHR	Miss Status Handling Register
DTLB	Data Translation Lookaside Buffer
PRG	Prefetch Requests Generated
PRE	Prefetch Requests Enqueued
PRD	Prefetch Requests Dropped

CHAPTER 1

INTRODUCTION

This chapter is an introduction about prefetchers, the processor for which prefetchers have been designed in this project and the language they are implemented in. The last section of the chapter gives an outline of the flow of the thesis.

1.1 Introduction to 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 a local memory (like caches or external buffers) before they are actually requested by the processor. It helps in the reduction of cache miss rate or miss penalty. Main memory latency is high compared to caches. 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 main memory. Hence, timely and accurate prefetching of data can result in improvement in performance. But the disadvantage with prefetchers is that, when they become too aggressive it can cause increase in traffic on the memory interconnection network which can lead to extra miss penalty for genuine demand requests from the current core or other cores in multi-core processors. 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.

The most common method of prefetching is one in which whenever there is a miss on a block, the processor fetches two blocks: the requested block and the next consecutive block from the main memory into the cache. There are several other types of prefetchers including Program Counter (PC)-based prefetchers like stride prefetcher and its modifications, address-based prefetchers and hybrid prefetchers. Most prefetchers predict what address needs to be prefetched based on past access patterns.

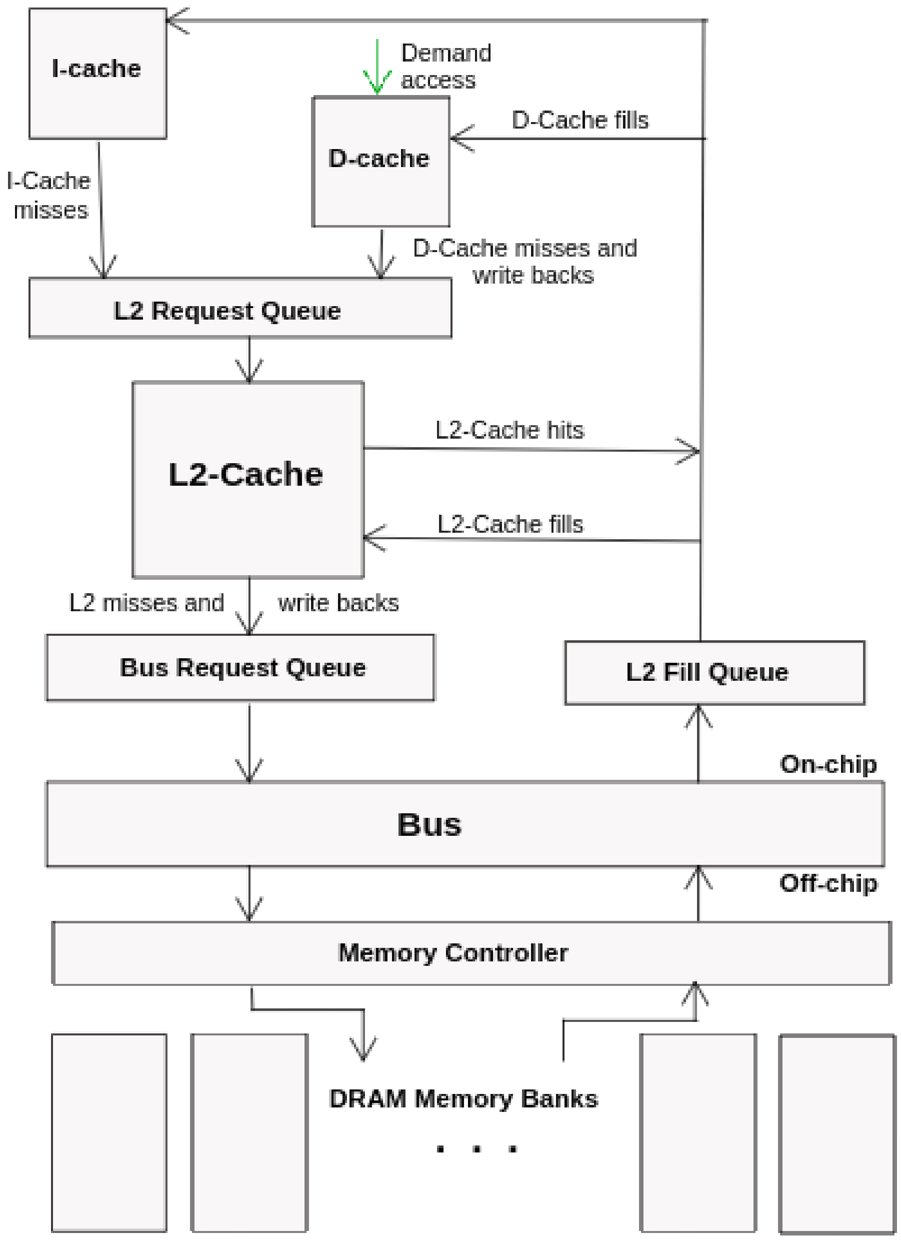

Figure 1.1: Block diagram of a memory system without prefetcher.

In this project, data prefetchers like next-line/previous-line prefetcher, PC-based stride prefetchers and its modifications have been implemented.

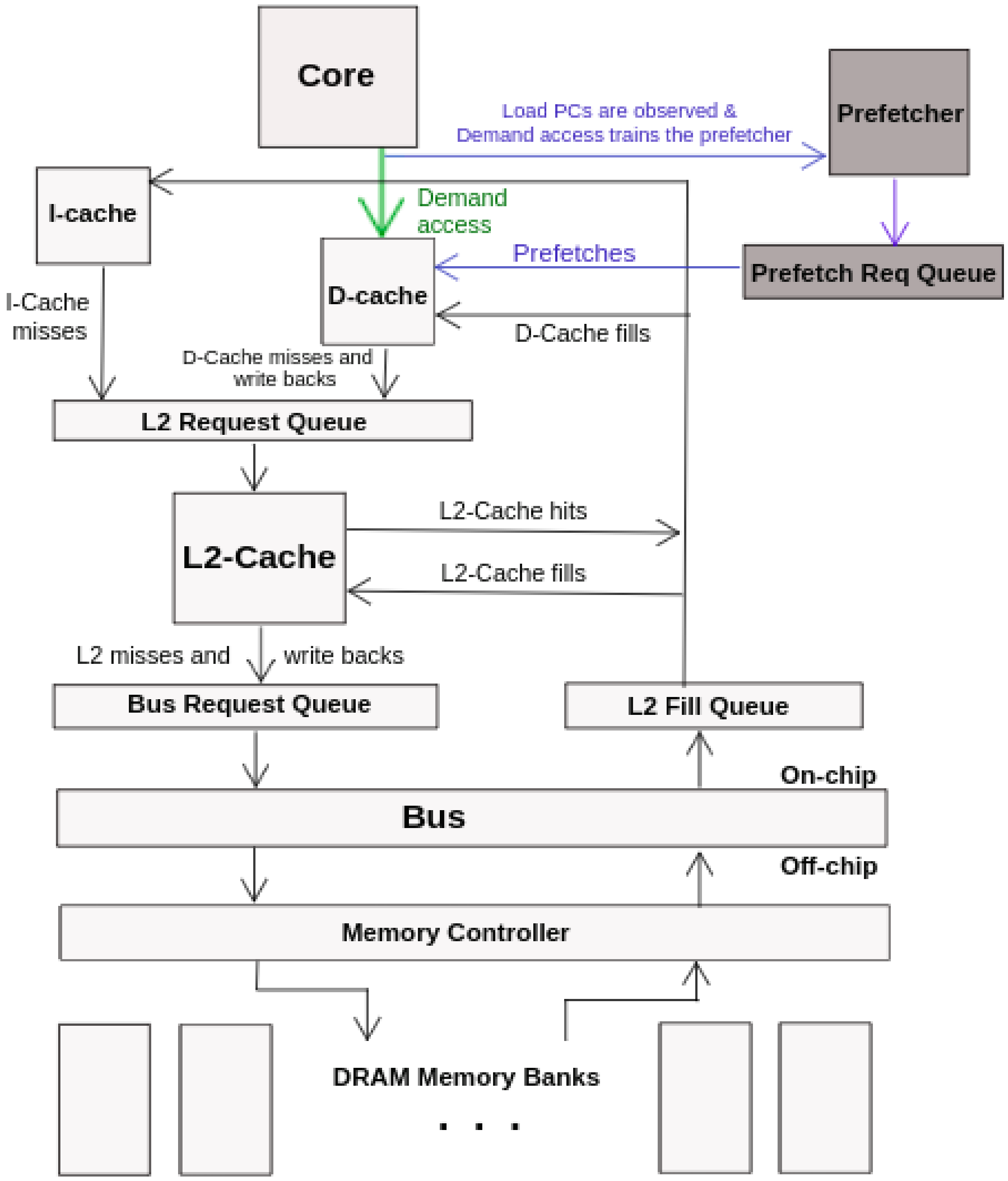

Figure 1.2: Block diagram of a memory system with a core side prefetcher (Present in the I-Class version of this project).

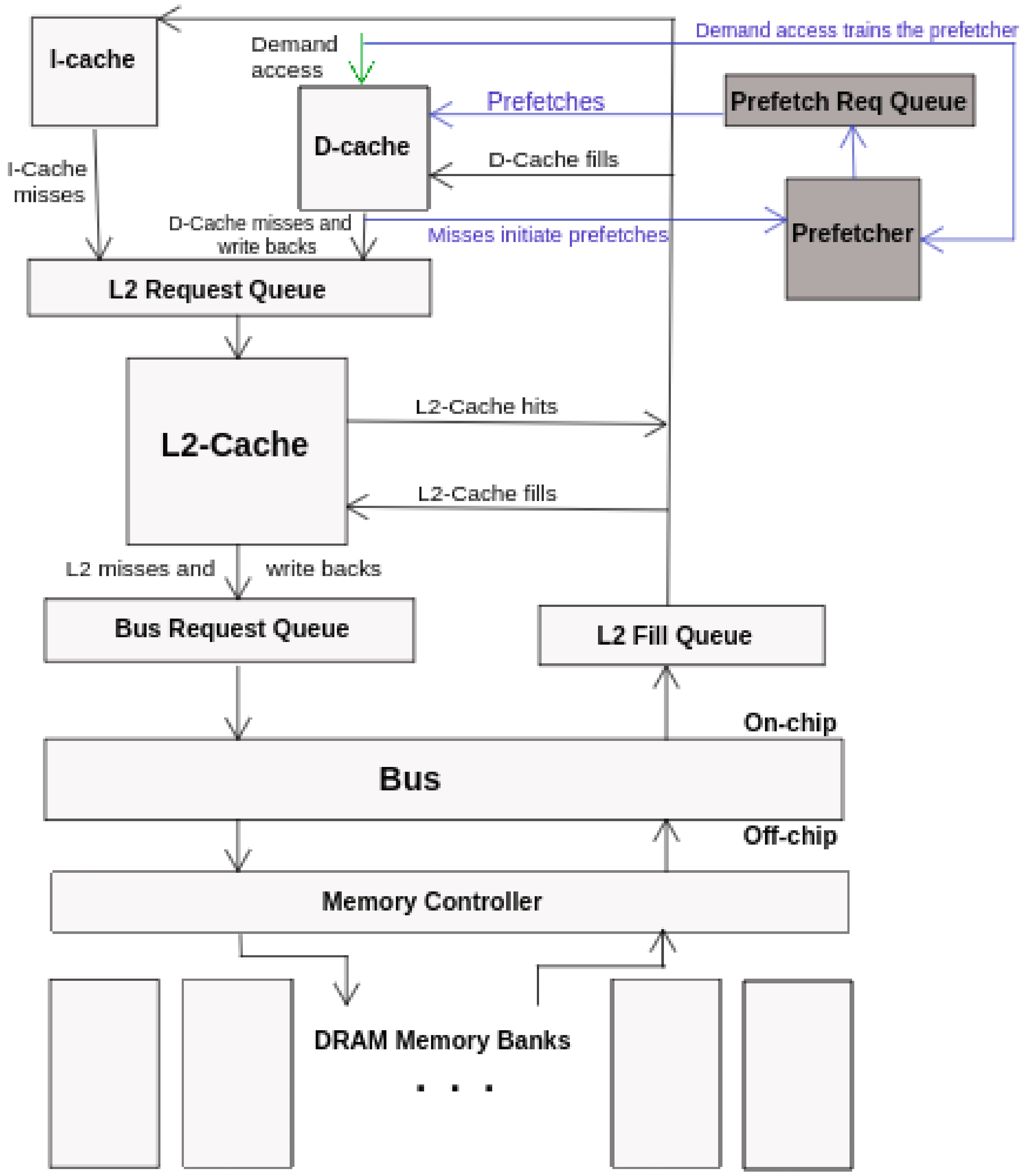

Figure 1.3: Block diagram of a memory system with an L1 data cache side prefetcher (Not present in the I-Class version of this project).

1.2 About the I-Class processor

The Shakti I-Class core for which prefetchers have been designed in this project, is a superscalar 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.

It is a 4-wide out-of-order core that can fetch/dispatch/issue/commit 4 instructions per 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 removes false dependencies along with checkpointing, reorder buffer that stores instruction metadata for all instructions in flight, operand bypass, memory dependence predictor and non-blocking cache supporting multiple outstanding misses with Miss Status Handling Registers (MSHRs).

1.3 Why Bluespec SystemVerilog?

All the prefetchers in this project have been designed and implemented in Bluespec SystemVerilog (BSV) language. BSV is a high-level functional hardware description programming language to handle chip design which comes with a SystemVerilog frontend. Concurrency can be realized reliably in BSV since it is a rule-based language where hardware is described as object-oriented modules. The model of atomic rules in BSV helps to eliminate race conditions that are unwanted and makes it easily scalable to large designs. BSV also supports more polymorphism. A designer can easily rework designs, reuse them and integrate them together in more flexible ways in BSV since rules, modules, interfaces, functions and actions are all objects in this language. This leads to greater level of correctness, lucidity and preciseness . With all these features, BSV can greatly raise a hardware designer's productivity and is hence preferred. The bsc compiler is used to generate Verilog from the BSV code. From Verilog, the further steps in the VLSI design flow like generation of netlist, physical layout etc. can be continued as in the usual process.

1.4 Organization of thesis

The outline of the thesis is as follows. Chapter 2 is an elaborate description of the design and functionality of the existing prefetchers, their enhancements or optimizations and also the new prefetchers implemented. Next, in Chapter 3, the various tests written to check the functional correctness and evaluate the performance of each prefetcher are discussed. Chapter 4 shows the experimental setup and the statistics and results from the performance analysis of the prefetchers. Finally towards the end, Chapter 5 concludes the thesis with the future scope of this project and further work that can be done. Appendix A contains examples of some the directed tests mentioned in Chapter 3. The references for this project have been cited at the end of the thesis.

CHAPTER 2

PREFETCHERS

As mentioned in the introduction chapter, most prefetchers predict what address needs to be prefetched based on past access patterns. This process takes place in two stages: training stage and generate stage. In the training stage, the prefetcher observes the load PCs and the distance between the memory addresses referenced by the load instruction. Here, the prefetcher learns and detects the pattern. If the prefetcher has enough confidence in the pattern and all conditions are met, it predicts and issues prefetch requests in the generate stage. The design and implementation approach of each prefetcher is described in various sections of this chapter. The first section defines the important parameters involved in prefetcher design.

2.1 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.

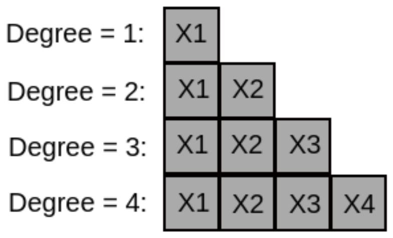

Figure 2.1: An illustration of prefetch requests issued for prefetch degree equal to 1 to 4.

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

- **Number of entries:** The total number of entries in the Stride Prefetch Table (SPT).

- **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. 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 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 log of number of bytes in a cache line. It is used to calculate line address in line stride prefetcher.

2.2 Next-line/Previous-line prefetcher

The most easy to implement version of the next-line prefetcher is one which simply prefetches the next cache line on a demand access. There are several enhancements possible to this simple prefetcher to improve its performance. Here, the prefetcher used in I-Class is explained.

2.2.1 Existing work

The next-line/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 next ‘n’ cache lines.

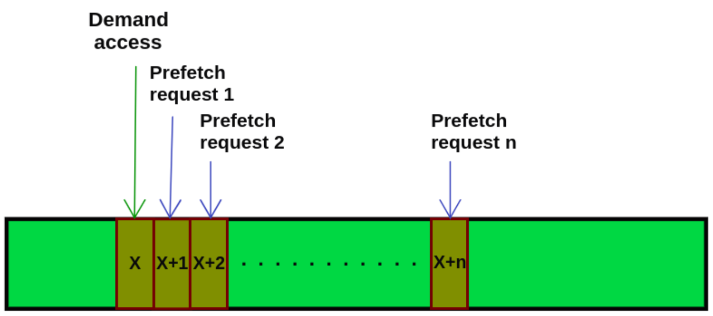

Figure 2.2: Next-line/Previous-line prefetcher case (i): Prefetching next n lines.

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

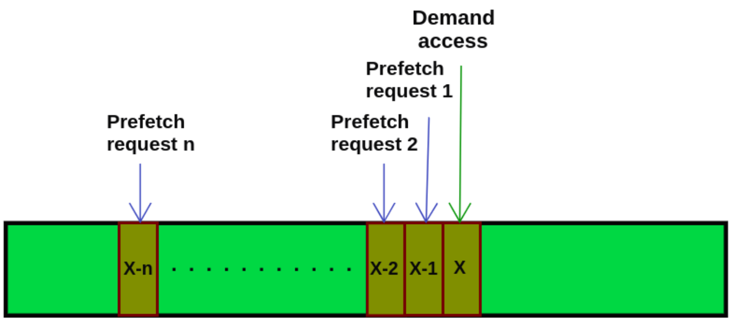

Figure 2.3: Next-line/Previous line prefetcher case (ii): Prefetching previous n lines.

2.2.2 Modification introduced

Prefetch requests mentioned in the existing prefetcher will be generated only if the prefetch address falls in the same page as that of the current demand load address.

2.3 Stride prefetcher

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 4. This difference is called stride. Here, the stride pattern is a constant 4. So, it can be predicted that 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. In other terms, observe and record 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. 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, prefetch [current address + stride] address (when prefetch degree and distance are both equal to 1). The addresses to be prefetched can be predicted accordingly when the prefetch degree or distance is a different value.

2.3.1 Existing work

The stride prefetcher implemented for the I-Class core uses a register array for the SPT. 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. Some bits of PC become the index used to locate a particular SPT entry and the other PC bits are used for 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 parameterizable (Num_Entries, stride width, maximum confidence, confidence threshold, prefetch degree and prefetch distance). Index width will be $\log_2(Num\_Entries)$ and confidence counter width will be $\log_2(Maximum\_confidence)$. Stride is calculated in bytes.

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

Tag	Previous address	Stride	Confidence counter
...	...	...	...
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
...	...	...	...

Table 2.1: Stride Prefetch Table

There are two stages involved. In the training stage, whenever a load instruction is seen, the table 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 default value (= 64). If not, decrement the confidence counter.

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

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

The most common stride seen for a particular load PC is a single positive or negative stride during one small execution window in the lifetime of a program. However, during other phases of execution in the same program, the stride observed for the same load instruction may change. For example, a load may be striding through the first field (with offset m) of a structure for the first 1000 loop iterations and may be striding through the second field (with offset n) in the next 1000 loop iterations. Both these scenarios can be easily captured by this stride prefetcher.

2.3.2 Enhancements

The following improvisations have been introduced to the existing stride prefetcher:

1. The prefetch addresses generated will now depend on prefetch distance value as:

   (Prefetch address) = (Current address) + (stride)*(prefetch distance)

2. A prefetch request will be issued only if the prefetch address doesn't fall in the same cache line as the previous prefetch request generated.
3. Prefetch requests will be generated only if the prefetch address falls in the same page as that of the current load address.

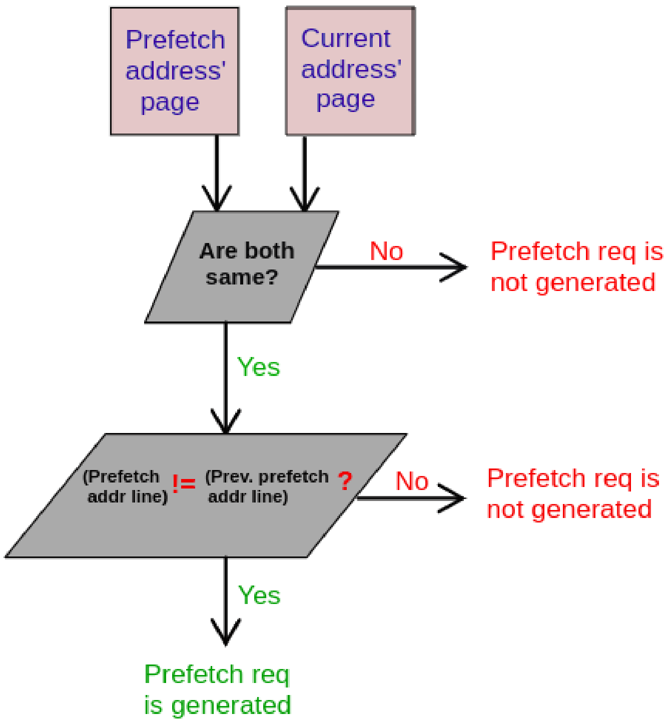
Prefetch address' page
Current address' page
Are both same?
No
Prefetch req is not generated
Yes
(Prefetch addr line) != (Prev. prefetch addr line) ?
No
Prefetch req is not generated
Yes
Prefetch req is generated

Figure 2.4: Enhancements to existing stride preftcher.

2.4 Line stride prefetcher

The line stride prefetcher is a modification of the stride prefetcher. Here, the stride is captured in multiples of cache lines instead of bytes. Though this may be less accurate comparatively, it saves a lot of space in the SPT. In the stride prefetcher, it takes more space to store stride value and the complete previous address. Here, the space for storing stride reduces and also, instead of storing the entire previous address, only the previous line address will be stored i.e. only the upper order bits of the address will be stored. 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, stride will be computed as the difference between the previous line address and the current line address and this stride has been called as line stride henceforth in the thesis.

$$\text{(Current line stride)} = \text{(Current line address)} - \text{(Previous line address)}$$

$$\text{(Prefetch line address)} = \text{(Current line address)} + \text{(line stride)}*\text{(prefetch distance)}$$

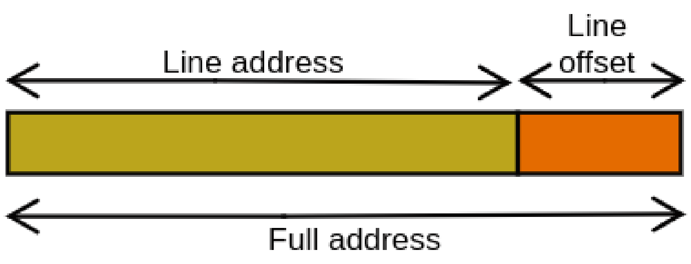

Figure 2.5: Extraction of line address from the full address.

Tag	Previous line address	Line Stride	Confidence counter
...	...	...	...
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
...	...	...	...

Table 2.2: Stride Prefetch Table for line stride 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
- Line Stride width = 5

2.4.1 Register array version

The working of register array based line stride prefetcher is same as that of stride prefetcher along with the enhancements except that line stride replaces stride in all computations and line addresses replace the full addresses.

2.4.2 BRAM version

In this version, the SPT is stored as a PC-indexed table in Blocked Random Access Memory (BRAM). The implementation uses the BRAMCore package of BSV. In BRAM, the read operation has one cycle extra latency (i.e. data is available one cycle after read request is put).

The design of the line stride prefetcher in this version is as follows: It is implemented with 1R and 1W dual ported BRAM (one port for reading and one port for writing to the BRAM). A simple 3-cycle pipelined design is chosen (cycle 0, 1, 2). In cycle 0, the load PC and address are received by the prefetcher and the address is latched in the read port of the BRAM. The load information is latched in the pipeline register for stage 0. In cycle 1, the SPT entry read is available and one of the four operations (same as that in stride prefetcher) and the computation for table entry update takes place. These values are latched in the pipeline register for stage 1. In cycle 2, if there is a need to update the table entry, the new entry is written using the write port. Computation and generation of prefetch requests also happens in this cycle if the generate conditions are satisfied.

An optimization 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 2 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 1. This problem is solved using bypassing. In stage 1, it is checked if the stage 2 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.

CHAPTER 3

DIRECTED 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. The directed tests written for the prefetchers are discussed in the below sections. The loop limit or number of iterations for the loops mentioned in the tests below is 2048.

3.1 Tests for next-line/previous-line prefetcher

- **test_prefetch_nl1:** This program has a simple loop in which values are loaded from addresses which are 64 bytes apart and the cumulative sum of them is calculated. The load address of each iteration is 64 bytes ahead of the that in the previous iteration. After the loop finishes execution, the sum is stored into a memory location. Since the addresses that are being referenced are in consecutive lines, the next-line/previous-line prefetcher is expected to perform well in this case and should improve the performance of the processor.
- **test_prefetch_nl2:** This test is a modification of the previous test. In this test instead of storing the sum at the end of the loop, it is stored into a memory location in every iteration of the loop. This is to observe how the prefetch requests are considered when there are several store operations along with the loads and how it affects the performance.
- **test_prefetch_nl3:** This test is same as the first test (test_prefetch_nl1) except that the addresses are 144 bytes apart. Since the addresses that are being referenced are not in consecutive lines but instead at a distance of two lines, the next-line/previous-line prefetcher is expected to degrade the performance as it always keeps prefetching the next line which is unused.
- **test_prefetch_nl4:** Similar to the second test (test_prefetch_nl2), this test also has store operations inside the loop. The rest of the program is same as the previous test (test_prefetch_nl3). It is also expected to show degraded performance with next-line/previous-line prefetcher.

3.2 Tests for stride prefetcher

- **test_prefetch_stride1:** This test has two simple loops. In the first loop values are loaded from addresses which are 32 bytes apart and the cumulative sum of them is calculated. The load address of each iteration is 32 bytes ahead of the that in the previous iteration. In the second loop also values are loaded from addresses which are 32 bytes apart and the cumulative sum of them is calculated but the load address of each iterations is 32 bytes behind of the that in the previous iteration. After the loop finishes execution, the sum is stored into a memory location. So this test covers scenarios of both positive and negative stride. Since the stride value is constant for the entire loop, the stride prefetcher is expected to identify this pattern easily and generate prefetches which should improve the performance.

- **test_prefetch_stride2:** This test is a modification of the previous test. In this test instead of storing the sum at the end of the loop, it is stored into a memory location in every iteration of the loop.

- **test_prefetch_stride3:** This test is similar to test_prefetch_stride1 but the stride in the loop keeps increasing by 16 in every iteration. Since the addresses that are being referenced are not with a constant stride, the stride prefetcher is not expected to improve the performance.

- **test_prefetch_stride4:** Similar to the second test (test_prefetch_stride2), this test also has store operations inside the loops. The rest of the program is same as the previous test (test_prefetch_stride3).

- **test_prefetch_saxpy_word:** This program has a simple Single-Precision A·X Plus Y (SAXPY) loop. SAXPY is a combination of scalar multiplication and vector addition: it takes as input two vectors of 32-bit floats X and Y with N elements each, and a scalar value A. It multiplies each element X[i] by A and adds the result to Y[i]. In this test, the elements of X are 4 bytes apart in memory and the elements of Y are 4 bytes part. So, this is again a case of constant positive stride of 4 which the stride prefetcher should be able to detect easily. So, stride prefetcher is expected to perform well in this context.

3.3 Tests for line stride prefetcher

- **test_prefetch_linestride1:** This test is same as test_prefetch_stride1 but the addresses are 128 bytes i.e. two lines apart. Since the line stride value is constant for the entire loop, the line stride prefetcher is expected to capture this pattern and generate prefetches which should improve the performance. The stride prefetcher can also capture this but the line stride prefetcher can do it while using lesser space for the SPT.

- **test_prefetch_linestride2:** This test is a modification of the previous test. In this test instead of storing the sum at the end of the loop, it is stored into a memory location in every iteration of the loop.

- **test_prefetch_linestride3:** This test is similar to test_prefetch_linestride1 but it has just a single positive stride loop.
- **test_prefetch_linestride4:** Similar to the second test (test_prefetch_linestride2), this test also has store operations inside the loops. The rest of the program is same as the previous test (test_prefetch_linestride3).
- **test_prefetch_linestride5:** This test is a modification of first test. The loop additionally has multiplication and division instructions. This is to simulate and observe the effect of these longer latency instructions.
- **test_prefetch_linestride6:** This test is a variant of test_prefetch_linestride5. Here, the addresses are 16 bytes apart instead of 128 bytes.
- **test_prefetch_saxpy_line:** This test is a modification of test_prefetch_saxpy_word. The difference is that in this test, the addresses in this test are 16 bytes apart.

Appendix A of the thesis shows assembly code of some of the tests mentioned above.

CHAPTER 4

PERFORMANCE ANALYSIS

This chapter contains the results and observations from the performance analysis of the prefetchers. The experimental setup and the different configurations for which the tests were run are explained in the first two sections of this chapter.

4.1 Experimental setup

4.1.1 I-Class core default configuration

The I-class core is highly parameterizable but the default configuration is being used for the experiments for performance analysis of the prefetchers. The values of some 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)	4
Commit stage width (instructions per cycle)	4
Number of entries in Load queue in Load Store Unit	16
Number of entries in Store queue in Load Store Unit	16
Number of registers in unified physical register file	128
Number of entries in Re-order buffer	96

Table 4.1: Values of parameters in the default configuration of I-Class core.

4.1.2 I-Class L1 data cache parameters

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 cache to the core per cycle: 1
- MSHR size: 8
- MSHR FIFO depth: 3
- DTLB size: 16

4.2 Experiments

To understand the effects of various parameters and evaluate the performance of prefetchers better, the 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, L2 cache is not yet present. A load hit in the L1 data cache would return the data back in 2 cycles. A miss will go the DRAM whose latency is typically over 100-200 cycles in a real-time system. To realize 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, 100 and 200 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. Two values of prefetch distance: 2 and 4, have been considered for the experiments 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 MSHR getting full and cache getting busy.
- **Prefetch throttling:** In the initial stages of the performance analysis, it was observed that most of the times MSHRs get full with prefetch requests causing the cache to put busy and affecting the performance of the processor. For all later 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 putting busy due to MSHRs getting full with the prefetch requests.

4.3 Results

The following are the statistics observed for the different prefetchers by running them with the directed tests mentioned in the previous chapter and various parameter values described in the previous section. The cycles column in the following tables represents the number of core cycles taken by the processor to finish the test execution.

Abbreviations used in tables:

PRG: Prefetch Requests Generated

PRE:Prefetch Requests Enqueued

PRD: Prefetch Requests Dropped

4.3.1 Next-line/Previous-line prefetcher

- **Additional latency = 0 cycles:**

Test	No prefetcher	Next-line/Previous-line prefetcher			
	Cycles	Cycles	PRG	PRE	PRD
prefetch_nl1	44171	43244	2029	412	385
prefetch_nl2	49487	49484	2028	1958	1920
prefetch_nl3	44171	43331	2029	396	375
prefetch_nl4	49487	49522	2028	1926	1895

Table 4.2: Statistics of no prefetcher performance vs next-line/previous-line prefetcher performance with no additional memory latency.

- **Additional latency = 100 cycles:**

Test	No prefetcher	Next-line/Previous-line prefetcher			
	Cycles	Cycles	PRG	PRE	PRD
prefetch_nl1	83749	83788	2030	638	617
prefetch_nl2	86341	86335	2028	1179	1163
prefetch_nl3	83749	83961	2030	619	600
prefetch_nl4	86341	86438	2030	1170	1154

Table 4.3: Statistics of no prefetcher performance vs next-line/previous-line prefetcher performance with additional memory latency of 100 cycles.

- **Additional latency = 200 cycles:**

Test	No prefetcher	Next-line/Previous-line prefetcher			
	Cycles	Cycles	PRG	PRE	PRD
prefetch_nl1	135541	135541	2029	360	334
prefetch_nl2	138143	138135	2029	1525	1499
prefetch_nl3	135541	135742	2029	351	327
prefetch_nl4	138143	138338	2030	1540	1515

Table 4.4: Statistics of no prefetcher performance vs next-line/previous-line prefetcher performance with additional memory latency of 200 cycles.

4.3.2 Stride prefetcher

- **Additional latency = 0 cycles, prefetch distance = 2:**

Test	No prefetcher	Stride prefetcher			
	Cycles	Cycles	PRG	PRE	PRD
prefetch_stride1	42099	42175	1901	1621	1363
prefetch_stride2	48631	49180	1689	1609	5
prefetch_stride3	43246	43246	0	0	0
prefetch_stride4	49483	49483	0	0	0
prefetch_saxpy_word	56159	55598	4323	1413	4

Table 4.5: Statistics of no prefetcher vs stride prefetcher performance for additional memory latency = 0 cycles, prefetch distance = 2.

- **Additional latency = 0 cycles, prefetch distance = 4:**

Test	No prefetcher	Stride prefetcher			
	Cycles	Cycles	PRG	PRE	PRD
prefetch_stride1	42099	42088	1830	1467	1052
prefetch_stride2	48631	48671	1323	1202	30
prefetch_stride3	43246	43246	0	0	0
prefetch_stride4	49483	49483	0	0	0
prefetch_saxpy_word	56159	55598	4315	1413	4

Table 4.6: Statistics of no prefetcher vs stride prefetcher performance for additional memory latency = 0 cycles, prefetch distance = 4.

- **Additional latency = 100 cycles, prefetch distance = 2:**

Test	No prefetcher	Stride prefetcher			
	Cycles	**Cycles**	**PRG**	**PRE**	**PRD**
prefetch_stride1	77554	79325	1916	1605	1195
prefetch_stride2	80922	84377	551	362	187
prefetch_stride3	83749	83749	0	0	0
prefetch_stride4	86357	86357	0	0	0
prefetch_saxpy_word	80798	66816	4200	2155	3

Table 4.7: Statistics of no prefetcher vs stride prefetcher performance for additional memory latency = 100 cycles, prefetch distance = 2.

- **Additional latency = 100 cycles, prefetch distance = 4:**

Test	No prefetcher	Stride prefetcher			
	Cycles	**Cycles**	**PRG**	**PRE**	**PRD**
prefetch_stride1	77554	78063	1854	1571	1195
prefetch_stride2	80922	82839	1082	756	187
prefetch_stride3	83749	83749	0	0	0
prefetch_stride4	86357	86357	0	0	0
prefetch_saxpy_word	80798	66816	4192	2151	3

Table 4.8: Statistics of no prefetcher vs stride prefetcher performance for additional memory latency = 100 cycles, prefetch distance = 4.

- **Additional latency = 200 cycles, prefetch distance = 2:**

Test	No prefetcher	Stride prefetcher			
	Cycles	**Cycles**	**PRG**	**PRE**	**PRD**
prefetch_stride1	123234	125211	961	664	407
prefetch_stride2	127165	166527	1961	1408	26
prefetch_stride3	135741	135741	0	0	0
prefetch_stride4	138362	138362	0	0	0
prefetch_saxpy_word	127165	85526	4288	1299	2

Table 4.9: Statistics of no prefetcher vs stride prefetcher performance for additional memory latency = 200 cycles, prefetch distance = 2.

- **Additional latency = 200 cycles, prefetch distance = 4:**

Test	No prefetcher	Stride prefetcher			
	Cycles	**Cycles**	**PRG**	**PRE**	**PRD**
prefetch_stride1	123234	123608	1833	1701	1499
prefetch_stride2	127165	138377	1113	856	30
prefetch_stride3	135741	135741	0	0	0
prefetch_stride4	138362	138362	0	0	0
prefetch_saxpy_word	127165	85783	4280	1229	2

Table 4.10: Statistics of no prefetcher vs stride prefetcher performance for additional memory latency = 200 cycles, prefetch distance = 4.

4.3.3 Register based line stride prefetcher

- **Additional latency = 0 cycles, prefetch distance = 2:**

Test	No prefetcher	Register line stride prefetcher			
	Cycles	**Cycles**	**PRG**	**PRE**	**PRD**
prefetch_linestride1	62751	60882	3486	301	244
prefetch_linestride2	70143	70088	3804	3771	3621
prefetch_linestride3	44171	43204	1749	16	4
prefetch_linestride4	70312	70304	3598	3323	2394
prefetch_linestride5	43259	43261	904	409	384
prefetch_linestride6	37127	37127	0	0	0
prefetch_saxpy_line	105435	85798	5932	4951	1944

Table 4.11: Statistics of no prefetcher vs register based version of line stride prefetcher performance for additional memory latency=0 cycles, prefetch distance=2.

- **Additional latency = 0 cycles, prefetch distance = 4:**

Test	No prefetcher	Register line stride prefetcher			
	Cycles	**Cycles**	**PRG**	**PRE**	**PRD**
prefetch_linestride1	62751	60861	3004	469	412
prefetch_linestride2	70143	70029	3561	3525	3386
prefetch_linestride3	44171	43193	1513	222	205
prefetch_linestride4	70312	70313	3285	3122	2386
prefetch_linestride5	43259	43277	42	34	11
prefetch_linestride6	37127	37127	0	0	0
prefetch_saxpy_line	105435	86413	5740	5364	2815

Table 4.12: Statistics of no prefetcher vs register based version of line stride prefetcher performance for additional memory latency=0 cycles, prefetch distance=4.

- **Additional latency = 100 cycles, prefetch distance = 2:**

Test	No prefetcher	Register line stride prefetcher			
	Cycles	Cycles	PRG	PRE	PRD
prefetch_linestride1	140794	138841	3387	410	349
prefetch_linestride2	140361	140346	3829	3403	3257
prefetch_linestride3	83728	83721	1720	330	314
prefetch_linestride4	140769	140810	3795	3397	3265
prefetch_linestride5	83843	83857	727	386	361
prefetch_linestride6	62783	62783	0	0	0
prefetch_saxpy_line	380610	231580	5943	5014	2372

Table 4.13: Statistics of no prefetcher vs register based line stride prefetcher performance for additional memory latency=100 cycles, prefetch distance=2.

- **Additional latency =100 cycles, prefetch distance = 4:**

Test	No prefetcher	Register line stride prefetcher			
	Cycles	Cycles	PRG	PRE	PRD
prefetch_linestride1	140794	138828	2920	616	563
prefetch_linestride2	140361	140411	3560	2758	2630
prefetch_linestride3	83728	83721	1478	328	314
prefetch_linestride4	140769	140832	3541	3083	2993
prefetch_linestride5	83843	83870	684	369	347
prefetch_linestride6	62783	63709	0	0	0
prefetch_saxpy_line	380610	213462	5751	5254	3574

Table 4.14: Statistics of no prefetcher vs register based line stride prefetcher performance for additional memory latency=100 cycles, prefetch distance=4.

- **Additional latency = 200 cycles, prefetch distance = 2:**

Test	No prefetcher	Register line stride prefetcher			
	Cycles	Cycles	PRG	PRE	PRD
prefetch_linestride1	240712	238756	3457	377	316
prefetch_linestride2	240278	240208	3828	3401	3258
prefetch_linestride3	135541	135532	1720	330	314
prefetch_linestride4	240669	240810	3795	3397	3265
prefetch_linestride5	135756	135770	727	386	361
prefetch_linestride6	93221	93221	0	0	0

Table 4.15: Statistics of no prefetcher vs register based line stride prefetcher performance for additional memory latency= 200 cycles, prefetch distance=2.

- **Additional latency = 200 cycles, prefetch distance = 4:**

Test	**No prefetcher**	**Register line stride prefetcher**			
	Cycles	**Cycles**	**PRG**	**PRE**	**PRD**
prefetch_linestride1	240712	238743	2969	609	555
prefetch_linestride2	240278	240321	3564	3131	3000
prefetch_linestride3	135541	135532	1478	328	314
prefetch_linestride4	240669	240832	3541	3083	2993
prefetch_linestride5	135756	135783	683	369	347
prefetch_linestride6	93221	93221	0	0	0

Table 4.16: Statistics of no prefetcher vs register based line stride prefetcher performance for additional memory latency=200 cycles, prefetch distance=4.

4.3.4 BRAM version of line stride prefetcher

- **Additional latency = 0 cycles, prefetch distance = 2:**

Test	**No prefetcher**	**BRAM line stride prefetcher**			
	Cycles	**Cycles**	**PRG**	**PRE**	**PRD**
prefetch_linestride1	62751	61300	1723	416	296
prefetch_linestride2	70143	70073	3804	3786	3641
prefetch_linestride3	44171	43204	796	168	159
prefetch_linestride4	70312	70302	3769	3683	3567
prefetch_linestride5	43259	43279	1492	679	661
prefetch_linestride6	37127	37127	0	0	0
prefetch_saxpy_line	105435	83595	5908	4450	1477

Table 4.17: Statistics of no prefetcher vs BRAM version of line stride prefetcher performance for additional memory latency=0 cycles, prefetch distance=2.

- **Additional latency = 0 cycles, prefetch distance = 4:**

Test	**No prefetcher**	**BRAM line stride prefetcher**			
	Cycles	**Cycles**	**PRG**	**PRE**	**PRD**
prefetch_linestride1	62751	60934	1520	126	53
prefetch_linestride2	70143	70021	2329	2322	2192
prefetch_linestride3	44171	43260	1372	52	34
prefetch_linestride4	70312	70302	3539	3440	3331
prefetch_linestride5	43259	43273	833	394	380
prefetch_linestride6	37127	37127	0	0	0
prefetch_saxpy_line	105435	84340	5681	4811	2294

Table 4.18: Statistics of no prefetcher vs BRAM version of line stride prefetcher performance for additional memory latency=0 cycles, prefetch distance=4.

- **Additional latency = 100 cycles, prefetch distance = 2:**

Test	No prefetcher	BRAM line stride prefetcher			
	Cycles	**Cycles**	**PRG**	**PRE**	**PRD**
prefetch_linestride1	140794	139471	868	263	155
prefetch_linestride2	140361	140390	1804	1410	1269
prefetch_linestride3	83728	83719	522	33	19
prefetch_linestride4	140769	140754	2774	2769	2653
prefetch_linestride5	83843	83858	1895	964	942
prefetch_linestride6	62783	62783	0	0	0
prefetch_saxpy_line	380610	223210	4906	3622	1101

Table 4.19: Statistics of no prefetcher vs BRAM version of line stride prefetcher performance for additional memory latency=100 cycles, prefetch distance=2.

- **Additional latency = 100 cycles, prefetch distance = 4:**

Test	No prefetcher	BRAM line stride prefetcher			
	Cycles	**Cycles**	**PRG**	**PRE**	**PRD**
prefetch_linestride1	140794	139137	854	321	258
prefetch_linestride2	140361	140408	3548	2284	2154
prefetch_linestride3	83728	83719	516	30	17
prefetch_linestride4	140769	140762	3539	3533	3428
prefetch_linestride5	83843	83870	1770	967	945
prefetch_linestride6	62783	62783	0	0	0
prefetch_saxpy_line	380610	210123	5716	5011	3324

Table 4.20: Statistics of no prefetcher vs BRAM version of line stride prefetcher performance for additional memory latency=100 cycles, prefetch distance=4.

- **Additional latency = 200 cycles, prefetch distance = 2:**

Test	No prefetcher	BRAM line stride prefetcher			
	Cycles	**Cycles**	**PRG**	**PRE**	**PRD**
prefetch_linestride1	240712	238743	2969	609	555
prefetch_linestride2	240278	240321	3564	3131	3000
prefetch_linestride3	135541	135532	1478	328	314
prefetch_linestride4	240669	240832	3541	3083	2993
prefetch_linestride5	135756	135783	683	369	347
prefetch_linestride6	93221	93221	0	0	0

Table 4.21: Statistics of no prefetcher vs BRAM version of line stride prefetcher performance for additional memory latency=200 cycles, prefetch distance=2.

- **Additional latency = 200 cycles, prefetch distance = 4:**

Test	No prefetcher	BRAM line stride prefetcher			
	Cycles	Cycles	PRG	PRE	PRD
prefetch_linestride1	240712	239250	1163	258	191
prefetch_linestride2	240278	240247	2547	1717	1592
prefetch_linestride3	135541	135532	516	29	17
prefetch_linestride4	240669	240662	3539	3533	3428
prefetch_linestride5	135756	135786	1770	967	945
prefetch_linestride6	93221	93221	0	0	0

Table 4.22: Statistics of no prefetcher vs BRAM version of line stride prefetcher performance for additional memory latency=200 cycles, prefetch distance=4.

4.4 Observations

- Next-line/Previous-line prefetcher: With next-line/previous-line prefetcher, the performance was expected to improve for tests ‘prefetch_nl1’ and ‘prefetch_nl2’ but it is observed that it improved only marginally and most of the prefetch requests that enqueued were dropped later as the MSHRs were all full. For the tests ‘prefetch_nl3’ and ‘prefetch_nl4’, the prefetcher degraded the performance like it was expected to: the processor took more cycles for test execution with the prefetcher.

- Stride prefetcher: For the test ‘prefetch_saxpy_word’ the stride prefetcher improves the performance significantly and the number of prefetches dropped is very low. This shows that the stride pattern was detected correctly and prefetches generated were timely for this type of test which has longer latency operations included. The performance was also expected to improve for tests ‘prefetch_stride1’ and ‘prefetch_stride2’ but it is observed that the performance has not improved and most of the prefetch requests that were enqueued were dropped later due to the demand requests. As expected, since there is no constant stride pattern for the tests ‘prefetch_stride3’ and ‘prefetch_stride4’, no prefetch requests are generated.

- Register based line stride prefetcher: Again, the saxpy test ‘prefetch_saxpy_line’ shows significant improvement in performance with the line stride prefetcher. For the tests ‘prefetch_linestride1’ to ‘prefetch_linestride5’ too it was expected that the line stride prefetcher would improve the performance but it is observed most of the prefetch requests that were enqueued were dropped later as all the MSHRs are full too soon and the cache puts out busy and thus the performance has not improved. It can also be observed that for tests that have store operations inside the loop (‘prefetch_linestride2’ and ‘prefetch_linestride4’), greater number of prefetches are generated compared to tests that do not have stores inside the loops (‘prefetch_linestride1’ and ‘prefetch_linestride3’).From the statistics of ‘prefetch_linestride6’ we see that the line stride prefetcher is unable to detect a line stride value that is less than 1. So it is suitable to use it when the stride value is more than 64 bytes or line stride is more than 1.

- BRAM version of line stride prefetcher: The statistics are similar to that of the register based version of line stride prefetcher. However due to the difference in the design, the number of prefetches is lesser in this case.

- Prefetch distance: It is observed that in most of the tests, there was not much change in the performance with different values (2 and 4) of prefetch distance. However, the test ‘prefetch_saxpy_line’ took fewer cycles in case of distance = 4 compared to distance = 2 with both register based and BRAM versions of line stride prefetcher when the additional memory latency was 100 cycles. It shows that in some contexts, having a higher value of distance can make the prefetches more timely and advantageous.

- Additional memory latency: In most of the tests, there is a marginal improvement in the relative performance with increased latency. With a large additional memory latency value (200 cycles), the test ‘prefetch_saxpy_line’ did not execute correctly and returned no results.

CHAPTER 5

CONCLUSION

From this project, we learn that prefetching, when working accurately and timely can improve the performance of the processor to some extent. However, several factors like the MSHR count and prefetch distance can affect the prefetcher performance. In the future version of I-Class where the cache can accept 2 requests per cycle, these prefetchers can be more beneficial as it may mitigate the problem of MSHRs getting full soon. Simple prefetchers like next-line/previous-line prefetcher can work well in certain situations while more advanced prefetchers like stride prefetchers perform better in other contexts. The advantage of PC-based stride prefetchers at the L1 level for I-Class is that the design is simpler to implement with fewer bytes of storage. Based on the requirements of the application that the processor is used for, a suitable prefetcher can be implemented and further modifications or optimizations can be done.

5.1 Future Work

Further work can be done in the performance analysis of the prefetchers implemented in terms of accuracy, coverage and timeliness. Detailed analysis can also involve aspects like prefetcher performance with higher MSHRs, interaction with memory dependence speculation and interaction with cache replacement policies. Timeliness or usefulness feedback from cache to prefetcher can also be thought of, to improvise the prefetchers. Parameter tuning can also be done to higher extents to find the suitable values of parameters for the best performance. There are also several other types of prefetchers that can be implemented like stream prefetchers, correlation based prefetchers, address based prefetchers, pre-computation or execution-based prefetchers etc. The implementation of instruction prefetchers, L2 cache prefetchers and prefetchers for heterogeneous multi-core processors can also be explored.

APPENDIX A

DIRECTED TESTS EXAMPLES

The following are excerpts of code from some of the directed tests mentioned in Chapter 3. The code section and beginning of data section of the tests are shown here.

A.1 test_prefetch_nl1

Test for next-line/previous-line prefetcher with a single loop where cumulative sum of data which are 64 bytes apart is calculated and stored at the end of the loop.

Assembly language code excerpt:

```
test_0:
   li t6, 0;       # loop index
   li t0, 0x40;    # loop limit
   la t1, tdat0;   # base addr
   li t3, 0;       # to store sum value

#loop
loop_0:
   lw t2, 0(t1);         # load data from address
   add t3, t3, t2;       # cumulative sum
   addi t1, t1, 64;      # next address
   addi t6, t6, 1;
   bne t6, t0, loop_0;
   la t1, tdat0;
   sw t3, 0(t1);         # store the sum
   j shakti_end;
```

```
# data section
.data

.align 6;
.global tohost;
tohost: .dword 0;
.align 6;
.global fromhost;
fromhost: .dword 0;

.align 6;
.global begin_signature;
begin_signature:
tdat0: .dword 0x0101010101010101
tdat1: .dword 0x2323232323232323
tdat2: .dword 0x4545454545454545
tdat3: .dword 0x6767676767676767
tdat4: .dword 0x8989898989898989
```

A.2 test_prefetch_saxpy_word

Test for stride prefetcher with a single SAXPY loop where elements of the array are at addresses which are 4 bytes apart.

Assembly language code excerpt:

```
# set up addresses
  li t0, 0x800; # num. elements/loop limit
  slli t1, t0, 2; # array size
  la t2, tdat0; # address of factor A
  flw ft0, 0(t2); # load A (0.5)
  la t2, tdat1; # base addr for array X
  add t3, t2, t1; # base addr for array Y
```

```
# initializing data values
  li t5, 0x3f000000; // To make y[i] = x[i]+0.5
  fmv.w.x ft1, t5;

  li t6, 0x3f800000; // To make x[i+1] = x[i]+1.0
  fmv.w.x ft2, t6;

# intializing X values
  li t4, 1; # loop index starting from 1 (first element already init
loop_0:
  flw ft3, 0(t2); //x[i]
  addi t2, t2, 4;
  addi t4, t4, 1;
  fadd.s ft3, ft3, ft2; // x[i+1] = x[i]+1.0
  fsw ft3, 0(t2); // store to x[i+1]
  bne t4, t0, loop_0;
  la t2, tdat1;          // Retrieving base addr of array X
  li t4, 0;              // Resetting loop index to 0
  j loop_1;

# initializing Y values
loop_1:
  flw ft3, 0(t2); // y[i]
  fadd.s ft3, ft3, ft1; // Initializing as y[i] = x[i]+0.5
  fsw ft3, 0(t3); // store to y[i]
  addi t3, t3, 4;
  addi t2, t2, 4;
  addi t4, t4, 1;
  bne t4, t0, loop_1;

  li t4, 0;              // Resetting loop index to 0
  la t2, tdat1;          // Retrieving base addr of array X
  add t3, t2, t1;        // Retrieving base addr of array Y
  j loop_2;

# loop Y = AX+Y
loop_2:
  flw ft1, 0(t2); # x[i]
  flw ft2, 0(t3); # y[i]
  fmul.s ft3, ft1, ft0; # (a * x[i])
  fadd.s ft4, ft3, ft2; # (result + y[i])
  fsw ft4, 0(t3); # store to y[i]
  addi t2, t2, 4;
  addi t3, t3, 4;
  addi t4, t4, 1;
  bne t4, t0, loop_2;
  j shakti_end;

# data section
.data

.pushsection .tohost,"aw",@progbits;
.align 6;
.global tohost;
tohost: .dword 0;
.align 6;
.global fromhost;
fromhost: .dword 0;
.popsection;

.align 4;
.global begin_signature;
begin_signature:
tdat0: .word 0x3f000000
tdat1: .word 0x3f800000
```

A.3 test_prefetch_linestride2

Test for line stride prefetcher with a positive stride (128 bytes) loop and a negative stride (128 bytes) loop. In each iteration of the loop, the cumulative sum up to that iteration is stored back in memory .

Assembly language code excerpt:

```
test_0:
   li t6, 0; # loop index
   li t0, 0x800; # loop limit
   la t1, tdat0; # base addr
   li t3, 0; # to store sum value
loop_0:
   lw t2, 0(t1);
   add t3, t3, t2;
   sw t3, 0(t1);
   addi t1, t1, 128;
   addi t6, t6, 1;
   bne t6, t0, loop_0;
   li t6, 0;
   j loop_1;
loop_1:
   lw t2, 0(t1);
   add t3, t3, t2;
   sw t3, 0(t1);
   addi t1, t1, -128;
   addi t6, t6, 1;
   bne t6, t0, loop_1;
   la t1, tdat0;
   sw t3, 0(t1);
   j shakti_end;
```

```
# data section
.data

.align 6;
.global tohost;
tohost: .dword 0;
.align 6;
.global fromhost;
fromhost: .dword 0;

.align 6;
.global begin_signature;
begin_signature:
tdat0: .dword 0x0101010101010101
tdat1: .dword 0x2323232323232323
tdat2: .dword 0x4545454545454545
tdat3: .dword 0x6767676767676767
tdat4: .dword 0x8989898989898989
tdat5: .dword 0xabababababababab
tdat6: .dword 0xcdcdcdcdcdcdcdcd
tdat7: .dword 0xefefefefefefefef
tdat8: .dword 0xaaaaaaaaaaaaaaaa
tdat9: .dword 0xbbbbbbbbbbbbbbbb
tdat10: .dword 0xcccccccccccccccc
```

REFERENCES

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

2. John L. Hennessey and David A. Patterson, *Computer Architecture: A Quantitative Approach*, Fifth edition, 2011.

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

4. Rishiyur S. Nikhil and Kathy Czeck, *BSV by Example*, Revision 2010.

5. Onur Mutlu, *Computer Architecture Lecture 24: Prefetching*, Notes from Carnegie Mellon University, Fall 2011.

6. Biswabandan Panda, *Modern Memory Systems Lecture 12: Hardware Prefetching*, Notes from Indian Institute of Technology Kanpur.

7. `https://en.wikipedia.org/wiki/Cache_prefetching`