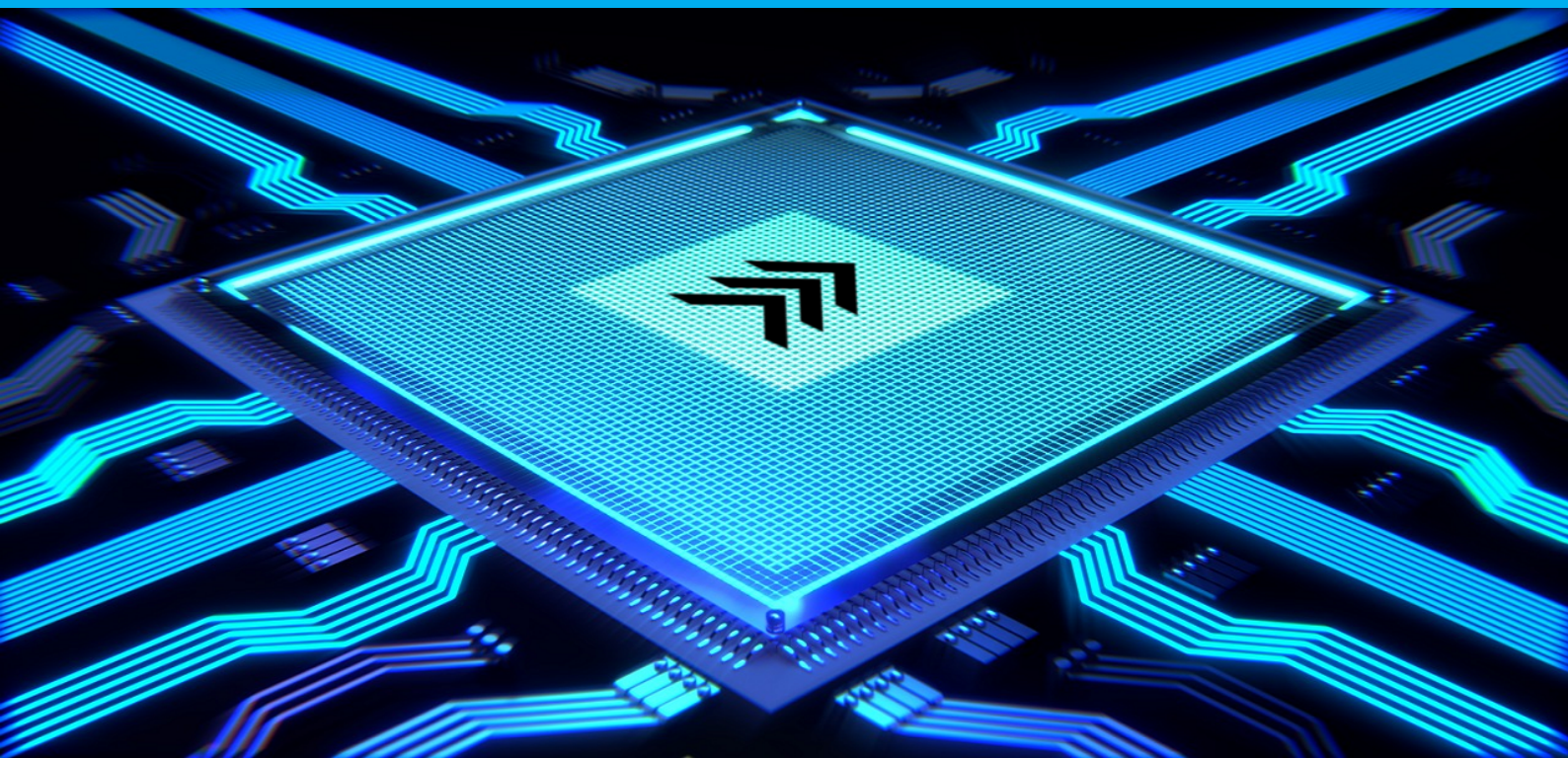


Arrow Prefetching

Prefetching Techniques for
High Performance Big Data
Processors using the Apache
Arrow Data Format

K.P. Metaxas

MSc Thesis Report
Embedded Systems



Arrow Prefetching

Prefetching Techniques for High Performance Big Data Processors using the Apache Arrow Data Format

by

K.P. Metaxas

to obtain the degree of Master of Science in Embedded Systems
at the Delft University of Technology,
to be defended publicly on Wednesday October 23, 2019 at 12:30 PM.

Student number: 4625439
Project duration: November 1, 2018 – October 23, 2019
Thesis committee: Prof. dr. ir. Z. Al-Ars, Quantum & Computer Engineering, TU Delft, Supervisor
Prof. dr. ir. P. Hofstee, Quantum & Computer Engineering, TU Delft
Prof. dr. ir. R. Van Leuken, Microelectronics, TU Delft
Ir. T. Ahmad Quantum & Computer Engineering, TU Delft

An electronic version of this thesis is available at <http://repository.tudelft.nl/>.

Abstract

As the digitisation of the world progresses at an accelerating pace, an overwhelming quantity of data from a variety of sources, of different types, organised in a multitude of forms or not at all, are subjected into diverse analytic processes for specific kinds of value to be extracted out of them. The aggregation of these analytic processes along with the software and hardware infrastructure implementing and facilitating them, comprise the field of *big data analytics*, which has distinct characteristics from normal data analytics. The systems executing the analysis, were found to exhibit performance weaknesses, significant front-end-bounding *Level 1 Data cache* miss rates specifically, for certain queries including, but not limited to, Natural Language Processing analytics. Based on this observation, investigations on whether, for data using the *Apache Arrow* format, its metadata could be used by certain prefetching techniques to improve cache behaviour and on the profile of the datasets and workloads which could profit from them, were conducted.

Architectural simulations of the execution of various microbenchmarks in an In-Order and an Out-Of-Order core were performed utilising different popular prefetchers and comparing the produced statistics with those of a perfect prefetcher. During this process, the performance of systems featuring the tested prefetchers was found to lag behind this of the perfect prefetcher system by a considerable margin for workloads featuring indirect indexing access patterns for datatypes of variable width. As a result, those patterns, which are readily available in the access mechanism of *Apache Arrow*, were identified as possible candidates for acceleration by more sophisticated prefetching techniques.

The first step in the investigation of such techniques was the introduction of a software prefetching scheme. This scheme, which is using prefetching and execution bursts, was embedded in a synthetic benchmark, specifically designed for the investigation of the effect the computation intensity of an algorithm has on the successful prefetching of indirect indexing access patterns. Its performance approached closely the performance produced by ideal prefetching for all the computation intensities except for the very low ones, with its resilience to possible variance in memory congestion being questionable though.

As hardware prefetching is more adaptable to runtime parameters like memory congestion, subsequently such a module was designed, with information about the memory locations and hierarchy of the *Apache Arrow columns*' buffers communicated to it by the software. Using this information, the prefetcher is able to distinguish between index and value accesses and prefetch the regular, iterative ones successfully. The hardware technique was able to almost match the performance of the ideal prefetcher for medium and high computation intensities and approach it, to the extent the bandwidth constraints allow, for the rest.

In terms of speed-up, the maximum attained of the hardware prefetcher over the top performing reference prefetchers was almost 7x and 4x for In-Order and Out-Of-Order execution respectively with the software prefetcher performing marginally worse in both cases. Those speed-ups were achieved for those algorithm's computation intensities which resulted in execution neither constrained by memory system's bandwidth limitations, nor by in-core computation resources' limitations.

Preface

“The saddest aspect of life right now is that science gathers knowledge faster than society gathers wisdom.”

Isaac Asimov

I am submitting this minor contribution with the hope that science moves towards a direction, in which the gathering of social wisdom will be favoured as much, if not more, as this of technical knowledge.

I would like to express my gratitude to professors Zaid Al-Ars and Peter Hofstee for their guidance throughout the course of this study and to my family and friends for their constant, multidimensional support.

K.P. Metaxas
Delft, October 18, 2019

Contents

List of Figures	ix
List of Tables	xvii
List of Acronyms	xix
1 Introduction	1
1.1 Context	1
1.1.1 Big Data Analytics Systems' Architecture	2
1.1.2 Big Data Management Frameworks	3
1.2 Problem	4
1.3 Research questions	5
1.4 Thesis outline	5
2 Background	7
2.1 Memory hierarchy	7
2.1.1 Main memory	8
2.1.2 Caches	8
2.2 Cache Prefetching	10
2.2.1 Hardware Prefetching	10
2.2.2 Software Prefetching	13
2.2.3 Hardware and Software Prefetching differences and trade-offs	14
2.3 Column-Oriented Databases.	14
2.4 Architectural Simulation	18
3 Experimental setup	21
3.1 System Configurations	21
3.2 Benchmarks	22
3.2.1 Matrix Operations.	23
3.2.2 Regular Stencil Algorithms	25
3.2.3 String Manipulation.	27
3.2.4 Indirect Accesses Synthetic Microbenchmark.	28
3.3 Simulation Statistics	29
4 Ideal Prefetcher	31
4.1 Ideal Prefetching Definition	31
4.2 Simulating Ideal Prefetching	32
4.3 Results	32
4.3.1 Reference Prefetchers' Tuning	32
4.3.2 Ideal Prefetcher Comparison	33
4.4 Discussion	41
5 Arrow Software Prefetching	45
5.1 Definition	45
5.2 Implementation	46
5.3 Results	48
5.4 Discussion	55

6	Arrow Hardware Prefetcher	57
6.1	Definition	57
6.2	Implementation	58
6.2.1	Targeted Access Patterns and Identification	58
6.2.2	Metadata Communication Mechanism	60
6.2.3	Algorithm	63
6.2.4	Configuration	66
6.3	Results	66
6.4	Discussion	70
7	Conclusions	73
7.1	Summary	73
7.2	Limitations	74
7.3	Suggestions for Future Research	75
A	Additional figures	77
A.1	Ideal Prefetcher.	77
A.2	Arrow Software Prefetching	96
	Bibliography	107

List of Figures

1.1	Illustration of the four Vs of <i>big data</i> as presented in [39]	1
1.2	The evolution of the <i>big data</i> chain as shown in [32]	2
1.3	The microarchitectural characteristics of the queries analysed in [46]	4
2.1	<i>Big data</i> systems memory hierarchy as shown in [23]	7
2.2	Direct-Mapped and 2-way Set Associative placement as shown in [8]	9
2.3	Illustration of prefetcher metrics and common access patterns as presented in [42]	11
2.4	Fraction of the total L1D cache misses the indirect access misses constitute as shown in [54]	12
2.5	Coverage, Accuracy and Latency of IMP from [54]	12
2.6	Row-Oriented and Column-Oriented organisation of a database as presented in [1]	15
2.7	Arrow common data layer as depicted in [1]	16
2.8	Record batch structure of the database shown in the figure on the right (based on [44] and [7])	17
2.9	Database representation example using the contents of <i>buffers</i> as presented in [44]	17
2.10	Model properties trade-offs as depicted in [24]	18
3.1	Matrix addition of two $m \times n$ matrices as shown in [14]	23
3.2	Matrix multiplication of one 4×2 matrix with one 2×3 matrix as presented in [15]	23
3.3	Example of calculating the first row of the matrix multiplication algorithm to exploit SIMD instructions	24
3.4	Von Neumann cell neighbourhood as depicted in [29]	25
3.5	Moore cell neighbourhood as depicted in [29]	25
3.6	Von Neumann 3D cell neighbourhood as depicted in [30]	26
3.7	Visualisation of capitalising the first letter of each word of an <i>Arrow array</i> of strings	28
3.8	List sizing conditions for the Synthetic Microbenchmark	28
3.9	Access patterns of the Synthetic Microbenchmark for <i>step</i> = 1	28
4.1	Execution time of systems featuring the tested prefetchers when matrix addition is executed on the In-Order core as their degree increases	35
4.2	Execution time of systems featuring the tested prefetchers when matrix addition is executed on the OOO core as their degree increases	35
4.3	Speed-up of the Ideal Prefetcher when matrix addition is executed on the In-Order core over the reference prefetchers as the input size increases	35
4.4	Speed-up of the Ideal Prefetcher when matrix addition is executed on the OOO core over the reference prefetchers as the input size increases	35
4.5	Execution time of systems featuring the tested prefetchers when matrix multiplication is executed on the In-Order core as their degree increases	36
4.6	Execution time of systems featuring the tested prefetchers when matrix multiplication is executed on the OOO core as their degree increases	36
4.7	Speed-up of the Ideal Prefetcher when matrix multiplication is executed on the In-Order core over the reference prefetchers as the input size increases	36
4.8	Speed-up of the Ideal Prefetcher when matrix multiplication is executed on the OOO core over the reference prefetchers as the input size increases	36
4.9	Execution time of systems featuring the tested prefetchers when a single iteration of Von Neumann Neighbourhood stencil is executed on the In-Order core as their degree increases	37
4.10	Execution time of systems featuring the tested prefetchers when a single iteration of Von Neumann Neighbourhood stencil is executed on the OOO core as their degree increases	37

4.11 Execution time of systems featuring the tested prefetchers when 10 iterations of the Von Neumann Neighbourhood stencil are executed on the In-Order core as their degree increases	37
4.12 Execution time of systems featuring the tested prefetchers when 10 iterations of the Von Neumann Neighbourhood stencil are executed on the OOO core as their degree increases	38
4.13 Speed-up of the Ideal Prefetcher when a single iteration of the Von Neumann Neighbourhood stencil is executed on the In-Order core over the reference prefetchers as the input size increases	38
4.14 Speed-up of the Ideal Prefetcher when a single iteration of the Von Neumann Neighbourhood stencil is executed on the OOO core over the reference prefetchers as the input size increases	38
4.15 Speed-up of the Ideal Prefetcher when a single iteration of the Moore Neighbourhood stencil is executed on the In-Order core over the reference prefetchers as the input size increases	38
4.16 Speed-up of the Ideal Prefetcher when a single iteration of the Moore Neighbourhood stencil is executed on the OOO core over the reference prefetchers as the input size increases	38
4.17 Execution time of systems featuring the tested prefetchers when a single iteration of 3D Von Neumann Neighbourhood stencil is executed on the In-Order core as the degree increases	39
4.18 Execution time of systems featuring the tested prefetchers when a single iteration of 3D Von Neumann Neighbourhood stencil is executed on the OOO core as the degree increases	39
4.19 Speed-up of the Ideal Prefetcher when a single iteration of the 3D Von Neumann Neighbourhood stencil is executed on the In-Order core over the reference prefetchers as the input size increases	39
4.20 Speed-up of the Ideal Prefetcher when a single iteration of the 3D Von Neumann Neighbourhood stencil is executed on the OOO core over the reference prefetchers as the input size increases	39
4.21 Speed-up of the Ideal Prefetcher when the word's first letter capitalisation algorithm is executed on the In-Order core over the reference prefetchers as the input size increases	40
4.22 Speed-up of the Ideal Prefetcher when the word's first letter capitalisation algorithm is executed on the OOO core over the reference prefetchers as the input size increases	40
4.23 Speed-up of the Ideal Prefetcher when the sentence's first letter capitalisation algorithm is executed on the In-Order core over the reference prefetchers as the input size increases	40
4.24 Speed-up of the Ideal Prefetcher when the sentence's first letter capitalisation algorithm is executed on the OOO core over the reference prefetchers as the input size increases	40
4.25 Speed-up of the Ideal Prefetcher when the paragraph's first letter capitalisation algorithm is executed on the In-Order core over the reference prefetchers as the input size increases	40
4.26 Speed-up of the Ideal Prefetcher when the paragraph's first letter capitalisation algorithm is executed on the OOO core over the reference prefetchers as the input size increases	40
5.1 Execution time of a system featuring the <i>stride</i> prefetcher when the synthetic benchmarks with 4+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase	49
5.2 Execution time of a system featuring the <i>stride</i> prefetcher when the synthetic benchmarks with 520+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase	49
5.3 Execution time of a system featuring the <i>stride</i> prefetcher when the synthetic benchmarks with 1000+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase	50
5.4 Execution time of a system featuring the <i>stride</i> prefetcher when the synthetic benchmarks with 4+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase	50

5.5	Execution time of a system featuring the <i>stride</i> prefetcher when the synthetic benchmarks with 520+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase	50
5.6	Execution time of a system featuring the <i>stride</i> prefetcher when the synthetic benchmarks with 1000+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase	51
5.7	Speed-up of the Ideal Prefetcher when the synthetic benchmark with the top performing SW prefetching configurations is executed on the In-Order core over the reference prefetchers as the computation intensity per load increases	52
5.8	Speed-up of the Ideal Prefetcher when the synthetic benchmark with the top performing SW prefetching configurations is executed on the OOO core over the reference prefetchers as the computation intensity per load increases	53
5.9	Percentage of total number of computation intensity cases, each combination of SW prefetching parameters performing within a 5% range of the top performer when executed in the In-Order core of the system featuring a <i>stride</i> prefetcher	53
5.10	Percentage of total number of computation intensity cases, each combination of SW prefetching parameters performing within a 5% range of the top performer when executed in the OOO core of the system featuring a <i>stride</i> prefetcher	53
5.11	Increase rate of the appearances of the SW prefetching parameters' combinations performing within a 5% range of the top performer when executed in the OOO core of the system featuring a <i>stride</i> prefetcher	54
5.12	Increase rate of the appearances of the SW prefetching parameters' combinations performing within a 5% range of the top performer when executed in the OOO core of the system featuring a <i>stream on miss</i> prefetcher	54
5.13	Increase rate of the appearances of the SW prefetching parameters' combinations performing within a 5% range of the top performer when executed in the OOO core of the system featuring a <i>stream on access</i> prefetcher	54
6.1	Structure of the <i>Columns Table</i> in the memory region reserved for the software to communicate the required metadata of the <i>Arrow columns</i>	61
6.2	Registration of the <i>columns</i> of the example database of figure 6.3 to the <i>Columns Table</i>	62
6.3	Example of the structure of the buffers of a database in <i>Apache Arrow</i> format	62
6.4	The structure of Internal Prefetcher Tables and the updating process using the information of the <i>Columns Table</i>	63
6.5	Speed-up of the <i>Arrow</i> Prefetcher for the different benchmarks executed on the In-Order core over the best of the reference prefetcher in each case	67
6.6	Speed-up of the <i>Arrow</i> Prefetcher for the different benchmarks executed on the OOO core over the best of the reference prefetcher in each case	67
6.7	Speed-up of the best performing HW-SW prefetching combination, the <i>Arrow</i> and the <i>Ideal</i> prefetchers when the synthetic benchmark is executed on the In-Order core over the best of the reference prefetchers in each case as the computation intensity per load increases	68
6.8	Speed-up of the best performing HW-SW prefetching combination, the <i>Arrow</i> and the <i>Ideal</i> prefetchers when the synthetic benchmark is executed on the In-Order core over the best of the reference prefetchers in each case as the computation intensity per load increases excluding the first value	69
6.9	Speed-up of the best performing HW-SW prefetching combination, the <i>Arrow</i> and the <i>Ideal</i> prefetchers when the synthetic benchmark is executed on the OOO core over the best of the reference prefetchers in each case as the computation intensity per load increases	69
A.1	Latency per cache block of systems featuring the tested prefetchers when matrix addition is executed on the In-Order core as their degree increases	77
A.2	Cache miss rate of systems featuring the tested prefetchers when matrix addition is executed on the In-Order core as their degree increases	78

A.3	Latency per cache block of systems featuring the tested prefetchers when matrix addition is executed on the OOO core as their degree increases	78
A.4	Cache miss rate of systems featuring the tested prefetchers when matrix addition is executed on the OOO core as their degree increases	79
A.5	Latency per cache block of systems featuring the reference prefetchers when matrix addition is executed on the In-Order core as the input size increases	79
A.6	Latency per cache block of systems featuring the reference prefetchers when matrix addition is executed on the OOO core as the input size increases	79
A.7	Cache miss rate of systems featuring the reference prefetchers when matrix addition is executed on the In-Order core as the input size increases	79
A.8	Cache miss rate of systems featuring the reference prefetchers when matrix addition is executed on the OOO core as the input size increases	79
A.9	Latency per cache block of systems featuring the tested prefetchers when matrix multiplication is executed on the In-Order core as their degree increases	80
A.10	Cache miss rate of systems featuring the tested prefetchers when matrix multiplication is executed on the In-Order core as their degree increases	80
A.11	Latency per cache block of systems featuring the tested prefetchers when matrix multiplication is executed on the OOO core as their degree increases	80
A.12	Cache miss rate of systems featuring the tested prefetchers when matrix multiplication is executed on the OOO core as their degree increases	81
A.13	Latency per cache block of systems featuring the reference prefetchers when matrix multiplication is executed on the In-Order core as the input size increases	81
A.14	Latency per cache block of systems featuring the reference prefetchers when matrix multiplication is executed on the OOO core as the input size increases	81
A.15	Cache miss rate of systems featuring the reference prefetchers when matrix multiplication is executed on the In-Order core as the input size increases	81
A.16	Cache miss rate of systems featuring the reference prefetchers when matrix multiplication is executed on the OOO core as the input size increases	81
A.17	Latency per cache block of systems featuring the tested prefetchers when a single iteration of the Von Neumann Neighbourhood stencil is executed on the In-Order core as their degree increases	82
A.18	Cache miss rate of systems featuring the tested prefetchers when a single iteration of the Von Neumann Neighbourhood stencil is executed on the In-Order core as their degree increases	82
A.19	Latency per cache block of systems featuring the tested prefetchers when a single iteration of the Von Neumann Neighbourhood stencil is executed on the OOO core as their degree increases	82
A.20	Cache miss rate of systems featuring the tested prefetchers when a single iteration of the Von Neumann Neighbourhood stencil is executed on the OOO core as their degree increases	83
A.21	Latency per cache block of systems featuring the tested prefetchers when 10 iterations of the Von Neumann Neighbourhood stencil are executed on the In-Order core as the degree increases	83
A.22	Cache miss rate of systems featuring the tested prefetchers when 10 iterations of the Von Neumann Neighbourhood stencil are executed on the In-Order core as the degree increases	84
A.23	Latency per cache block of systems featuring the tested prefetchers when 10 iterations of the Von Neumann Neighbourhood stencil are executed on the OOO core as the degree increases	84
A.24	Cache miss rate of systems featuring the tested prefetchers when 10 iterations of the Von Neumann Neighbourhood stencil are executed on the OOO core as the degree increases	84
A.25	Latency per cache block of systems featuring the reference prefetchers when a single iteration of the Von Neumann Neighbourhood stencil is executed on the In-Order core as the input size increases	85

A.26 Latency per cache block of systems featuring the reference prefetchers when a single iteration of the Von Neumann Neighbourhood stencil is executed on the OOO core as the input size increases	85
A.27 Cache miss rate of systems featuring the reference prefetchers when a single iteration of the Von Neumann Neighbourhood stencil is executed on the In-Order core as the input size increases	85
A.28 Cache miss rate of systems featuring the reference prefetchers when a single iteration of the Von Neumann Neighbourhood stencil is executed on the OOO core as the input size increases	85
A.29 Execution time of systems featuring the tested prefetchers when a single iteration of the Moore Neighbourhood stencil is executed on the In-Order core as the degree increases	85
A.30 Latency per cache block of systems featuring the tested prefetchers when a single iteration of the Moore Neighbourhood stencil is executed on the In-Order core as the degree increases	86
A.31 Cache miss rate of systems featuring the tested prefetchers when a single iteration of the Moore Neighbourhood stencil is executed on the In-Order core as the degree increases	86
A.32 Execution time of systems featuring the tested prefetchers when a single iteration of the Moore Neighbourhood stencil is executed on the OOO core as the degree increases	86
A.33 Latency per cache block of systems featuring the tested prefetchers when a single iteration of the Moore Neighbourhood stencil is executed on the OOO core as the degree increases	87
A.34 Cache miss rate of systems featuring the tested prefetchers when a single iteration of the Moore Neighbourhood stencil is executed on the OOO core as the degree increases	87
A.35 Latency per cache block of systems featuring the reference prefetchers when a single iteration of the Moore Neighbourhood stencil is executed on the In-Order core as the input size increases	87
A.36 Latency per cache block of systems featuring the reference prefetchers when a single iteration of the Moore Neighbourhood stencil is executed on the OOO core as the input size increases	87
A.37 Cache miss rate of systems featuring the reference prefetchers when a single iteration of the Moore Neighbourhood stencil is executed on the In-Order core as the input size increases	88
A.38 Cache miss rate of systems featuring the reference prefetchers when a single iteration of the Moore Neighbourhood stencil is executed on the OOO core as the input size increases	88
A.39 Speed-up of the Ideal Prefetcher when 10 iterations of the Moore Neighbourhood stencil are executed on the In-Order core over the reference prefetchers as the input size increases	88
A.40 Speed-up of the Ideal Prefetcher when 10 iterations of the Moore Neighbourhood stencil are executed on the OOO core over the reference prefetchers as the input size increases	88
A.41 Latency per cache block of systems featuring the reference prefetchers when 10 iterations of the Moore Neighbourhood stencil are executed on the In-Order core as the input size increases	88
A.42 Latency per cache block of systems featuring the reference prefetchers when 10 iterations of the Moore Neighbourhood stencil are executed on the OOO core as the input size increases	88
A.43 Cache miss rate of systems featuring the reference prefetchers when 10 iterations of the Moore Neighbourhood stencil are executed on the In-Order core as the input size increases	89
A.44 Cache miss rate of systems featuring the reference prefetchers when 10 iterations of the Moore Neighbourhood stencil are executed on the OOO core as the input size increases	89
A.45 Latency per cache block of systems featuring the tested prefetchers when a single iteration of 3D Von Neumann Neighbourhood stencil is executed on the In-Order core as the degree increases	89
A.46 Cache miss rate of systems featuring the tested prefetchers when a single iteration of 3D Von Neumann Neighbourhood stencil is executed on the In-Order core as the degree increases	89

A.47 Latency per cache block of systems featuring the tested prefetchers when a single iteration of 3D Von Neumann Neighbourhood stencil is executed on the OOO core as the degree increases	90
A.48 Cache miss rate of systems featuring the tested prefetchers when a single iteration of 3D Von Neumann Neighbourhood stencil is executed on the OOO core as the degree increases	90
A.49 Latency per cache block of systems featuring the reference prefetchers when a single iteration of the 3D Von Neumann Neighbourhood stencil is executed on the In-Order core as the input size increases	90
A.50 Latency per cache block of systems featuring the reference prefetchers when a single iteration of the 3D Von Neumann Neighbourhood stencil is executed on the OOO core as the input size increases	90
A.51 Cache miss rate of systems featuring the reference prefetchers when a single iteration of the 3D Von Neumann Neighbourhood stencil is executed on the In-Order core as the input size increases	91
A.52 Cache miss rate of systems featuring the reference prefetchers when a single iteration of the 3D Von Neumann Neighbourhood stencil is executed on the OOO core as the input size increases	91
A.53 Speed-up of the Ideal Prefetcher when 10 iterations of the 3D Von Neumann Neighbourhood stencil are executed on the In-Order core over the reference prefetchers as the input size increases	91
A.54 Speed-up of the Ideal Prefetcher when 10 iterations of the 3D Von Neumann Neighbourhood stencil are executed on the OOO core over the reference prefetchers as the input size increases	91
A.55 Latency per cache block of systems featuring the reference prefetchers when 10 iterations of the 3D Von Neumann Neighbourhood stencil are executed on the In-Order core as the input size increases	91
A.56 Latency per cache block of systems featuring the reference prefetchers when 10 iterations of the 3D Von Neumann Neighbourhood stencil are executed on the OOO core as the input size increases	91
A.57 Cache miss rate of systems featuring the reference prefetchers when 10 iterations of the 3D Von Neumann Neighbourhood stencil are executed on the In-Order core as the input size increases	92
A.58 Cache miss rate of systems featuring the reference prefetchers when 10 iterations of the 3D Von Neumann Neighbourhood stencil are executed on the OOO core as the input size increases	92
A.59 Latency per cache block of systems featuring the reference prefetchers when the word's first letter capitalisation algorithm is executed on the In-Order core as the input size increases	92
A.60 Latency per cache block of systems featuring the reference prefetchers when the word's first letter capitalisation algorithm is executed on the OOO core as the input size increases	92
A.61 Cache miss rate of systems featuring the reference prefetchers when the word's first letter capitalisation algorithm is executed on the In-Order core as the input size increases	93
A.62 Cache miss rate of systems featuring the reference prefetchers when the word's first letter capitalisation algorithm is executed on the OOO core as the input size increases	93
A.63 Latency per cache block of systems featuring the reference prefetchers when the sentence's first letter capitalisation algorithm is executed on the In-Order core as the input size increases	93
A.64 Latency per cache block of systems featuring the reference prefetchers when the sentence's first letter capitalisation algorithm is executed on the OOO core as the input size increases	93
A.65 Cache miss rate of systems featuring the reference prefetchers when the sentence's first letter capitalisation algorithm is executed on the In-Order core as the input size increases	94
A.66 Cache miss rate of systems featuring the reference prefetchers when the sentence's first letter capitalisation algorithm is executed on the OOO core as the input size increases	94

A.67 Latency per cache block of systems featuring the reference prefetchers when the paragraph's first letter capitalisation algorithm is executed on the In-Order core as the input size increases	94
A.68 Latency per cache block of systems featuring the reference prefetchers when the paragraph's first letter capitalisation algorithm is executed on the OOO core as the input size increases	94
A.69 Cache miss rate of systems featuring the reference prefetchers when the paragraph's first letter capitalisation algorithm is executed on the In-Order core as the input size increases	95
A.70 Cache miss rate of systems featuring the reference prefetchers when the paragraph's first letter capitalisation algorithm is executed on the OOO core as the input size increases	95
A.71 Execution time of a system featuring the <i>stream on miss</i> prefetcher when the synthetic benchmarks with 4+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase	96
A.72 Execution time of a system featuring the <i>stream on access</i> prefetcher when the synthetic benchmarks with 4+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase	96
A.73 Execution time of a system featuring the <i>stride</i> prefetcher when the synthetic benchmarks with 240+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase	97
A.74 Execution time of a system featuring the <i>stream on miss</i> prefetcher when the synthetic benchmarks with 240+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase	97
A.75 Execution time of a system featuring the <i>stream on access</i> prefetcher when the synthetic benchmarks with 240+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase	97
A.76 Execution time of a system featuring the <i>stream on miss</i> prefetcher when the synthetic benchmarks with 520+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase	98
A.77 Execution time of a system featuring the <i>stream on access</i> prefetcher when the synthetic benchmarks with 520+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase	98
A.78 Execution time of a system featuring the <i>stride</i> prefetcher when the synthetic benchmarks with 760+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase	98
A.79 Execution time of a system featuring the <i>stream on miss</i> prefetcher when the synthetic benchmarks with 760+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase	99
A.80 Execution time of a system featuring the <i>stream on access</i> prefetcher when the synthetic benchmarks with 760+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase	99
A.81 Execution time of a system featuring the <i>stream on miss</i> prefetcher when the synthetic benchmarks with 1000+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase	99
A.82 Execution time of a system featuring the <i>stream on access</i> prefetcher when the synthetic benchmarks with 1000+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase	100
A.83 Execution time of a system featuring the <i>stream on miss</i> prefetcher when the synthetic benchmarks with 4+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase	100
A.84 Execution time of a system featuring the <i>stream on access</i> prefetcher when the synthetic benchmarks with 4+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase	100
A.85 Execution time of a system featuring the <i>stride</i> prefetcher when the synthetic benchmarks with 240+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase	101

A.86 Execution time of a system featuring the <i>streamOnMiss</i> prefetcher when the synthetic benchmarks with 240+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase	101
A.87 Execution time of a system featuring the <i>streamOnAcc</i> prefetcher when the synthetic benchmarks with 240+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase	101
A.88 Execution time of a system featuring the <i>streamOnMiss</i> prefetcher when the synthetic benchmarks with 520+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase	102
A.89 Execution time of a system featuring the <i>streamOnAcc</i> prefetcher when the synthetic benchmarks with 520+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase	102
A.90 Execution time of a system featuring the <i>stride</i> prefetcher when the synthetic benchmarks with 760+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase	102
A.91 Execution time of a system featuring the <i>streamOnMiss</i> prefetcher when the synthetic benchmarks with 760+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase	103
A.92 Execution time of a system featuring the <i>streamOnAcc</i> prefetcher when the synthetic benchmarks with 760+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase	103
A.93 Execution time of a system featuring the <i>streamOnMiss</i> prefetcher when the synthetic benchmarks with 1000+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase	103
A.94 Execution time of a system featuring the <i>streamOnAcc</i> prefetcher when the synthetic benchmarks with 1000+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase	104
A.95 Percentage of total number of computation intensity cases, each combination of SW prefetching parameters performing within a 5% range of the top performer when executed in the In-Order core of the system featuring a <i>stream on miss</i> prefetcher	104
A.96 Percentage of total number of computation intensity cases, each combination of SW prefetching parameters performing within a 5% range of the top performer when executed in the In-Order core of the system featuring a <i>stream on miss</i> prefetcher	104
A.97 Percentage of total number of computation intensity cases, each combination of SW prefetching parameters performing within a 5% range of the top performer when executed in the OOO core of the system featuring a <i>stream on miss</i> prefetcher	105
A.98 Percentage of total number of computation intensity cases, each combination of SW prefetching parameters performing within a 5% range of the top performer when executed in the OOO core of the system featuring a <i>stream on miss</i> prefetcher	105

List of Tables

2.1	Estimation of average latencies for the different levels of memory hierarchy expressed in terms of time units and number of cycles for a 4GHz CPU (data taken from [47]) . . .	8
3.1	System configurations	22
5.1	Top performing combinations of parameters in the form <i>degree, distance</i> for the systems featuring the reference prefetchers as the computation intensity per load increases in the In-Order core	51
5.2	Top performing combinations of parameters in the form <i>degree, distance</i> for the systems featuring the reference prefetchers as the computation intensity per load increases in the OOO core	52

List of Acronyms

Acronym	Definition
ICT	Information and Communications Technology
SoC	System on Chip
FP	Floating Point
CPU	Central Processing Unit
L1I	Level 1 Instruction
TLB	Translation Lookaside Buffer
IPC	Instructions Per Cycle
LLC	Last Level Cache
TLP	Thread Level Parallelism
NLP	Natural Language Processing
SIMD	Single Instruction Multiple Data
L1D	Level 1 Data
SDRAM	Synchronous Dynamic Random Access Memory
SRAM	Static Random Access Memory
FIFO	First In First Out
PC	Program Counter
IPD	Indirect Pattern Detection
IMP	Indirect Memory Prefetching
OOO	Out-Of-Order
OS	Operating System
ISA	Instruction Set Architecture
SW	Software
HW	Hardware

Introduction

Since the introduction of modern computing systems, the scientific and industrial community are in a constant struggle of co-optimising the hardware and software architectures and the interface between them in pursuit of ever-growing performance. The thesis, which is documented in this report, attempts to contribute to this purpose, with its context, *big data* and related prior work, being documented in section 1.1. The problem of identifying architectural improvements targeting specifically *big data analytics* applications is analysed subsequently in 1.2, followed by a concrete statement of the research questions in 1.3 and the outline of the rest of the report in 1.4.

1.1. Context

In the ever-expanding "digital dimension" of the world, an exponentially-growing, vast amount of data from various domains and sources, such as sensors, users, companies' databases and systems etc, is produced and stored, pending a specialised analysis for value to be extracted out of them [32]. The total amount accumulated over the years is estimated to reach 40 zettabytes by 2020 [28]. Except for their volume, this accumulation of data, referred to by the term "*big data*" in the past two decades [38], exhibits different characteristics in terms of format and structure, or the lack of thereof, and frequently their analysis should be performed in real time. Such a combination of sheer *Volume* of data, varying in structure (*Variety*) and quality (*Veracity*) with tight time constraints (*Velocity*) for their analysis, exceeds the capabilities of the typical systems and software tools. These features, commonly referred as the "four Vs" (illustrated in figure 1.1), along with the requirements stemming from them, define a new field. In it, the different techniques for the acquisition, transformation and evaluation of *big data*, as well as the algorithms and the libraries implementing those techniques and their distribution across the processing systems, usually clusters of computation units, are incorporated. Both the science, as defined by the study and formalisation of those techniques, and the frameworks evolve with the time along with the needs of the fields contributing the data and the technology respectively [32]. This evolution of the *big data* chain with time is depicted in figure 1.2.

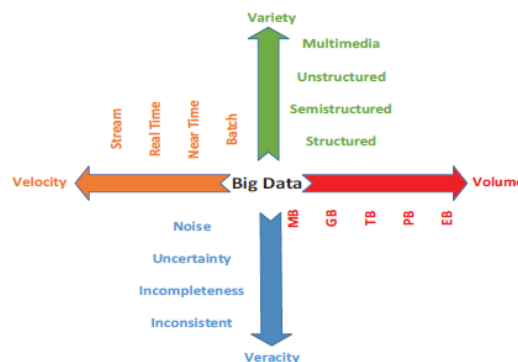


Figure 1.1: Illustration of the four Vs of *big data* as presented in [39]

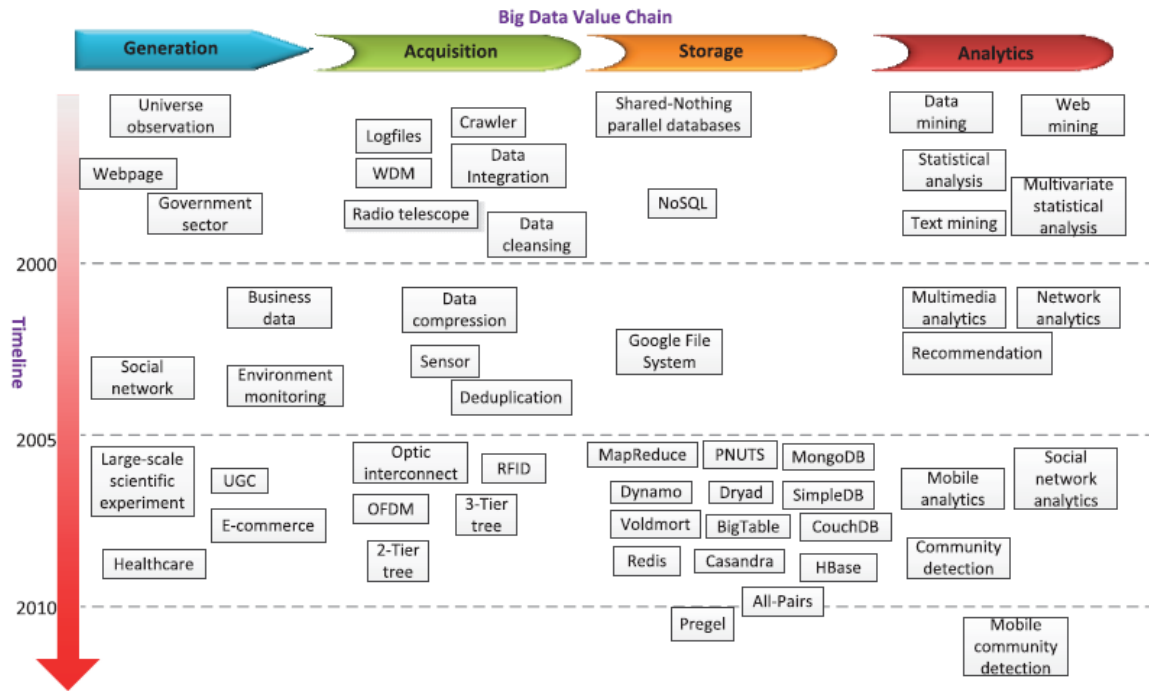


Figure 1.2: The evolution of the *big data* chain as shown in [32]

Having very diverse datasets, in terms of origin and properties, as inputs and aiming to extract from them very different conclusions of value, the field of *big data analytics* is very diverse as well. Most of those applications can be grouped, based on the analytical methods they implement, in six technical areas: "structured data analytics, text analytics, multimedia analytics, web analytics, network analytics and mobile analytics". For a general definition covering all those areas to be given, it can be said that data analysis utilises domain-specific analytical methods and tools to "inspect, transform and model the data" with the goal to extract value out of them [32]. Because of the variety of areas they cover, describing specific examples or even subcategories of data analytics methods and algorithms deviates from the scope of the current report. Those, used as a mean to test the techniques to be presented in the next chapters, will be described in section 3.2.

1.1.1. Big Data Analytics Systems' Architecture

Due to the aforementioned properties of the field, a need to revisit the prevalent concepts behind the systems' architecture and data frameworks emerged, aiming to produce systems able to accommodate efficiently such an increased demand in computational power. Towards this direction, several performance and (micro)architectural studies have been carried out using different processing cores to run benchmarks advocated to be representative of the *big data* workloads, in order to characterise them, identify their bottlenecks and evaluate the performance and energy efficiency advantages of the leading (micro)architectures of the industry. It needs to be noted that for the high-performance systems used by the industry for *big data*, the power efficiency is becoming increasingly important as the ICT ecosystem is projected to exceed 20% of the global electricity consumption by 2030 with the data centres being responsible for one-third of it [33].

A citation of a representative sample of the studies mentioned above follows, in order to form an adumbration of the (micro)architectural features a *big data* platform should incorporate in order to sustain high-performance and be energy efficient. The first study, "ARM Wrestling with *big data*: A Study of Commodity ARM64 Server for Big Data Workloads" [34], is focusing on a performance and energy efficiency comparison of an ARM64 SoC powered server manufactured by AMD with a native AMD x64 one, for various *big data* workloads including Intel's benchmark suite *HiBench*. The results show no favourite when it comes to integer performance, while for floating point operations the more powerful shared FP unit of the x64 chip results in better performance for the parts of the workloads exhibiting low data parallelism, with the field becoming more even when the data parallelism increases and the ARM chip is able to use all of its multiple FP units. Nevertheless, when it comes to energy consumption

and efficiency, the ARM CPU confirms the company's reputation of producing low power cores by a comfortable margin, leaving the choice between the two architectural approaches open to the specific requirements of each industry.

In "System and architecture level characterisation of big data applications on big and little core server architectures" [39], a system and microarchitecture analysis of several *big data* applications was performed on a server platform featuring a big high performance *Intel Xeon* core and on one featuring a little low power *Intel Atom* core. With both cores sharing the same architecture with the *Intel 64* instruction set, the focus was set on the microarchitectural level and its impact in correlation with the characteristics of the workloads. While on the performance metrics, the dominance in terms of resources of the big core renders it superior, when it comes to efficiency there is no clear winner as the results vary with the type, with both the computational and the I/O intensity of the application run, and with the data size. Several improvements could be performed targeting performance flaws which affect both types of cores, such as the TLB miss overhead, the L1I cache miss rate and the branch prediction. In addition, for the little core, the refinements needed to tailor it to the characteristics of the *big data* programs are greater in number. Despite their performance shortcomings, the power efficiency for various *big data* workloads leaves room for the little cores to be considered as candidates for the *big data* processing infrastructure.

Based on the above analysis, which was focused on general performance and energy metrics of the different representative chips, there is no clear favourite for the *big data* infrastructure among the prevalent architecture and microarchitecture instances, as they exhibit different strengths and thus, are suitable for different use cases and requirements. Consequently, an alternative approach should be followed to single out specific architectural features, the improvement of which could lead to better performance. Such an approach was followed by "Amdahl's Law in Big Data Analytics: Alive and Kicking in TPCx-BB (BigBench)" [46] study, which is using more specific performance metrics able to describe better the nature and the characteristics of typical *big data* applications. With its results, produced by executing the whole *BigBench* suite sequentially, the popular assumption that *big data* analysis scale-out is only limited by the resources was debunked and the diversity among the various queries in terms of computation intensity, memory behaviour and parallelism was showcased. For the computation aspect, the queries were found to be mostly backend bound with healthy branch prediction rates, though with significant variation in IPC. For the memory, while the bandwidth and the LLC size requirements were over-provisioned, a notable number of cache misses in several queries, especially affecting the L1D cache, was recorded. The most counter-intuitive observation though is the limited thread level parallelism exhibited by some of the queries, failing to exploit all the available cores which spent a significant percentage of the execution time in the halted state. The natural language processing queries especially exhibited low TLP in general, while many of the rest scaled-out well on the available cores in the initial part of the map and reduce phases, but in the end had long converging parts in which only a few cores were active.

1.1.2. Big Data Management Frameworks

Except for the systems' architecture, *big data* evolution triggered a revision of the data organisation and formats. Data management frameworks aim to organise data in an optimal way for processing and analysis [32]. One specific case towards this direction, which among others specifically aims for better cache behaviour for *big data analytics* is *Apache Arrow*. According to its website [1], Arrow "specifies a standardized language-independent columnar memory format for flat and hierarchical data, organized for efficient analytic operations on modern hardware". More specifically, it enables the creation of immutable objects, which can be shared between different applications, development platforms and languages, containing the data stored in an efficient for streaming, columnar memory layout, as well as metadata describing the hierarchy, the structure and the data types. A trivial example of data analysis, showcasing the benefits from memory access optimisation Arrow yields, can be found in the personal blog of professor Daniel Abadi [19]. In this example, due to the in-memory columnar format, searching a particular attribute (*column*) of a database for a specific value could be optimised by the compiler to use SIMD operations resulting in a total speed-up of almost 4x scaling-out fully to the length of the available SIMD registers.

Other frameworks implementing columnar formats exist, but mostly as a mean of optimised storage, like *Apache Parquet*, *Google BigTable*, *Apache Cassandra*, *Apache ORC* and *Apache HBase* [44] [32]. *Arrow*, on the other hand, is an in-memory representation able to interact with all the above frameworks

when it comes to storage, as well as with other programs and programming languages [44]. Thus, despite their common purpose, this of optimising the execution of analytics tasks, the difference in granularity commands for different properties and results in different benefits. A more extensive presentation of Apache Arrow follows in section 2.3.

1.2. Problem

In the previous chapter, several studies were presented attempting to escape the prevalent paradigm of treating the *big data* workloads simply as programs with large data inputs. Instead, it was tried to characterise them and identify specific (micro)architectural features *big data* processing systems should include in order to achieve the best performance and energy efficiency. If such features could be designed based on or combined with specialised data formats like *Apache Arrow*, then such a solution would be tailored to the needs of the field of *big data analytics*.

From the *BigBench* study [46], it can be deduced that since a notable percentage of the *big data* workloads exhibit, in certain phases or in general, limited thread-level parallelism, the single-core performance is still significant. Such a conclusion is supported also by the results for the FP-intensive parts in [34] for which the x64 architecture showed superior performance due to the more capable FP unit and the extent of the non-parallelisable parts. Since most of the workloads were found to be mostly back-end bound, the obvious solution would be to strengthen the execution units, a feat which is essentially limited by the semiconductor technology and the performance/energy trade-off. An alternative path could be chosen based on the observation that the NLP queries (q10, q18, q19 and q27 in figure 1.3), which sustain low TLP throughout their execution, exhibit also a large number of L1D cache misses and their front-end to back-end bound cycles ratio is more balanced. As a result, due to the percentage of the front-end bound cycles, a possible reduction in the L1D cache misses could have a measurable impact at least in the single-core performance which in turn, because of the limited TLP, would increase the overall performance.

Metric	Q01	Q02	Q03	Q04	Q05	Q06	Q07	Q08	Q09	Q10	Q11	Q12	Q13	Q14	Q15
CPI	1.38	0.757	0.826	0.783	1.52	1.32	1.13	0.903	1.99	1.08	1.36	0.980	1.32	1.46	1.18
Front-End Bound Cycles (%)	13.3	16.1	16.0	16.3	13.2	13.9	13.6	15.8	12.6	15.2	12.9	15.8	13.6	12.6	14.0
Back-End Bound Cycles (%)	60.9	45.8	48.6	47.2	63.2	60.3	57.5	51.2	66.2	46.4	62.5	53.0	60.9	63.6	58.3
Branch Mispredict Rate (%)	1.94	0.819	0.629	0.917	0.409	1.21	1.83	1.12	0.301	2.43	2.08	0.721	1.57	1.24	1.78
L1D MPKI	16.0	9.00	10.2	11.1	10.8	17.0	15.4	11.8	5.69	21.2	17.1	6.51	17.4	12.3	17.2
L2 MPKI	12.4	3.87	3.58	3.85	4.47	12.3	11.4	6.40	3.92	17.1	12.7	4.68	11.7	9.13	11.6
LLC MPKI	2.27	0.731	0.674	0.678	1.21	3.73	1.96	1.19	1.05	1.26	2.43	0.914	2.54	1.96	2.20
Memory Bandwidth (MB/s)	7640	6100	5660	6070	5610	25600	4720	7490	4840	3130	12200	5110	14800	10100	6340
IO Bandwidth (MB/s)	52.1	119	89.8	111	7.75	34.9	36.0	54.7	45.8	21.5	55.0	36.8	19.5	34.5	14.2

Metric	Q16	Q17	Q18	Q19	Q20	Q21	Q22	Q23	Q24	Q25	Q26	Q27	Q28	Q29	Q30
CPI	1.04	1.21	0.885	1.09	1.45	1.06	0.970	1.34	1.38	1.42	1.42	1.61	1.17	1.74	0.786
Front-End Bound Cycles (%)	15.9	13.5	14.1	15.0	12.9	14.5	13.8	13.6	13.3	13.5	13.0	12.1	14.7	12.8	16.4
Back-End Bound Cycles (%)	53.9	59.0	49.1	48.6	62.6	55.6	54.7	60.6	61.5	62.0	62.4	64.1	54.9	64.8	46.9
Branch Mispredict Rate (%)	1.19	2.01	1.64	2.51	1.38	1.70	1.61	1.88	2.03	1.25	0.938	1.93	2.71	0.507	0.689
L1D MPKI	12.2	18.2	14.1	22.7	13.9	15.1	12.6	15.3	16.8	14.3	12.8	16.1	17.9	5.58	8.85
L2 MPKI	9.43	12.3	11.8	17.8	10.2	10.1	9.95	13.2	13.0	10.2	7.32	12.2	15.6	4.46	3.54
LLC MPKI	2.10	2.24	2.16	1.65	2.26	1.75	1.79	2.49	2.31	2.30	1.57	2.49	2.56	1.30	0.705
Memory Bandwidth (MB/s)	13800	5540	12100	2560	9330	8400	9410	15400	10800	11400	4560	6340	4870	6150	6330
IO Bandwidth (MB/s)	274	41.9	23.8	22.9	28.3	115	53.2	27.9	19.9	17.5	21.8	7.91	67.4	26.2	90.7

Figure 1.3: The microarchitectural characteristics of the queries analysed in [46]

As Arrow was proven to have a therapeutic influence for the cache behaviour of *big data analytics*, a possible microarchitectural improvement could build on top of that to produce even better results mainly for those workloads, the profile of which shows that could profit from such a feature, like the NLP queries. Since Arrow's specification already optimises the data format for analytics workloads, a further attempt towards this direction would yield very limited acceleration, if any. Though the metadata, in which important information about the properties of the data is coded in a compact and efficient way, could be exploited to render the hardware aware of the structure and the hierarchy of the data in order to be able to better identify and optimise the patterns based on which they are accessed. Since the target is the curtailment of cache misses, a prefetching scheme aware of this information would be the intuitive path to be followed.

1.3. Research questions

Based on the above analysis, the problems it identified and the path chosen to tackle the challenge in the bigger picture, which is to increase the performance and energy efficiency of the execution of *big data analytics* workloads in the processing systems, the following questions emerge:

1. Which access patterns the traditional prefetchers fail to identify, while there is exploitable information about them in the *Apache Arrow* objects' metadata and which categories of datasets use such patterns the most?
2. Can the information contained in the metadata of *Apache Arrow* objects be exploited by a prefetching scheme, to improve the cache behaviour and further accelerate the execution of *big data analytics* workloads?
3. What is the profile of algorithms to benefit the most from prefetching techniques exploiting the metadata of *Apache Arrow* objects?

1.4. Thesis outline

Addressing these research questions, required the definition of the involved topics as well as studying previous work on them. This process is documented in chapter 2 starting from the more general concepts and progressing to the more specific elements used throughout this study. Subsequently, the configuration of those elements based on the targeted systems along with the benchmarks used to quantify their performance are analysed in chapter 3. The initial step of evaluating the possibility for improvement of the memory behaviour with the use of a more sophisticated prefetching scheme through the definition of an ideal prefetching behaviour is presented in chapter 4 and based on the findings regarding which specific types of access patterns are to benefit the most, a software and a hardware prefetching solution are proposed in chapters 5 and 6 respectively. Finally, the conclusions produced during the above steps are presented in chapter 7 along with the limitations, which arose during their course, and suggestions on how this study could be continued in the future.

2

Background

The purpose of this chapter is to provide a short description of the concepts comprising or stemming from the research questions introduced in chapter 1. The caches, which are parts of the memory hierarchy in modern computing systems and the prefetching techniques attempting to exploit their fast access times will be outlined in sections 2.1 and 2.2 respectively. *Apache Arrow*, the data format and the columnar formats in general, the exploitation of the metadata of which was investigated during this study, will be presented in section 2.3. Finally, the system simulator used to evaluate the cache behaviour and the introduced performance improvements is explained in section 2.4.

2.1. Memory hierarchy

As it was described in section 1.1, the volume of data worldwide is growing rapidly. Consequently, the applications performing analytics tasks for *big data* need to access an ever-growing data space as fast as possible in order to achieve realistic execution times. Though, the requirements of having very large memory able to sustain low latency and high bandwidth with uniform access times at a reasonable cost are conflicting and unattainable using the current technology. To be more specific, larger memories require more time or much more hardware (larger chip area) to determine the accessed location and respond, while fast memories are far more expensive in terms of implementation costs (cell technology, frequency) when it comes to latency and in terms of area (more hardware used in parallel), interconnection (ports) and implementation (control logic, frequency) costs when it comes to bandwidth [43].

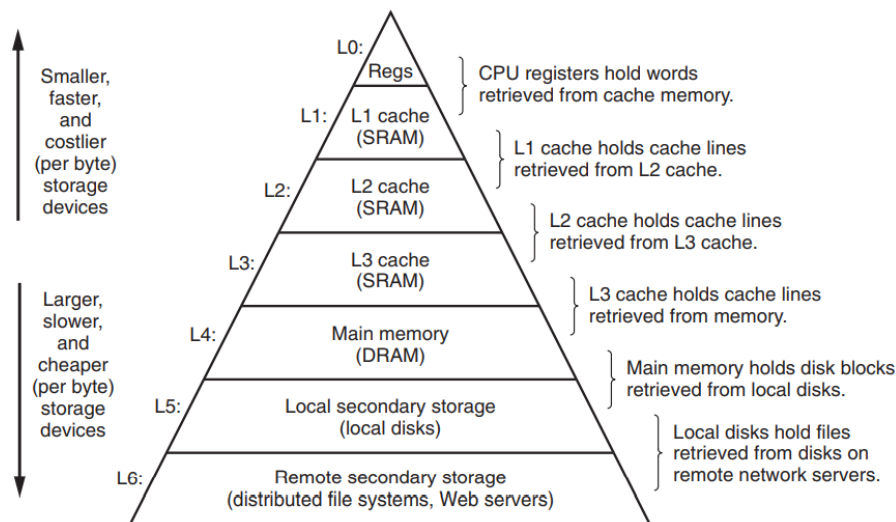


Figure 2.1: *Big data* systems memory hierarchy as shown in [23]

The prevalent solution for this problem is a memory hierarchy with multiple levels starting from very fast minuscule memories, like the register file, very close to the processing units, to progressively larger and slower memories, all the way to huge data centres kilometres away from where the processing happens connected through the internet (figure 2.1). In order for such a scheme to succeed its purpose, the most relevant data, those to be accessed more frequently in the short term, should be in the closest possible level to the processing units. In this way, memory appears from the perspective of the programmer as a single component, which is almost as fast as the levels closest to the CPU and almost as large as the furthest levels. The criticality of this condition can be seen in table 2.1 where an estimation of the average latencies for the different components in memory hierarchy is listed:

Hierarchy Level	Latency	Latency in cycles
Register File	<0,25ns	<1
Level 1 Cache reference	0,5ns	2
Level 2 Cache reference	7ns	14
Main Memory reference	100ns	400
Read 4K randomly from Main Memory	1μs	4.000
Read 1K bytes over a 1Gb/s network	10 μs	40.000
Read 4K randomly from SSD	150μs	600.000
Read 1MB sequentially from Main Memory	250μs	1.000.000
Round trip within the same Datacenter	500μs	2.000.000
Read 1MB sequentially from SSD	1ms	4.000.000
Rotational Disk seek	10ms	40.000.000
Read 1MB sequentially from Disk	20ms	80.000.000
Round trip packet California-Netherlands	150ms	600.000.000

Table 2.1: Estimation of average latencies for the different levels of memory hierarchy expressed in terms of time units and number of cycles for a 4GHz CPU (data taken from [47])

2.1.1. Main memory

Main memory is the only essential level in the hierarchy a CPU needs in order to function. The prevalent form used in modern systems is the Synchronous Dynamic Random Access Memory (SDRAM), which resides most frequently outside the CPU chip. Like all the random access memories, it provides virtually equal access times regardless of the physical location of the accessed address, while in addition to that it is volatile and its function is synchronised with an internal clock. Its low cost is based on the fact that it uses only a single transistor and a capacitor in order to store the value each bit, which is also the key reason for its lower speed in comparison to SRAMs as the capacitors leak and drain their charge (dynamic) both over time and when read and need to be periodically refreshed to retain their values [43].

The need to constantly refresh an ever-increasing number of cells causes overheads, which as the technology progresses become increasingly significant leading up to 20% performance loss for large sizes [25]. Such a trend is deteriorating the problem of "memory wall" the computing systems industry experiences, due to the inability of DRAM to follow the pace of acceleration of processing units and to the limited communications bandwidth between the chips leading to memory-induced stalls in processing. The most widely adopted approach to mitigate the problem is by increasing the number and sizes of the upper levels of the memory hierarchy, as well as use locality (more details in section 2.1.2) and prefetching (explained in section 2.2) to keep the most relevant data on them [41]. Despite those measures, there are still workloads suffering from significant CPU idle times waiting for the data to arrive.

2.1.2. Caches

Caches were introduced as a measure to bridge the expanding speed gap between the CPU and the main memory, as it was mentioned in the previous sections, since the existence of the faster forms like SRAM allowed but the performance and costs would largely decrease and increase respectively for larger sizes sufficient for replacing DRAM as main memory [27]. An important factor contributing

to this cost is that each SRAM cell is nearly 5 times larger than a DRAM cell, which would increase the size decisively requiring longer interconnect elements and thus, mostly eliminating any meaningful performance benefits.

Despite the concept of smaller and faster memories between the CPU and the main memory being discussed since 1965 [53], the memory hierarchy was introduced in commercial chips with the inclusion of an off-chip cache memory located in the motherboard for the *Intel 386DX* chip in 1985, with another level being added inside the CPU later by Intel for the next generation *486DX* CPU and later split into separate instruction and data modules for the first *Intel Pentium*. As the CPU technology evolved and entered the multi-core era, *level 1 caches* were multiplied in order for each core to have its dedicated L1I and L1D caches and *level 2 cache* was introduced as well [49], mostly shared by all the cores. For high performance many-core architectures *level 3* and in some cases *level 4 system caches* followed to further expand the hierarchy and curtail the effects of the "memory wall".

Inside the cache, the organisation is implemented using blocks, frequently called cachelines as well, hosting the data of a fixed-sized group of contiguous memory addresses, which either begins from address 0 or from a multiple of group size. Since the size of a cache is smaller than this of the main memory, more than one group of memory addresses should be mapped to the same block. If each group can be mapped to only one block as shown in the left part of figure 2.2, then the cache is called Direct-Mapped. Such a design decision simplifies and speeds up the placement of each group to the respective block, but prohibits groups mapped to the same block to be simultaneously in the cache, which could result into very high miss rates when the access patterns match the mapping patterns (conflict misses). If it is possible for each group to be mapped to N different blocks instead of 1, then these N blocks comprise a N -way set and the cache is called N -way Set Associative (example for $N = 2$ in the right part of figure 2.2). In this way, N blocks the presence of which in the cache in a memory-mapped design would be mutually exclusive, can be in the cache at the same time, but the placement complexity of each block increases as there are N candidates to be evicted in case the set is full. As associativity increases, the likelihood of conflict misses decreases further up to a point of diminishing returns, but the required hardware and time for placing increases as well. If N reaches the number of available cache blocks, then the cache is called fully-associative as each group can be placed in any block [43].

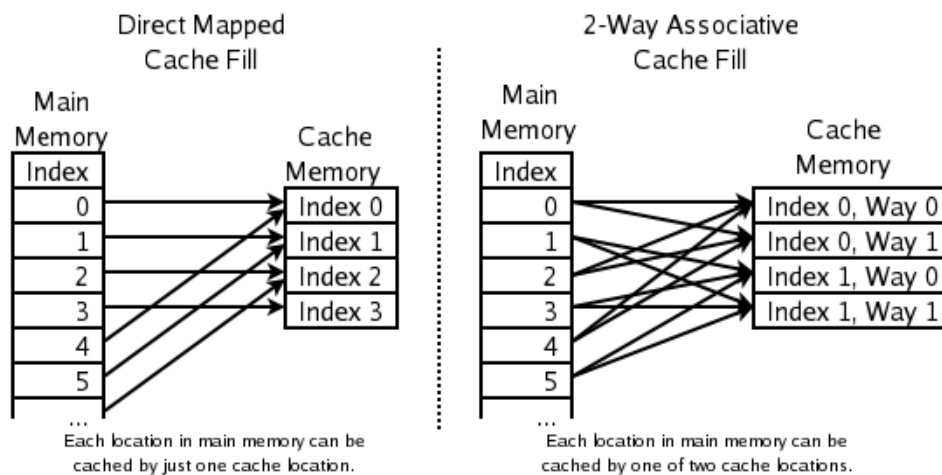


Figure 2.2: Direct-Mapped and 2-way Set Associative placement as shown in [8]

In order to sustain higher hit rates and fulfil their purpose, caches should exploit locality as it is a good indicator of which data will be relevant for the under-execution piece of code. Locality when it comes to time is an indicator of possible reuse. More specifically within the framework of processing, temporal locality indicates that data which were used recently, are likely to be referenced again in the near future and thus, they should remain in cache. A degree of cache associativity and a sensible replacement policy guarantee that temporal locality will be indeed exploited up to a level allowed by the cache size. Another type of locality is the spatial locality which indicates that data in proximity to those just accessed have a higher likelihood to be accessed as well. Design decisions that guarantee the exploitation of spatial locality is the existence of cache blocks, which are fetched on their entirety instead

of fetching just their referenced part [43], and the operation of the DRAMs in bursts. In order to further take advantage of locality but to identify more complex access patterns as well, various prefetching schemes have been introduced, the most relevant of which will be discussed in the next section.

2.2. Cache Prefetching

Since cache misses become increasingly costly as the on-chip core count grows at a faster rate than the memory bandwidth, other techniques should be implemented on top of the memory hierarchy to curtail the resulting performance impact. Prefetching, which is defined as the set of techniques of fetching instructions or data to a level of the memory hierarchy closer to the processing units based on the prediction that they will be needed shortly, has been proven to be the most efficient of those since with negligible extra costs can bring the performance close to that of a perfect cache and accelerate both serial and parallel workloads. Nevertheless, the implementation of those techniques is demanding as it involves complicated predictions of future access patterns and thus, if not designed carefully, it may lead to cache pollution with irrelevant blocks which in turn consume useful bandwidth and may immediately cause or bring forward the eviction of other useful cache lines causing even more misses [42].

Several terminologies and metrics have been introduced to define and quantify the success of prefetching techniques when dealing with these challenges except for the intuitive reduction in the cache miss rate. Namely, the percentage of the original misses eliminated by each specific technique is called *coverage*, while the *accuracy* of prefetching is defined as the percentage of the *useful* prefetches, which are the prefetched cache blocks used at least once before evicted, relative to the total number of prefetches, both useful and *harmful*. Moreover, the term *timeliness* refers to the prefetch requests which are satisfied before their contents are referenced, in comparison to *late* prefetches which just reduce and not eliminate the observed latency. However, the concept of timeliness is not exhausted by the completion of the request as the target should not be evicted before referenced, thus the *lookahead* of prefetching should remain appropriately confined. Lastly, *redundant* are termed the prefetches targeting cache blocks already present in the corresponding cache [42].

Due to the generality of the concept of prefetching, there is a variety of techniques proven to be effective in specific cases. They can be categorised based on the hierarchy level they target, on how they are perceived by the executed code, meaning whether they are to be treated as instructions or data, the means of implementation, which can be either software or hardware, the access patterns they can identify and the data they base their predictions on. Each of those categories and their intersection, face different implementation challenges to comply with the requirements mentioned above. A short presentation of the most relevant subset to this report, which is both the software and hardware prefetching of data targeting the *L1 data cache (L1D)* follows.

2.2.1. Hardware Prefetching

The first form of prefetching was introduced in order to expand the spatial locality properties of the memory hierarchy by adding a specialised hardware block inside the cache called prefetcher. Its task was to sequentially fetch the next cache lines in *IBM 370/168* and *Amdahl 470V* computers and is an essential part of many levels of every memory hierarchy ever since. Such components are trying to correlate runtime information, such as past accesses [50], and other metadata in order to estimate the access patterns and predict future accesses based on them. Since no information about the under-execution algorithm is available, the achieved accuracy is doubtful and thus additional complexity is necessary in order to prevent cache pollution [42]. Different amounts of pollution can be tolerated in each level of the hierarchy as its performance impact is more significant in the levels closer to the execution units, which have stronger bandwidth and capacity constraints. Fortunately, in these levels it is easier to propagate relevant runtime information from the CPU to minimise the number of such events.

As the number of hardware prefetching techniques and variations proposed by scientific and industrial sources is large, only the logic of simple instances popular in modern systems, which were used as a reference, and of those relevant to the proposed techniques will be briefly described in this section.

Stream Prefetchers

The simplest prefetching technique, and the first to be introduced as mentioned above, is the stream prefetching, which, in an attempt to exploit the spatial locality and sequentiality characterising many applications, it prefetches after a distance D sequentially the next K memory blocks, as shown in figure

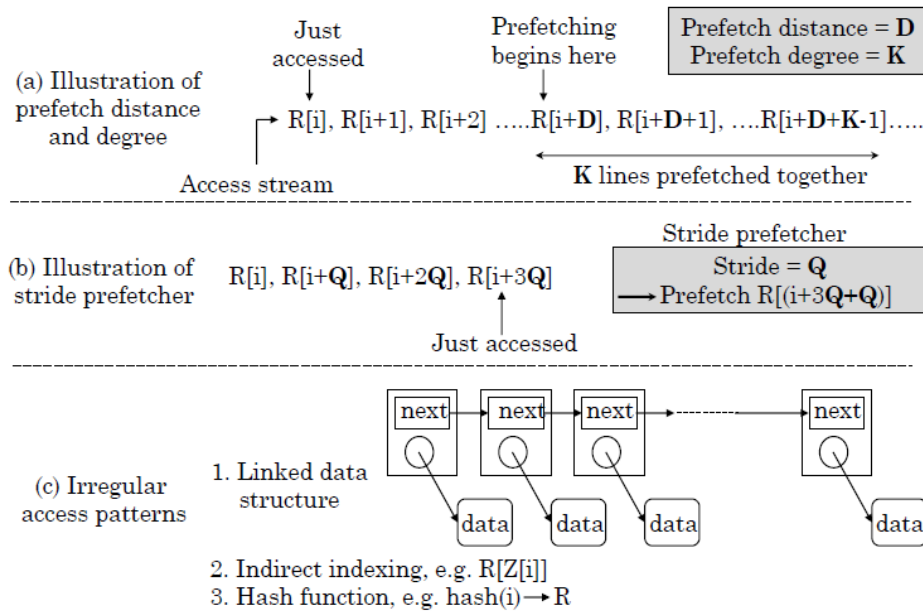


Figure 2.3: Illustration of prefetcher metrics and common access patterns as presented in [42]

2.3a, either on every access or only on misses. The number D of the memory blocks skipped between the block of the access triggering the prefetching mechanism and the first block of the stream is called *Prefetch Distance* and the number K of the contiguous blocks to be prefetched sequentially, the length of the stream, is called *Prefetch degree* [42]. The most common case is prefetch on access with $D = 1$ and K to be determined based on the desired aggressiveness of the prefetcher, as large values reduce significantly the cache misses in pieces of code with good spatial locality but pollute the cache otherwise and low values may lead to late prefetches hiding only partially miss latencies [50].

Placing prefetched blocks directly in the cache is the simplest approach, though due to the simplicity of the prediction scheme, this could lead to cache pollution for more complex access patterns than streaming algorithms. Thus, different approaches have been proposed such as placing the prefetched data on separate stream FIFO buffers, the first entry of which will be checked on every miss and if it matches it will be brought to the cache and the rest of the entries to the front of the queue else the buffer will be flushed. When data from the stream are used, more prefetches will be issued to keep the buffers sufficiently ahead of the execution [42] [50]. The cost of avoiding pollution is the extra area of the separate buffers and the extra bandwidth needed to keep the buffers full.

Stride Prefetchers

A more sophisticated technique than sequentially prefetch the next addresses is to identify the underlying access pattern and based on that to predict and prefetch the next blocks based on it. The simplest form of a pattern is a constant stride, meaning the systematic accessing of addresses with a constant distance between them by the same memory instruction. In order for such a pattern to be identified, when a memory instruction is first issued, the address of the instruction (the value of the PC when the instruction is issued) and the address of the accessed data are saved. During the second issue, the previously accessed address is subtracted from the address of the accessed data to calculate the stride Q which is saved. When the third access occurs, the same subtraction takes place and if the newly-calculated stride matches the previous one, K entries are fetched, from $R[i + 3Q + Q]$ to $R[i + 3Q + (K - 1) \cdot Q]$ as shown in figure 2.3b. In case of stride mismatch, the stride is updated with the new value and no prefetch is issued. Thus, for each memory instruction, the address of the instruction, the address of the data previously accessed and the previously calculated stride should be stored. Because it is impossible to store those data for every memory accessing instruction, the prefetcher stores this information for the most recent accesses in a separate small cache [50].

The above mechanism can be further enhanced by adding a confidence counter for each entry to measure how confident is the prefetcher about each identified stride by increasing the counter on stride matches and decreasing it on stride mismatches. Based on such a counter several thresholds

can be set like a prefetch threshold, which, if exceeded, prefetches based on the identified pattern will be initiated. In this way the number of accesses required to confirm the identification of a specific pattern can be specified, determining how cautious or aggressive the prefetcher should be in order not to pollute the cache or not to stall, resulting in an increase of the number of cache misses and late prefetches respectively. Another threshold can be set for the confidence of an entry below which its stride is updated with a new identified value, guaranteeing that temporary jumps after multiple accesses with a constant distance between them, for example in cases like a new row in a table, will have no negative impact. In [26], the above thresholds are unified in a single confidence threshold, below which the entry is in an identification phase with its stride value being updated by any new computed stride and above which the entry is in a ready phase with its stride value remaining stable and prefetches being issued based on it. A maximum threshold can also be set, which when reached the confidence counter cannot be further increased, in order to guarantee that the confidence value will not grow much larger than the value of the threshold, making the prefetcher insensitive to a pattern change. Finally, with a similar logic, a minimum threshold can be set to prevent an out of proportion counter decrease preventing the entry from reaching fast enough the ready state when upon several accesses following the same constant pattern.

Indirect Access Prefetchers

Nevertheless, not all the data structures are regular due to the nature of their elements (e.g. variability in length), to the need for being stored efficiently (e.g. sparse matrices) or to the hierarchical relations between the data (e.g. linked lists) and thus, accessing them directly would be cumbersome or even impossible. From this property, a need emerges to introduce methods of indirect accessing using functions or lookup tables as intermediates, in order to translate a specific index or key to the actual location of its corresponding data, as shown in figure 2.3c. Such an access mechanism renders stream prefetchers harmful, as in most cases limited spatial locality exists between the accesses, and the stride prefetchers unhelpful, as they fail to identify any constant offset access pattern and issue any prefetches.

For a prefetcher to be able to identify such irregular access patterns, information about the intermediate translation scheme should be available to it or it should attempt to infer it during runtime. Attempting to pass information for functions or for data structures to cover all possible cases of indirection during the design or runtime would not be beneficiary, as it would increase drastically the complexity and the memory needs of the prefetcher, resulting in a negative cost-benefit relation.

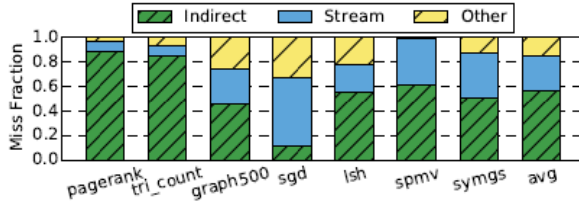


Figure 2.4: Fraction of the total L1D cache misses the indirect access misses constitute as shown in [54]

Application	Streaming Prefetcher			Streaming + IMP		
	Cov.	Acc.	Lat.	Cov.	Acc.	Lat.
<i>Pagerank</i>	0.08	1.00	3.07	0.96	1.00	1.13
<i>Tri_Count</i>	0.07	0.50	5.78	0.91	0.72	3.57
<i>Graph500</i>	0.28	0.87	3.36	0.74	0.94	1.61
<i>SGD</i>	0.56	0.70	2.21	0.67	0.72	1.68
<i>LSH</i>	0.22	0.50	5.90	0.78	0.66	4.04
<i>SpMV</i>	0.38	0.98	1.89	0.99	0.98	1.00
<i>SymGS</i>	0.37	0.97	3.25	0.87	0.96	2.01
<i>Average</i>	0.28	0.79	3.64	0.85	0.85	2.15

Figure 2.5: Coverage, Accuracy and Latency of IMP from [54]

Solutions for the simplest case of indirect access prefetchers, which is the indirect indexing (see 2.3c2), have been proposed for both CPUs [54], [21] and GPUs [35]. Indirect indexing is the method of determining the accessed index of a data structure using the contents of another. In the form presented in figure 2.3c2, the contents of $Z[i]$ determine which index of R will be accessed. Many algorithms like those who operate on sparse datasets and arrays of variable length elements use such kind of accesses regularly as it can be seen in figure 2.4. Specialised approaches for specific algorithms like in [21] will not be discussed, as tackling the general cases is the aim this study. Such an approach to identify this kind of a relationship between 2 data structures is documented in [54], which is based on the remark that the address of an indirect access can be split into 3 parts: a base address, an index and a coefficient with which the index is multiplied. This relation is formalised by the following equation:

$$\begin{aligned}
 Addr(R[Z[i]]) &= BaseAddr_R + coeff \cdot Z[i] \\
 Addr(R[Z[i+1]]) &= BaseAddr_R + coeff \cdot Z[i+1]
 \end{aligned} \tag{2.1}$$

Which, based on the premise that the step with which a data structure is traversed, is a multiple of the size of its elements, most of the times a power of 2, can be simplified to:

$$\begin{aligned} \text{Addr}(R[Z[i]]) &= \text{BaseAddr}_R + (Z[i] \ll \text{shifts}) \\ \text{Addr}(R[Z[i+1]]) &= \text{BaseAddr}_R + (Z[i+1] \ll \text{shifts}) \end{aligned} \quad (2.2)$$

As a first step on the attempt to identify $(\text{Addr}(R[Z[i]]), Z[i])$ pairs, candidate indices $(Z[i])$ are determined from stream prefetcher hits and put on the Indirect Pattern Detection (IPD) table. For a predefined small number of cache misses after the index load and a predefined small number of *shifts* considered, the corresponding Base Addresses are calculated, using the first equation in 2.2 with the *MissAddr* as the $\text{Addr}(R[Z[i]])$ and the index read from the *stream buffer* as $Z[i]$, and put on the *BaseAddr* array of the entry. When the next index $(Z[i+1])$ on the corresponding stream is accessed, the Base Addresses for a number of the subsequent cache misses are calculated and stored in the same way as for the first index, using the second equation in 2.2. The Base Addresses of the two indices for the same value of *shifts* are compared and if there is a match, then an entry is created in the Indirect Memory Prefetch (IMP) table containing the value of the Program Counter of the associated stream, the value of the *BaseAddr_R* and the value of the *shifts* field, with the entry in the IPD being erased. If there is no match, the same procedure is followed for the third index $(Z[i+2])$, which is placed in IMP in case of match, otherwise the decision that no pattern can be identified using this scheme is taken and the IPD entry is erased [54].

For the entries in the prefetch table, the index value is updated for every new index read from the *stream buffer* and every access thereafter is checked whether it conforms to the identified pattern. A confidence scheme similar to the one described for stride prefetchers in section 2.2.1 is used to evaluate the confidence for the correctness of the identified pattern. The saturating confidence counter is increased on compliant accesses and decreased if the index field is updated before a match for its previous value is found. Once the value of the counter reaches a threshold a predefined number of prefetches are issued with a small initial distance which increases linearly with the value of the saturating counter [54].

Such a solution is limited by the ability of the prefetcher to identify only streaming indirect accesses, those for which the step the index table is traversed with should be 1 ($i, i+1, i+2$), by the constraint that the resulting index $Z[i]$ should be multiplied only with coefficients which are powers of 2 and by the lack of information on the contents of which addresses can be classified as indices and which as data. The result of those limitations is the acceptance of the proponents of the IMP prefetcher that "it is fine to fail to recognize many index-address pairs", though the coverage and accuracy results for the benchmarks studied shows that the scheme performed better than expected with enough room for improvement for some [54].

2.2.2. Software Prefetching

Except for relying solely on specialised hardware components to identify possible access patterns on runtime, simple or more complicated prefetch instructions, for the programmer or the compiler to take advantage of, are included in the instruction sets of all popular architectures at a negligible hardware cost. From the initial implementation of the simple instructions as normal load instructions fetching the referenced data to registers hardwired to zero [50], currently, complex specialised instructions exist which except from the targeted address, they provide information about the destination cache, the temporality and the potential use of the data [2] [9]. Software prefetching usually exploits the knowledge of the algorithm and the data dependencies to issue prefetch instructions for data that are known to be used in the short term.

As hardware prefetchers are featured in virtually all modern CPUs, software prefetches should take into account their characteristics and parameters in order to target missing patterns. The process of determining the distance, in terms of instructions, between the prefetch and the instruction using the data is called *prefetch scheduling* and in addition to the other prefetching schemes in the system, it must take into account both the execution time required by the code between the instructions and the memory hierarchy parameters. Execution time and latencies may vary and thus, they cannot be known but only approximated during compile time, in which a miscalculation might lead to bandwidth starvation and untimely prefetches and in turn to the opposite effect of optimising memory accesses. For this reason, it must be performed with caution and mostly be left for the compiler to optimise large loops which

can have a more predictable execution behaviour [50]. Another parameter that can possibly degrade performance is the instructions overhead induced by the additional prefetch instructions, as well as possible alterations in the structure of the produced code, as the compiler might not be able to perform optimisations which would be possible otherwise [36].

Various studies have evaluated the use of software prefetches in popular benchmark suites exhibiting different memory access characteristics and for different architectures. The performance varies with the workload as expected, with the indirect indexing accesses being a subset favoured significantly by such techniques. But except for the variability by workload, a less expected variability by architecture, as the studied architectures allow the same amount of information about the prefetch to be specified in the instruction, is observed. This is attributed to the significant performance impact memory stalls have on in-order execution, as on a cache miss the CPU cannot continue execution regardless if there are independent instructions available for execution, in contrast to Out-Of-Order (OOO) machines [21].

2.2.3. Hardware and Software Prefetching differences and trade-offs

As described in sections 2.2.1 and 2.2.2, the differences of these approaches are not limited to their implementation. With hardware prefetching, no software overhead is induced, in terms of both extra code and compiler complexity, while in some cases the compiler is able to better optimise the code without the inserted prefetches altering its structure. Scheduling prefetches and determining their parameters, like degree and distance, becomes easier and more accurate as well, with runtime information available to be exploited, which the programmer and the compiler should otherwise estimate and hardcode during compilation, if the runtime dependencies allows them to at all. Furthermore, the produced code may be incompatible with or vastly underperforming in different microarchitectures or memory hierarchies [50] [36] [54].

Nevertheless, the full complexity of creating prefetchers able to identify and correctly schedule prefetches is just transferred to the design of the hardware. Specifically, for more sophisticated prefetchers a significant amount of metadata should be retrieved from many parts of the processor, stored and processed for prefetch targets to be identified, resulting in area, routing and design time overheads [50]. Since no information about the under-execution program is available, a number of misses is required before the prefetcher can speculate a certain pattern, thus no performance gains can be attained for short patterns. Also, any expansion in the number of prefetch table entries or the complexity of the identifiable patterns requires an increase to the cost of the prefetcher both in terms of area, power, integration and routing, in contrast to the sole software overhead of software prefetches. Finally, hardware prefetchers, due to their speculative nature, are prone to inaccurate prefetches possibly caused by both misidentified patterns and loop bounds, which are impossible for the prefetcher to predict without any information about the software structure [36].

A selective combination of both techniques produces the best results, though if not carefully tailored to the system configuration can even decrease performance in comparison to using them in separation [21]. Leaving for the hardware the simple streaming and constant offset patterns, while targeting specific irregular accesses with software prefetches can lead to the optimal performance, given that the memory capabilities of the system are taken into account for the scheduling and tuning of software prefetching in order to prevent antagonism for bandwidth between the prefetching mechanisms [36].

2.3. Column-Oriented Databases

Except from the techniques used to keep the most relevant data in the fastest possible level of the memory hierarchy, the organisation and the format of the data themselves are important to minimise latency and maximise throughput. In *big data analytics*, many queries scan, analyse and process large portions of only a few columns of a table without utilising the rest of the data, which are loaded along with the useful data from the lower levels of the memory hierarchy when stored in the traditional row-wise format. The unused attributes of each entry (row) occupy cache blocks, which could be used by useful data otherwise, inducing an extra amount of latency. Although OOO execution may alleviate a small part of this latency, the fact that the accesses of such queries make little use of the fundamental attribute of spatial locality in memory accesses, around which many of the principles of hierarchical memory is built, especially for tables with rows spanning multiple cachelines, remains. In this particular case of large rows, prefetching would be of little help or even harmful, while a large part of the data contained in each cacheline would be irrelevant, wasting the valuable high-speed memory resources

of the system and the bandwidth required to transfer them [20].

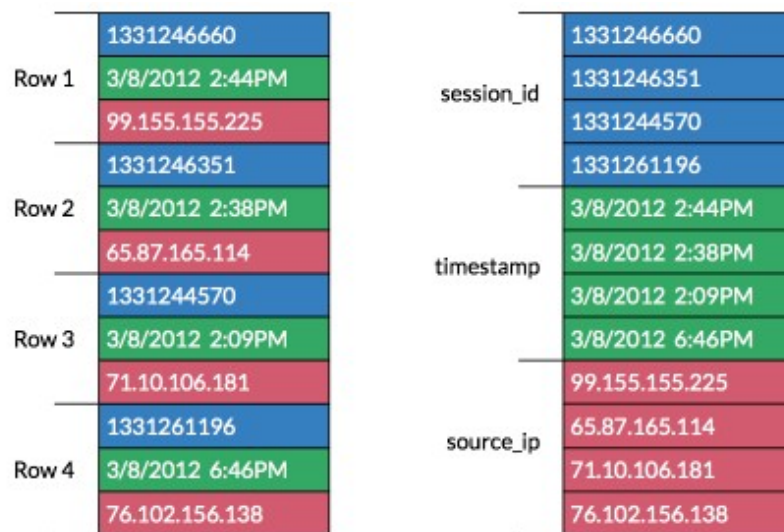


Figure 2.6: Row-Oriented and Column-Oriented organisation of a database as presented in [1]

For this increasing number of queries, organising the databases in memory in columns instead of rows can be of great benefit. In such formats, the databases instead of storing in memory each attribute of each entry (row) one after the other followed by the attributes of the next entry stored in the same order and so on (left part of figure 2.6), the first attribute of all entries is stored followed by the second attribute of all entries and so on, creating columns of attributes (right part of figure 2.6). In this way, queries are able to read from the memory only the attributes they need, instead of wasting bandwidth and cache space by loading all the attributes of the referenced rows [20]. Examples of such queries operating on the database presented in figure 2.6, would be finding the sessions which started before 6 pm or geolocating the sources using their IP address, requiring only the timestamp or source_ip columns respectively. Nevertheless, columnar format is not ideal for any workload, as some queries require many or all the attributes of a specific number of entries. In such cases, it is the columnar format not exploiting the spatial locality as the attributes of each entry may reside in addresses distant from each other and thus, most probably not in the cache in perfect analogy with the case described in the previous paragraph. Consequently, each orientation benefits different kind of queries and for the field of *big data analytics* specifically, it is the columnar one as it operates mostly on small sets of attributes of large portions, if not the whole, of the databases [20].

The benefits of columnar orientation of databases in *big data analytics*, exceed the better cache behaviour due to locality. Since the data in columns are of the same type and usually undergo the same transformations and processing, the property of being stored in a contiguous way can be exploited by efficiently loading them in SIMD registers and process them in parallel, resulting in a significant improvement in throughput. Performance can be further improved for filtering, if the database is sorted based on the attribute used to express the criteria of the filter, as fast algorithms, like binary search, can be utilised just in this specific column. It is not uncommon for several copies of columns sorted using different properties, to be created for such purposes [20].

This per-column efficient processing, optimised with SIMD when possible, can be sustained, in most queries, until the very end and only then assemble the resulting entries, if needed at all. In this way, tight loops with CPU friendly access patterns are used throughout the whole query resulting in a considerable increase in memory bandwidth and, as a result, in performance. Furthermore, compressing data of the same type is more effective as less variability exist in them, as a consequence simpler compression algorithms can achieve good compression ratios when compressing columns, something which mostly does not hold for rows. Moreover, for the different compression schemes, there are operations which can be performed directly on the compress data, allowing decompression to be postponed until it is really needed. Consequently queries can fit more data in every level of the memory hierarchy for most of their duration [20].

Apache Arrow

Such a cloumnar format is *Apache Arrow*, which, according to the project's website, "is a cross-language development platform for in-memory data", organised in columnar memory format [1]. Thus, Arrow exploits all the advantages of columnar orientation, but differs in the hierarchy level it targets, which, in contrast to most of the platforms featuring columnar data representation formats, is the main memory and not the non-volatile levels below it. As a result, different trade-offs arise since, in the case of disks, IO dominates latency and aggressive compression is employed in order to address it, adding significant burden to the CPU as otherwise many cycles, in the order of magnitude of millions (see table 2.1), would be spent idle. On the contrary, since Arrow targets the much faster level of main memory, the price of which declined significantly in the past years enabling a considerable increase in its size, optimisation of processing through cache locality, pipelining and SIMD is the target, leading to different design decisions [47]. For instance, its batch size is much smaller because of the considerably small difference between the random and sequential reads in RAM compared to disks, especially the rotational ones [19].

Apache Arrow's contribution to the field of *big data analytics* is not limited to a columnar format tailored to high-speed processing. A significant amount of time and resources, both processing and memory ones, are lost in the differences between the way each programming language and tool utilises them and the serialisation, deserialisation and reformatting processes. Arrow solves this issue through a standardised, optimised data layer, which can be efficiently accessed and shared between many different, Arrow-enabled ecosystems, languages, applications and tools spanning all the *big data* value chain as depicted in figure 2.7. Using this layer, persistent, immutable Arrow objects, able to represent diverse data structures, from simple flat tables with primitive data types to rich-structured, nested data hierarchies with complex data types, eliminate all the overheads of data sharing [47], improving performance of many workloads up to 10x-100x [44].

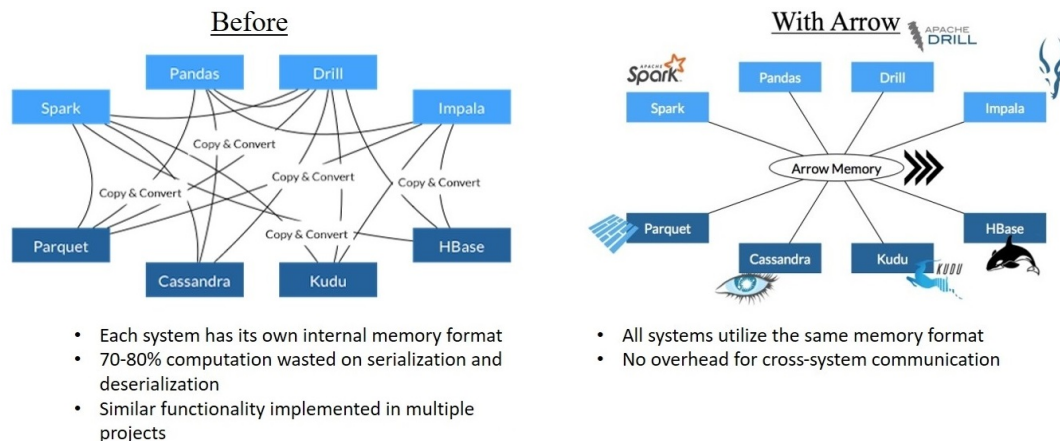


Figure 2.7: Arrow common data layer as depicted in [1]

The flexibility of Arrow in describing diverse data structures is a result of its concise but potent metadata. In them, the full information of the hierarchy, types and structure of the dataset is encoded, allowing an aggregation of contiguous memory buffers to represent complex databases. An overview of how the combination of the Arrow in-memory format and the metadata produce a highly-efficient and versatile representation follows, starting from the high level and moving progressively downwards to some details of its physical memory layout.

Any particular dataset can be represented in Arrow as a *Table* or a series of *Record Batches*, which are collections of *Columns* adhering to a specified *Schema* [5]. The *schema* consists of a sequence of *fields*, one for each *column*, containing information about the name, the logical type (it can be a primitive one, like integer, double or character, etc, or a nested one like structure, string or union, etc), the nullability (the allowance for null values), the dictionary encoding (if present) and, for nested structures, the *fields* of its children (members). Each logical type has "a type name and metadata along with an explicit mapping to a physical memory representation". In this way, a specific logical type along with its children, if any, and an explicit memory mapping is associated with each *column* defining a concrete data hierarchy of structures and how the *Buffer(s)* (explained later) of each *column* should be

used in order to access and interpret any specific value of the dataset [6].

Each *column* consists of an *array*, possibly chunked, and optionally some column-specific metadata. An *array* "represents a known-length sequence of values all having the same type". In memory, "those values are represented by one or more buffers", the meaning of which depends on the *column*'s logical type declared in the corresponding *field* [5]. The *buffers* of each *array*, if not chunked, or of each chunk are contiguous virtual memory regions. *Arrays* consist at least of one *buffer* containing the value data and for nullable *arrays*, also of a *bitmap buffer* indicating which values are null. For non-primitive types more *buffers* are needed in order to represent the hierarchy of the values and possibly the values themselves [7].

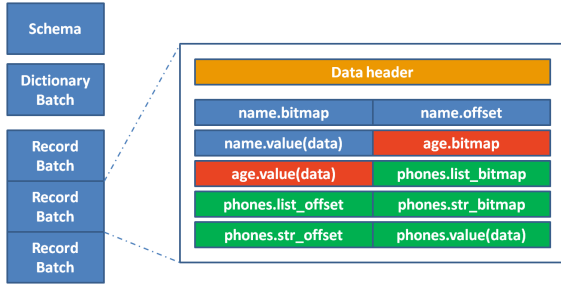


Figure 2.8: Record batch structure of the database shown in the figure on the right (based on [44] and [7])

Columnar data

```
persons = [{
  name: 'Joe',
  age: 18,
  phones: [
    '555-111-1111',
    '555-222-2222'
  ]
}, {
  name: 'Jack',
  age: 37,
  phones: [
    '555-333-3333'
  ]
}]
```

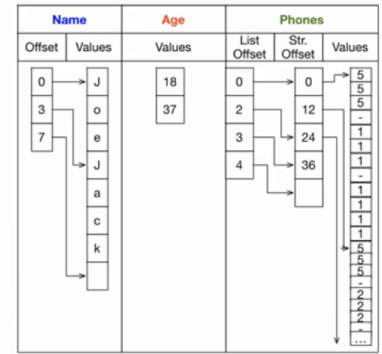


Figure 2.9: Database representation example using the contents of *buffers* as presented in [44]

The relative to a base address location and length of the *buffers* in virtual memory are defined in the *Data Header* [6], allowing, in combination with the corresponding *field*, for each value of the dataset to "be accessible in constant time, with complexity growing linearly in the nesting level" [7]. An example of a database, which consists of three nullable *columns*: *name* (string), *age* (unsigned integer) and *phones* (list of strings), is given in figure 2.9. The simplest of them is the *age column* as due to its primitive type and thus, its constant data length, except for its *bitmap buffer*, it just needs a single *value buffer* to be represented. The offsets, in this case, are constant and equal to the length of the datatype in bytes (here denoted by the C function *sizeof*) and as a result the contents of each index *i* can be accessed in the address (Addr):

$$Addr_{age[i]} = baseAddr_{age.value} + i \cdot sizeof(integer) \quad (2.3)$$

A more complex case with one level of nesting is the *name column*, which contains strings, represented in Arrow as lists of characters, and thus having variable length. Due to this variability, except for the *bitmap buffer* and the *value buffer*, a separate *offsets buffer* is required to indicate in which offset from the base address each string starts. Consequently, each string indexed by *j* and each character indexed by *i* in a string indexed by *j* can be accessed in the respective addresses (Addr):

$$\begin{aligned} Addr_{name[j]} &= baseAddr_{name.value} + offset[j] \\ Addr_{name[j].character[i]} &= baseAddr + (offset[j] + i) \cdot sizeof(character) \end{aligned} \quad (2.4)$$

Finally, the *phones column* exhibits two levels of nesting as a list of strings is essentially a list of lists of characters. As mentioned above, with each level of nesting the complexity grows linearly and so does the number of required *buffers*. Specifically for this case, separate *offset* and nullability *buffers* are required for the lists and the strings, denoted respectively as *list_offset*, *str_offset*, *list_bitmap* and *str_bitmap*. Each list indexed by *k*, each string indexed by *j* in a list indexed by *k* and each character index by *i* in a string indexed by *j* in a list indexed by *k* can be accessed in the respective addresses (Addr):

$$\begin{aligned}
Addr_{phones[k]} &= baseAddr_{phones.value} + str_offset[list_offset[k]] \cdot sizeof(character) \\
Addr_{phones[k].string[j]} &= baseAddr_{phones.value} + str_offset[list_offset[k] + j] \cdot sizeof(character) \\
Addr_{phones[k].string[j].character[i]} &= baseAddr_{phones.value} + \\
&\quad (str_offset[list_offset[k] + j] + i) \cdot sizeof(character)
\end{aligned} \tag{2.5}$$

In case of a null value in index i , the i th bit of the bitmap will be set to 0, as the default value for bits representing non-null values is 1, and for the respective offsets the following relation will hold: $offset[i + 1] = offset[i]$. More information about more complex cases of the Arrow memory layout can be found in [7].

2.4. Architectural Simulation

Modern processing systems become increasingly complex as the semiconductor technology allows more and more transistors to fit in the same area. Due to this complexity, integrating new peripherals or exploring different parameters on existing ones using accurate hardware description models like VHDL specifications, can be very expensive in terms of time and resources [24], while analytical models fail to accurately capture the variations in system metrics seemingly small details can cause. As a result, simulators featuring models of varying level of detail, accuracy, configurability and speed emerged, rendering simulation the standard performance modelling and design space exploration method [22]. Unsurprisingly, any increase in abstraction and thus, in simulation speed, reduces the accuracy and fidelity of a model, while it allows for more configurability and an expanded design space as depicted in figure 2.10. Simulators can be classified in various categories based on their level of detail, the scope of the target and their input.

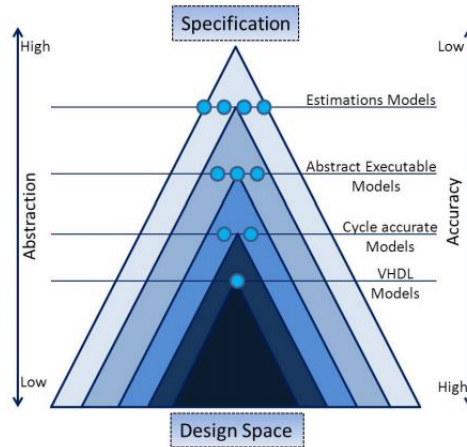


Figure 2.10: Model properties trade-offs as depicted in [24]

Functional simulators emulate the behaviour of the simulated instruction-set architecture producing the same results per instruction but fail to capture any data on the microarchitectural parameters as they are not modelled. Such simulators can be used for identifying architectural features of programs, such as memory access locality, the number of branches etc. *Timing* simulators increase the level of detail by simulating all the microarchitectural features of a processor producing detailed statistics for each of them. The detail level can be further parametrised with time granularity of the simulator which can either be event-based or interval-based. In *event-driven* simulation, a set of meaningful events is determined and when one of those is scheduled, is put on a queue. Simulation then is fast-forwarded to the time the next event in the queue is scheduled. In *interval-based* simulations, portions of the simulators are used to determine the timings of miss events interrupting normal execution, such as cache misses and branch mispredictions. Then various methods can be used to estimate the duration and other statistics for each of the intervals. If intervals are reduced to one cycle and full simulation of the functionality of the processor is used as an estimation method, then the simulator is called *cycle-based*. If parameters from the RTL implementation are integrated in the model to further increase

accuracy, then the simulator is called *cycle-accurate*. Integration of both functional and timing models in the same simulator is possible in order to focus only on points of interest of the executed programs, fast-forwarding by functionally simulating the rest [22].

Classifying simulators in terms of scope refers to whether the operating system, which each program is compiled for and runs in, is simulated as well, allowing for the impact of OS-specific routines like system calls to be taken into account or the program of interest is run in isolation emulating each OS-specific routine and estimating its impact on the produced statistics. Those two approaches are called *full-system* simulation and *application-level/user-mode* simulation respectively. Finally, the inputs classification has to do with whether the binary of the program is executed by the simulator as is, called *execution-driven* simulation, allowing the user to pass the desired input on runtime or "pre-recorded streams of instructions executed by benchmarks with some fixed inputs" on real machines called traces are used instead, called *trace-driven* simulation, limiting the results and the conclusions to those produced for the fixed inputs [22].

The criteria to be taken into account when choosing among the available simulators though are not limited to the above-mentioned classifications. The support for different ISAs, microarchitectural features, parallel execution and other, external to CPU, features like peripherals, as well as the freedom for configuring and combining them, is crucial for modelling modern systems. Another important kind of support is the community support, which allows users and developers of each simulator to help each other solving problems and improving the accuracy of their results. Finally, when more metrics, for example related to the energy/power cost of a system, are integrated as well, enables for better-informed estimations of the cost/benefit relation of the under-test features [22].

gem5

A simulator aggregating almost every single of the above advantages is *gem5*, which was created based on the best properties of the full-system simulator *m5* and *GEMS*, a simulator focusing on memory systems. Despite being event-driven, it can keep track of events on cycle granularity, rendering its accuracy comparable to *cycle-based* simulators. It supports all popular ISAs, allowing for them to be simulated both in functional and timing mode, which is equipped with many configurable microarchitectural features and extensive statistics for each of them. Characterising it as *full-system* or *application-level*, called *System Emulation* in *gem5*, is not possible since it is capable of running in both modes, though, despite its ability to produce traces, it operates only as *execution-driven* [22]. Those characteristics render *gem5* a flexible, modular simulator, allowing for both software and hardware investigations in different abstraction levels [24].

The accuracy of *gem5* has been tested against actual processors to estimate its accuracy. Most of the experiments were performed on *ARM* architecture which has the most extensive support due to the massive contribution of the company in the project. The results from several studies shown an average inaccuracy close to 10% for various benchmarks suites, with a range from 1.39% to 17.94% [22]. Both [24] and [51] partially attributed the error to modelling inaccuracies of L2 cache and DRAM latencies, with the accuracy improving in the latest of the studies. The fidelity of the single-core behaviour though seems to be better, further reducing the average error for those cases [31]. Outside the *ARM* architecture, the accuracy of *gem5* seems to deteriorate as for the *x86* architecture the error in IPC values is significant. Nevertheless, the overall accuracy rate along with the desired properties of flexibility, modularity and configurability lead *gem5* to be one of the most extensively used simulators both in academia and industry for evaluation of new ideas and proposals [22].

3

Experimental setup

As documented in section 2.4, *gem5* is a highly configurable simulator allowing for multiple levels of abstraction and inputs as it is *execution-driven*. Selecting the appropriate accuracy level, simulation mode, system and (micro)architectural configurations and benchmarks to run, in order to estimate the impact of each feature, can be challenging with such a plethora of options. Although a persistent attempt to match each and every parameter of the model to this of a real system does not necessarily lead to better accuracy [22], only realistic configurations and benchmarks modelling parts or the entirety of popular algorithms allow for the production of detailed statistics that can be processed to derive meaningful conclusions. In this section, the selection process will be presented along with the resulting configurations in section 3.1, the benchmarks in section 3.2 and the statistics in section 3.3.

3.1. System Configurations

Except for the simple functional model, *gem5* includes 3 more models for the more accurate *timing* simulations: the *Timing Simple* model, which is the simplest model able to simulate the memory system but not pipelined, the *Minor* model, which is in-order as well but with a short 4-stage pipeline, and the *O3* model, which is an OOO CPU with a more detailed 7-stages pipeline. Both pipelined models are of interest as the powerful *O3* processor can be used as a model for the big powerful, power-hungry cores featured in most of the high-performance clusters, while the *Minor* In-Order CPU could model the little, power-efficient cores used for offloading less performance-critical tasks in heterogeneous architectures, featuring both, to save energy. This combination, which shows promising energy efficiency results on specific analytics workloads [37], could gain traction due to the increasing energy costs of the servers systems [33]. Nevertheless, only a single frequency for both models was selected for uniformity purposes to focus on the impact of the resources without having to correct for the difference in frequency. Since no dynamic frequency scaling mechanism will be simulated, the frequency was set to a value between the normal frequency and the maximum turbo frequency of the high-end server processing systems available in the market [18] [10] [17].

As discussed in section 1.1.1, the different ISAs have different characteristics which benefit specific workloads more than others and thus, all have a place in architectural investigations for *big data analytics*. Though, using a particular simulator for the initial phases introduces several limitations. As discussed in section 2.4, limited work on validating the accuracy of the simulator have been performed for ISAs other than *ARM*, and those examining those cases shown that *gem5* models the rest of the ISAs with less fidelity. Except for the accuracy of the model, the overall support of the ISAs varies significantly, with *ARM* being the most supported architecture as well in all aspects due to the active participation of the company in the project [4]. During the selection of configurations, all the popular ISAs, namely *x86*, *POWER*, *ARM* and *RISC-V*, were considered. Though, several cache-related instructions needed were not implemented correctly in *gem5* for each of them. For the case of *ARM*, those were related to L2 cache and were limiting only for certain scenarios that could be examined only for completeness. For *x86*, *POWER* and *RISC-V*, there were unimplemented instructions for L1 cache, prohibiting the test of scenarios highly relevant to the performed research. Contributing to the development of the missing instructions for the less accurate ISAs would increase the workload with-

out guaranteeing an improvement in the accuracy of the results. Furthermore, utilising a single ISA simplifies the integration of assembly instructions and ISA-specific intrinsics in the benchmarks in parts when compiler optimisations are needed.

Determining the parameters of the memory hierarchy is important due to the scope of the research questions. Since the systems of interest are high performance clusters with a large amount of resources, high-end memory components were selected. The sizes and associativity of the L1 and L2 caches were selected based on the high-end server processing systems used as a reference [18] [10] [17], with the rest of the parameters left to the predefined of the example high-performance configuration file as they are not disclosed for any of the systems. An L3 cache was not included as for all the tests only single core scenarios were tested and a system level cache would not contribute to the accuracy of the tests. For the main memory, the fastest available model on the used version of *gem5* was selected with ample size to host any data size required for the benchmarks.

When running specific benchmarks using system calls or programs dependent on the performance of specific peripherals, *application-level* simulation introduces a significant error as the OS-specific routines are emulated and their effect on performance is roughly approximated. Since all the benchmarks run during the course of this study, described in section 3.2, do not use any peripherals and their memory needs are fully-accommodated by the main memory, *full-system* simulation would increase the simulation time without improving the accuracy significantly. Consequently, *system emulation* was selected for running all the benchmarks in order to limit simulation time.

The resulting configurations based on the above selection process are listed in table 3.1. Any parameter not included in the table was left with the default value of the reference ARM script modelling a high performance system [11] [12] [13].

Feature	In-Order (Minor)	Out-Of-Order (O3)
Superscalar	✓	✓
Pipeline stages	Fetch1, Fetch2, Decode, Execute	Fetch, Decode, Rename, Issue, Execute, Writeback, Commit
Fetch Width	Fetch1: 1, Fetch2: 2	8
Decode Width	2	8
Rename Width	N/A	8
Issue Width	N/A	8
Execute Units	Int ALU: 2, Int MUL: 1, Int DIV: 1, FP/SIMD: 1, RD/WR: 1	Int ALU: 6, Int MUL/DIV: 2, FP ALU: 4, FP MUL/DIV: 2, SIMD: 4, RD/WR: 4
Writeback Width	N/A	8
Commit Width	N/A	8
Frequency	3 GHz	
L1I size, associativity	32kB, 8-way	
L1D size, associativity		
L1I latency		
L1D latency	3 cycles	
L2 size, associativity	1MB, 16-way	
L2 latency	12 cycles	
Main memory configuration	16GB DDR4-2400, 8 devices per rank, 2 ranks per channel, 2 channels	
Main memory timings	[16] [13]	
Cache block size	64 bytes	
DRAM burst size		

Table 3.1: System configurations

3.2. Benchmarks

The evaluation process of architectural features in addition to specific metrics to quantify their impact on important aspects of computing, like execution time, power consumption etc, require programs which are representative of the real workloads the system is designed to execute. Such programs are called benchmarks and are aggregated in benchmark suites targeting all the relevant components of a system and comparing it to those of reference systems. If the scope of the benchmark is limited to a small piece of an algorithm or a small hardware part, usually both, it is called a microbenchmark. Microbenchmarks are useful to prove that a concept is working as designed, the real value of it though can only be proven

by its contribution to real workloads or at least larger benchmarks modelling them.

Since the aim of the reported research was to prove and not quantify the potential benefit of several prefetching schemes and due to the limitations, time and architectural simulation introduce, only microbenchmarks were used. The selection was based on algorithms popular on scientific and other *big data analytics* applications, while other were synthesised specifically to quantify certain concepts as explained later. All of them were implemented using C++ and were cross-compiled using the ARM64 7.4.0 version of the *gcc* compiler for the *Linux Ubuntu* distribution. The compiler was instructed to use the highest level of optimisation (-O3) and statically link all the required libraries as *System Emulation* in *gem5* requires. Each benchmark was written both using SIMD intrinsics and in its simple form allowing the compiler to optimise it and the fastest version of the two was selected to be used. It should be noted that using the *Apache Arrow* library with the ARM64 architecture was not possible as there is not a precompiled version for this architecture and any attempt to cross-compile it was not successful. As a result, a simplified version of the relevant parts of *Apache Arrow* had to be developed using C++ as well.

For each of the benchmarks, the data were produced beforehand and stored in the simplified *Arrow* format to be accessed later. In order to make sure that there is no head start with any of the data remaining in the cache from the production phase, all the related blocks were invalidated from the caches by issuing specifically cache invalidation instructions, affecting both caches, for every memory block containing offsets or values of all the produced *Arrow arrays*. A short description of the microbenchmarks used is provided in the following sections.

3.2.1. Matrix Operations

Matrix operations are a fundamental part of many scientific workloads and thus, *big data analytics* on these fields are using such operations in their queries. Matrix addition and multiplication lie on the extremes of computation intensity of the matrix operations as the first is mostly memory-intensive with just one arithmetic operation performed per 2 cells read from memory, reduced even further with SIMD instructions and the second has a computation complexity of $O(N^3)$ making it fairly intensive. This computation intensity and popularity of matrix multiplication rendered it a research topic with *big data* specific approaches proposed in certain cases [45] [52]. All the above, along with their straightforward, easy to implement algorithms contributed to their selection to be part of the benchmarks.

Matrix addition is performed by forming a new matrix, each element of which is produced by adding the elements on the same position in the operand matrices as shown in figure 3.1. Instead of performing those operations per element, SIMD instructions can be employed to perform up to 4 integer additions in parallel, increasing performance and stressing the memory further as the $64 \div 4 = 16$ integers of each of the operands cache blocks are consumed in $16 \div 4 = 4$ SIMD additions as it can be seen in listing 3.1. Meaning that in case the required addresses are not present in L1D cache, in the In-Order execution case they should be fetched with a rate of 2 cache blocks every 4 SIMD instructions plus the required instructions to calculate the next pointers. For the OOO execution the challenge is aggravated even further since there is no dependence between the instructions and cache blocks are requested and consumed at a higher rate.

$$\mathbf{A} + \mathbf{B} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix} = \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} & \cdots & a_{1n} + b_{1n} \\ a_{21} + b_{21} & a_{22} + b_{22} & \cdots & a_{2n} + b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} + b_{m1} & a_{m2} + b_{m2} & \cdots & a_{mn} + b_{mn} \end{bmatrix}$$

Figure 3.1: Matrix addition of two $m \times n$ matrices as shown in [14]

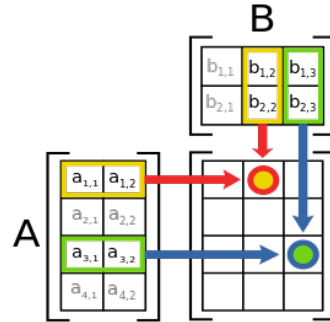


Figure 3.2: Matrix multiplication of one 4×2 matrix with one 2×3 matrix as presented in [15]

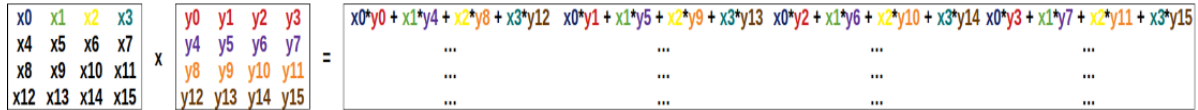


Figure 3.3: Example of calculating the first row of the matrix multiplication algorithm to exploit SIMD instructions

```
static inline void matrix_add(int32_t* __restrict ml1, int32_t* __restrict ml2,
    int32_t* __restrict res, const uint32_t size)
{
    /*Set initial pointers to the operand and result matrices
    **for the sets of 4 32-bit signed integers
    */
    int32x4_t* vml1 = (int32x4_t*) ml1;
    int32x4_t* vml2 = (int32x4_t*) ml2;
    int32x4_t* nRes = (int32x4_t*) res;

    //For each set(size/4 in total) add the operands
    for(uint32_t i=0; i < size/4; i++)
    {
        nRes[i] = vaddq_s32(vml1[i], vml2[i]);
    }
}
```

Listing 3.1: Code for Matrix addition using ARM SIMD intrinsics as implemented for the corresponding benchmark

```
inline void matrix_mult(NPowerMatrixBuffer<int32_t>* mat1,
    NPowerMatrixBuffer<int32_t>* mat2, int32_t* __restrict res, const uint32_t size)
{
    /*k is the index of the row of mat1 used
    **and of the row of res to be computed
    */
    for(uint32_t k=0; k<size; k++)
    {
        int32x4_t* nRes = (int32x4_t*) &(res[k*size]);
        /*i is the index of the element of the k-th row of mat1
        **and of the row of mat2
        */
        for(uint32_t i=0; i < size; i++)
        {
            int32_t scal = mat1->at(k,i);
            /*j traverses each(i-th) row of mat2
            **and repeatedly the same(k-th) row of res
            */
            for(uint32_t j=0; j<size/4; j++)
            {
                /*Multiply the ith row of mat2 with the ith element
                **of kth row of mat1 and accumulate the results
                **to the ith row of res
                */
                int32x4_t* nRow = (int32x4_t*) &(mat2->at(i,4*j));
                nRes[j] = vmlaq_n_s32(nRes[j], *nRow, scal);
            }
        }
    }
}
```

Listing 3.2: Code for Matrix multiplication using ARM SIMD intrinsics as implemented for the corresponding benchmark

For matrix multiplication, each row of the matrix on the left of the operator is element-wise multiplied with each column of the matrix on the right of the operator and the intermediate products are added to calculate a single element of the product matrix. For example to calculate element $R_{(i,j)}$ when multiplying matrices $a \cdot b$, each element in column i of matrix a is multiplied with the respective element in

column j of matrix b with the resulting products being added as it is shown in the example in figure 3.2.

Using the vector-by-scalar multiply-accumulate SIMD instruction, the algorithm *ARM* proposes for column-major order [40], is replicated for row major order. As depicted in figure 3.3 initially each element of the first row of the matrix (y) on the right of the operator (in red) is multiplied with the first element of the first row of the matrix (x) on the left of the operator (in blue) and accumulated with the first row of the result matrix. Subsequently, the second row of y (in purple) is multiplied with the second element of the first row of x (in green) and accumulated with the first row of the result matrix. The process continues until the whole first row of the result matrix is calculated and then the algorithm proceeds to the second row of the x matrix to calculate similarly the second row of the result matrix and so on until all the whole result matrix is computed. In this way, except for the SIMD units the cache locality is exploited as well since most of the accesses are sequential and most cachelines reused.

3.2.2. Regular Stencil Algorithms

Another widely used category of algorithms is the stencil algorithms, which, due to their importance in numerical solvers and physical simulation codes, are of particular interest to scientific computing research [48]. In such algorithms, specific cells in the neighbourhood of the cell to be updated are used to determine its new value. The number and the relative locations of the cells are depended on the algorithm and determine whether it mostly stresses the CPU or the memory hierarchy. Two widely used cases of neighbourhoods in cellular automata are Von Neumann, in which the updated value of the current cell depend on its immediate neighbours, namely the cell above it, the cell below it, the cell on its left and the cell on its right as depicted in figure 3.4, and Moore, in which except from the above, the immediate diagonal cells are included as well (see figure 3.5), increasing the value of the contributing cells to 9 [29].

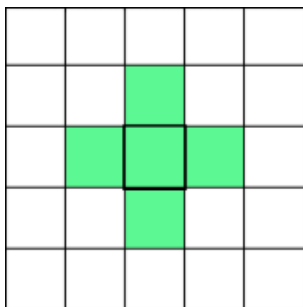


Figure 3.4: Von Neumann cell neighbourhood as depicted in [29]

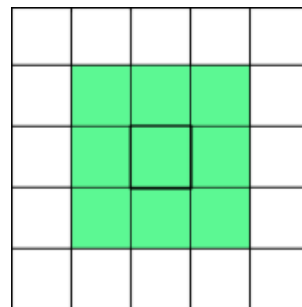


Figure 3.5: Moore cell neighbourhood as depicted in [29]

A particular algorithm using the stencil neighbourhood described above is the *Jacobian* iterative method used in numerical linear algebra. Each cell is updated with the average of the value of its neighbours minus the value of a reference matrix. Fixed-point arithmetic implementations of a similar logic algorithm for both Von Neumann and Moore neighbourhoods, presented in listings 3.3 and 3.4 respectively, were used to test the effect of an increase in computations per cell. In this case, hand-tuning using SIMD intrinsics did not provide an improvement in comparison to the code produced by the compilers and thus, the later was preferred.

Cellular automata and their stencil updating process expand to cases with more dimensions than 2 like numerical simulations of chemical processes [30]. A 3-dimensional neighbourhood expands the above concept to different pages except for row and column, resulting to 6 Von Neumann neighbours as depicted in figure 3.6. Again the hand-written SIMD implementation was outpaced by the one produced for the straightforward implementation by the compiler (see listing 3.5) and thus, the latter was used.

```

static inline void jacobLine4p_row_update(NPowerMatrixBuffer<fp>* inp,
    NPowerMatrixBuffer<fp>* bin, fp* __restrict res, const uint32_t row,
    const uint32_t size)
{
    for(uint32_t i=1; i<size-1; i++)
    {
        /*Update each cell of the row with the average of its
        **Von Neumann neighbours minus the value of a reference matrix
        */
        res[i] = (inp->at(row-1,i) + inp->at(row+1,i) + inp->at(row,i+1) +
            inp->at(row,i-1) - bin->at(row,i)) / 4;
    }
}

```

Listing 3.3: 2D Stencil using Von Neumann neighbourhood as implemented for the corresponding benchmark

```

static inline void jacobLine8p_row_update(NPowerMatrixBuffer<fp>* inp,
    NPowerMatrixBuffer<fp>* bin, fp* __restrict res,
    const uint32_t row, const uint32_t size)
{
    for(uint32_t i=1; i<size-1; i++)
    {
        /*Update each cell of the row with the average of its Moore neighbours
        **minus the value of a reference matrix
        */
        res[i] = (inp->at(row-1,i-1) + inp->at(row-1,i) + inp->at(row-1,i+1)
            + inp->at(row+1,i-1) + inp->at(row+1,i) + inp->at(row+1,i+1)
            + inp->at(row,i+1) + inp->at(row,i-1) - bin->at(row,i)) / 8;
    }
}

```

Listing 3.4: 2D Stencil using Moore neighbourhood as implemented for the corresponding benchmark

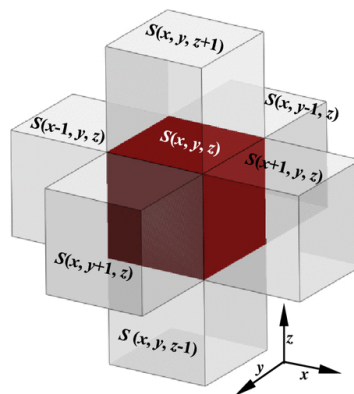


Figure 3.6: Von Neumann 3D cell neighbourhood as depicted in [30]

```

static inline void jacobLine6p_row_update(NPowerMatrixBuffer<fp>* inp,
                                         NPowerMatrixBuffer<fp>* bin, fp* __restrict res, const uint32_t page,
                                         const uint32_t row, const uint32_t size)
{
    for(uint32_t i=1; i<size-1; i++)
    {
        /*Update each cell of the row with the average of its
        **Von Neumann neighbours minus the value of a reference matrix
        */
        res[i] = (inp->at(page-1,row,i) + inp->at(page+1,row,i) +
                  inp->at(page,row-1,i) + inp->at(page,row+1,i) +
                  inp->at(page, row, i-1) + inp->at(page, row, i+1) -
                  bin->at(page, row, i)) / 6;
    }
}

```

Listing 3.5: 3D Stencil using Von Neumann neighbourhood as implemented for the corresponding benchmark

Despite the large distance of the values in the different pages, this algorithm, similarly to all the above, shows a high degree of both spatial and temporal locality as with each increase of the index, all the pointers to the contributing cells move one cell to the left, allowing the cache hierarchy and simple prefetching techniques like a hardware stream prefetcher to produce high cache hit rates.

3.2.3. String Manipulation

As documented in section 1.1.1, Natural Language Processing is a popular field of *big data analytics*. This field utilises an extensive collection of string manipulation algorithms, which due to the variability in length of the strings, sometimes lead to irregular access patterns, in contrast to the above documented scientific algorithms. Consequently, it is of particular interest to test the performance of memory hierarchy and prefetchers for such cases.

A simple algorithm of this sort is capitalising or lowercasing the first letter of a string as depicted in figure 3.7. For this process, just the first byte of the whole string is needed with the rest being irrelevant. When the length of the string is short, for example when the capitalising algorithm is executed in words granularity, then multiple strings reside on the same cache block and every cache block of the table is needed, resulting in a streaming application like the aforementioned algorithms in this section. But when the granularity decreases to sentence or paragraph, then a single cacheline is not sufficient to host the whole string, which might span to more than two blocks in the second case. Then the access patterns are dependent on the size of the string, leading to an irregularity. In addition to that, this algorithm is extremely memory-intensive as for each string loaded just a logic operation is performed, which in the case of the paragraph granularity, this translates to one operation per cacheline.

An implementation of the above algorithm, shown in listing 3.6, along with a program to pre-process pieces of text to produce arrays of strings with word, sentence and paragraph granularity was implemented. Due to the gather and scatter nature of the algorithm, SIMD performed worse as expected. The text fed as an input to the benchmark was produced by a random text generator based on the "Lorem ipsum" part of the Cicero document "De finibus bonorum et malorum".

```

static inline void capitalize_first(ListArrowBuffer<char>* arr, const uint32_t len)
{
    for(uint32_t i=0; i<len; i++)
    {
        //Get the first letter of each string and reset the corresponding bit
        arr->at(i,0) &= ~0x20;
    }
}

```

Listing 3.6: Algorithm for capitalising the first letter of each string in an array

element of every list and every third list respectively for $step = 1$ as shown in figure 3.9 or of every N -th list and every $3N$ -th list for $step = N$ was to ensure that there will be some overlapping in the cache blocks accessed by the two different patterns as it is frequently the case in many algorithms. This implementation allows the study of the effect of the amount of processing per load, of the database size and of the distance between the indirect accesses.

```

const int32_t limit = num_clients/3;

for(int32_t i=INIT; i<limit; i+=step)
{
    //Load the fifth profit value of every (step)-th list
    uint32_t k = profit.at(i,5);
    //Load the fourth profit value of every (3*step)-th list
    uint32_t j = profit.at(3*i,4);
    int32_t l = ops;
    //Perform (4*ops+1) arithmetic operations with those values
    do
    {
        j+=(j-k)>>1;
        l--;
    }while(l>0);
    result += j;
}

```

Listing 3.7: Synthetic benchmark to measure the effect of the amount of processing per indirect access

3.3. Simulation Statistics

The level of detail in timing simulations *gem5* provides, allows for specific results for each simulated module to be provided for the whole duration or for a part of the simulation. Collecting, deciding which parts are relevant for each specific investigation and processing this information is an important step of exploiting the full potential of the simulator. Nevertheless, this configurability of the simulated modules and the difference in their implementation causes a non-uniformity in the results as specific statistics which are available for a module may not be produced for modules of the same category, limiting possible comparisons only to statistics available for all of them. The selection of statistics to compare the prefetching techniques is described in this section, using as a reference the metrics proposed in section 2.2.

As established in the chapter 2, the main goal of *memory hierarchy* and *prefetching* is to create the illusion of a memory system as fast as the levels close to the processing cores with a size similar to the levels further away in order to achieve the best possible performance. The established metric of performance in computing systems is the minimisation of execution time and for comparisons the introduced speed-up. The definition of execution time, regardless if measured in terms of CPU cycles or time units, is the intuitive quantification of the time required by a system to execute a specific piece of code. When the relative performance of two systems needs to be measured, speed up, which is defined as the ratio of the execution time in a reference system to the execution time in the under evaluation system as depicted in equation 3.1, is the standard metric utilised.

$$SpeedUp = \frac{ExecutionTime_{Reference}}{ExecutionTime_{UnderEvaluation}} \quad (3.1)$$

In order to improve performance, prefetching techniques attempt to predict which data are to be referenced shortly and request them from lower levels. Ideally, the prefetches will be timely and the data will be already in the cache when the reference takes place resulting in a hit. If the prefetch request is not issued early enough, the reference will result in a cache miss, though less costly in terms of latency and execution stalling. For both those cases, metrics to quantify frequency of such events are of interest to evaluate the coverage and timeliness of prefetching techniques. A combination of the *cache miss ratio* with the *overall miss latency* is an adequate indicator of the effectiveness of prefetching as it takes into account both the cases of successful prefetching. *Cache miss ratio* is defined as the ratio

of the cache access resulting in a miss to the total number of cache access as formalised in equation 3.2.

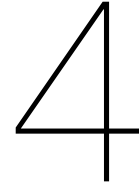
$$CacheMissRatio = \frac{CacheAccesses_{Miss}}{CacheAccesses_{Total}} \quad (3.2)$$

Despite the techniques attempting to hide the latency of cache misses in modern systems, like superscalar OOO execution, execution stalls due to memory misses are common. Measuring to what extent prefetching is able to alleviate those stalls when they occur could give an estimate of the number of misses actually covered by prefetching but were not timed successfully due to its limited aggressiveness or due to memory system related restrictions. Several statistics are offered by *gem5* for the cache miss latencies including an accumulation of the latencies of all cache misses and an average latency per miss. A raw accumulation of the latencies disregards the fact that many of those latencies overlap in OOO execution and thus, despite each possibly stalling a specific unit, the execution may partially continue rendering this latency less harmful performance-wise. An average of latency per miss, in addition to the overlapping effect, divides this latency with the total number of misses and as the number of which decreases, any outlier can create the illusion of high miss cost. For the above reason, the first option is preferred though scaled down by the data size to create a latency per accessed data cacheline metric as formalised in equation 3.3. Since this metric does not take into account the latencies of the cache hits or the overlapping, it is not an accurate estimation of the latency of for each cache block and its more relevant when used for comparisons.

$$LatencyPerCacheline = \frac{\sum_{\forall CacheMiss} Latency}{\frac{Size_{Dataset}}{Size_{CacheBlock}}} \quad (3.3)$$

Metrics for the accuracy and the redundancy of prefetches are more difficult to extract as while the number of unused cache blocks, which were evicted without being referenced, is reported, there is no statistic provided for the total number of prefetched cache blocks as only the total number of prefetch requests are counted, which accumulates both useful and redundant requests. An addition for measuring the number of used prefetched blocks was developed for the simulator, but for hardware prefetching only on misses cannot be updated correctly as the prefetcher, which is the only module able to keep track of the prefetch requests, is notified only on cache miss events. For prefetching on access it can give an estimate of the accuracy by measuring the ratio of the used prefetches to the sum of the used and the, evicted before used, prefetches as formalised in equation 3.4. Nevertheless, this ratio is not exact as there may be a non-negligible number of unnecessary prefetched blocks in the cache not evicted yet.

$$AccuratePrefetchRatio = \frac{PrefetchedBlocks_{Used}}{PrefetchedBlocks_{Used} + PrefetchedBlocks_{EvictedUnused}} \quad (3.4)$$



Ideal Prefetcher

An essential step to approach the research questions is to determine the effectiveness of the studied prefetchers on accelerating the execution of the selected microbenchmarks and specifically their ability to identify the particular access patterns they contain. Quantifying that involves the definition of an ideal prefetching technique, as documented in section 4.1, and configure the simulator in a way that the results of such a behaviour could be approximated. A proposal of such a configuration follows in section 4.2. Comparing the resulting cache behaviour with the one of the system when the different reference prefetchers are included, would provide evidence for whether there are cases in which popular prefetchers fail and how much room for performance improvement exists for more sophisticated techniques. The presentation of the results of such investigation and a discussion based on it follow in sections 4.3 and 4.4 respectively.

4.1. Ideal Prefetching Definition

As documented extensively in chapter 2.2, prefetching is an essential part of the modern processing systems to release the full potential of the memory hierarchy. Based on its properties and the associated metrics specified there, the ideal behaviour of a prefetching scheme comprises of certain requirements. Most importantly, its basic mission is to eliminate any misses by identifying the access patterns, use them to infer future accesses and issue prefetches, so, when accessed, the data to be already in the targeted memory hierarchy level (perfect coverage). Such a scheme should be able to identify exactly the patterns and issue prefetches only for addresses that will indeed be used on the short-term (perfect accuracy). Otherwise, irrelevant cache blocks will result in wasting memory bandwidth and possibly in immediate or accelerated eviction of useful data from the targeted cache level or the levels below it. A waste of memory resources could also take place when prefetches are issued for cache blocks already present in the targeted level. An ideal prefetcher should be able to distinguish which of the predicted future access can be satisfied without any prefetch and initiate requests only for the rest (eliminate redundancy).

Nevertheless, the requirements of ideal prefetching are not exhausted in perfect prediction of which data need to be prefetched. Timing those prefetches is crucial as well (perfect timeliness), as late requests will increase the access latency and consequently hinder performance, while an early issuing could lead to an eviction of the prefetched data before they are actually used or deprive other more high priority requests of the memory resources. Even if a prefetching scheme has all the required logic and information to identify any pattern and predict accurately the future accesses, timing the prefetches is a very demanding task which cannot be perfectly optimised as the necessary system information cannot be gathered from other units or is not available to this level at all. Furthermore, the more complex the systems become with more independent units able to request simultaneously data from the memory, ideal prefetching for all of them, becomes, if not already, unattainable.

Specific prefetching logic is suitable for different kind of access patterns, which leads semiconductor companies to include multiple prefetcher modules for their most powerful chips. For algorithms accessing sequentially contiguous memory regions like matrix operations (3.2.1) and regular stencil algorithms (3.2.2), the prefetching logic of streaming and stride prefetchers, as described in section 2.2.1,

could provide good coverage. For algorithms with indirect indexing accesses like those described in sections 3.2.3 and 3.2.4, the arrays of indices, if accessed in a regular way, can be accommodated by stride prefetchers and if accessed sequentially by stream prefetchers as well. Though prefetching the actual data in those cases needs a more complex logic and thus, the aforementioned prefetchers are expected to perform inadequately.

4.2. Simulating Ideal Prefetching

No module to simulate the behaviour of ideal prefetching is available in *gem5* and implementing one would not be a simple task mainly due to the issue of satisfying the timeliness requirement, which would necessitate gathering information from various modules of the simulator and develop a fairly complex algorithm with dubious results. Setting aside the timeliness requirement disregards how the prefetch-induced data transferring across the system affects and is affected by memory traffic contending for the available bandwidth, but is the only viable solution for this initial investigation. On the other hand, such a decision simplifies significantly the simulation process since designing a separate module is not necessary for algorithms with deterministic access patterns as prefetching the data beforehand would be sufficient to emulate the rest of the properties of ideal prefetching discussed above.

More specifically, exploiting the information available before the execution of which data will be accessed by each benchmark and prefetching them before the start of the execution will result indeed in perfect coverage and accuracy and no redundancy. In order to implement this, a similar to the invalidation process described in chapter 3.2 was followed, but this time instead of invalidation instructions for every block containing offsets or values of any of the produced *Arrow arrays*, prefetch instructions targeting the L1D cache were issued with the hints that they will be used for load and should be retained in the cache (PLD1KEEP [3]). The hint about the use of the data does not have any effect in this particular cache level as there is no other core to share the data and thus, the coherence protocol does not have any impact.

Though for such an approach to be realisable, the size of the cache should be able to accommodate each and every piece of data the algorithm is going to access. This would constrain the data size to be smaller than the size of the targeted cache, the L1D cache in this case. For larger data sizes, closer to the case of *big data* but sufficiently small not to require unrealistic simulation times, to be tested, the size of the cache should be increased to exceed comfortably the size of the data.

Due to the cache associativity, discussed in section 2.1.2, memory blocks of the dataset could be mapped to the same cache block, especially with the relatively small values of associativity of the L1 caches. To prevent that from happening, either the size of the cache should be increased until the low value of associativity does not result in any replacements or in addition to the size, the associativity should be increased as well to a value high enough to render the cache almost fully-associative. For the validation of the data being fully accommodated by the L1D cache, 0 cache misses and replacements should be reported in the simulation results.

4.3. Results

The comparative evaluation of the effectiveness of the studied reference prefetchers in comparison to an ideal one, has to be fair and realistic in terms of their configuration. Aiming towards this, a simple tuning process focusing on the main parameters of prefetching was performed initially, the results of which are presented in section 4.3.1. After tuning the prefetchers, an evaluation of their effectiveness for different sizes and access patterns was conducted (section 4.3.2) to estimate whether there is room for improvement for more sophisticated approaches in any of those cases. Except for the *Stride* prefetcher, two versions of the stream prefetcher were tested, one more aggressive issuing prefetches on every access (denoted as "*streamOnAccess*") and one more conservative which prefetches only after misses (denoted as "*streamOnMiss*").

4.3.1. Reference Prefetchers' Tuning

During the short presentation of prefetching in section 2.2, two main parameters were discussed, namely the prefetch distance and degree. Out of these two, the implementation of the *stride* and *stream* prefetcher modules in *gem5* allowed only for the degree to be configured, limiting any possible investigations, with the distance being hardcoded to 1. Consequently, in the initial investigations for determining whether each of the reference prefetchers can approach the performance of the ideal

prefetcher in the, more tailored to their capabilities, streaming benchmarks, only the degree was tested as a parameter leaving the rest of the configuration options of the modules (like queue and table sizes, confidence thresholds etc) to the default values specified in the corresponding *gem5* script. As the field of *big data* was the main focus, for the tuning process of the prefetchers the input size was set to the maximum of the tested options, in order to comfortably outsize the L2 cache. The results of tuning process, grouped by benchmark, follow.

Matrix Operations

For both the In-order (figures 4.1, A.1 and A.2) and OOO (figures 4.2, A.3 and A.4) cores, a saturating improvement in all aspects of the memory accessing behaviour is observed as the prefetch degree increases in matrix addition. For all the execution time curves (figures 4.1 and 4.2), the knee is located around the degree of 3 with the latency curves (figures A.1 and A.3) exhibiting a very similar trend. For the OOO core though, the saturating behaviour does not expand to the cache miss rate curve (figure A.4) which despite the almost complete saturation in the decline of the execution time, it continues to decrease as the degree increases.

While the aforementioned saturating trend with a knee around 3 expands to the In-Order execution of matrix multiplication (figures 4.5, A.9 and A.10) and the OOO execution when prefetchers operating on access basis are used, the findings are different for the *Stream On Miss* prefetcher, the benefit of which is partially reversed when its degree is increased beyond 3 as shown in figures 4.6 and A.11. The miss rate exhibits the same strictly decreasing trend though with more abrupt changes in rate as can be observed in figure A.12.

Regular Stencil Algorithms

For the regular stencil algorithms, gradual saturation is again observed. Only the distance between the extreme values and the rate of saturation differ, while the location of the knee is shifted closer to the degree of 2. Increasing the number of iterations does not have any notable impact on the shape of the curves, thus only for the Von Neumann neighbourhood stencil the curves for both the execution of 1 iteration (figures 4.9, A.17 and A.18 for the In-Order and 4.10, A.19 and A.20 for the OOO) and 10 iterations will be included (figures 4.11, A.21 and A.22 for the In-Order and 4.12, A.23 and A.24 for the OOO). Increasing the size of the neighbourhood produces a reduction in cache misses for both In-Order and OOO cores as shown in figures A.31 and A.34 respectively, which moves the execution time curves closer to the ones of the ideal prefetcher, but increases the overall execution time only for the In-Order core (figure A.29) as the numerous computation resources of the OOO are able to nullify the effect of the extra computation load (figure A.32). Finally, for the 3D Von Neumann neighbourhood, the more conservative *Stride* prefetcher seems to be affected in terms of latency (figures A.45 and A.47 for In-Order and OOO respectively) and miss rates (figures A.46 and A.48 for In-Order and OOO respectively) by the irregularity of the partial updating of the matrix with data of the intermediate smaller matrix implemented for memory size optimisation purposes. However, as it can be observed in figure 4.18 the OOO core was able to almost completely hide the extra latency in contrast with the In-Order core which failed to do so adequately (figure 4.17).

4.3.2. Ideal Prefetcher Comparison

Setting aside the particularities of the tuning results, which will be discussed more extensively in the next section, the common denominator of all of them is that the saturation of the speed up each prefetcher contributes was reached before the degree of 4. Choosing a degree of 3 would be sufficient to capitalise on most of the benefits of all the prefetchers, though since performance improvement is the main goal and no multicore scenario was tested to identify possible inter-core effects due to the pressure an increasing degree could cause in the memory system, the more aggressive choice of 4 was preferred. After using the streaming benchmarks to determine which degree is sufficient to approach an ideal performance without overloading the memory system with requests which would yield little further improvements, the tuned prefetchers were tested for various sizes and access patterns to understand the effect of those parameters and conclude whether more targeted approaches are worth to be considered.

Matrix Operations

Matrix addition is a fairly memory intensive algorithm as analysed in 3.2.1 and as a result when there are sufficient resources to exploit it, any improvement in memory behaviour could have a significant

impact. This was confirmed as the powerful OOO core with ideal prefetching was able to achieve a 25x speed up over a configuration without prefetching and close to 15x speed up over the rest of the prefetchers as shown in figure 4.4. The smaller In-Order core on the other hand was not able to achieve a similar speed up with the ideal prefetching being less than 3 times faster than the On-Access prefetchers (figure 4.3). While the latencies seem to have an almost identical shape with the speed-up curves (figures A.5 and A.6), the cache miss rates present significant differences (figures A.7 and A.8).

For matrix multiplication the input size seems to have a distinct performance impact for the both the latencies (figures A.13 and A.14) and the miss rates (figures A.15 and A.16) due to the re-use of the data and the presence of the L2 cache. The computation intensity of the algorithm though allows both CPUs with the contribution of prefetching to almost eradicate any performance impact for larger input sizes as depicted in figures 4.7 and 4.8.

Regular Stencil Algorithms

Two dimensional stencil algorithms do not exhibit any variability in execution time with the increasing size for a single iteration with both latencies and miss rates remaining very low for both In-Order and OOO execution as it can be observed in figures A.25, A.26, A.35, A.36, A.27, A.28, A.37 and A.38. Prefetching on access succeeds in almost matching the execution times of the ideal prefetcher (figures 4.13, 4.14, 4.15 and 4.16), leaving virtually no room improvement. When adding more iterations, the L2 cache is able to limit even further the execution time for data sizes fully accommodated by it, nullifying any speed-up of the ideal prefetcher over the reference ones (figure 4.15 and A.40).

The 3D stencil case is more of interest as the effect of the L2 cache is visible even in the single iteration case (figures 4.19 and 4.20), as the accesses on the previous page are evicted by the L1D cache but for small input sizes the data are still in the L2 cache with high probability. Similarly to the 2D stencils, the prefetchers are able to identify and prefetch accurately the access patterns limiting the speed-up of the ideal prefetcher very close 1x. For multiple iterations, the aforementioned effect of the L2 cache is shrinking even further any advantage of the ideal prefetcher as shown in figures 4.19 and A.54.

String manipulation

As documented in section 3.2.3, string manipulation algorithms contain a significant amount of indirect accesses due to the variability in length of the strings. The capitalising algorithm with word granularity produces very low miss rates for all the sizes in both cores as it can be seen in figures A.61 and A.62, close to 0 when a *Stream On Access* prefetcher is used, as expected due to its streaming nature and the multiple references per cacheline. As a result streaming prefetcher achieves an execution time very close to the ideal prefetcher for this case (figures 4.21 and 4.22).

As the granularity decreases, the first letter of each sentence may potentially reside in the same, the next or the one after the next cacheline and thus, each cache block is accessed at most twice, leaving less time for the prefetch requests to be fetched from main memory and thus, increasing the miss rates significantly as presented in figures A.65 and A.66. In the OOO execution this leads to performance more than 5 times faster for the ideal prefetcher (figure 4.24, while for the In-Order core which has less resources to exploit the ideal prefetching, the speed-up is limited to a bit more than 2x (figure 4.23).

The above trend is aggravated in the case of the paragraph granularity, as in addition to the prefetchers only being able to follow and prefetch the sequential accesses on the *offsets buffer* and not the *value buffer* accesses due to the variable, possibly multi-cacheline distance between them, each cache block is accessed at most once. This is reflected on the increased cache miss rates presented in figures A.69 and A.70, though this allows for more time for the memory to proceed with the prefetch request making subsequent misses less costly as thus lowering the average miss latency per cache line for the OOO core (figure A.68). The resulting speed-up for the ideal prefetcher as a result is only marginally increased in comparison to the sentence case as shown in figure 4.26. For the In-Order core the gains are much more significant as shown in figure 4.25.

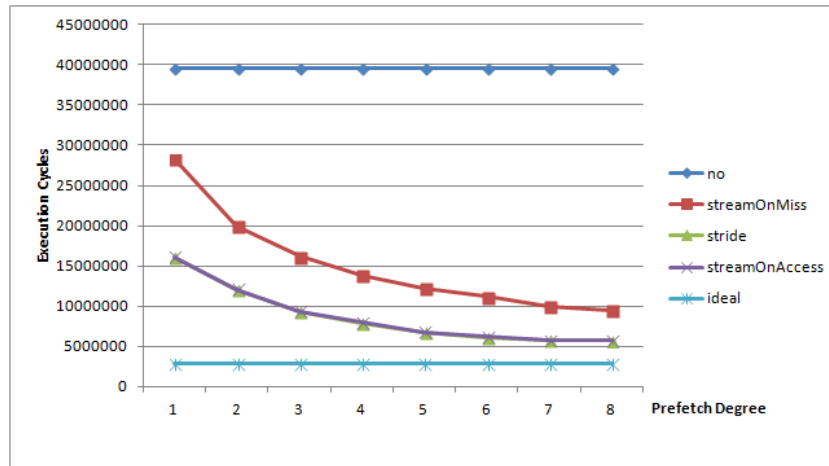


Figure 4.1: Execution time of systems featuring the tested prefetchers when matrix addition is executed on the In-Order core as their degree increases

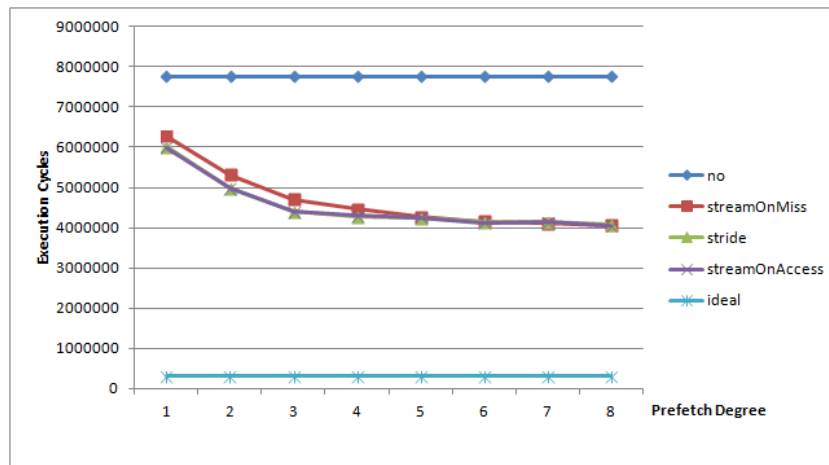


Figure 4.2: Execution time of systems featuring the tested prefetchers when matrix addition is executed on the OOO core as their degree increases

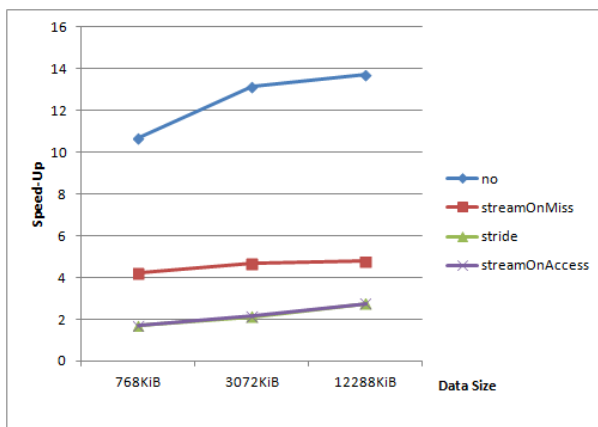


Figure 4.3: Speed-up of the Ideal Prefetcher when matrix addition is executed on the In-Order core over the reference prefetchers as the input size increases

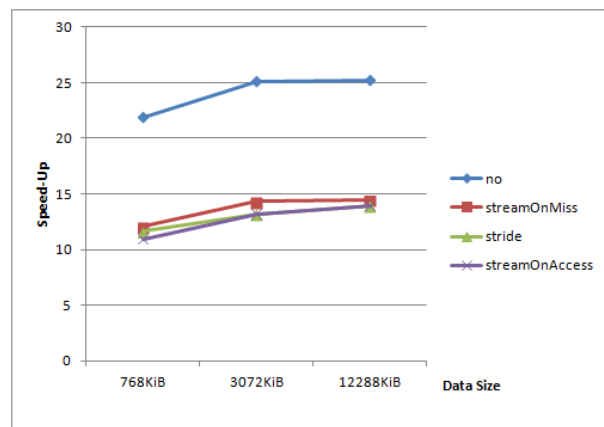


Figure 4.4: Speed-up of the Ideal Prefetcher when matrix addition is executed on the OOO core over the reference prefetchers as the input size increases

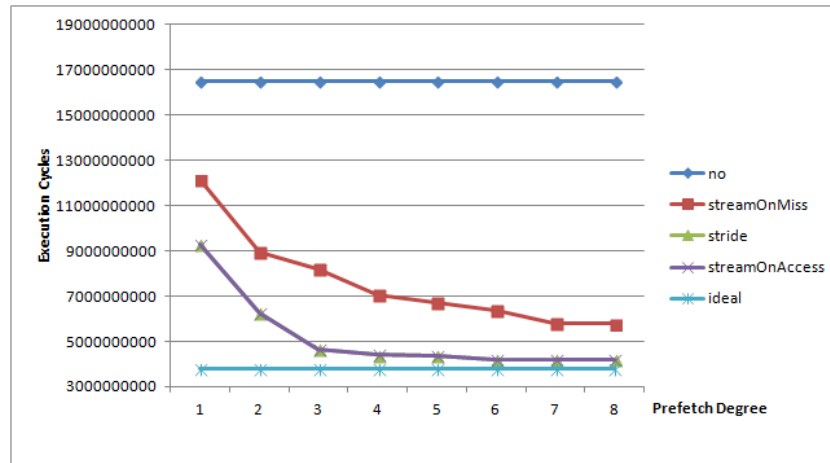


Figure 4.5: Execution time of systems featuring the tested prefetchers when matrix multiplication is executed on the In-Order core as their degree increases

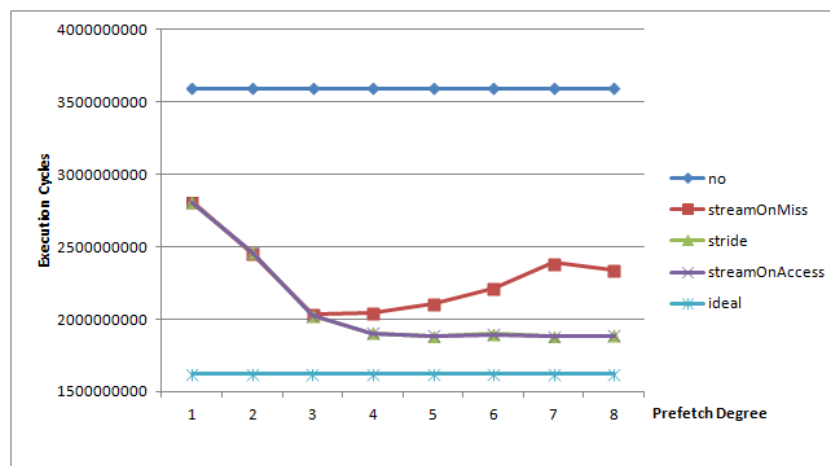


Figure 4.6: Execution time of systems featuring the tested prefetchers when matrix multiplication is executed on the OOO core as their degree increases

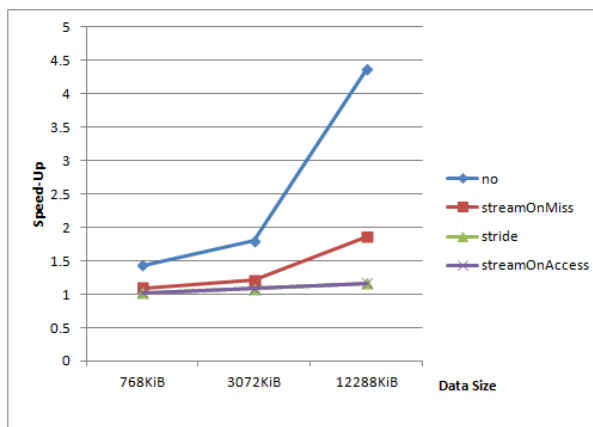


Figure 4.7: Speed-up of the Ideal Prefetcher when matrix multiplication is executed on the In-Order core over the reference prefetchers as the input size increases

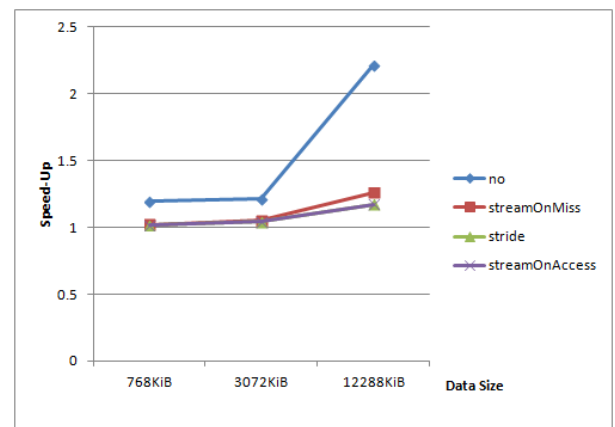


Figure 4.8: Speed-up of the Ideal Prefetcher when matrix multiplication is executed on the OOO core over the reference prefetchers as the input size increases

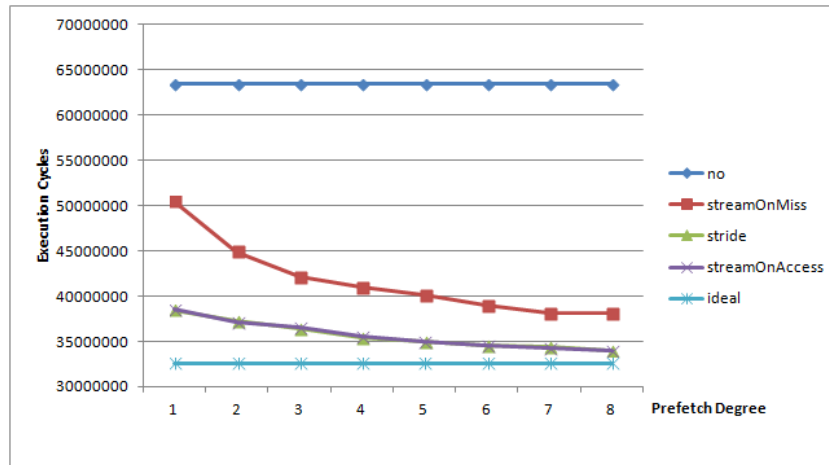


Figure 4.9: Execution time of systems featuring the tested prefetchers when a single iteration of Von Neumann Neighbourhood stencil is executed on the In-Order core as their degree increases

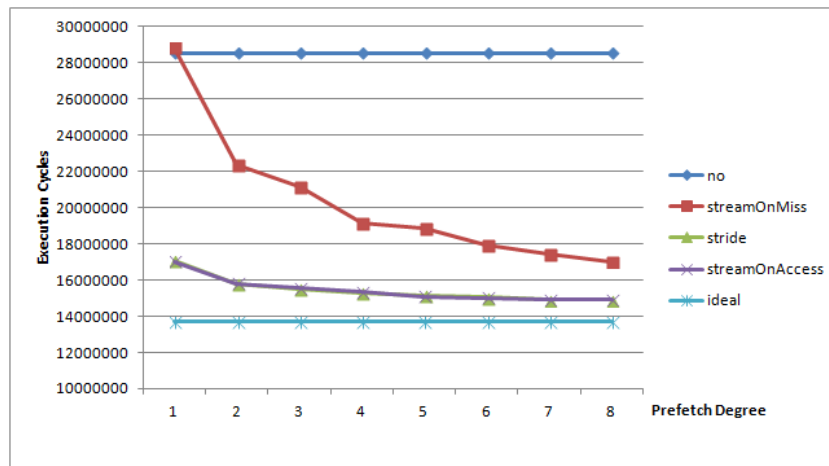


Figure 4.10: Execution time of systems featuring the tested prefetchers when a single iteration of Von Neumann Neighbourhood stencil is executed on the OOO core as their degree increases

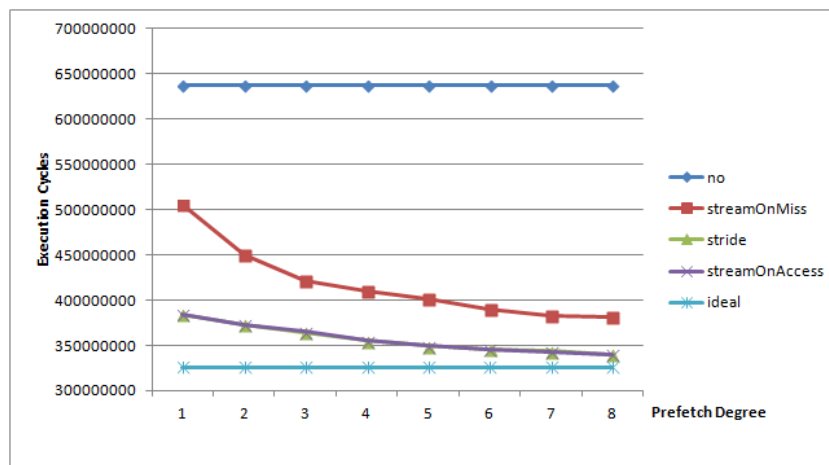


Figure 4.11: Execution time of systems featuring the tested prefetchers when 10 iterations of the Von Neumann Neighbourhood stencil are executed on the In-Order core as their degree increases

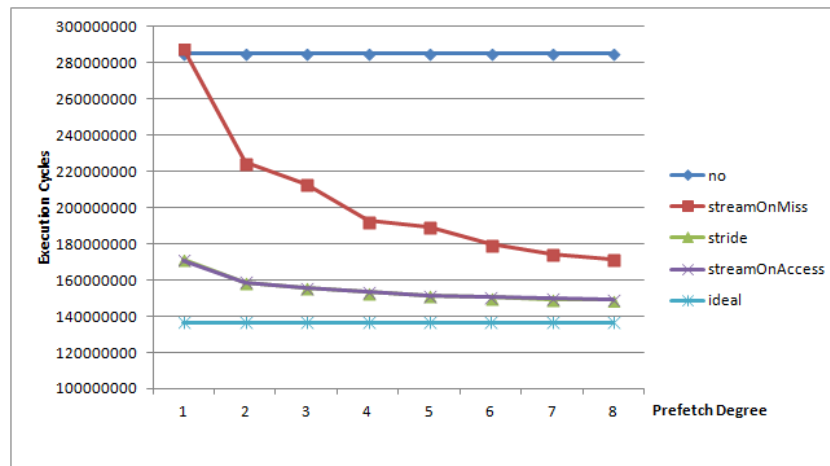


Figure 4.12: Execution time of systems featuring the tested prefetchers when 10 iterations of the Von Neumann Neighbourhood stencil are executed on the OOO core as their degree increases

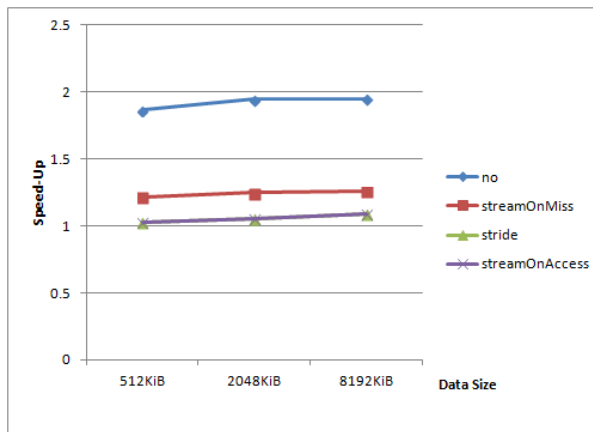


Figure 4.13: Speed-up of the Ideal Prefetcher when a single iteration of the Von Neumann Neighbourhood stencil is executed on the In-Order core over the reference prefetchers as the input size increases

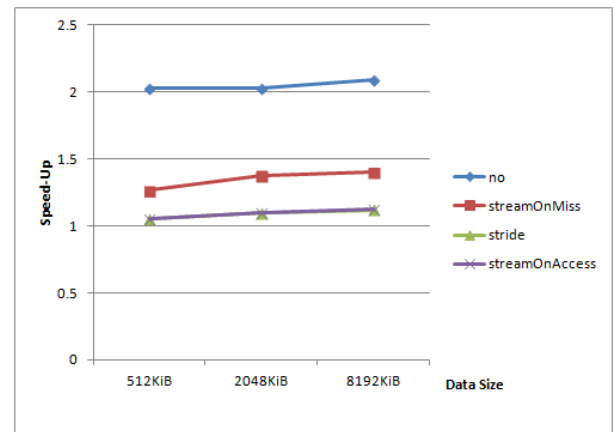


Figure 4.14: Speed-up of the Ideal Prefetcher when a single iteration of the Von Neumann Neighbourhood stencil is executed on the OOO core over the reference prefetchers as the input size increases

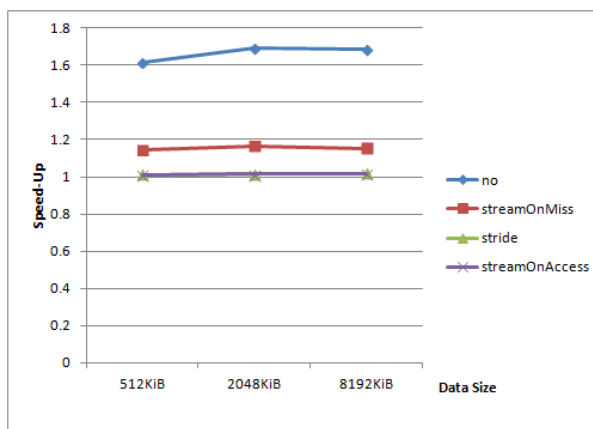


Figure 4.15: Speed-up of the Ideal Prefetcher when a single iteration of the Moore Neighbourhood stencil is executed on the In-Order core over the reference prefetchers as the input size increases

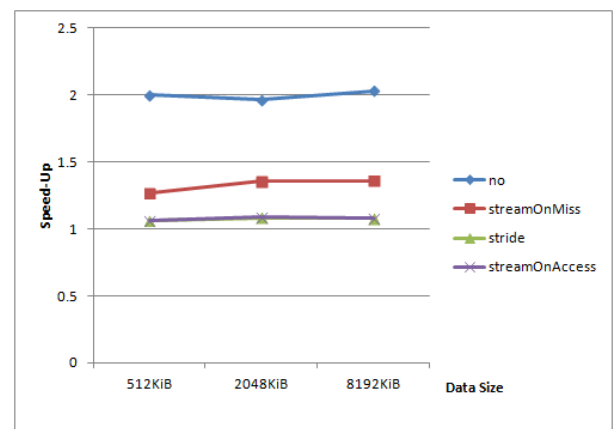


Figure 4.16: Speed-up of the Ideal Prefetcher when a single iteration of the Moore Neighbourhood stencil is executed on the OOO core over the reference prefetchers as the input size increases

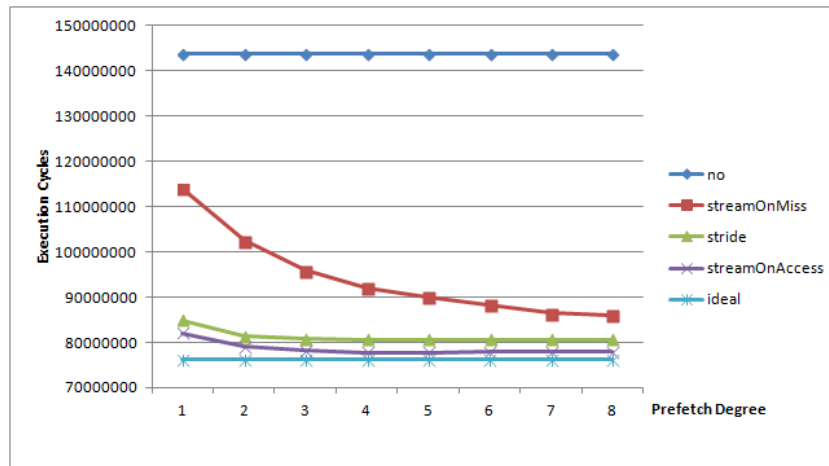


Figure 4.17: Execution time of systems featuring the tested prefetchers when a single iteration of 3D Von Neumann Neighbourhood stencil is executed on the In-Order core as the degree increases

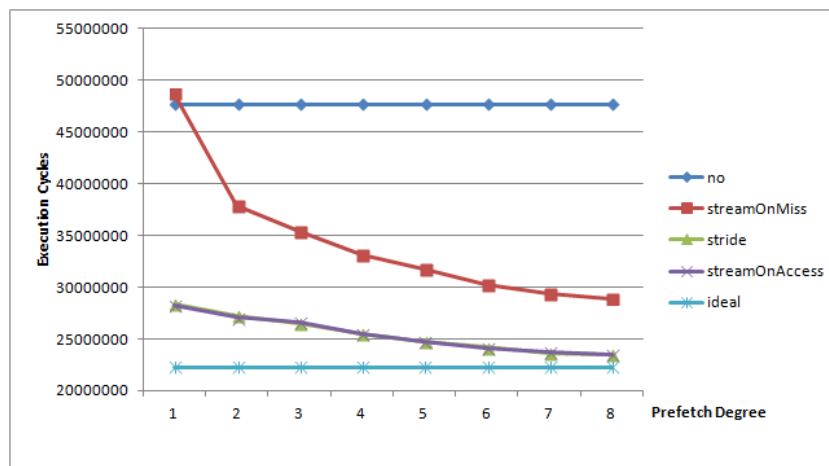


Figure 4.18: Execution time of systems featuring the tested prefetchers when a single iteration of 3D Von Neumann Neighbourhood stencil is executed on the OOO core as the degree increases

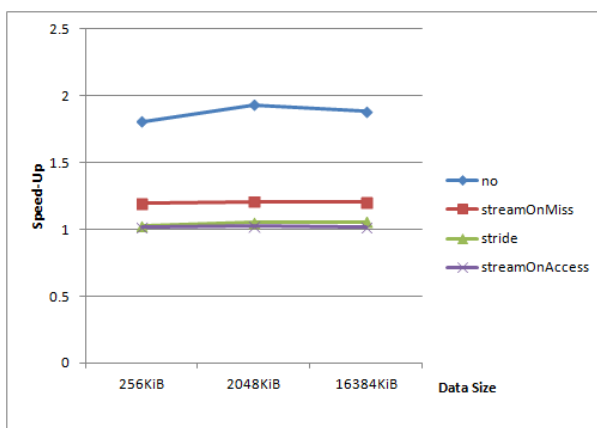


Figure 4.19: Speed-up of the Ideal Prefetcher when a single iteration of the 3D Von Neumann Neighbourhood stencil is executed on the In-Order core over the reference prefetchers as the input size increases

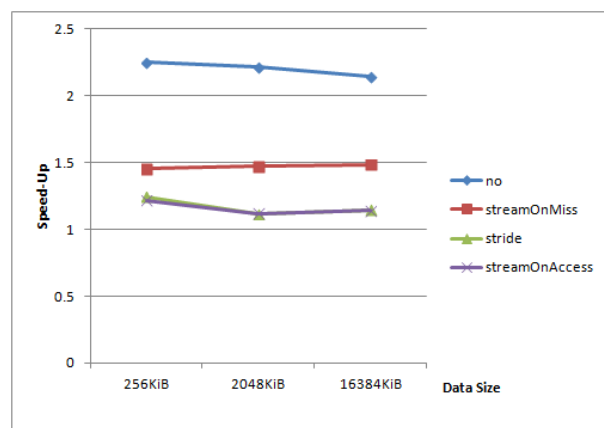


Figure 4.20: Speed-up of the Ideal Prefetcher when a single iteration of the 3D Von Neumann Neighbourhood stencil is executed on the OOO core over the reference prefetchers as the input size increases

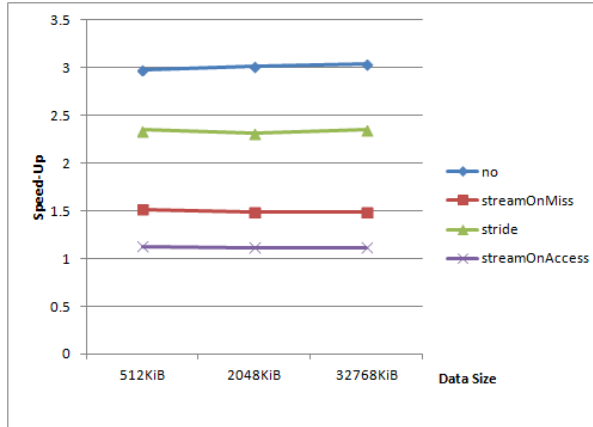


Figure 4.21: Speed-up of the Ideal Prefetcher when the word's first letter capitalisation algorithm is executed on the In-Order core over the reference prefetchers as the input size increases

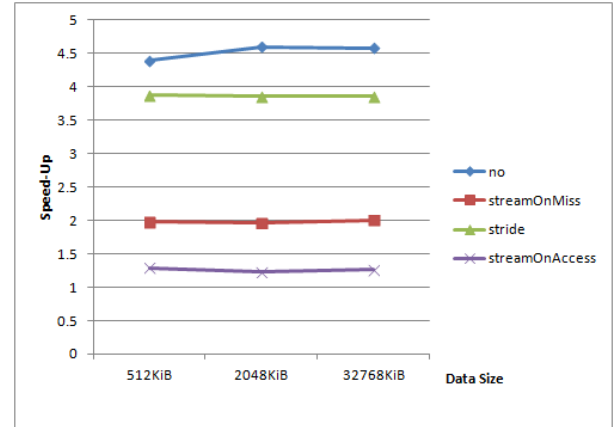


Figure 4.22: Speed-up of the Ideal Prefetcher when the word's first letter capitalisation algorithm is executed on the OOO core over the reference prefetchers as the input size increases

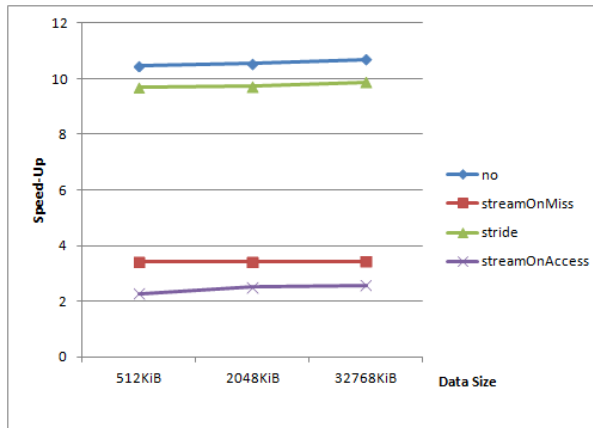


Figure 4.23: Speed-up of the Ideal Prefetcher when the sentence's first letter capitalisation algorithm is executed on the In-Order core over the reference prefetchers as the input size increases



Figure 4.24: Speed-up of the Ideal Prefetcher when the sentence's first letter capitalisation algorithm is executed on the OOO core over the reference prefetchers as the input size increases

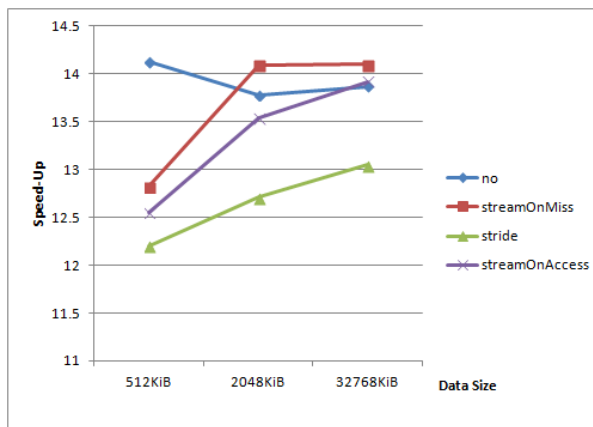


Figure 4.25: Speed-up of the Ideal Prefetcher when the paragraph's first letter capitalisation algorithm is executed on the In-Order core over the reference prefetchers as the input size increases

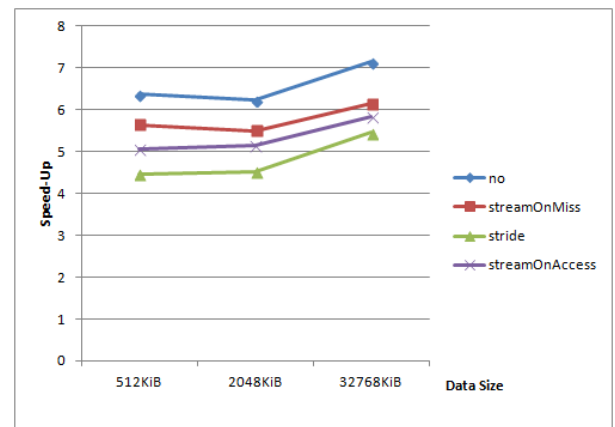


Figure 4.26: Speed-up of the Ideal Prefetcher when the paragraph's first letter capitalisation algorithm is executed on the OOO core over the reference prefetchers as the input size increases

4.4. Discussion

Although most of the aspects of the behaviour of the prefetchers, as presented in the figures of the previous section (4.3), were intuitively expected, there were several points worthy of more discussion. Those will be grouped and discussed in a per benchmark basis, followed by an overview of the important remarks which were used to decide on the next steps.

Matrix Operations

The execution time and speed-up plots for matrix addition confirm that due to the streaming nature of the benchmark, execution time scales almost proportionally with the size for all the prefetching configurations. The slight increase in speed-up of the ideal prefetcher, the execution time of which increases exactly proportionally with the size, over the rest of the configurations, especially between the values 768KiB and 3072KiB can be attributed to the main memory configuration. The DRAM of the system is comprised of 2 parallel channels, with 2 ranks per channel, 8 devices per rank with each device featuring a row buffer with a length of 1KB and, as the DDR4 specification requires, 16 banks. The number of bytes which can be accessed with internal DRAM latency close to the minimum possible (t_{CL}) is calculated below:

$$DataSize_{Lat \approx t_{CL}} = 2Channels \cdot 2 \frac{Ranks}{Channel} \cdot 8 \frac{Devices}{Rank} \cdot 16 \frac{Banks}{Device} \cdot 1KB \frac{RowBufferSize}{Bank} = 512KB \quad (4.1)$$

After the above size, larger latencies are experienced internally in the DRAM rendering the optimisation of the accesses more difficult for the memory controller. The extra latency has initially a positive impact for the miss rate of the OOO core for prefetching configurations as the costlier misses allow for more time for the prefetched blocks to be transferred from memory resulting in cache hits, but as the congestion in memory increases with the size due to the memory intensity of the algorithm and the number of prefetches, this positive effect is diminished. This effect is not experienced though by the In-Order core as after a miss the execution stalls and thus no more accesses and in turn more prefetches are produced until it continues. As a result, the very low miss rate increases as the accesses become costlier.

The increasing degree of the tested prefetchers has predictable results as well, with the latency per cache block and the miss rate dropping until they saturate. This saturation occurs due to the exhaustion of the capabilities of the memory system with more requests contending for bandwidth as the degree increases. For the *Stream on Miss* prefetcher in the OOO execution, the saturation is not reached on its miss rate as groups of consecutive misses trigger series of prefetches which result in more hits as the series gets larger. But this bursting scheme renders the fewer misses costlier thus the saturation is reached in both the per cacheline latency and the execution time.

Matrix multiplication execution time is significantly affected by size due to the data reuse. For the size of 768KiB, both cores are able to take advantage both of the minimal DRAM latency for most of the accesses when the data are brought from the memory initially and of the much smaller latencies of the L2 cache every time they are referenced again. As the size increases and the dataset cannot be fully accommodated by the L2 cache, larger latencies are experienced both on the first and on the subsequent references. The effect is stronger the more conservative is the prefetching, with the no prefetcher configuration being the most affected and the aggressive, prefetch on access, configurations being only marginally slower than the ideal prefetcher one. The same picture is observed for the latencies per cacheline for the same reasons. Nevertheless, the plots of the miss rate differ as the In-Order core, due to its slow execution and the stalls, allows for more time for the prefetching requests to arrive from the lower levels and when they fail to do so on time, only one miss is recorded. In the OOO order core in contrast, each lane accesses the memory independently since there are no data dependencies and thus, multiple misses are recorded for the same cacheline when the different lanes attempt to access it.

Increasing the degree cause the same speed-up saturation in both cores for all prefetchers except for the *Stream On Miss* prefetcher on the OOO core. Its difference in behaviour is caused by the access pattern of the algorithm as scalars from the first matrix are multiplied with whole rows from the second. When prefetching on miss, for the first matrix a prefetch will be issued only on the miss when the first scalar of the cacheline is accessed, while for the second matrix, prefetches will be produced on

a much more regular basis as the rate of cache blocks consumption is high especially when SIMD is used. Consequently, the prefetched cachelines of the first matrix will be evicted before they are used, rendering misses, when the border of a cache block is crossed, inevitable. Initially when the value of degree is low, any increase reduces the miss rate and cost of missing accesses to the second matrix. As the degree increases and those benefits start to saturate and the memory resources contention intensifies due to the number of requests issued per miss, the latency of the misses of the first matrix become significant and their impact on execution time crucial as they stall all the lanes simultaneously. In the In-Order core due to the stalling which limits the pressure to memory system and the absence of independent lanes, this behaviour is not replicated.

Regular Stencil Algorithms

The two dimensional stencils have a very predictable behaviour as the execution time scales proportionally to the data size, while the cache miss rate and the latency per cache block hardly change. The on access prefetchers are able to almost match the performance of the ideal prefetcher, limiting its speed-up to values very close to 1. Performing more iterations reduces significantly the execution time of the more conservative configurations as they are able to exploit the low latencies of the L2 cache. When the degree is increased, the saturation in performance is evident in all the plots regardless of the number of iterations.

In the case of the 3D stencil, the picture is almost the same in terms of performance as the size increases, with *Stride* prefetcher performing marginally worse. This can be attributed to the inability of the *Stride* prefetcher to identify a constant stride for the *memcpy* instructions used to copy the updated pages of the matrix from a smaller temporary one, like the following:

```
//Update previous page with the new values as the old are no longer needed
memcpy(&(V.at(i-1,0,0)), &(temp[((i+1)%2)*size*size]), sizeof(fp) * size * size);
```

Listing 4.1: memcpy instruction used to update the matrix with the calculated value stored in a small temporary matrix

With the use of the modulo operator, the first and the second page of the temporary matrix are copied alternately reversing the sign of the stride in each iteration and preventing the stride prefetcher to issue any prefetches for these accesses. This is a trade-off to decrease the memory usage of the algorithm as having the whole initial and updated matrix simultaneously is not needed. Its impact is negligible in terms of performance, close to 0 for the OOO, but is more visible in the latency and miss rate plots. Except for this constant accumulation of misses, the saturating with the degree speed-up follows the same trend as the other stencils.

String manipulation

The string manipulation with its indirect access patterns is of particular interest. The capitalisation algorithm with word granularity seem to be perfectly prefetched by the *Stream On Access* prefetcher as expected due to its streaming nature. *Stride* prefetcher is only able to identify a stride for the sequential accesses of the *offsets buffer* while fails to discern any for the value accesses due to the variability in the length of the words. Increase in size has no noticeable effect either in any of the metrics.

For the case of sentences there is more potential for the ideal prefetcher in terms of speed-up as it exceeds 2x and 5x for the In-Order and the OOO cores respectively. The under-performance of the *Stride* prefetcher, which reaches almost the level of the configuration without prefetching, stands out. The no prefetcher configuration performs worse than expected as well when compared to the other granularities. A possible cause for this observation is the erratic memory behaviour of the algorithm, as each cacheline may be accessed twice, once or not at all, producing every possible combination of hits and miss bursts, which pose a demanding challenge to the memory controller to optimise. Stream prefetching both *Offsets* and *Values* flatten out these random spikes, which improves performance significantly, while *Stride Prefetcher* is affected by it because as it is unable to foresee any pattern, it does not issue any prefetches for the *value buffer* at all. Size again is having a negligible performance effect with the execution times scaling normally with it.

More fluctuations are observed in the case of paragraphs, with a behaviour similar to the matrix addition as likewise the accesses produce streams of misses. The reason of those misses is again the fast consumption of the cachelines, but not due to the SIMD units but due to the single operation

per cacheline of the algorithm. The inability of the prefetchers to reliably prefetch data from the *value buffer* deteriorate performance, which for small sizes is affected less due to reasons related to the main memory configuration which were elaborated in the matrix operations subsection of the current section. The cautious approach of *Stride Prefetcher*, prefetching only the *Offsets* results in a better behaviour as there are no redundant or inaccurate prefetches contending for the bandwidth with the actual accesses. Nevertheless, the possible speed-up of an ideal prefetcher is further increased in comparison to the finer granularities for the In-Order core reaching 13x, while it stays relatively the same for the OOO at a bit more than 5x.

Overview

As discussed exhaustively in section 4.3 and in the current section so far, increasing the prefetching degree leads initially to a tangible improvement in performance, which is rapidly saturating after a value of 3. Further increases yield little benefit, while it accumulates pressure on the memory system, which in turn may lead to reversing part of the gains in some cases. Thus, from this point on, all the prefetcher configurations will be set to a prefetching degree of 4, in order to simulate a behaviour that is close to the saturation knee, but on the aggressive side of it as the current study aims at exploring high performance systems and, since only single core scenarios were studied, there is no evidence of negative interactions between cores from overloading the memory system.

Further increases to the size beyond the capacity of the L2 do not have any significant performance impact, especially when prefetching on access is enabled. The regular stencil and the matrix multiplication algorithms are perfectly accommodated by those prefetchers as well, with the speed-up of an ideal prefetcher over them very close to 1x, rendering any research for a more sophisticated prefetching scheme for them, irrelevant. In the matrix addition and the string first letter capitalisation algorithms, the ideal prefetcher is able to produce large speed-ups and thus, studying them is more of interest. For the matrix addition case though, all the speed-up can be attributed to reaching the bandwidth limit of the main memory, 19.2 GB/s, as the 4, 128 bits long, SIMD units of the OOO core are able to sustain a throughput between $1 \frac{SIMDInstr}{cycle} \cdot 4 \frac{Integers}{SIMDInstr} \cdot 4 \frac{Bytes}{Integer} = 16 \frac{Bytes}{Cycle}$ to $4 \frac{SIMDInstr}{cycle} \cdot 4 \frac{Integers}{SIMDInstr} \cdot 4 \frac{Bytes}{Integer} = 64 \frac{Bytes}{Cycle}$ (48 to 192 GB/s).

For the sentence and paragraph first letter capitalisation algorithms, SIMD was not used due to their gather and scatter nature, though this same property limits the ability of the main memory to sustain high bandwidth for those acceses. As a result, part of the speed-up of the ideal prefetcher can indeed be attributed to bandwidth limitations. Nevertheless, comparing the miss rate with the latency plots for both the In-Order (figures A.67 and A.69) and the OOO (figures A.68 and A.70) cores, while both the *Stride* and *Stream On Access* prefetchers sustain the same miss rates, the latency per cache block is higher for *Stream On Access*. A possible cause could be the issue of inaccurate prefetches, which consume bandwidth and increase the latency of the useful accesses, by the *Stream On Access* prefetcher. This argument along with the assumption that those inaccurate prefetches are stemming from the inability of the prefetchers to accurately prefetch the *value buffer* of the strings array are put to test in the following chapters with the use of the software and more targeted hardware prefetching.

5

Arrow Software Prefetching

In the previous chapter an investigation of the effect of ideal prefetching scheme on several algorithms was explored and compared with the cache behaviour of popular reference prefetchers. From this investigation, the indirect access patterns of string manipulation algorithms showed promising speed ups when a more sophisticated prefetching technique is used. The simplest path to follow in order to evaluate whether the speed-up of the ideal prefetcher is an artefact of the limited memory bandwidth or if prefetching could utilise memory more efficiently to bring the execution time closer to the ideal, is the use of software prefetching instructions for the patterns, which are speculated to be causing the high miss rates and hurting performance, namely the indirect indexing access. The definition of the requirements of a software prefetching technique applied on *Apache Arrow* data structures should satisfy and how it could be implemented will be discussed in sections 5.1 and 5.2 respectively. Simulation results on the benchmarks quantifying the possible benefits determining the profile of algorithms profiting the most will be subsequently presented in section 5.3 and discussed in section 5.4.

5.1. Definition

As presented in section 2.2.2, software prefetching is the technique of inserting specific prefetching instructions, which exploit information known before or during compile-time about the access patterns of the algorithm to improve its memory behaviour and its performance as a result. Despite the programmer having the choice to manually insert prefetching instructions, without careful consideration they can downgrade performance by issuing mistimed or redundant prefetches and thus, it is better if the technique can be implemented by the compiler in a more systematic way. Even the compiler implementations are facing the above challenges though, as neither the memory load in the system level, nor the contents of the caches could be predicted at compile time. A possible solution is the application of heuristic methods by the compiler, which depending on the targeted microarchitecture and the specific characteristics of each algorithm, will determine whether to introduce any software prefetches and the appropriate parameters for them.

Those software prefetching instructions are integrated, in the vast majority of cases, in programs executed by systems implementing already one or multiple hardware prefetching schemes. Consequently, a cooperation of hardware and software prefetching is essential to avoid any of the adverse effects and exploit the benefits documented in section 2.2.3. In order to achieve this cooperation, the access patterns the hardware prefetcher would be able to accommodate, should be estimated. From the remaining accesses, repeating ones performed in a systematic way with destinations predictable early enough in the execution should be considered as candidates for software prefetching.

More specifically, within the framework of *Apache Arrow*, in which indirect indexing is very common when accessing nested structures, there are several opportunities for software prefetching. Regardless of the nesting level, when any of the *offsets buffers* is traversed sequentially, with a constant stride or by using any other function able to be computed beforehand or early enough, in order to index the data, software prefetching can be considered. Examples of such accesses are any computation using only specific parts or members of multiple variable length structures like lists or strings stored in array, like the case of capitalising the first letter, computing the average grade of all students for any course using

an array containing the list of their grades etc.

Sequential or constant stride accesses either to the *offsets buffer* or to the *value buffer* can be sufficiently accommodated by a *Stride Prefetcher*. For the regular case of indirect indexing accesses (denoted in its general form by equation 5.1 with i increasing iteratively) resulting in variable strides, the loading of the *Offsets* ($Offset[index_{off}]$) for calculating the indices ($Offset[index_{off}] + index_{val}$), if performed using any constant stride ($s_{offset_{const}}$), can be identified by a *Stride Prefetcher* as well. However, the addresses of the accessed data in the *value buffer* (equation 5.3) depend on the contents of the corresponding offset. As a result, those accesses cannot be accommodated by any of the reference prefetchers, except if the difference of the indices ($Offset[index_{off_{cur}}] + index_{val_{cur}} - (Offset[index_{off_{prev}}] + index_{val_{prev}})$) remain small enough for the values to reside within either the same or subsequent cache blocks, a case in which a stream prefetcher can perform well. For accesses with larger indices difference, software prefetches could be beneficial if the indices arrive on the cache early enough. In order to ensure this, software prefetches could be issued for the indices as well, but such a choice would result in both software and hardware consuming bandwidth to prefetch the same data for the same accesses, increasing the possibility of adverse effects. The use of 32-bit long indices stored in contiguous buffers by *Apache Arrow* instead of pure memory pointers, enables more indices to be loaded per cacheline and increases the possibility of those required by the software prefetching to be in the cache sufficiently beforehand with the help of hardware prefetching.

$$V(index_{offset}, index_{value}) = V(s_{offset_{const}} * i + c_{offset_{const}}, c_{value_{const}}) = Value[Offset[s_{offset_{const}} * i + c_{offset_{const}}] + c_{value_{const}}] \quad (5.1)$$

$$Addr(Offset[s_{offset_{const}} * i + c_{offset_{const}}]) = baseAddr_{offset} + dataSize_{offset} * (s_{offset_{const}} * i + c_{offset_{const}}) \quad (5.2)$$

$$Addr(V(index_{offset}, index_{value})) = baseAddr_{value} + dataSize_{value} * (Offset[s_{offset_{const}} * i + c_{offset_{const}}] + c_{value_{const}}) \quad (5.3)$$

Except for the requirement for the cooperation between software and hardware prefetching, the form of the accessing schemes and the timely fetching of the indices used by the software prefetch instruction to determine the destination address in the *value buffer*, the *distance* (D) and the *degree* (K) of the prefetches, as depicted in figure 2.3, should be decided appropriately. Within this specific context, the distance can be defined as how far the next offset index ($i + D$) which will be used to prefetch the next *Value Address* ($Addr(Value[Offset[s_{offset_{const}} * (i + D) + c_{offset_{const}}] + c_{value_{const}}])$), should be from the current offset index (i). Similarly, the degree can be defined as the number of consecutive *Value Addresses* ($Addr(Value[Offset[s_{offset_{const}} * (i + D + N) + c_{offset_{const}}] + c_{value_{const}}])$, $\forall N \in [1, K]$) will be prefetched per software prefetch. The optimisation of both those parameters depend on the characteristics of the algorithm, such as the memory and compute intensity, on some architecture- and memory-related properties of the system, which can be partially taken into account during the compilation and some system quantities that change dynamically during the execution, like the memory resources contention. Therefore, a robust software prefetching scheme should take into account as many of those parameters as possible and be resilient to normal variations of the rest.

5.2. Implementation

From the requirements set above, it evident that determining whether software prefetching can benefit memory behaviour and as a result performance is dependent on the properties of both the executed program and the system it runs on. An exhaustive exploration of the effects of system parameters is impossible, though designing particular scenarios as representative as possible of categories of real systems could lead to useful conclusions. On the other hand, the memory behaviour of each program and how it interacts with software prefetching and its parameters is easier to be studied. Instead of

studying specific algorithms in isolation, a more general profile of programs able to be benefited was chosen to be synthesised and an attempt to correlate the parameters of this profile with the software prefetching parameters, in order to form a heuristic compilers can follow decide the value of the latter, followed.

The investigation of the effect of ideal prefetching in chapter 4, revealed that even if a prefetching scheme is capable of fully identifying the access patterns of an algorithm, its memory intensity may distance its memory performance from the ideal. Based on this observation, an attempt to determine which amount of computations per load is sufficient to render software prefetching profitable, and for those values, the combinations of prefetching distance and degree yielding the highest performance, is presented subsequently. As a base, the synthetic benchmark documented in section 3.2.4, was used, expanded with a parametric software prefetching scheme (listing 5.1).

```

const int32_t limit = num_clients/3;

int32_t prf_ind = INIT+DISTANCE*step;

/*Partition the algorithm in (limit/step*DEGREE + 1) loops,
**each performing bursts of DEGREE prefetches per load pattern
**and bursts of DEGREE iterations
*/
for(int32_t i=INIT; i<limit; i+=step*DEGREE)
{
    /*Initiate a prefetch burst by issuing DEGREE prefetches
    **for each of the access patterns without exceeding the array bounds
    */
    for(int32_t d=0; d<DEGREE && prf_ind < limit; d++)
    {
        /*Prefetch the cacheline containing
        **the 5th element of the (prf_ind)-th list of the array
        */
        prefetchCacheline(((uint64_t)&(profit.at(prf_ind,5)) & ~(CACHELINE_SIZE-1)));

        /*Prefetch the cacheline containing
        **the 4th element of the (3*prf_ind)-th list of the array
        */
        prefetchCacheline(((uint64_t)&(profit.at(3*prf_ind,4)) & ~(CACHELINE_SIZE-1)));
        prf_ind+=step;
    }

    /*Initiate a burst of DEGREE computation iterations
    **without exceeding the array bounds
    */
    for(int32_t d=i; d<i+DEGREE*step && d<limit; d+=step)
    {
        // Load the fifth profit value of every (step)-th list
        uint32_t k = profit.at(d,5);
        uint32_t j = profit.at(3*d,4);
        int32_t l = ops;
        // Perform (4*ops+1) arithmetic operations with those values
        do
        {
            j+=(j-k)>>1;
            l--;
        }while(l>0);
        result += j;
    }
}

```

Listing 5.1: A SW prefetching optimised version of the Synthetic benchmark listed in 3.7,

The algorithm was partitioned in pairs of bursts, with first a burst of *DEGREE* software prefetches per access pattern and subsequently a burst of *DEGREE* execution iterations of the original algorithm (explained for listing 5.1 in section 3.2.4). Consequently, in each iteration of the outer loop initially the prefetching of future accesses (indexed by *prf_ind*) is initiated and then the computations for the current indices are performed. The issue of prefetching in bursts will allow the memory controller to

better optimise the accesses, while adjusting the value of *DEGREE* can tailor the length bursts to the characteristics of the memory system and lead to an improvement in timeliness. In order to test the possibilities for the timing of the bursts, the prefetch index (*prf_ind*) before each iteration is always *DISTANCE* indices ahead of the current index until it reaches iteration limit. An increased value for distance would result in misses initially because the first *DISTANCE* values would not be prefetched, but may improve performance in the later parts of the execution because of the increased number of timely prefetches.

Organising the software prefetches in this way, reduces the contribution of the program in the timeliness of the prefetches, to the tuning of the degree, which determines the length of the bursts, and the distance, which determines the head start the prefetching has over the normal accesses. By tuning those parameters for different amounts of computations per execution iteration, an approximation of their optimal value for different computation intensities could be made. If the compiler could estimate the computation intensity in terms of computation operations per memory load for any algorithm accessing memory using such access patterns, common with *Apache Arrow*, it could use those estimations in a heuristic way to estimate the values of distance and degree. Of course in this process, the effect of the loads of the indices should be taken into account, but, at least for those close to the current index, cache hits should be expected with a higher probability because of the hardware prefetching.

Nevertheless, this method overlooks the impact of the memory congestion induced by other parts of the system. A more complete study should include this parameter in the tuning process in order to make it as resilient as possible to congestion fluctuations. In addition to that, architecture-related parameters such as the impact of the specific prefetching instructions were not taken into account as well. Since compilers target specific architectures while allow even more specialised targeting with the specification of microarchitectures as options, separate specialised studies for each architecture or even microarchitecture could be considered to improve the implementation of such a heuristic method by the compilers.

5.3. Results

In this section, the parameters chosen for the aforementioned investigation are briefly explained in the beginning, followed by the presentation of the analysis performed on the results. A more detailed discussion on them along with the reflection contribution in the process of deciding the next steps of the study follows in the next section (5.4).

Since the orientation of the prefetching mechanism is towards *big data*, a sufficiently large size was chosen for the dataset. Specifically, the number of clients was set to 196608 with the transaction profits for each client occupying on average 5 (1 to 9) cache blocks, which amount to $5 \cdot 64bytes = 320bytes$ per client and $196608 \cdot 320 = 60MiBs$. The initial value and the step were set to 0 and 1 respectively for the maximum number of iterations to be performed. For the tested set of values for the prefetching parameters, the tested degrees varied in the range of 1 to 8, while the distance ranged from 0 to 8, resulting in 64 combinations of values. Systems featuring each of the reference prefetchers were tested for all those combinations resulting in a sufficient accumulation of data under the limitations described in the previous section.

While for the minimum number of computations, increases in the degree and distance, up to a point yielded saturating speed-ups for both cores as it can be seen in figures 5.1, A.71 and A.72 for the In-Order core and 5.4, A.83 and A.84, their effect is more irregular for larger sizes (figures 5.2 and 5.3 and A.73 to A.82 for In-Order and 5.5 and 5.6 and A.85 to A.94 for OOO execution). Those irregularities are possibly caused by the memory controller being able to better optimise the bursts produced by some combinations of prefetching parameters than others. Thus, different levels of memory congestion may alter the memory behaviour of a particular combination, with the neighbouring combinations giving an indication of the direction they might move to.

Since no regular trend was identified in the aforementioned figures and determining the top performing combinations of parameters is not possible directly from them, those combinations for each case of computation loads were detected (table 5.1 for In-Order execution and 5.2 for the OOO) in the results and their performance was compared with this of the ideal prefetcher (figures 5.7 and 5.8 for In-Order and OOO cores respectively). For most of the computation loads, despite achieving a performance close the ideal, the top performing combinations are neighbouring with one or more configurations exhibiting considerably high execution times (peaks in the execution time figures) and any increase in

memory congestion due to bursts of requests from other units in the system could drive the memory behaviour towards these directions. In order to avoid such a scenario, further analysis of the results was performed in order to find configurations performing within a 5% range of the top performing one of the system with the same hardware prefetcher ($executionTime_{candidate} \leq 105\% \cdot executionTime_{top}$). Occurrences of each combination being within 5% of the top performance were accumulated for all the 26 computation intensity cases and they were plotted as a percentage of the total number of cases in figures 5.9, A.95 and A.96 for the In-Order and 5.10, A.97 and A.98 for the OOO cores.

While for the In-Order core there is an area of combinations around the $deg4 - dst4$ one, sustaining appearance percentages over 90%, for the OOO core not such regular behaviour is observed and thus, there is not a single-combination-fits-all candidate. Calculating the derivative of the above metric as the computation intensity increases, could provide an indication on which combinations are performing well in which intervals of computation intensity, as a combination sustaining an increase rate of 1 throughout the whole interval means that it keeps having an execution time within a 5% range of the top performing configuration. Combinations with direct neighbours having in less than 10% of the cases execution times within that range were excluded. In the produced graphs for the different prefetchers (figures 5.11 to 5.13), different configurations sustain the required performance for most of the intensities, namely $deg5 - dst3$ and $deg4 - dst4$ for the system featuring a *stride* prefetcher, $deg4 - dst4$ for the system featuring a *stream on miss* one and $deg1 - dst4$ for the *stream on access* system. For all three of them though, there is no reliable combination sustaining consistently a performance near the best one for the memory intensive cases (less than 80 operations). The same observation holds for the In-Order core as well, with the missing 10% of the appearances being in those cases.

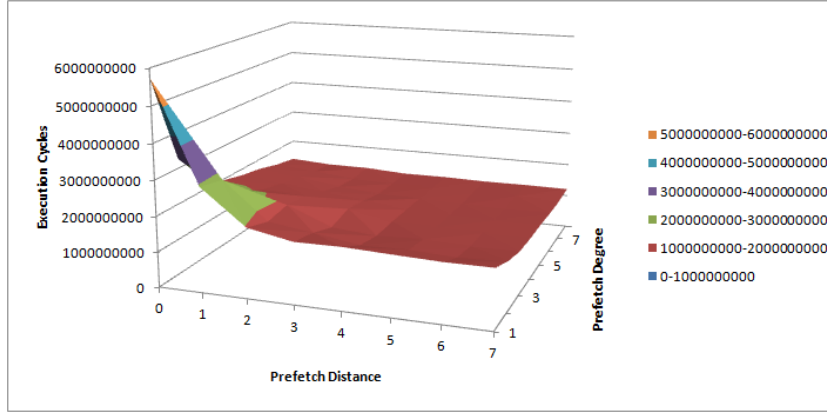


Figure 5.1: Execution time of a system featuring the *stride* prefetcher when the synthetic benchmarks with 4+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase

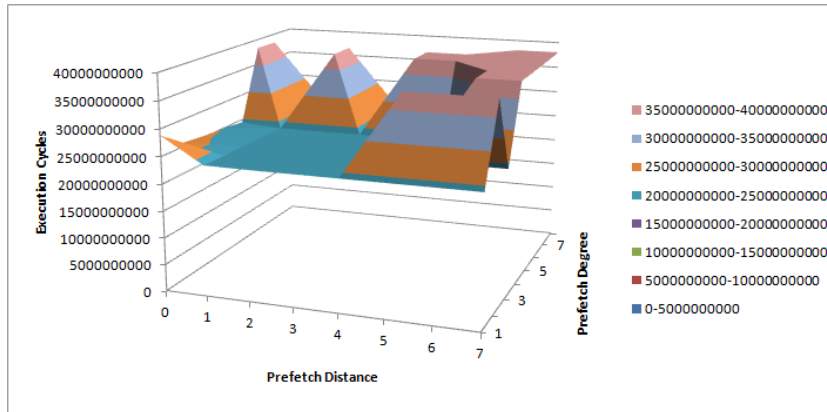


Figure 5.2: Execution time of a system featuring the *stride* prefetcher when the synthetic benchmarks with 520+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase

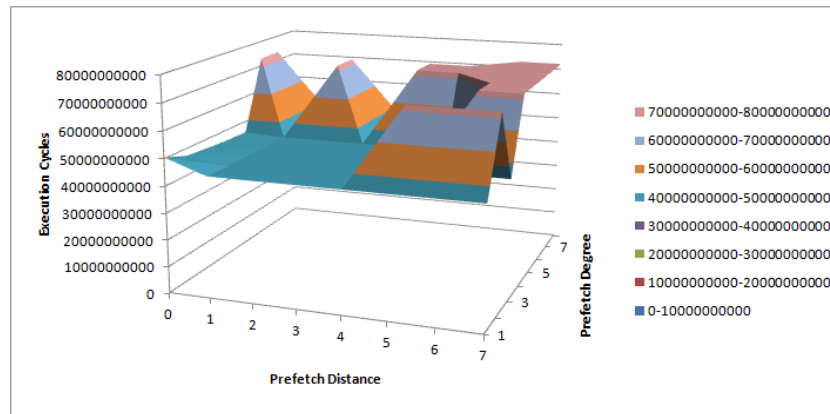


Figure 5.3: Execution time of a system featuring the *stride* prefetcher when the synthetic benchmarks with 1000+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase

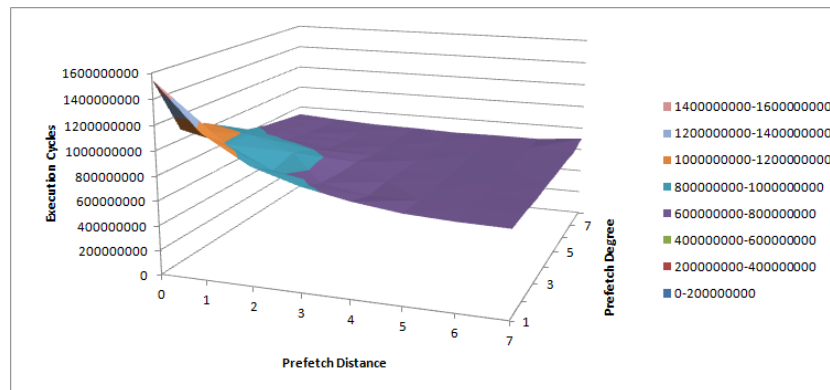


Figure 5.4: Execution time of a system featuring the *stride* prefetcher when the synthetic benchmarks with 4+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase

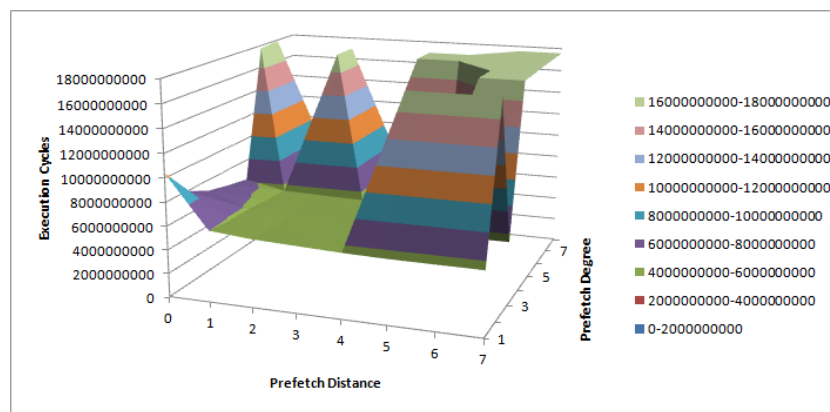


Figure 5.5: Execution time of a system featuring the *stride* prefetcher when the synthetic benchmarks with 520+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase

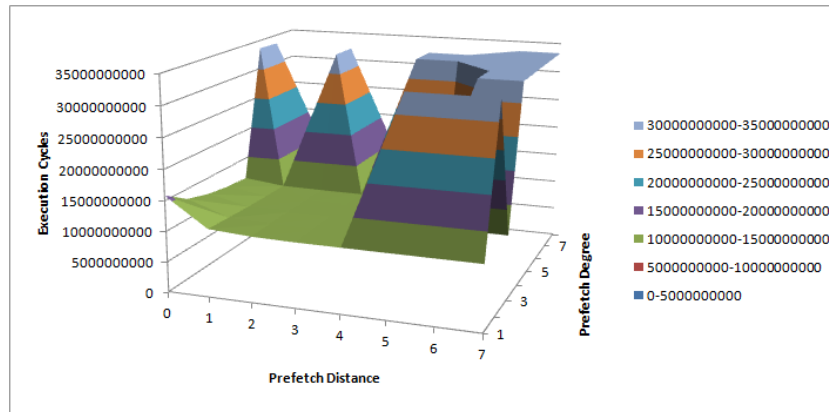


Figure 5.6: Execution time of a system featuring the *stride* prefetcher when the synthetic benchmarks with 1000+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase

Arithmetic Operations	<i>stride</i> + SW	<i>streamOnMiss</i> + SW	<i>streamOnAccess</i> + SW
4	7,7	5,6	7,7
40	5,5	8,5	5,5
80	5,5	5,5	5,5
120	5,5	8,5	5,5
160	5,5	8,5	5,5
200	5,5	8,5	5,5
240	5,5	5,2	5,5
280	5,5	5,3	5,4
320	5,3	5,5	5,3
360	5,4	5,5	5,3
400	5,3	5,3	5,5
440	5,3	8,5	5,4
480	5,4	5,4	5,5
520	5,3	5,5	5,3
560	5,2	5,2	5,4
600	5,5	4,3	5,3
640	5,2	5,4	5,2
680	5,5	5,4	5,4
720	5,2	8,5	5,5
760	5,3	5,3	5,3
800	5,5	5,5	5,5
840	5,2	5,5	5,3
880	5,3	5,3	5,4
920	5,2	6,3	5,4
960	5,2	7,3	5,4
1000	5,2	7,1	5,3

Table 5.1: Top performing combinations of parameters in the form *degree, distance* for the systems featuring the reference prefetchers as the computation intensity per load increases in the In-Order core

Arithmetic Operations	stride + SW	streamOnMiss + SW	streamOnAccess + SW
4	7,7	8,7	8,5
40	3,7	1,7	3,7
80	5,5	3,6	3,6
120	5,5	1,7	1,6
160	5,4	3,6	1,7
200	3,7	5,5	1,7
240	4,5	4,5	1,7
280	5,5	5,5	1,7
320	5,5	3,7	3,7
360	5,5	5,5	4,5
400	5,5	5,5	4,5
440	5,5	5,5	5,5
480	5,5	5,5	5,5
520	5,5	5,5	4,4
560	5,5	5,5	5,5
600	5,5	5,5	4,5
640	5,5	5,4	4,5
680	5,5	5,4	5,4
720	5,5	5,5	5,5
760	5,5	4,4	5,5
800	5,5	5,5	4,5
840	5,5	5,4	5,5
880	5,5	5,5	4,4
920	5,5	5,4	4,5
960	5,5	5,5	4,5
1000	5,5	5,5	5,4

Table 5.2: Top performing combinations of parameters in the form *degree, distance* for the systems featuring the reference prefetchers as the computation intensity per load increases in the OOO core

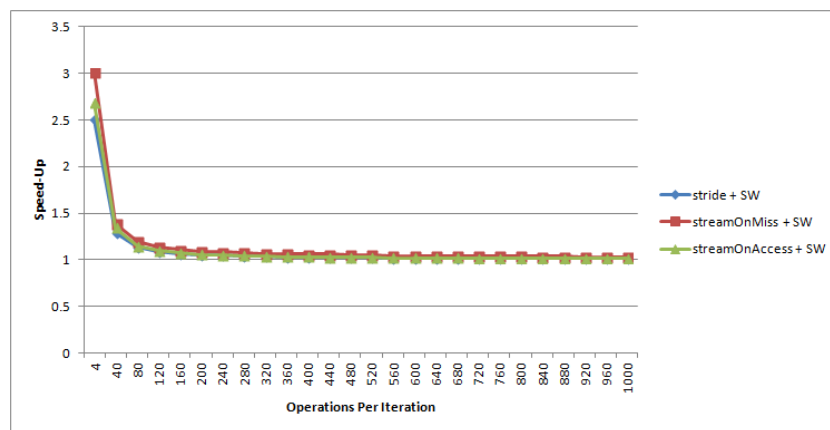


Figure 5.7: Speed-up of the Ideal Prefetcher when the synthetic benchmark with the top performing SW prefetching configurations is executed on the In-Order core over the reference prefetchers as the computation intensity per load increases

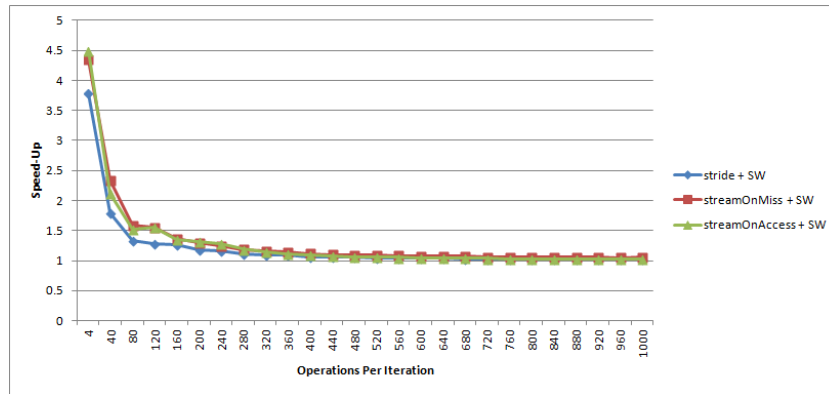


Figure 5.8: Speed-up of the Ideal Prefetcher when the synthetic benchmark with the top performing SW prefetching configurations is executed on the OOO core over the reference prefetchers as the computation intensity per load increases

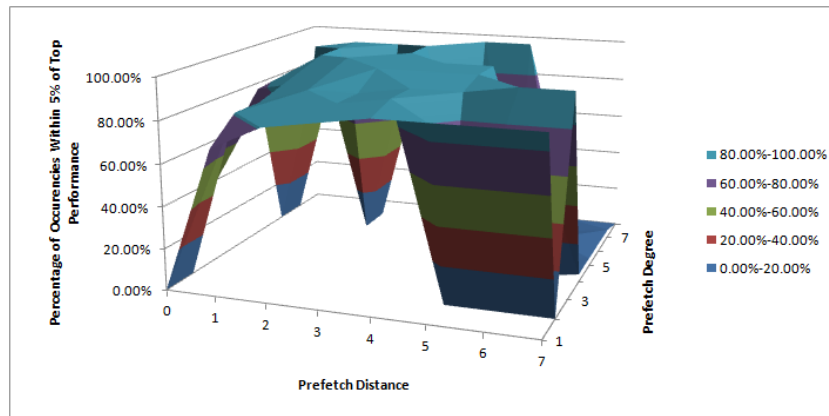


Figure 5.9: Percentage of total number of computation intensity cases, each combination of SW prefetching parameters performing within a 5% range of the top performer when executed in the In-Order core of the system featuring a *stride* prefetcher

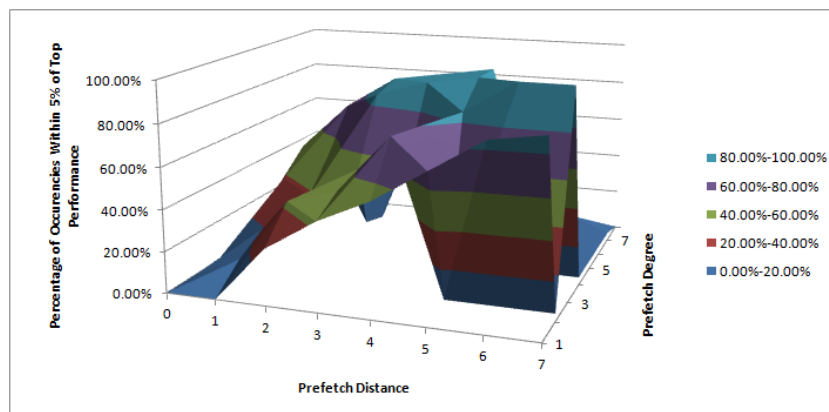


Figure 5.10: Percentage of total number of computation intensity cases, each combination of SW prefetching parameters performing within a 5% range of the top performer when executed in the OOO core of the system featuring a *stride* prefetcher

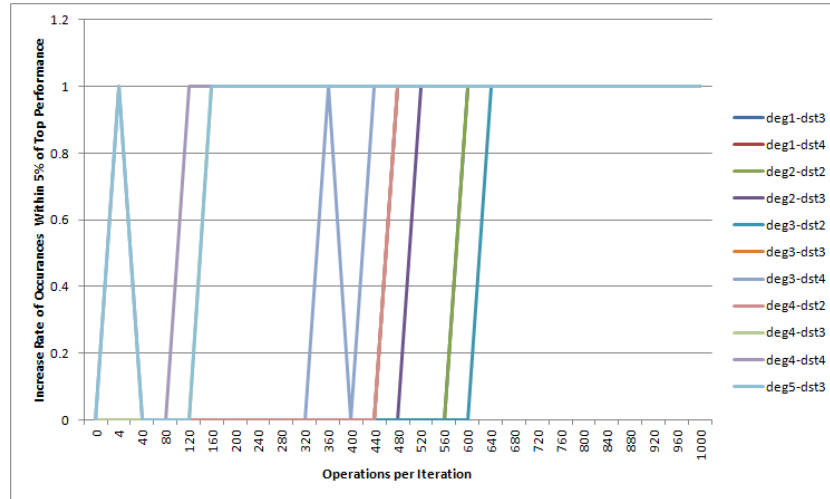


Figure 5.11: Increase rate of the appearances of the SW prefetching parameters' combinations performing within a 5% range of the top performer when executed in the OOO core of the system featuring a *stride* prefetcher

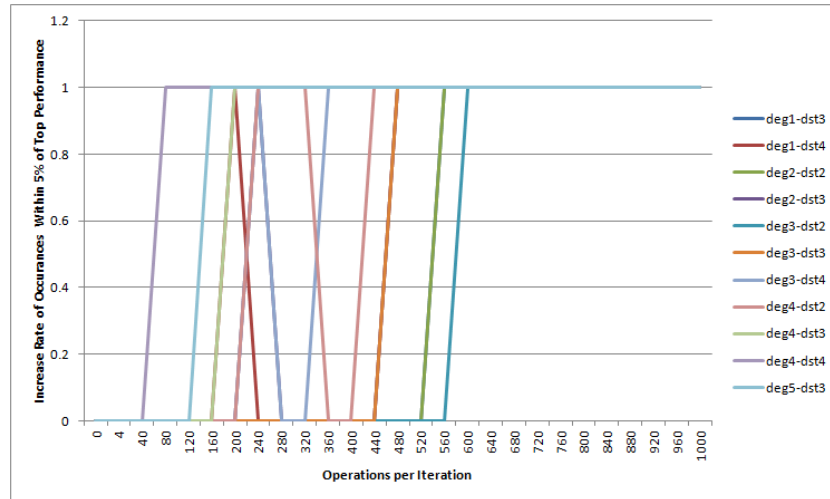


Figure 5.12: Increase rate of the appearances of the SW prefetching parameters' combinations performing within a 5% range of the top performer when executed in the OOO core of the system featuring a *stream on miss* prefetcher

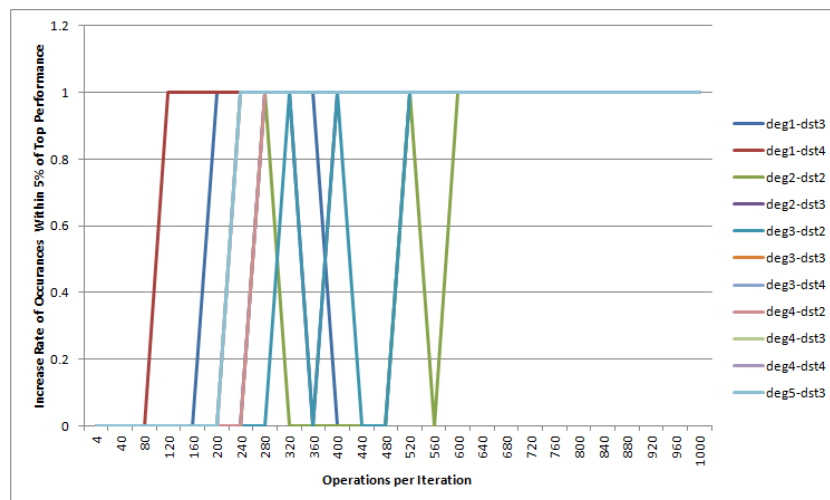


Figure 5.13: Increase rate of the appearances of the SW prefetching parameters' combinations performing within a 5% range of the top performer when executed in the OOO core of the system featuring a *stream on access* prefetcher

5.4. Discussion

The results (presented in section 5.3) of the investigation of determining a systematic way of issuing software prefetches for the indirect accesses of *Apache Arrow* databases analysed in section 5.2, were promising but inconclusive. The identification of the optimal combination of parameters for each case of computation intensity was not straightforward as substantial fluctuation in the execution times of the different combinations were exhibited in the more balanced and the more computation intensive cases. Different lengths of prefetch burst with different distances from the current indices affect the optimisation capability of the memory controller differently. Attributing this behaviour to the handling of the software prefetch bursts by the controller stems from the observation that this behaviour was common for all systems featuring the different reference prefetchers using either core with only the height of the irregularities' peaks changing, while the parameter combinations affected remain the same.

Despite the shape of the 3-dimensional curves remaining the same across the different prefetchers, the best performing parameters differed both between the different prefetchers for the same computation intensities and between the different computation intensities for the same prefetchers. Those combinations achieved a performance very close to the one of the ideal prefetcher as the computation intensity increased. The best performing hardware prefetcher was stride prefetcher as upon failing to recognise a pattern in *value buffer* accessing, it does not attempt to issue any prefetches for it, allowing the useful accesses to exploit the bandwidth the other configurations waste with inaccurate prefetching. For the high memory intensity cases, considerable room for improvement exists even if the best-performing combination for the exact characteristics of the system and of the workloads executed in parallel could be identified.

Different memory congestion profiles could result in memory controller experiencing circumstances similar to those leading to the irregularities. Those top-performance combinations neighbour frequently with some of the ones mishandled by the memory controller and being in this neighbourhood increases the possibility of such events. As one of the requirements of software prefetching is resilience to the different run-time parameters affecting the memory behaviour of the system, it is essential to identify combinations able to satisfy this requirement with performance close to the best possible. Since the simulations under different workloads in multicore configurations were not conducted, the safest path is to select combination satisfying the performance criteria without having immediate neighbours producing irregularities.

For the In-Order core, a neighbourhood of combinations around the combination of *deg4 – dst4* are able to sustain execution times within 5% of the top performance. Nevertheless, for the most memory intense cases those combinations, despite performing adequately, they cannot approach the top performance, which is produced by combinations with high degree and distance parameters. For those cases, these combinations should be considered if software prefetching is to be used. For the rest of the cases, the *deg4 – dst4* combination should be preferred regardless of the used configuration.

Similar to the In-Order, the OOO core achieves top performance for the memory intensive cases when high degree, high distance combinations are used. Though in this case, the top-performing combination fail to approach the performance of ideal prefetcher by a larger margin and this effect is sustained for a longer interval until it saturates. In addition to that, the performance of *stride* prefetcher, issuing prefetches only for the strides which is confident to have successfully identified and thus, exploiting the available bandwidth more efficiently, is noticeably better for the 25% most memory intensive cases. As the computation intensity increases, the same neighbourhood of combinations performing constantly close to the top performance is not present. In this case a more sophisticated analysis identified combinations with an adequate performance, resilience trade-off, which are *deg5 – dst3* and *deg4 – dst4*, *deg4 – dst4* and *deg1 – dst4* for the systems featuring a *stride*, *stream on miss* and *stream on access* prefetchers respectively.

Concluding the software prefetching investigations, it should be pointed out that this scheme is able to almost match the performance of the ideal prefetcher for the medium and high computation intensities. Nevertheless, for the memory intensive cases there is a significant gap between them, while the resilience of the best-performing parameter combinations for different memory congestion profiles is questionable. In general, based on the behaviour exhibited for the different parameter combinations, the performance improvement achieved by software prefetching seems to be vulnerable to the memory behaviour of the whole system during run-time.

Arrow Hardware Prefetcher

Software prefetching can adequately approach the performance of ideal prefetching for most cases of computation intensity as documented in chapter 5, but its benefits seem to be rather sensitive to the identification of the optimal combination of parameters, which is a challenging task to be performed during compile-time. Hardware prefetching on the other hand is more resilient to changes in those parameters and allows for more degrees of adaptability during execution time at the expense of extra hardware and design complexity. The design of such a prefetching module was attempted in the framework of this study with its requirements being defined in sections 6.1 and the details of its logic and interaction with the software in section 6.2. A performance comparison of a system featuring this prefetcher when executing the studied benchmarks with the performance of the rest of the aforementioned prefetching schemes is presented and discussed in sections 6.3 and 6.4 respectively.

6.1. Definition

The purpose of hardware prefetching, as analysed in section 2.2.1, is to identify specific kinds of targeted access patterns and based on them predict the future accesses and issue prefetch requests for them. Such a scheme shifts the burden and complexity of prefetching from the software level to the hardware. This shift, however, deprive the prefetching initiator of some information readily available by the algorithm, in exchange for access and possibly adaptability to run-time information. A trade-off between the two is usually hardware prefetching modules, some aspects of which are parametrisable using software or which can use information passed to them by the software. Nevertheless, the more programmable a module is designed to be, the more complex is its implementation for the designers and its configuration for the software developer, leading in many cases to results, which are mediocre or even to the opposite direction. Thus, less of a requirement and more of a direction, is to limit the interaction between the software and the prefetching module to the minimum sufficient amount for enabling the prefetcher's mission.

As design and hardware resources are invested for the design, the integration and the implementation of such a module, it is required for it to induce a considerable performance benefit compared to the reference prefetchers for the majority of algorithms using the identifiable patterns. More specifically, it should be able to produce accurate and timely prefetches improving the memory behaviour of the system and approaching, to the extent the memory resources and the memory intensity of the executed algorithms allow, the performance of the ideal prefetcher for those patterns. For an *Apache Arrow* prefetcher, those patterns are both the constant-offset, strided patterns and the indirect indexing accesses *Arrow* uses to access the variable length or nested elements.

In section 2.2.1 previous work on indirect indexing prefetching was explained, with the study in [54] trying to infer the indirect access patterns from accesses with instruction addresses close to each other. This approach though, was not aware and consequently could not use any information about the structure and hierarchy of the data, which, for the case of *Apache Arrow*, is coded in the schema, the headers and the *offsets buffers* of each *column*. Using an approach which allows the prefetcher to access this information could simplify drastically the identification process of indirect indexing patterns for accesses with instruction addresses close to each other, as the prefetcher will know which *column*

was accessed and whether each access targeted any of the *offsets buffer* and thus is the load of an index due to the immutability of *Arrow* objects, or the *value buffer*. As those metadata are defined using virtual addresses, an additional requirement for the prefetcher emerges, which is for it to operate on virtual addresses in order to be able to use this information.

Furthermore, since the field of interest is *big data* and the powerful systems processing them usually feature powerful OOO cores, the prefetcher should be able to handle occasional out of order accesses, a functionality that stride prefetchers already implement. This mechanism of handling such accesses should not make the prefetcher extremely insensitive to possible changes in any aspect of the identified access pattern for either part of the pair of index-data loads, which will produce inaccurate prefetches wasting memory resources. Consequently a trade-off between tolerance for occasional deviations, responsiveness to pattern changes and accuracy should be considered.

In pursuit of satisfaction of all the aforementioned requirements, the complexity of the resulting prefetch algorithm increases. Nevertheless, all of them are necessary to a certain degree for the implementation of such a module to be justifiable and thus, should be taken into account in any attempt to come up with a prefetching algorithm. In the next section such an attempt will be proposed, though the idea of a prefetching scheme with those characteristics is not restricted to this particular proposal, which could be optimised or revised when possible hardware implementations for it are considered.

6.2. Implementation

The implementation was initiated by the definition of the patterns to be targeted and the formulation of the identification process (section 6.2.1) in order to be understood which parts of the *Apache Arrow metadata* could be of use for prefetcher. After the identification process was defined, the communication mechanism of the required metadata from the software to the prefetcher module and the prefetching algorithm are analysed in sections 6.2.2 and 6.2.3 respectively, taking into account most of the major challenges stemming from the location of the prefetcher. Subsequently, the configuration of the prefetcher through both its parameters is discussed in section 6.2.4.

6.2.1. Targeted Access Patterns and Identification

Approaching the creation of the algorithm starts with rephrasing the indirect indexing access patterns and more specifically the simplest case of indirection, which is accessing *columns* through a single *offsets buffer*, with the terminology of *Apache Arrow*. Equations 5.1, 5.2, 5.3 from section 5.2, where the same *value buffer* accesses were targeted using software prefetches, for two consecutive accesses are repeated for the convenience of the reader:

$$\begin{aligned} V(index_{offset_1}, index_{value_1}) &= V(s_{offset_{const}} * i + c_{offset_{const}}, c_{value_{const}}) = \\ &\quad Value[Offset[s_{offset_{const}} * i + c_{offset_{const}}] + c_{value_{const}}] \\ V(index_{offset_2}, index_{value_2}) &= V(s_{offset_{const}} * (i + 1) + c_{offset_{const}}, c_{value_{const}}) = \\ &\quad Value[Offset[s_{offset_{const}} * (i + 1) + c_{offset_{const}}] + c_{value_{const}}] \end{aligned} \quad (6.1)$$

$$\begin{aligned} Addr(Offset[s_{offset_{const}} * i + c_{offset_{const}}]) &= \\ &\quad baseAddr_{offset} + dataSize_{offset} \cdot (s_{offset_{const}} \cdot i + c_{offset_{const}}) \\ Addr(Offset[s_{offset_{const}} * (i + 1) + c_{offset_{const}}]) &= \\ &\quad baseAddr_{offset} + dataSize_{offset} \cdot (s_{offset_{const}} \cdot (i + 1) + c_{offset_{const}}) \end{aligned} \quad (6.2)$$

$$\begin{aligned} Addr(V(index_{offset_1}, index_{value_1})) &= \\ &\quad baseAddr_{value} + dataSize_{value} \cdot (Offset[s_{offset_{const}} * i + c_{offset_{const}}] + c_{value_{const}}) \\ Addr(V(index_{offset_2}, index_{value_2})) &= \\ &\quad baseAddr_{value} + dataSize_{value} \cdot (Offset[s_{offset_{const}} * (i + 1) + c_{offset_{const}}] + c_{value_{const}}) \end{aligned} \quad (6.3)$$

As with the case of stride prefetching, iterative regular accesses following this pattern are targeted as providing support for any additional level of irregularity could increase complexity significantly. These

accesses (equation 6.1) can be reduced to two loads with first the $Offset[index]$ being loaded using the address ($Addr$) of equation 6.2 and subsequently, by using its contents, the address of the data is calculated, as shown in equation 6.3, and loaded. For both those loads an initial constant offset ($c_{Offset_{const}}$ for *Offsets* and $c_{Value_{const}}$ for *Values*) can be accommodated, while for the case of the indices a different step than 1 as well ($s_{Offset_{const}}$). Consequently any access of the form described in equation 6.1 with $s_{Offset_{const}} \in \mathbb{Z}$ and $c_{Offset_{const}}, c_{Value_{const}} \in \mathbb{N}$ within a loop with i increasing in each iteration, can be successfully prefetched if the number of iterations is greater than 2. In reality, a larger number of iterations is needed for any tangible performance gains, as for any prefetching scheme, but the field of *big data* provides large loops in abundance. Of course a prefetcher should be able to identify and accurately prefetch multiple such accesses per loop as well as multiple loops, the number of which is limited by the memory the internal tables of a prefetcher can occupy.

Passing the information about the address and the length of the *offsets buffers* and the *value buffer* for each *column*, would render the prefetcher capable to distinguish which of them is targeted by each access. When an accesses targeting a specific *offsets buffer* occur by a specific instruction address (PC) for the first time, then except for which *column* was targeted and through which *offsets buffer*, also the accessed index can be calculated using the first equation in 6.2:

$$index_{Offset_{accd1}} = \frac{Addr_{accd1} - baseAddr_{Offset}}{dataSize_{Offset}} \quad (6.4)$$

When a second access occurs from the same instruction address (PC), then by subtracting the accessed addresses of equation 6.2 the *offset index stride* ($s_{Offset_{const}}$) can be calculated as well. Thus, for each subsequent access, the *offset index stride* can be recalculated and as occurrences of it matching the previous calculated one accumulate, the prefetcher can be confident that the correct value has been identified. After the prefetcher becomes confident enough, it can start prefetching the next *DEGREE* addresses containing the next indices ($Addr(Offset[index_{Offset_{accd}} + s_{Offset_{const}} * n])$, $\forall n \in [1, DEGREE]$). This is a typical stride prefetching behaviour as was analysed in section 2.2.1 since changes in the value of *index stride* ($s_{Offset_{const}}$), if existent at all, are rare and followed by a large number of iterations with it remaining constant.

When an access targeting the *value buffer* occur by a specific instruction address (PC) for the first time, it is unknown which *offsets buffer* was used, if any. Instructing the compiler to place the load of the *offset* within several instructions before the load of the *value* and not to interpose any other *offset* loads between them, then the *value* load can be linked to the *offset* load entry with the closest smaller instruction address (PC). This allows the prefetcher to use the information, deduced with the above explained reasoning, of the linked *offset* load to identify which *offsets buffer* is used. Though, except for this piece of information, access to the actual contents of the *offsets buffer* both to determine which *offset index* was used in order to predict the next indices based on the *offset stride* of the linked *offset* load, and to use them to calculate the *value buffer* addresses to be prefetched based on equation 6.3, is needed. With the assumption that the *offset index* loaded by the initial access of the linked load will be used first, it is checked whether the accessed address of the *value buffer* is between the address pointed by this index ($index_{Offset_{init}}$) and the next ($index_{Offset_{init}} + 1$) as denoted by the next equation:

$$\begin{aligned} Addr(V(index_{Offset_{init}}, 0)) \leq Addr_{accd1} < Addr(V(index_{Offset_{init}} + 1, 0)) \quad (6.5) \\ \stackrel{6.3}{\Leftrightarrow} Offset[index_{Offset_{init}}] \leq \frac{Addr_{accd1} - baseAddr_{Value}}{dataSize_{Value}} < Offset[index_{Offset_{init}} + 1] \end{aligned}$$

If this relation is satisfied, then the accessed index was identified ($index_{Offset_{accd1}} = index_{Offset_{init}}$), else the next w indices ($index_{Offset_{init}} + n \cdot s_{Offset_{const}}$, $\forall n \in [1, w]$) are tested in the same way until the index is found. In case this does not happen within this window of w indices, it is possibly due to the OOO execution or due to parts of the contents of the *offsets buffer*, which were needed to test the indices of the window, not being in the cache. Then the index is speculated to be equal to the first loaded index by the linked load instruction ($index_{Offset_{init}}$) but with low confidence so it can be updated immediately if its actual value is found in later iterations or dropped quickly if not. If a value for the accessed index is found then equation 6.3 can be used to calculate the *value constant* ($c_{Value_{const}}$):

$$index_{Value_{accd1}} = c_{Value_{const}} = \frac{Addr_{accd1} - baseAddr_{Value}}{dataSize_{Value}} - \mathbf{Offset}[index_{Offset_{accd1}}] \quad (6.6)$$

When a second access occurs from the same instruction address (PC), a similar process is followed for the identification of the accessed *offset index*, with the previously identified index augmented by the *offset index stride* ($index_{Offset_{accd1}} + s_{Offset_{const}}$):

$$\mathbf{Addr}(\mathbf{V}(index_{Offset_{accd1}} + s_{Offset_{const}}, 0)) \leq Addr_{accd2} < \mathbf{Addr}(\mathbf{V}(index_{Offset_{accd1}} + s_{Offset_{const}} + 1, 0)) \quad (6.7)$$

$$\begin{aligned} &\stackrel{6.3}{\Leftrightarrow} \mathbf{Offset}[index_{Offset_{accd1}} + s_{Offset_{const}}] \\ &\leq \frac{Addr_{accd2} - baseAddr_{Value}}{dataSize_{Value}} < \\ &\mathbf{Offset}[index_{Offset_{accd1}} + s_{Offset_{const}} + 1] \end{aligned}$$

The constant value ($c_{Value_{const}}$) index is then calculated again for the new values of the accessed address ($Addr_{accd2}$) and the accessed *offset index* ($index_{Offset_{accd2}}$) as in equation 6.4. If the *offset index* was not found or either it or the constant value do not match the expected, then the confidence drops. As occurrences of them matching the expected values accumulate, the prefetcher becomes confident that the pattern has been identified correctly. If the prefetcher is confident enough, it can start prefetching the next *DEGREE* addresses containing the next values predicted to be accessed based on this pattern ($Addr(Value[Offset[index_{Offset_{accd}} + s_{Offset_{const}} * n] + c_{Value_{const}}]), \forall n \in [1, DEGREE]$). It should be noted that except for the elements of the *offsets buffer* needed to identify the *offset index*, more elements are needed to calculate the addresses to be prefetched. Consequently, likewise to the software prefetching case, having the needed elements of the *offsets buffer* early enough in the cache is crucial for the prefetching of the elements of the *value buffer*. The immutability of the *Apache Arrow* objects and by extension of its *offsets buffers* though, could possibly be exploited with the use of specialised memory types to guarantee the satisfaction of this requirement.

An observation that could be made from equations 6.1, 6.2 and 6.3 is that while an *offset index stride* ($s_{Offset_{const}}$) could be accommodated, this is not the case for the *value buffer* for which only a constant displacement ($c_{Value_{const}}$) from the corresponding offset is allowed. Algorithms in which the *offset index stride* is zero and there is a non-zero *value index stride* changing the index in every iteration ($c_{Value_{const}} + s_{Value_{const}} * i$) exhibit a typical constant stride behaviour, to which the prefetcher defaults for *Value Load entries* with no or inaccurate *Offset Load entry* link, so they can be accurately prefetched after an initial training phase. Though cases, in which both *index strides* are non-zero, were not found during this study, so in order to reduce the complexity of the algorithm, support for them was not included. If this feature was to be added, the verification of the constant ($c_{Value_{const}}$) would be replaced with the recalculation of the stride which would be as well added to the addresses to be prefetched ($Addr(Value[Offset[index_{Offset_{accd}} + s_{Offset_{const}} * n] + s_{Value_{const}} * n + c_{Value_{const}}]), \forall n \in [1, DEGREE]$).

6.2.2. Metadata Communication Mechanism

Except from the limitations the identification process introduces, which were discussed in the previous section, further constraints are imposed by the hardware. Prefetcher modules are in the vast majority of cases within the caches they issue prefetch requests for. This allows them to be able to observe the addresses accessed and to know the addresses of the instructions which issued those requests, the purpose of the accesses, whether they are reads or writes and the execution cycles during which they were initiated. Furthermore, it is usual for a prefetching module to feature internal tables which allow it to store information from previous access events and use them for its identification and prediction algorithms. The rest of the system information is difficult to become available to the prefetcher as it would require a complex routing, which reduce significantly the benefit to cost ratio.

During the analysis of the identification process, several pieces of the *Apache Arrow* metadata were mentioned. Specifically, the location in virtual memory and the length of both the *offsets buffers* and the *value buffer* are needed as well as the data size of the values. This information could be passed to

the prefetcher by the software using specialised instructions or memory-mapped registers, though this is not possible for the actual elements of the *offsets buffers*, which are needed by the prefetcher both to identify the pattern and to calculate the addresses to be prefetched, due to their size and the limited internal memory capacity of the prefetcher. One possible solution for this constraint is to use specialised internal buffers to which the prefetched elements of the *offsets buffer* would be stored in addition to the L1D cache. Nevertheless, this solution would further increase the internal memory demands of the prefetcher, which already has to accommodate the deduced information of the previous accesses for multiple instruction addresses. The solution which was eventually preferred is to allow the prefetcher to access the contents of the cache it resides in through a dedicated read-only port. In this way, the prefetcher will be able to access any part of the memory as long as it is located in its cache and since the prefetcher is able to issue requests for any address, this expands to the whole memory. Using such a scheme, reduces the communication between the software and the prefetcher to structured writes into specific reserved parts of the main memory, which the prefetcher can fetch to the cache and access.

After ensuring that all the required information would be able to reach the prefetcher, the communication of this information by the software and their organisation internally to prefetcher into tables for it to be accessed systematically by the prefetching algorithm, is subsequently discussed. This communication is performed through a reserved memory region containing a table (*Columns Table* in figure 6.1) in which the information of at most N *columns* can simultaneously be passed to and used by the prefetcher. For the registered *columns*, it is required for their *buffers* to be contiguous in virtual memory. Consequently, for *chunked arrays*, each chunk should be registered separately for this requirement to be satisfied. In case the table is full, the timestamp of each entry is used to replace the oldest one. Using a more efficient replacement scheme like *Least Recently Used* would require extra instructions and memory writes for each access to a *column*. Except for the details of the virtual space occupied by the *value buffer* and the size of the value datatype, the location in virtual memory of a number of *offsets buffers* per *column*, at most M , can be registered to the table as well. The responsibility of writing the table lies completely at the software level upon the instantiation of each *column* and could be performed in a systematic way by a specialised library or even the compiler. An example of forming the entries of the table from a set of *columns* (6.3) is presented in figure 6.2.

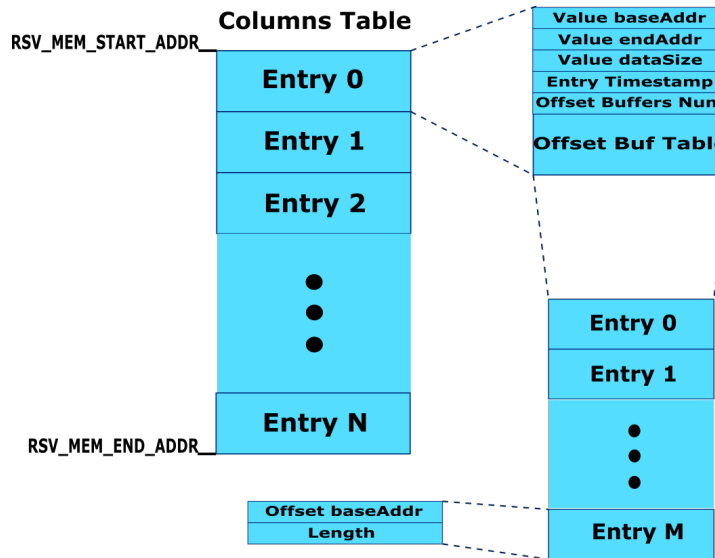
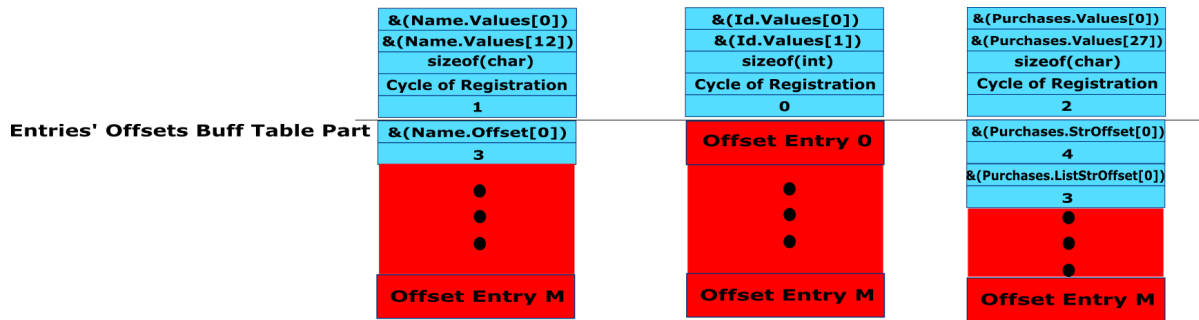
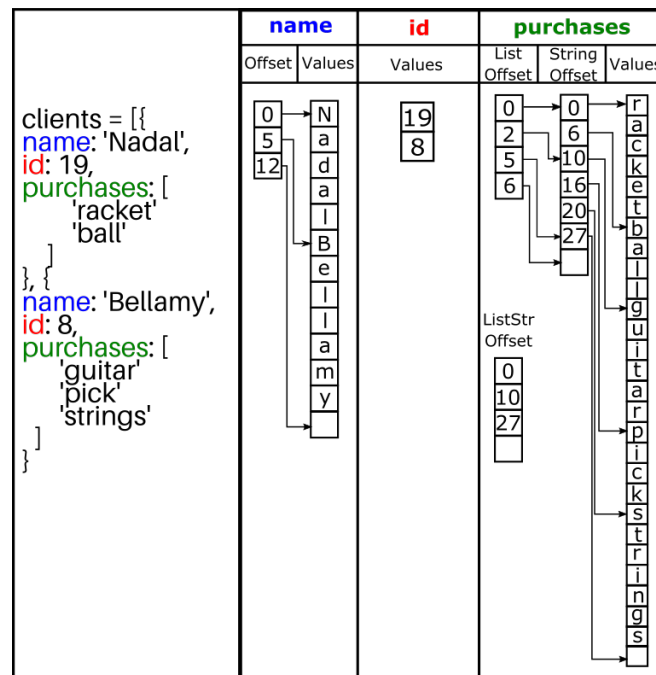


Figure 6.1: Structure of the *Columns Table* in the memory region reserved for the software to communicate the required metadata of the *Arrow columns*

As nested indirect indexing would complicate significantly the structure of the prefetcher and the complexity of the algorithm, supplementary *offset buffers* flattening the nesting level to one by recursively replacing the offsets of the logical datatypes with the offsets of the lower level until their element correspond to offsets of the *value buffer*. An example of such a supplementary buffer can be observed at the case of the *ListStr Offset buffer* of figure 6.3, for the construction of which the elements of the *List Offset buffer* were replaced by the the elements of the *String Offset buffer* they point to. In cases

Figure 6.2: Registration of the *columns* of the example database of figure 6.3 to the *Columns Table*Figure 6.3: Example of the structure of the buffers of a database in *Apache Arrow* format

for which the cost of the extra memory for the supplementary buffers or accessing the data through them is considered to outweigh the performance benefits, it can be selected to register only the *offsets buffer* closest to the data, the *String Offset buffer* in this case.

Each entry of the *Columns Table* is mapped to an internal table on a one-by-one correspondence with its structure being depicted in figure 6.4. As the software registers one or more *columns*, the prefetcher observes the burst of access to the reserved memory region and sets a global dirty flag for the internal tables of the prefetcher. As soon as the first access outside the reserved memory region occurs, the prefetcher initiates the process of updating its internal tables using the new metadata registered in the *Columns Table*. In order to do so, all the cachelines containing entries of the table are needed and thus, should be on the L1D cache in order for the prefetcher to have access on them. In case any of the needed cache blocks is missing, a prefetch request is issued for it. While waiting for the missing blocks to arrive the prefetcher is inactive in order not to issue any inaccurate prefetches. When all the required cache blocks are on the cache, for each internal table is checked whether it needs to be updated with the data of the corresponding entry of the *Columns table*. If an update is necessary, it proceeds and all the entries of the *offset* and *value* subtables (see figure 6.4) of the updated internal table are invalidated as they correspond to a deprecated *column* structure. This process as a whole is described in lines 4 to 21 of listing 6.1.

With this mechanism, the prefetcher always acts on an up-to-date version of the registered *columns*, while it avoids constantly accessing the reserved memory region of the *Columns Table*. This scheme

allows the cache blocks of the *Column Table* to be replaced when not needed, which is the greatest part of the execution time as updates to it are expected to be sporadic and be performed in bursts with multiple *columns* being registered at a time. Nevertheless, this choice leads to an increase of the memory needed for each of the internal tables and consequently to the total area cost of the prefetcher.

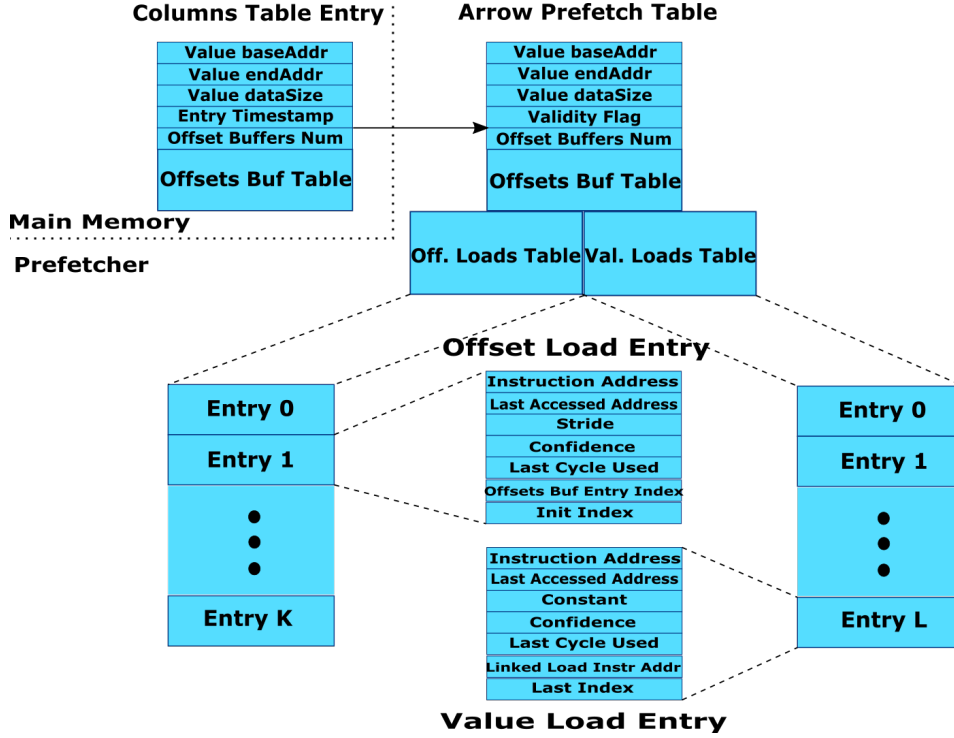


Figure 6.4: The structure of Internal Prefetcher Tables and the updating process using the information of the *Columns Table*

6.2.3. Algorithm

Having all the necessary metadata information stored internally in the prefetcher allows its use in the way described in section 6.2.1 to identify the access patterns, predict the next addresses to be accessed and issue prefetches for them. This logic along with the support for the communication mechanism was incorporated to an algorithm similar to what was described in section 2.2.1 for *stride prefetcher*. The algorithm in the form of pseudocode is presented in listing 6.1. Besides the sporadic communication events between the software and the prefetcher through the reserved memory region (lines 4 through 21), the algorithm consists of three main parts: determining whether the initiating access targeted any of the registered *columns* (case statements in lines 24, 38 and 65), creating new internal table entries (lines 27 and 41) or updating existing ones (29 and 43) and determining whether and which cache blocks to prefetch (lines 31 to 35, 45 to 62, 67-69 for the cases of loads targeting any *offsets buffer*, any *value buffer* and any other address outside the reserved memory region respectively).

Determining whether an access targets any of the *columns* requires for each of the valid internal tables, the accessed address (*accdAddr*) to be compared with its *value buffer* limits and with each of its *offsets buffer* limits. In case the address is within the value address space of the internal table ($accdAddr \in [baseAddr_{value}, endAddr_{value})$), then the corresponding *value buffer* of the *Arrow column* was accessed. In case the address is within any of the offset address spaces of the internal table ($accdAddr \in [baseAddr_{offset_i}, endAddr_{offset_i})$, checked $\forall i \in [0, Offset_Buffers_Num)$), then the corresponding *offsets buffer* of the *Arrow column* was accessed. If none of the above relations holds, then the access did not target any relevant memory region. Those comparisons are all independent and can be performed in parallel or grouped by table if deemed necessary by the resources constraints, as there is no possibility of an access corresponding to two tables simultaneously.

```

1  input: accdAddr, instrAddr, curCycle
2  output: prfAddrs
3  begin
4      if accdAddr ∈ [RSV_MEM_START_ADDR, RSV_MEM_END_ADDR) then
5          dirtyFlag ← True
6      else
7          if dirtyFlag == True then
8              tableLoaded ← True
9              for each blockAddr in ColumnsTableRegion
10                 block ← readBlock(blockAddr)
11                 if block == NULL then
12                     tableLoaded ← False
13                     prfAddrs.push(blockAddr)
14                 else
15                     colTableBuffer.push(block)
16                 end
17             end
18             if tableLoaded == True then
19                 dirtyFlag ← False
20                 InternalTables.update(colTableBuffer)
21             end
22         else
23             for each table in InternalTables
24                 case table.isOffsetLoad(accdAddr)
25                     entry ← table.offLoadtab.find(instrAddr)
26                     if entry == NULL then
27                         table.offLoadtab.createOffsetEntry(accdAddr, instrAddr, curCycle)
28                     else
29                         entry.updateOffsetEntry(accdAddr, instrAddr, curCycle)
30                     end
31                     if entry.confidence ≥ threshConfidence then
32                         stridePrefetch ← max(entry.stride, cacheBlockSize)
33                         for each n ∈ [1, DEGREE]
34                             prfAddrs.push(cacheBlockAddr(accdAddr) + stridePrefetch * n)
35                         end
36                     end
37                 case table.isValueLoad(accdAddr)
38                     entry ← table.valLoadtab.find(instrAddr)
39                     if entry == NULL then
40                         table.valLoadtab.createValueEntry(accdAddr, instrAddr, curCycle)
41                     else
42                         entry.updateValueEntry(accdAddr, instrAddr, curCycle)
43                     end
44                     if entry.confidence ≥ threshConfidence then
45                         linkedOffEntry ← table.offLoadtab.find(LinkedLoadInstrAddr)
46                         usedOffsetBuffer ←
47                             table.offsetsBufTable[linkedOffEntry.offBuffEntryIndex]
48                         usedOffsetStride ← linkedOffEntry.stride
49                         lastPrfAddr ← cacheBlockAddr(accdAddr)
50                         for each n ∈ [1, DEGREE]
51                             newPrfAddr ←
52                                 cacheBlockAddr(table.valueBaseAddr + table.valueDataSize *
53                                     (table.usedOffsetBuffer[lastindex + n * usedOffsetStride]))
54                             if newPrfAddr - lastPrfAddr != 0 AND
55                                 (signof(newPrfAddr - lastPrfAddr) ==
56                                     signof(usedOffsetStride)) then
57                                 prfAddrs.push(newPrfAddr)
58                             else
59                                 prfAddrs.push(lastPrfAddr +
60                                     signof(usedOffsetStride) * cacheBlockSize)
61                             end
62                         lastPrfAddr ← newPrfAddr
63                     end
64                 end
65             end
66         default
67             stridePrefetch ← cacheBlockSize
68             for each n ∈ [1, DEGREE]
69                 prfAddrs.push(cacheBlockAddr(accdAddr) + stridePrefetch * n)
70             end
71         end
72     end
73 end
74 end
75 end
76 end

```

Algorithm 6.1: Algorithm of Arrow Prefetcher in the form of pseudocode

When an access is identified to target any of the registered, in the *Offsets Buffer Table*, *offsets buffer*, its instruction load address is checked against the instruction load addresses of the entries of *Offset Loads* subtable. If a load from this instruction is already registered in the table, the corresponding entry's *Last Accessed Address* and *Last Cycle Used* fields are updated with the new access address and the value of the cycle counter respectively, while the stride is recalculated as explained in section 6.2.1. In case of a match, the entry's confidence counter is increased, if not maxed out, else it is decreased, if not already at the minimum value. If after the update of the confidence counter, its value is below a threshold, the value of the stride is updated with the newly calculated stride, otherwise it remains unchanged. This enables the offset stride to be resilient to temporary jumps of the accessed address or accesses appearing out of order, while it enables occasional changes of the access pattern.

If there is no entry present for this *Instruction Address*, a new entry is created with the *Offset Buffer Entry Index* value being set with the index the accessed *offsets buffer* was registered to in the *Offsets Buffer Table* of the *Columns Table*, which was subsequently copied to the one of the internal table corresponding to the *column*. The initial index is identified using the process described in 6.2.1, the stride is initialised to 0 and the confidence is set to the threshold value in order for the calculated stride during the second access to update immediately the field of the entry. The rest of the fields are updated in the same way, as explained in the previous paragraph.

For accesses identified to be targetting the *value buffer* of a registered *column*, the *Value Loads* subtable is looked-up for an entry with the same *Instruction Address*. In case such an entry is not found, a new is created with the entry of the *Offset Loads* subtable with the closest smaller *Instruction Address* being linked to it, by setting the *Linked Load Instruction Address* with this address. This allows the entry to access information related to the used *offsets buffer*, such as the *Offset Buffer Entry Index* and the *Init Index* discussed above. Using this information, the accessed index and the value of the constant can be calculated through the process described in section 6.2.1 and initialise the value of the *Last Index* and *Constant* fields respectively. In case the calculation is not possible, the *Confidence* counter is set to the threshold value, the *Last Index* field with the value of the *Init Index* of the linked *Offset Loads* entry and the *Constant* to 0. Otherwise, the *Confidence* counter is initialised with a predetermined during design time value. The selection process for the initial value of the confidence counter, as well as its maximum, minimum and threshold values are discussed in section 6.2.4. In the event of the linking process fails, the entry defaults to a normal stride prefetcher entry using the same fields to produce the behaviour described in section 2.2.1.

In case an entry for this instruction address is found at the subtable, then its *Last Index* and *Constant* fields are updated using the process described in section 6.2.1. When the accessed index does not match the expected one or the constant is different, the *Confidence* counter is decreased unless it is already at its minimum value, otherwise since both match, the counter is increased by 2, if not maxed out. In the event of the confidence dropping below the threshold value, the entry's link to *Offset Loads* entry is dropped and any accesses stemming from the *Instruction Address* of the entry produce a stride prefetcher behaviour using its fields.

The final part of the algorithm is the initiation of prefetches, which takes place for registered *Column* accesses when the *Confidence* counter of the entry corresponding to the instruction addresses initiating the access exceeds the threshold and for irrelevant accesses every time. In the second case, a simple stream prefetching behaviour is initiated, prefetching on access basis the next *DEGREE* cache blocks after the accessed address. The modelling of this behaviour is included as the simulator does not allow multiple prefetching units for the same cache, despite this being the case in the real systems, which use sophisticated logic to coordinate the multiple prefetchers they feature.

For the registered-*column* access, the prefetched patterns are described in section 6.2.1. Though, redundancy in prefetching has to be limited as much as possible by striving to issue prefetches for cachelines not requested already. Based on this direction, for *offset* loads, in case the entry's stride is smaller than the size of the cache block, the stride used for prefetches is set to cache block size (line 32). With this adjustment, redundant prefetches targeting the cacheline of the initiating access are not issued, while multiple prefetches targeting subsequent cachelines are issued only once. Instead the prefetcher takes advantage of the bandwidth it is allowed to use based on its *degree*, by issuing a single prefetch for each of the next *DEGREE* cache blocks as described in lines 33 to 35.

Preventing redundant prefetches for *column value* accesses is more complicated as calculating the address of a prefetch requires an access to the corresponding element of the *offsets buffer* in order to produce the index of the *value buffer* to be prefetched. In cases of small offsets differences like the

String Offsets buffers for the names and the description of the purchases in figure 6.3, multiple values pointed by subsequent objects are on the same cacheline. For access patterns accessing those values through such subsequent, close to each other, offsets, multiple prefetches would be issued for the same cacheline based on the prefetching scheme described in section 6.2.1. To avoid such a scenario, the address of the last issued prefetch of the current iteration of the algorithm is stored. After the new address is calculated, it is allowed to proceed only if it moved, towards the direction of the stride of the linked *Offset Loads* entry, to a different cacheline. Otherwise, a prefetch for the subsequent cache block to this direction is issued. Such a check further increases the complexity of the prefetcher, but is crucial for improving the memory behaviour of algorithms using access pattern like these.

6.2.4. Configuration

From the above analysis of the algorithm, it can be noted that most of the important decisions like initiating prefetches, updating the stride or dropping an entry are taken based on the value of the *Confidence* counter of the entry involved and some predetermined thresholds. As described in section 2.2.1 for the stride prefetcher, the selection of such values plays a major role in determining the aggressiveness and the responsiveness of the prefetcher. The distance between the initial value of the entry's counter and the confidence threshold determines how many accesses are needed in the training phase of the prefetcher before it is confident enough that the correct pattern was identified to start prefetching. The same holds for the distance between the minimum allowed value of the confidence counter, which the more it differs from the threshold value, the longer the training phase required after a large spell of access anomalies for the involved entry. On the other hand, the distance between the maximum value of the counter and the prefetch threshold determines how insensitive is the prefetcher to disturbances such as out of order accesses, but also affects its responsiveness to actual changes of the access pattern.

As the effect of those parameters become evident, their selection becomes important. For simplification of both the logic and the hardware, the minimum value of the counter was chosen to be 0. In ascending order, the threshold was set subsequently to a value of 4, as after any number of misses, four accesses following the same pattern are considered sufficient for prefetching to be reinitiated. As the threshold value is used for both the initialisation of the *Offset Loads* entries and of the *Value Loads* entries when the accessed index and constant calculation fail, the initial value of the confidence counter is only used for successful initialisations of *Value Loads* entries. In the OOO core, out of order appearance of the accesses is not uncommon during the first few accesses after the registration since the beneficial effect of prefetching of the corresponding offset elements is not yet fully present and thus, there is variability in miss latency for the different lanes which allows them to proceed in an arbitrary order. To reduce the sensitivity of the prefetcher in those initial irregularities, the initial value for setting the confidence counter was set to 20, which would produce several inaccurate prefetches in case of wrong identification of the pattern initially. Though due to the complexity of the identification process, such events are expected to be rare. Finally, the maximum value of the counter was set to 49, which is a very distant value from the threshold rendering the prefetcher fairly insensitive to pattern changes. The reasoning behind this choice is the vulnerability of the *value buffer* elements prefetching mechanism to memory congestion, as if the required elements of the *offsets buffer* are not in cache, the calculation of the accessed index and constant will fail reducing the confidence even if the pattern is unchanged in reality.

This choice of parameters and the use shared values for them for both *offsets buffer* and *value buffer* prefetching is only an attempt to simplify the algorithm and constrain the number of parameters included in the design space. For achieving the optimal performance, possibly separate confidence parameters should be investigated for them as their properties and sensitivity differ. The same holds for different prefetch degrees and possibly the introduction of prefetch distances, which all of them are left as proposals for future research.

6.3. Results

Based on the implementation documented in section 6.2, the *Arrow* prefetcher is expected to perform like a normal *stride* prefetcher for the constant offset accesses without any indirection, while for access introducing one or more levels of indirection, its performance is expected to be superior than the reference prefetchers to the extent the bandwidth limitations allow. In order to test this hypothesis,

the developed prefetcher model was tested using all the benchmarks listed in section 3.2, while a more in depth comparison of its performance with both the software and the ideal prefetching followed.

Arrow prefetcher behaviour defaults to simple stride prefetching when no nesting is involved and the *value buffer* is accessed in multiples of its value data size. In figures 6.5 and 6.6, the performance of the developed prefetcher was tested in comparison to the best performing of the reference prefetchers by calculating its speed-up over it for all the studied benchmarks. As expected, the speed-up is very close to 1 for the constant stride accessing benchmarks. A similar behaviour is observed for both the word and sentence granularity of the first letter capitalisation algorithms since the use of almost every cacheline allows the *stream on access* prefetcher to perform very similarly. For the paragraph granularity, the speed-up is limited from the bandwidth capabilities of the memory system. Consequently, greater speed-up is observed for the In-Order core in which the execution is slower due to its limited computation resources and a marginal increase compared to the other granularities for the OOO core.

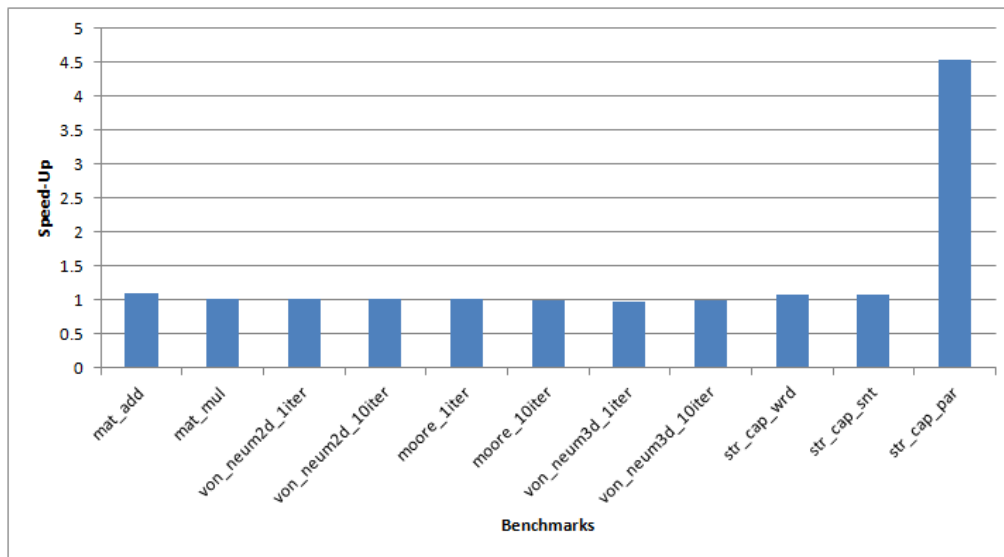


Figure 6.5: Speed-up of the *Arrow* Prefetcher for the different benchmarks executed on the In-Order core over the best of the reference prefetcher in each case

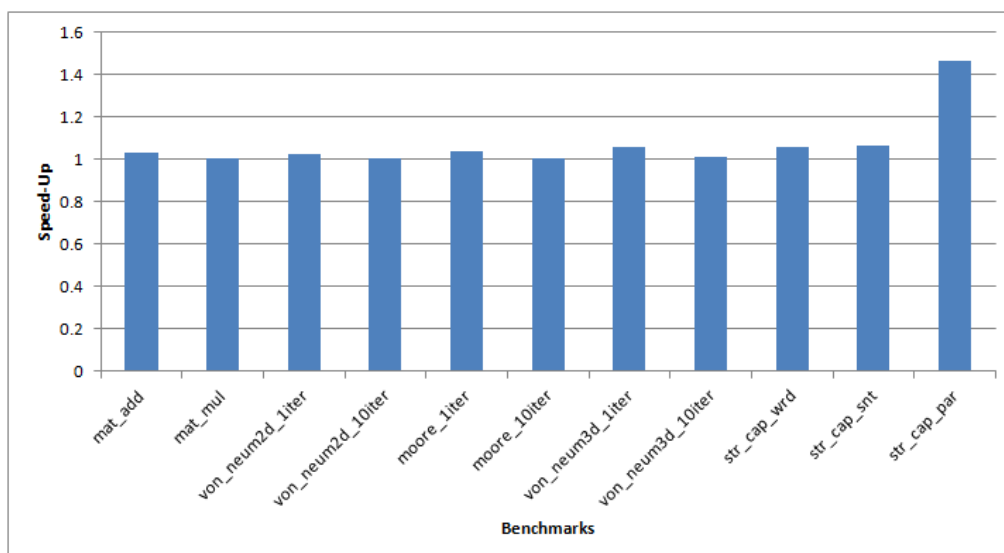


Figure 6.6: Speed-up of the *Arrow* Prefetcher for the different benchmarks executed on the OOO core over the best of the reference prefetcher in each case

Since the limited speed-up of the indirect accessing was expected for the extremely memory in-

tensive string capitalisation algorithm, a more detail investigation was performed using the synthetic benchmark for an increasing number of operations to quantify the impact of computation intensity on the capabilities of the prefetcher to timely prefetch the indirect access patterns. Like for the case of software prefetching, the same large dataset (60MiBs) was used to emulate the effect *big data* execution would have. Similarly the initial value of the loop and the step were set to 0 and 1 respectively to maximise the number of iterations. The produced graph for the increasing number of iterations for the In-Order core can be seen in figure 6.7.

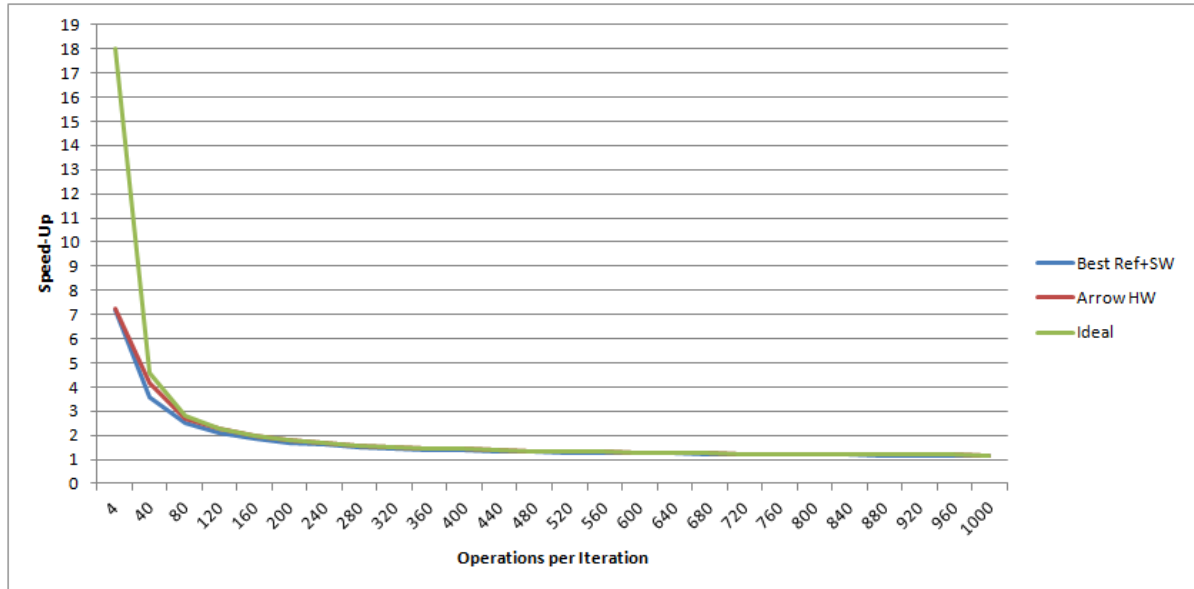


Figure 6.7: Speed-up of the best performing HW-SW prefetching combination, the *Arrow* and the *Ideal* prefetchers when the synthetic benchmark is executed on the In-Order core over the best of the reference prefetchers in each case as the computation intensity per load increases

The speed-up of the *Ideal* prefetcher for the most memory intensive case due to bandwidth constraints faced by the actual implementation skews the general picture and limits the capability of the reader to estimate the actual speed-up relation of the prefetching schemes. For this reason, the same figure with the first value of the operations per iteration excluded is repeated for the reader's convenience in figure 6.8. For the rest of the values, the *Arrow HW* prefetcher is very close to the ideal one and almost matches its behaviour from the value of 120 operations and onward. The best performing reference prefetcher plus software prefetching combination (Ref HW-SW) asymptotically approaches the same behaviour with its performance being almost indistinguishable from the other two schemes after the value of 560 operations. The same diagram for the OOO prefetcher follows in figure 6.9.

The OOO core is able to hide partially the latency of late prefetching of the first values caused by the memory intensity. Though as this property in combination with the bandwidth constraints affect all the realistic prefetchers, the initial speed-up is fairly low for all the prefetching schemes except for the *Ideal* prefetcher. As more computations per load are added, the effect of timely prefetching of the indirect accesses starts to be noticeable and peaks at the 160 operations for both the *Arrow HW* prefetcher and the HW+SW combination. After this point, the memory behaviour starts to have progressively a decreasing effect on the execution time with the algorithm becoming more and more computation intensive. The speed-up for both techniques remains above 2x throughout all the first half of values except for the initial one and their behaviour is very close to the ideal from the 200 operations and onward, while it becomes almost indistinguishable as the speed-up saturates due to the increase in computation intensity, reaching in the final value the 1.5x mark.

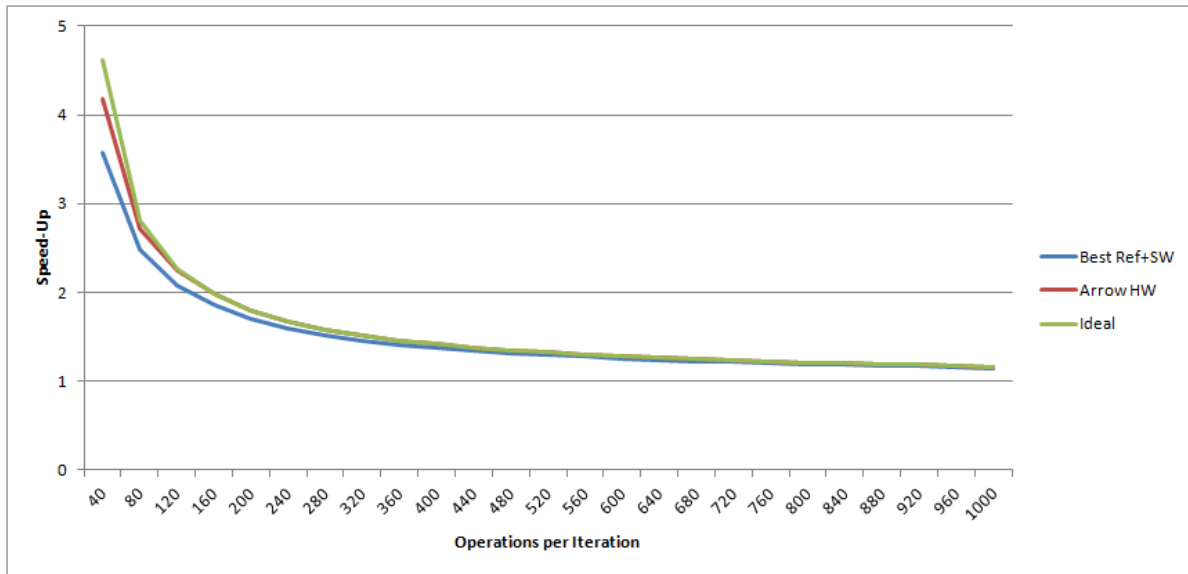


Figure 6.8: Speed-up of the best performing HW-SW prefetching combination, the *Arrow* and the *Ideal* prefetchers when the synthetic benchmark is executed on the In-Order core over the best of the reference prefetchers in each case as the computation intensity per load increases excluding the first value

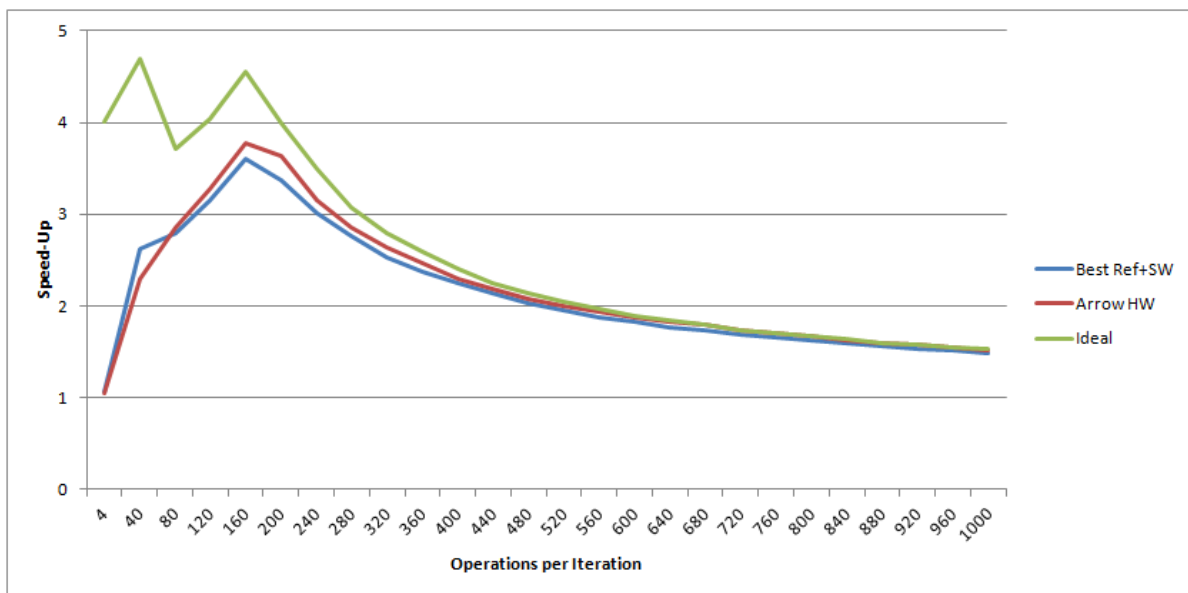


Figure 6.9: Speed-up of the best performing HW-SW prefetching combination, the *Arrow* and the *Ideal* prefetchers when the synthetic benchmark is executed on the OOO core over the best of the reference prefetchers in each case as the computation intensity per load increases

6.4. Discussion

The results presented in the previous section mostly validated the expectations regarding the behaviour of a more sophisticated prefetcher targeting the indirect indexing scheme of *Apache Arrow*, expected to approach the behaviour of an ideal prefetcher even more than the software prefetching scheme of chapter 5, when the system's bandwidth constraints allow.

More specifically, for the constant stride accesses, *Arrow* prefetcher is able to at least match the performance of the best of the reference prefetchers in each of the studied benchmarks, except for the Von Neumann 3D stencil when executed in the In-Order core. The reason of the marginal slowdown (0.98x) is that the prefetcher defaults to the behaviour of *stride* prefetcher for these access patterns and thus, similarly to this scheme, it fails to identify a constant stride for the *memcpy* instructions used to copy the updated pages from the temporary smaller matrix, which was introduced for memory use optimisation purposes as discussed in chapter 4.4. Nevertheless the slowdown is much smaller than the one observed there and thanks to the ability of the OOO core to hide part of the latency, it is converted to speed-up for this core.

The reason behind this speed-up and the marginal speed-up achieved for the rest of the benchmarks is that the prefetcher is able to exploit the contiguity of the *Arrow buffers* with page spanning prefetching. In section 2.3, where the specification of *Apache Arrow* was briefly discussed, the requirement for the *buffers* of each *Arrow array* or each *chunk* of it, to be contiguous in virtual memory was established. As this requirement was enforced for the *buffers* of all registered *Columns* and since the prefetcher operates on virtual addresses, it can issue prefetch requests spanning the virtual pages' boundaries, in contrast to the rest of the prefetchers, which without this information, block page spanning prefetching. This feature becomes more influencing for benchmarks which are either fairly memory intensive and can exploit any marginal improvement on the memory behaviour such as the matrix addition algorithm or include large jumps on their access patterns like the case of Von Neumann 3D stencil. Consequently, for the latter, by exploiting page spanning prefetches, the In-Order core is able to almost nullify any slowdown, while the OOO one is even able to produce a measurable speed-up.

For the first letter capitalisation algorithm, as mentioned earlier, due to the streaming accesses for both the word and sentence granularity, a marginal speed-up is expected and this only because of the prefetching across page boundaries. For the case of paragraphs, the speed-up follows exactly the trend recorded for the minimum number of operations in figures 6.7 and 6.9. As it was mentioned earlier for those cases, the In-Order prefetcher is able to achieve a considerable speed-up as its limited resources hide the bandwidth constraints of the memory system. Though exactly because of this same resources limitation, this speed-up is saturated at a fast pace as the number of operations per load increase, because any further improvement in access times is hidden by the time required for performing the computations. For the cases that the memory behaviour still has a noticeable effect on performance, the software prefetching technique lags slightly behind the developed hardware prefetcher, which almost matches the one of the ideal prefetcher the more the bandwidth constraints become insignificant.

The OOO core, for the high memory intensity cases like the paragraph first letter capitalisation, is constrained by the memory bandwidth, a limitation experienced by each and every of the actual hardware prefetchers and thus, their relative speed-up remains fairly low. As the number of operations per iteration increases, the load of the execution units of the core follows the same trend, resulting in the bandwidth constraints having a progressively decreasing effect and allowing the superior hit rates achieved by the *Arrow* prefetcher to impact performance. The speed-up peaks for 160 operations per iteration at almost half the value (3.78x) of the In-order execution (7.27x). Nevertheless, its saturation is much slower sustaining a speed-up larger than 2x from before the point of 40 operations per iteration until this of 520, and larger than 3x within the [100, 250] interval. The software techniques prefetching at much longer distance are able to capitalise on the available bandwidth earlier, but for larger loads than 80 operations per iteration constantly lags behind the *Arrow HW* prefetcher. This trend continues until they both almost match the ideal performance at the point of 520 operations per iteration as the memory induced speed-up saturates with the increasing computation intensity.

Except for the speed-up, both techniques approach the accuracy of the ideal prefetcher as well. This accuracy, nevertheless, does not have a measurable effect on performance due to the limited data reuse profiles of the benchmarks, with the data being used again either very close to the initial access or very far from it. Because of that, any inaccurate prefetches of the stream prefetchers do not evict data, which would have remained on the cache otherwise until they were used again. The extra bandwidth they consume has an effect in memory intense cases in which the bandwidth is constraining,

though their effect would be more notable in cases when multiple preemptable threads would run on the same core or for workloads with a different reuse profile.

Since both software and hardware prefetching are able to approach the behaviour of ideal prefetching when not constrained by the memory resources, the software-hardware prefetching trade-off, as outlined in the last 2 chapters, lies primarily on the much lower implementation cost of the software prefetching and the far superior reliability and adaptability of the hardware prefetcher. As hardware prefetcher issues both offset and value prefetches on a per access basis, the issue rate is affected by the rate the accesses are serviced and allow for the next to proceed, in contrast to the software prefetching for which both the value prefetches and accesses are issued in bursts, without being able to take into account the run-time parameters. This behaviour except for the ability to adapt the issuing of prefetches, it renders the hardware prefetching technique more resilient to fluctuations in memory congestion and thus, allows it to sustain a speed-up more consistently. Nevertheless, the prefetcher comes at a design complication and area cost, which increases the more complex the logic of the prefetcher becomes and the more resources it requires, which for the case of the developed prefetcher are non-trivial. Attempting to increase reliability and to automate the process of software prefetching by integrating its issuing logic to the compiler incurs some research and design costs as well though, which should not be neglected when deciding on which method is preferable for any particular application.

Conclusions

Over the course of this study, several steps have been made towards the direction of answering the research questions formulated in chapter 1. In section 7.1 an attempt to summarise the results of this study, which provide answers to those questions, to the extent the limitation to be discussed in section 7.2 allowed, followed by some suggestions for potential contributions in the final section 7.3.

7.1. Summary

The feasibility of accelerating *big data analytics* applications through an improvement of the memory behaviour was the problem identified in section 1.2. As the *Apache Arrow* has been proven to have a therapeutic influence on this aspect, it was used as a base for the introduction of further improvements.

Using a simplified version of the relevant parts of *Apache Arrow* developed in C++, an Ideal prefetching behaviour was simulated in chapter 4, identifying that the constant offset accesses using multiples of the data size of the *value buffer* elements as strides, are sufficiently identified and accommodated by the reference prefetchers documented in section 2.2.1, to the extent the bandwidth constraints of the memory system allows. For the indirect indexing accesses, such as those performed by the first letter capitalisation, string manipulation algorithm, the reference prefetchers lagged considerably behind the ideal prefetcher, allowing it to achieve a speed-up, over the best performing of them, up to 13x and more than 5x for systems featuring In-Order and OOO cores respectively. This speed-up can be partially attributed to the bandwidth constraints, which the ideal prefetcher does not simulate, but the observation of increased latency per cache block of the *stream on access* prefetcher, possibly due to the issuing of inaccurate prefetches for those patterns, hinted that part of the speed-up can be attributed to the inability of the reference prefetchers to accurately identify and prefetch those patterns.

Further investigation of the indirect indexing patterns, which *Apache Arrow* uses for all its nested accesses as described in section 2.3, followed with the evaluation of the impact on execution time the issuing of software prefetches for them could have. Software prefetching was employed only for the *value buffer* accesses, using a scheme partitioning algorithms into bursts of prefetches and burst of execution iterations. The prefetching of the needed elements of the *offsets buffers* was left to the reference prefetchers in order to avoid a waste of bandwidth with both software and hardware prefetching issuing requests for the same patterns. The best performing combinations of prefetching degree and distance parameters improved the performance over the reference prefetchers on both the In-Order and OOO cores for all the different computation intensities and approached the ideal behaviour for medium or higher computation intensities. Despite the best performing combination for each computation intensity case heavily underperforming in other cases or neighbouring with underperforming combinations, towards different profiles of memory congestion might move their performance, more stable combinations with performance very close to the best ones were successfully determined.

Consequently, it can be concluded that the indirect indexing patterns used by *Apache Arrow* to access any nested structure, cannot be identified sufficiently by the reference prefetchers in many cases. The information about their offsets contained in the *Apache Arrow* metadata can be exploited to accurately prefetch them, achieving a performance close to the ideal when the memory bandwidth constraints allow. Arrays of variable length structures such as lists and strings are favoured the most,

especially when their size exceeds the number of cachelines stream prefetchers issue requests for (degree). Though smaller sizes could benefit as well, as long as the indirect indexing patterns do not access (nearly) every cacheline sequentially, as is the case of the word and sentence granularities of the first letter capitalisation algorithm.

In an attempt to find a more reliable way of prefetching those patterns, the information regarding the *Arrow Columns* and more specifically the locations in virtual memory and the lengths of their *offset* and *data buffers* were exploited to develop a hardware prefetching algorithm described in chapter 6. Based on the simulation results, this algorithm has been able to approach even closer than the software prefetching technique the performance of the ideal prefetcher, to the extent the memory characteristics allow, for both the In-Order and the OOO cores. Despite both the software and hardware techniques approaching this behaviour sufficiently by exploiting those metadata, the hardware algorithm achieves that in a more adaptable to fluctuations in memory congestion and reliable way, but at higher cost. This consistency-cost trade-off has to be decided per case, with both an investigation on the performance of those schemes for different profiles of memory congestion and an evaluation of the cost of a concrete implementation of the hardware prefetching algorithm, being necessary parts of this process.

The effect of prefetching on algorithms with different computation intensities was studied as well, revealing a particular profile of algorithm with a computation intensity within a certain range, which profit the most from it. In the initial investigations of the performance of the reference prefetchers in comparison to the ideal one, extremely computation intensive cases, like the matrix addition algorithm, seem to benefit by a prefetching scheme of increased aggressiveness, but despite the capacity of prefetchers to perfectly identify and issue prefetches for their access patterns, they fail to approach the ideal memory behaviour due to bandwidth limitation. On the other side of the spectrum, the performance of the prefetchers in very computation intensive algorithms is very close to the ideal, but they do not have any significant impact on the execution time as the data brought to the cache are used over and over again. Applications in the middle of this spectrum, profit from prefetching the most, as the data reuse keeps the pipeline sufficiently full allowing the memory bandwidth to accommodate accurate prefetches on time before the new data are requested.

More specifically for the indirect indexing patterns and the, *Arrow* metadata exploiting, prefetching techniques which were proposed in the framework of this study, algorithms with less than 360 operations per iteration are benefited the most when executed in the In-Order core by experiencing a more than 1.5x speed-up for these intensities. Furthermore, the speed-up initially starts fairly high (7x) for the very memory intensive cases and saturates rapidly onward. For the OOO execution, several intervals of improved performance due to prefetching can be identified. Namely, from the point that the bandwidth limitations of the memory cease to dictate performance, which is after the point of 40 iterations, the speed-up remains above 1.5x, while inside the [40,250] and [100,250] the speed-up is above 2x and 3x for the hardware prefetcher and close to those values for the software scheme. The computation intensity profiles described by those ranges cover a large spectrum of algorithms in general, while distinct examples for the indirect indexing pattern specifically should be determined and tested to confirm the findings.

7.2. Limitations

During the course of this study, several limitations were faced in different steps. Most of them are already documented in the corresponding sections of the report, but for the convenience of the reader they will be gathered and summarised in this section as well.

As in all studies using simulation techniques to investigate several properties of any system, the accuracy of the simulator is always a limitation to the extent that it can alter the results in a way leading to differences in the extracted conclusions. As explained in chapter 3, *ARM* architecture, which is the most accurate of the ISAs, was used in order to limit the error to the minimum. Nevertheless, the trade-off between accuracy and invested time for this kind of initial investigations has to be accepted, as their goal is to prove that there is merit in pursuing the under-evaluation concepts further.

Another limitation affecting the findings of this study is the limited variety in the tested system configurations, which was confined to just two systems with fundamentally different single-core CPUs, though with the same underlying system parameters. Although the structuring of the system and the selection of those parameters were based in a particular reasoning (explained in section 3.1), the absence of multi-core scenarios, in which the multiple cores would contest for the use of memory resources, limited

both the ability to determine the parameters of the prefetching modules, in a way to be representative of a real multi-core system and the capability to generalise the results.

Furthermore, several limitations concerning the benchmarks used to quantify the effectiveness of the prefetchers were identified. Specifically, the use of the real *Apache Arrow* library in combination with this particular ISA in the simulator was not possible. The use of a different ISA, except for limiting the accuracy, would significantly complicate the experiments as necessary prefetch and cache invalidation instructions are not implemented for other ISAs. Without the use of the actual library though, another source of error was introduced as the differences in implementation and its extra complexity could have a minor impact on the results, which is not expected to alter their qualitative characteristics though. In addition to that, only microbenchmarks were used in the evaluation process, limiting the conclusions to algorithms of very specific profiles without the ability to generalise with for more complex realistic combinations. The synthetic benchmark contributed to the introduction of more variety, but still more is needed to reach definitive conclusions.

As documented in section 2.2.2, the implementation and parameters of software prefetching are ISA-dependent and thus, the performance improvement it is able to achieve as well as the optimal combination of distance and degree parameters are dependent on it as well. For a more complete study of software prefetching for *Arrow*, more ISAs should be explored, especially those popular in the *big data analytics* field. Nevertheless, identifying the optimal parameters is a separate process for each ISA and thus, the results identified in chapter 5 are representative for ARM. To further solidify the findings though, the testing of different scenarios of memory congestion is required, as the process of determining well-performing, resilient prefetch parameters combinations have a significant dependence on them.

The scope of this study is to investigate the merit of further exploration of the concept of prefetching schemes utilising the *Apache Arrow*'s metadata and identify the datasets and workloads which would benefit the most from such schemes. This property disqualifies limitations related to the implementation cost of either of the proposed techniques or the coverage of the different cases of indirect indexing they achieve, from being included. Nevertheless, those concepts were considered throughout the duration of this study and influenced many decisions over its course.

7.3. Suggestions for Future Research

Except for the obvious suggestions of tackling the aforementioned limitations and implementing the prefetching techniques to real systems in order to obtain results that will allow for a real case evaluation for the potential use of this schemes in *big data analytics* systems, several other proposals for concepts to be investigated will be presented in this chapter.

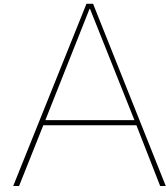
Both the proposed prefetching techniques targeted the L1D cache, taking into account that all prefetch request will have to be serviced from memory, without attempting to exploit the larger L2 cache. Furthermore, a common property of both is the necessity of being able to access early enough the *offsets buffer* elements needed in order to calculate the indices of the *value buffer* to be prefetched. Based on this remark, a less sophisticated prefetching scheme for the L2 cache could be designed to attempt to aggressively prefetch the contents of the *offsets buffers* used in L2 cache. Such a scheme could be implemented using other software or hardware, without having to be as accurate and as timely as the L1D prefetching.

A suggestion related to software prefetching is to include the option in existing compilers or to create dedicated ones able to compile the applications specifically for any particular *Arrow object*. Such compilers would be able to substitute the indirect indexing accesses with the actual, relative to the base of the *value buffer*, offsets reducing the number of accesses by half and the data dependencies significantly, allowing for better filling of the CPU's pipeline. Of course, such a scheme would hardcode the access patterns to those for the specific object the code was compiled for requiring recompilation for using the same code for a different object. Though if the speed-up is measured to be more significant than the compilation-per-object overhead, then such a method could be of interest.

As mentioned in the configuration section (6.2.4) of the hardware prefetching algorithm, the properties and the sensitivity to memory congestion of the confidence parameters of the *Offsets Loads* and *Value Loads* are different. Consequently, having shared confidence and degree parameters may have a negative impact overall on the timeliness and thus, on the performance benefit the prefetcher is able to induce. Testing several combinations of separate parameters along with adding the option for a

prefetching distance larger than 1 could possibly provide a better tuning of the hardware prefetcher and thus, increase its speed-up to approach even more the ideal prefetching.

Finally, different kinds of memory could be employed capitalising on the immutability of the *Arrow objects*. A use for such a memory could be the introduction of a scratchpad being used for storing the offsets or any part of the *object's* data or metadata. For the case only the offsets are chosen to be stored in the scratchpad while the data go through the cache system in the normal way, the hardware prefetcher, which needs constant access to the offsets, could instead of being inside the L1D cache, be attached to the scratchpad with a port allowing it to access the *offsets buffers'* elements, and issue normal prefetching requests targeting the data cache. In this way, bandwidth from the constant transferring of the offsets could be saved and be exploited by other accesses and the prefetching algorithm could be simplified as it would not have to issue prefetches for the offsets or provide support for the cases the required offsets are not found in the cache for value prefetching.



Additional figures

A.1. Ideal Prefetcher

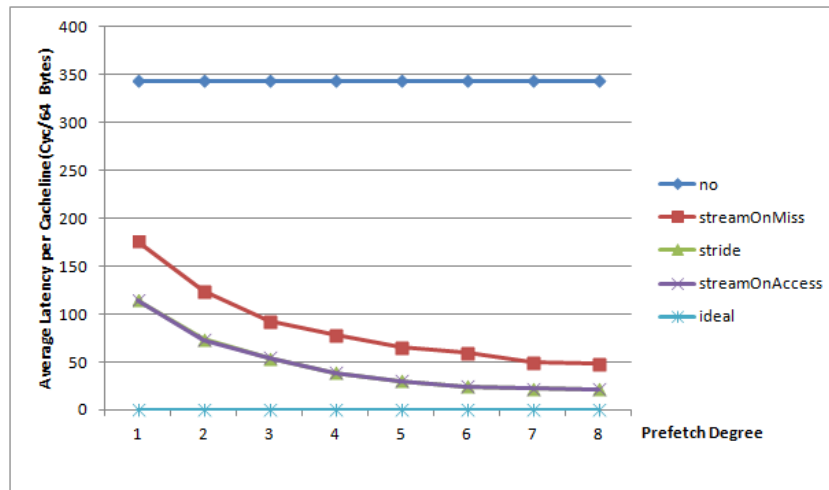


Figure A.1: Latency per cache block of systems featuring the tested prefetchers when matrix addition is executed on the In-Order core as their degree increases

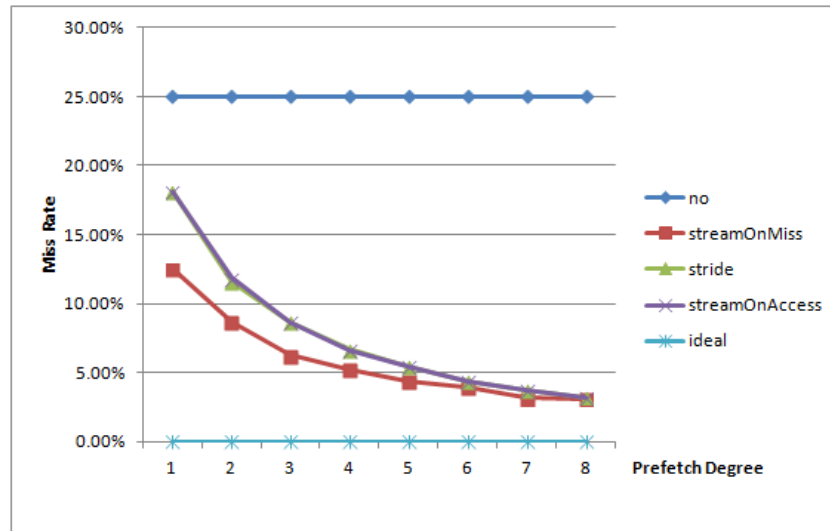


Figure A.2: Cache miss rate of systems featuring the tested prefetchers when matrix addition is executed on the In-Order core as their degree increases

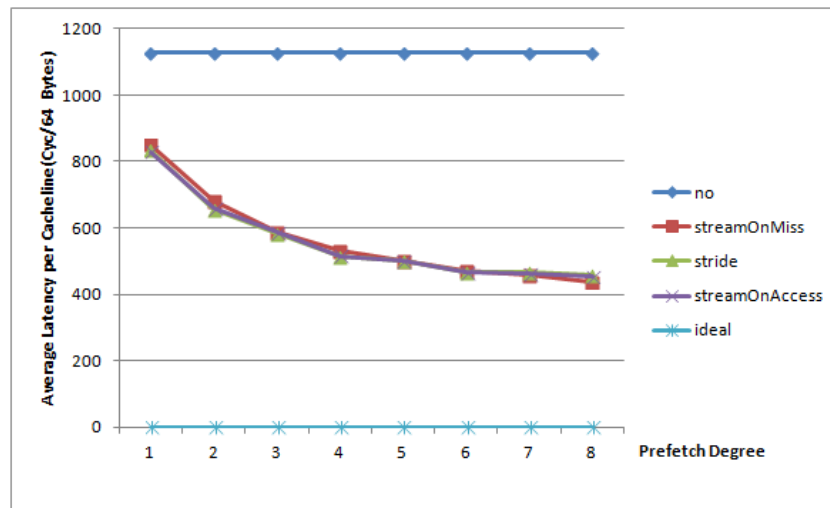


Figure A.3: Latency per cache block of systems featuring the tested prefetchers when matrix addition is executed on the OOO core as their degree increases

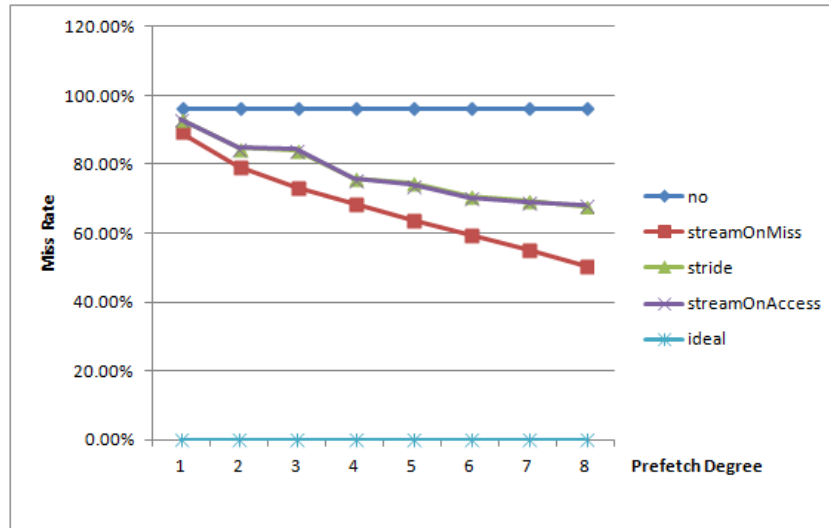


Figure A.4: Cache miss rate of systems featuring the tested prefetchers when matrix addition is executed on the OOO core as their degree increases

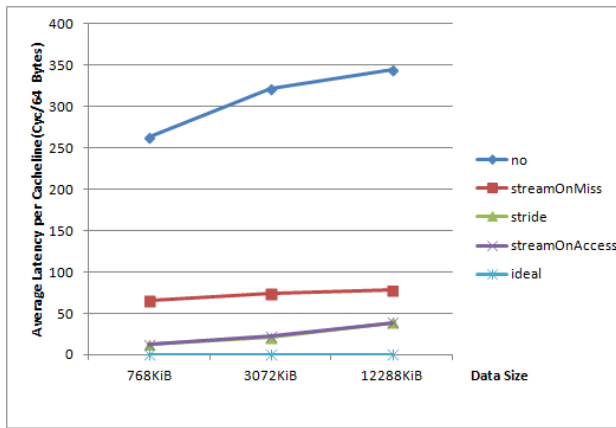


Figure A.5: Latency per cache block of systems featuring the reference prefetchers when matrix addition is executed on the In-Order core as the input size increases

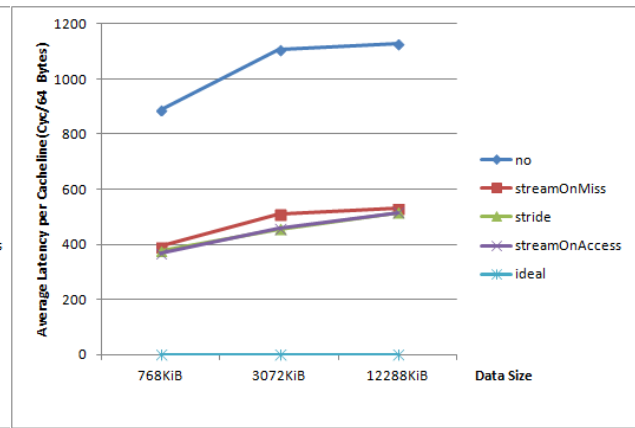


Figure A.6: Latency per cache block of systems featuring the reference prefetchers when matrix addition is executed on the OOO core as the input size increases

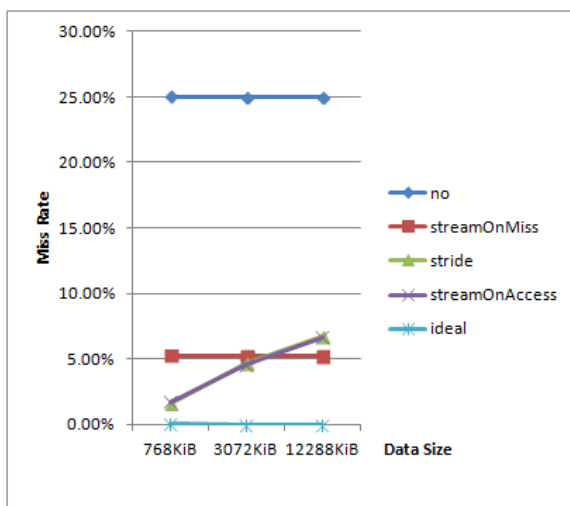


Figure A.7: Cache miss rate of systems featuring the reference prefetchers when matrix addition is executed on the In-Order core as the input size increases

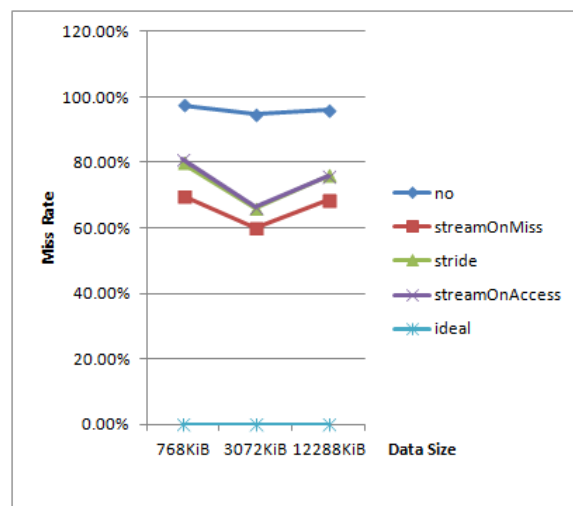


Figure A.8: Cache miss rate of systems featuring the reference prefetchers when matrix addition is executed on the OOO core as the input size increases

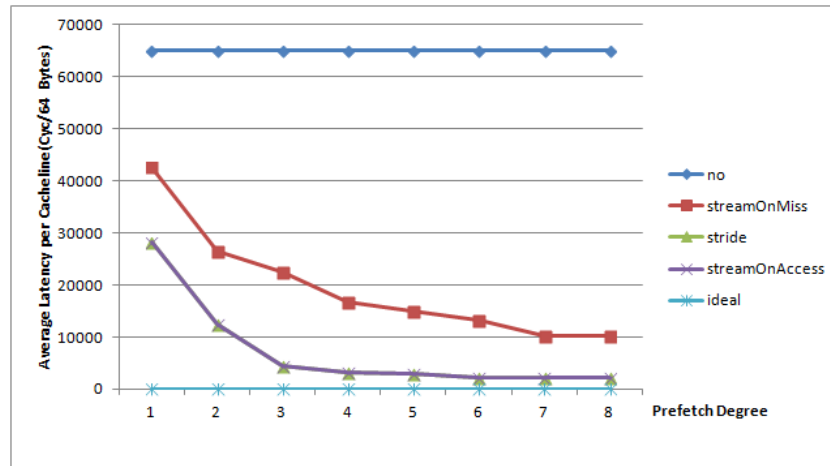


Figure A.9: Latency per cache block of systems featuring the tested prefetchers when matrix multiplication is executed on the In-Order core as their degree increases

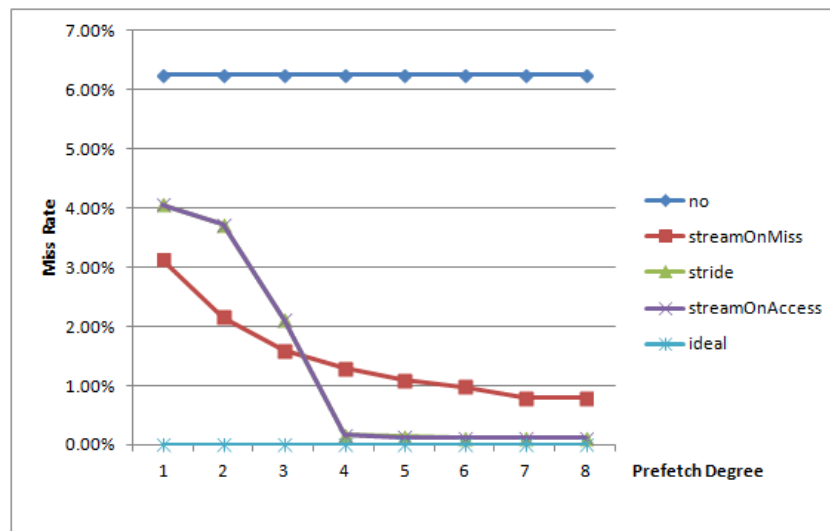


Figure A.10: Cache miss rate of systems featuring the tested prefetchers when matrix multiplication is executed on the In-Order core as their degree increases

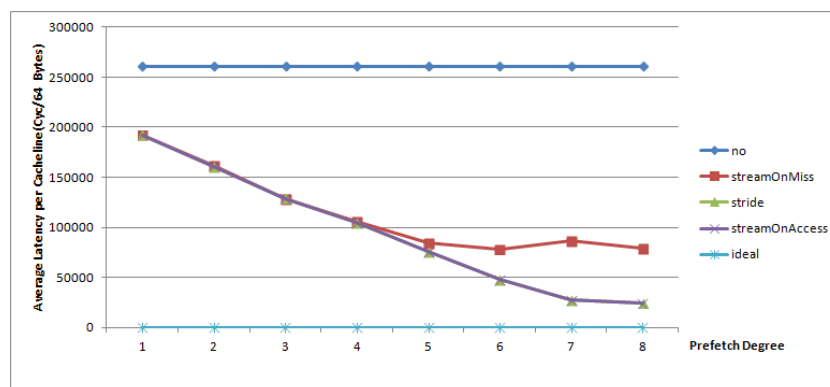


Figure A.11: Latency per cache block of systems featuring the tested prefetchers when matrix multiplication is executed on the OOO core as their degree increases

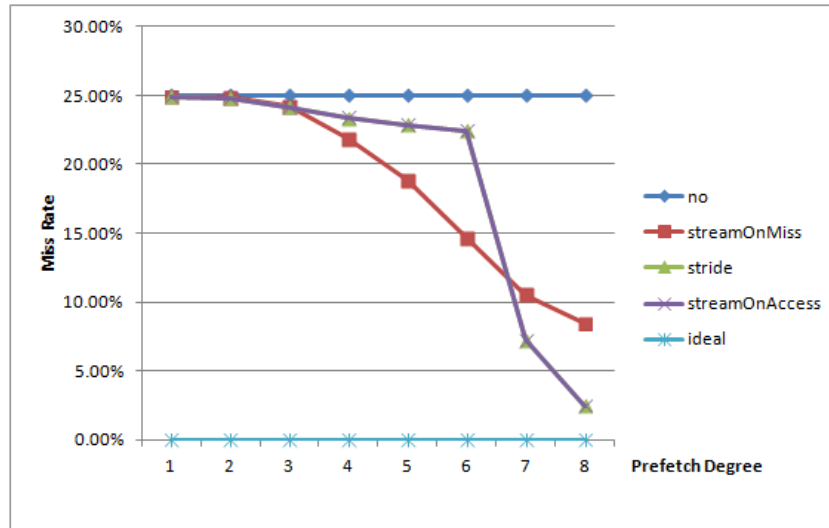


Figure A.12: Cache miss rate of systems featuring the tested prefetchers when matrix multiplication is executed on the OOO core as their degree increases

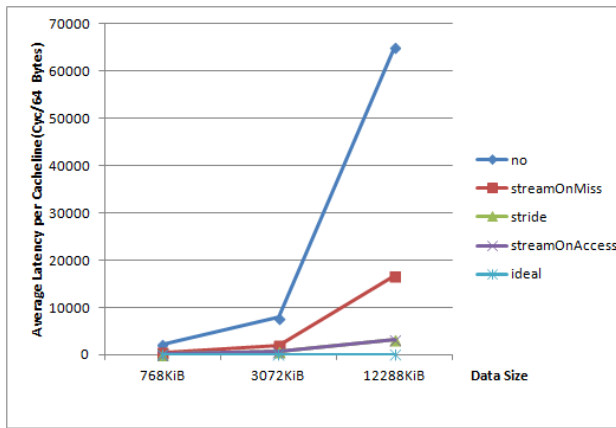


Figure A.13: Latency per cache block of systems featuring the reference prefetchers when matrix multiplication is executed on the In-Order core as the input size increases

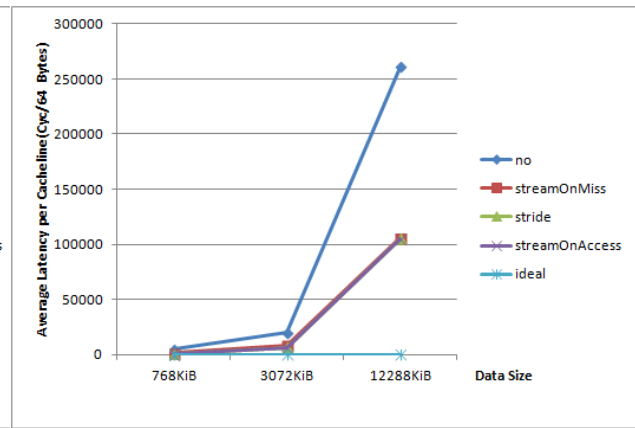


Figure A.14: Latency per cache block of systems featuring the reference prefetchers when matrix multiplication is executed on the OOO core as the input size increases

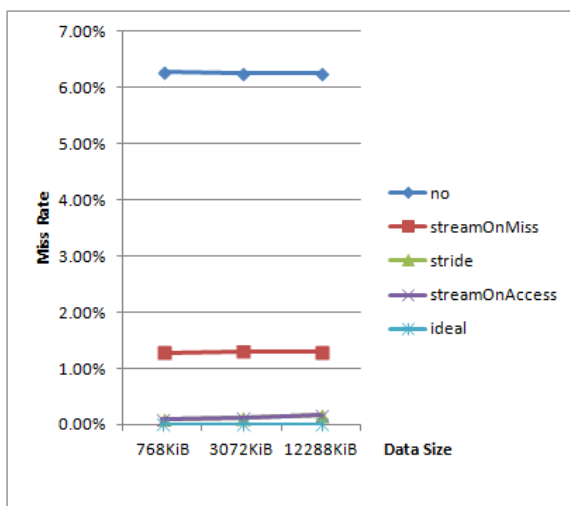


Figure A.15: Cache miss rate of systems featuring the reference prefetchers when matrix multiplication is executed on the In-Order core as the input size increases

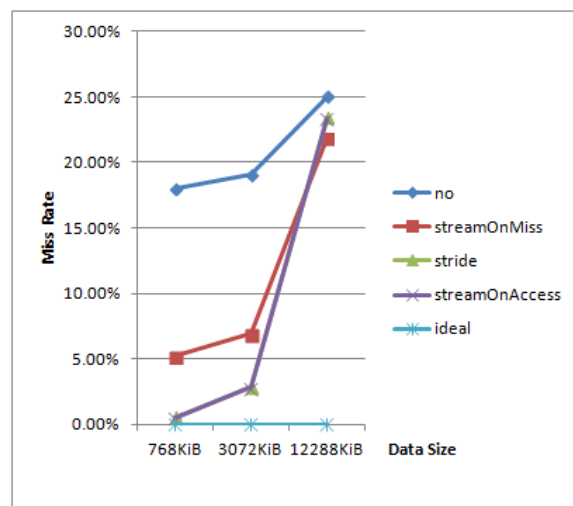


Figure A.16: Cache miss rate of systems featuring the reference prefetchers when matrix multiplication is executed on the OOO core as the input size increases

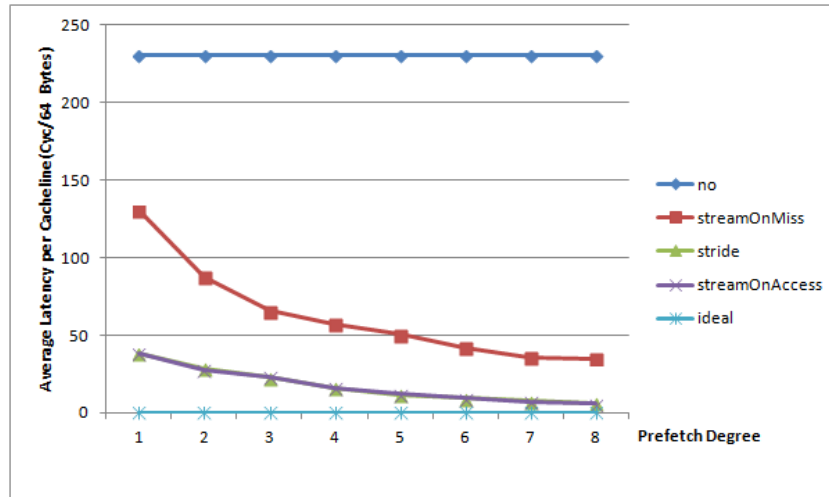


Figure A.17: Latency per cache block of systems featuring the tested prefetchers when a single iteration of the Von Neumann Neighbourhood stencil is executed on the In-Order core as their degree increases

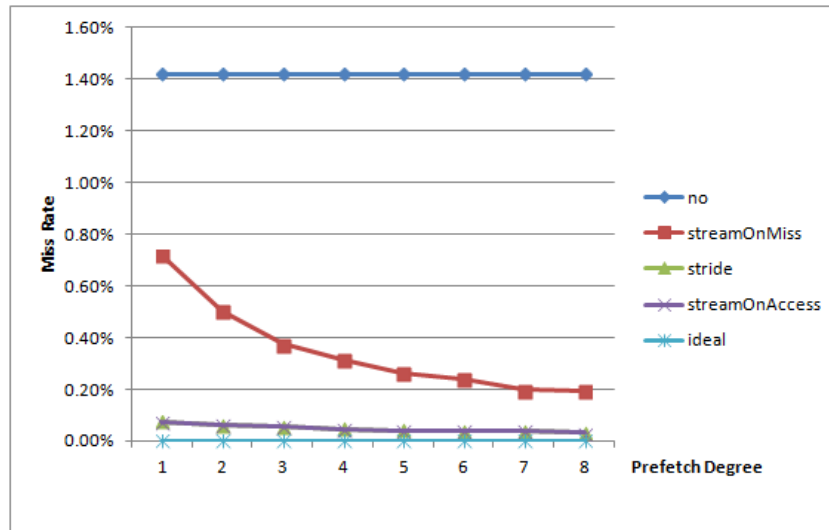


Figure A.18: Cache miss rate of systems featuring the tested prefetchers when a single iteration of the Von Neumann Neighbourhood stencil is executed on the In-Order core as their degree increases

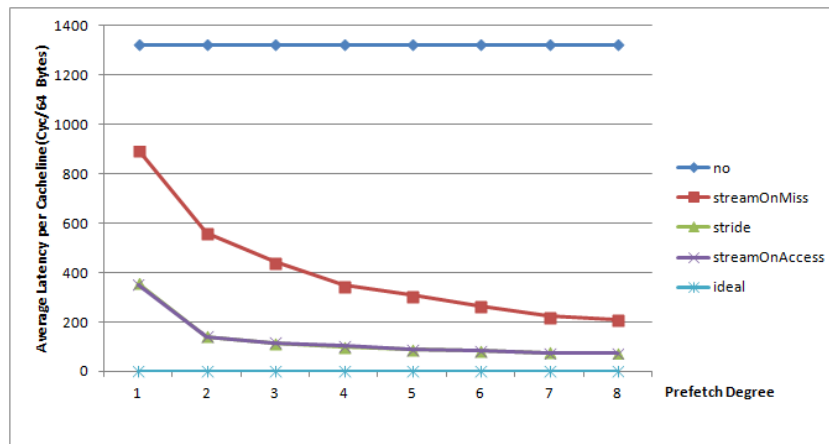


Figure A.19: Latency per cache block of systems featuring the tested prefetchers when a single iteration of the Von Neumann Neighbourhood stencil is executed on the OOO core as their degree increases

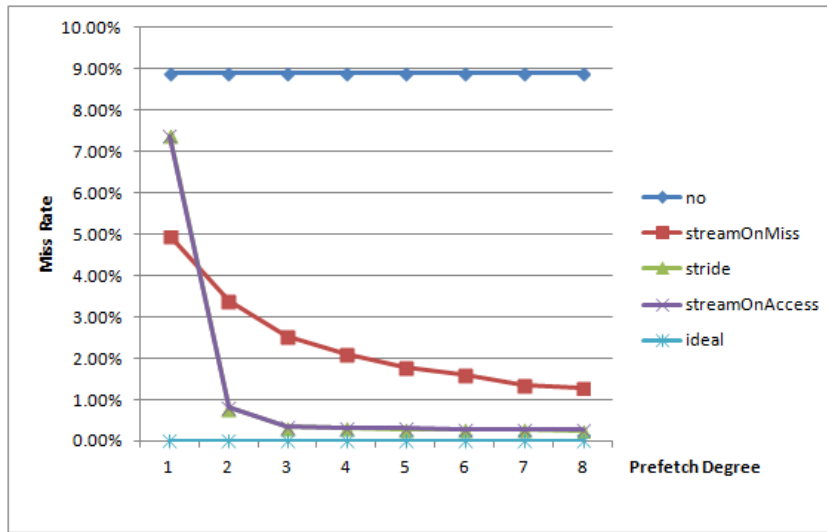


Figure A.20: Cache miss rate of systems featuring the tested prefetchers when a single iteration of the Von Neumann Neighbourhood stencil is executed on the OOO core as their degree increases

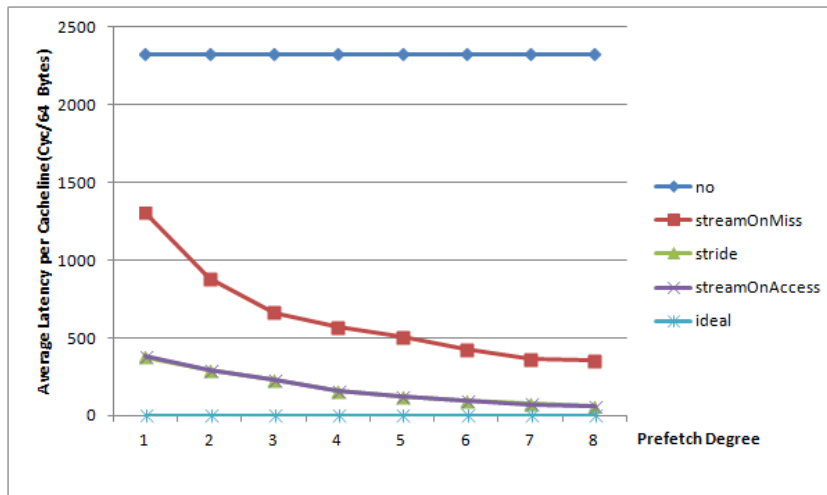


Figure A.21: Latency per cache block of systems featuring the tested prefetchers when 10 iterations of the Von Neumann Neighbourhood stencil are executed on the In-Order core as the degree increases

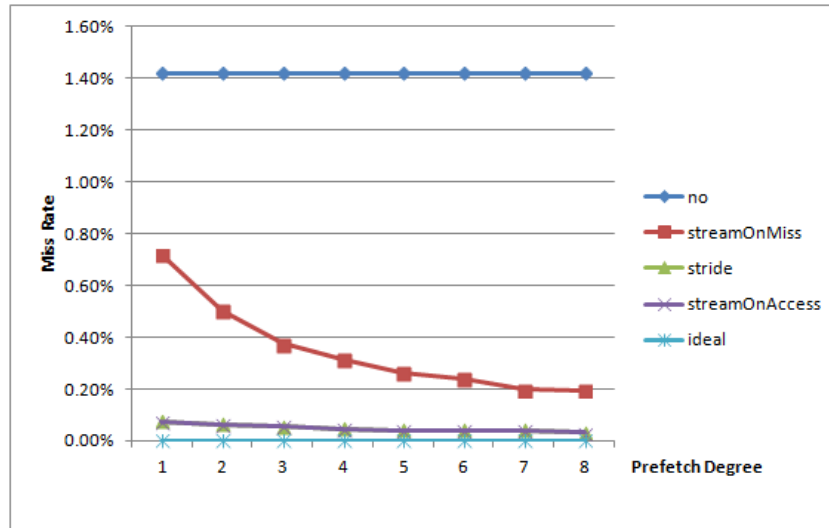


Figure A.22: Cache miss rate of systems featuring the tested prefetchers when 10 iterations of the Von Neumann Neighbourhood stencil are executed on the In-Order core as the degree increases

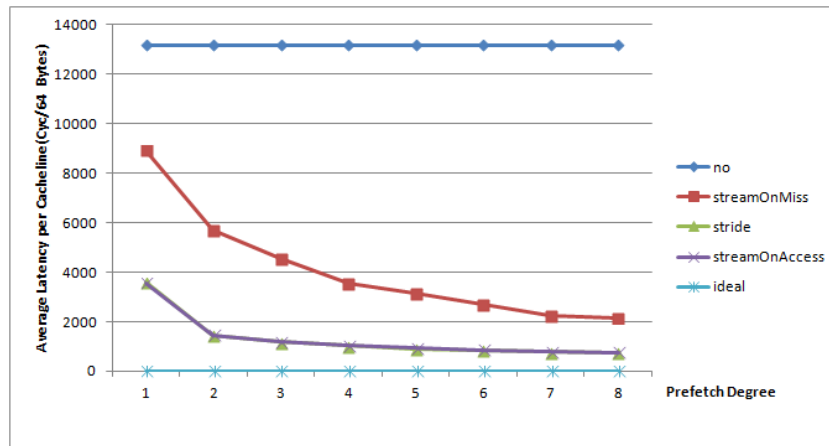


Figure A.23: Latency per cache block of systems featuring the tested prefetchers when 10 iterations of the Von Neumann Neighbourhood stencil are executed on the OOO core as the degree increases

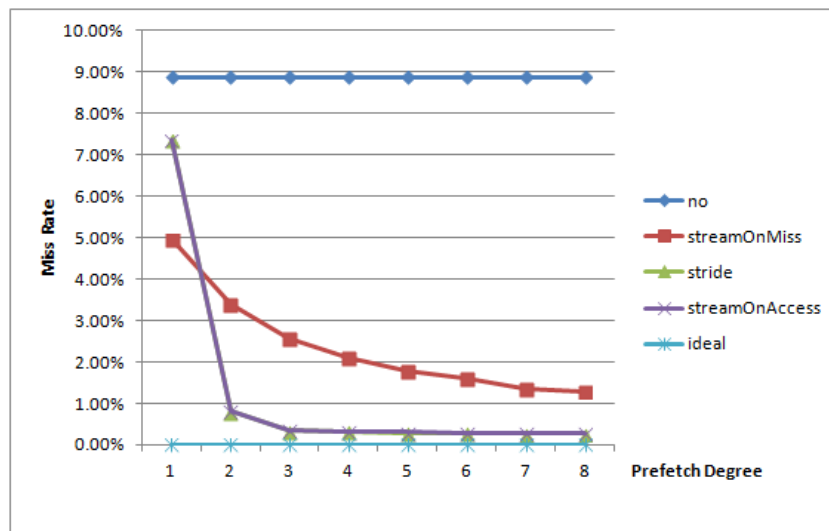


Figure A.24: Cache miss rate of systems featuring the tested prefetchers when 10 iterations of the Von Neumann Neighbourhood stencil are executed on the OOO core as the degree increases

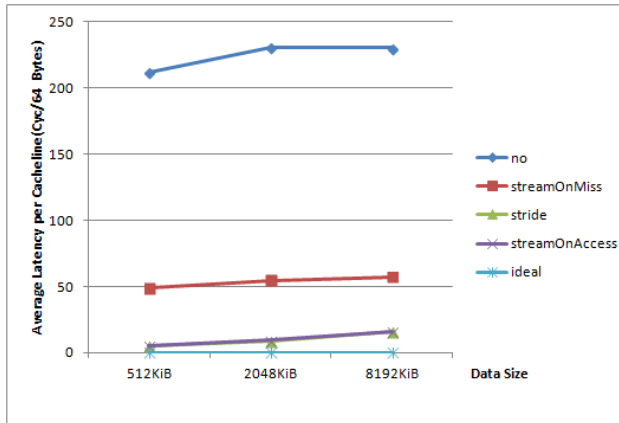


Figure A.25: Latency per cache block of systems featuring the reference prefetchers when a single iteration of the Von Neumann Neighbourhood stencil is executed on the In-Order core as the input size increases

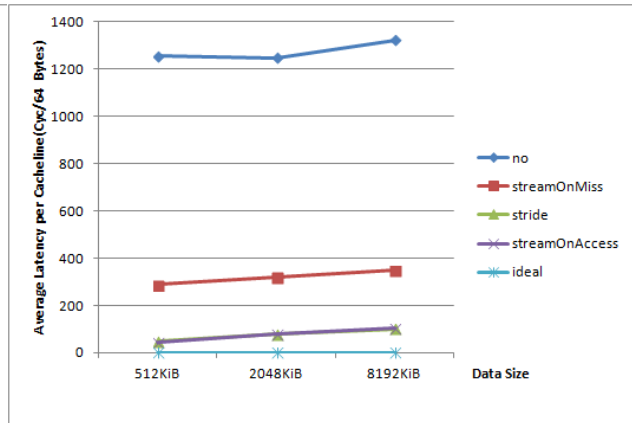


Figure A.26: Latency per cache block of systems featuring the reference prefetchers when a single iteration of the Von Neumann Neighbourhood stencil is executed on the OOO core as the input size increases

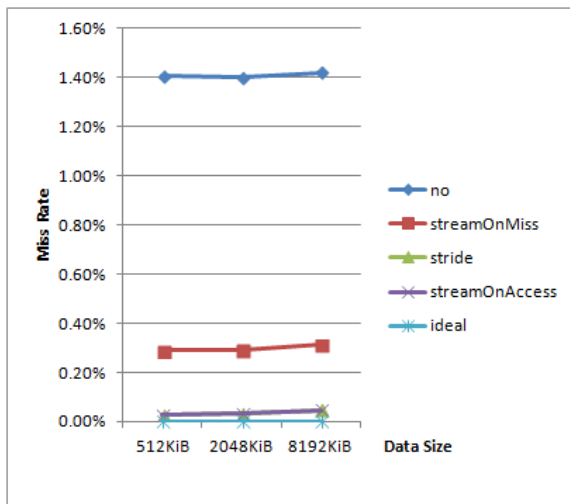


Figure A.27: Cache miss rate of systems featuring the reference prefetchers when a single iteration of the Von Neumann Neighbourhood stencil is executed on the In-Order core as the input size increases

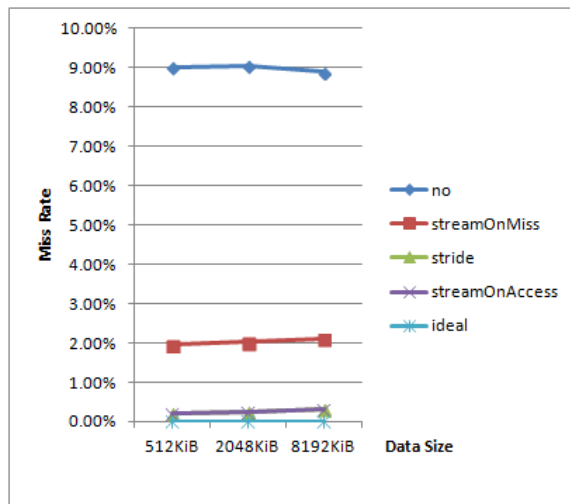


Figure A.28: Cache miss rate of systems featuring the reference prefetchers when a single iteration of the Von Neumann Neighbourhood stencil is executed on the OOO core as the input size increases

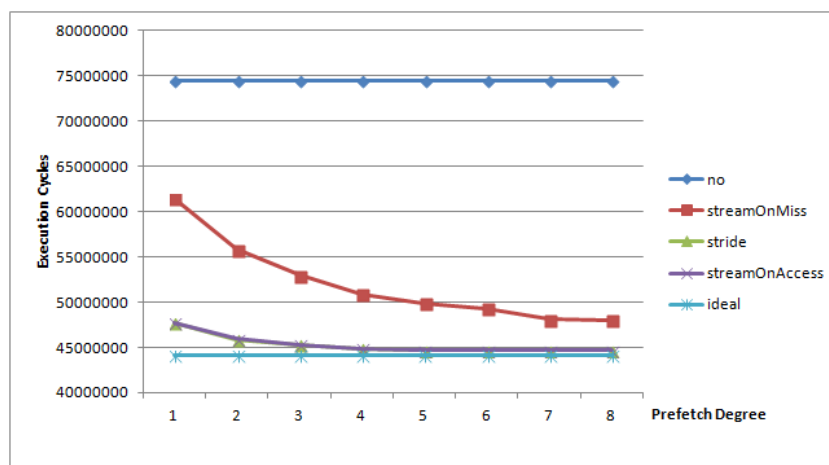


Figure A.29: Execution time of systems featuring the tested prefetchers when a single iteration of the Moore Neighbourhood stencil is executed on the In-Order core as the degree increases

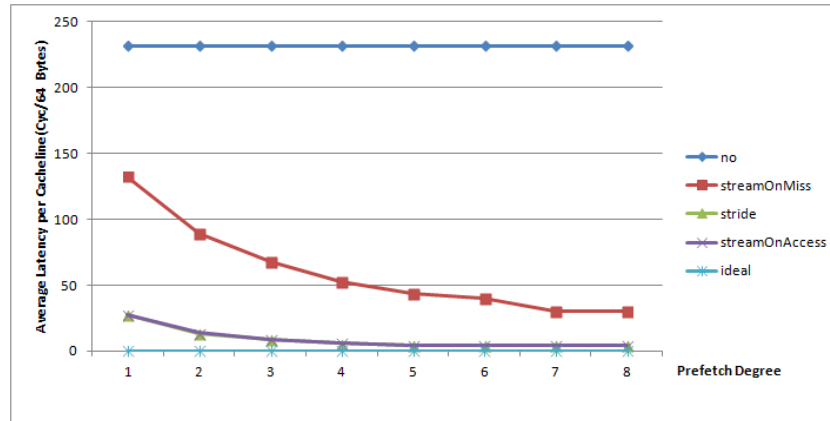


Figure A.30: Latency per cache block of systems featuring the tested prefetchers when a single iteration of the Moore Neighbourhood stencil is executed on the In-Order core as the degree increases

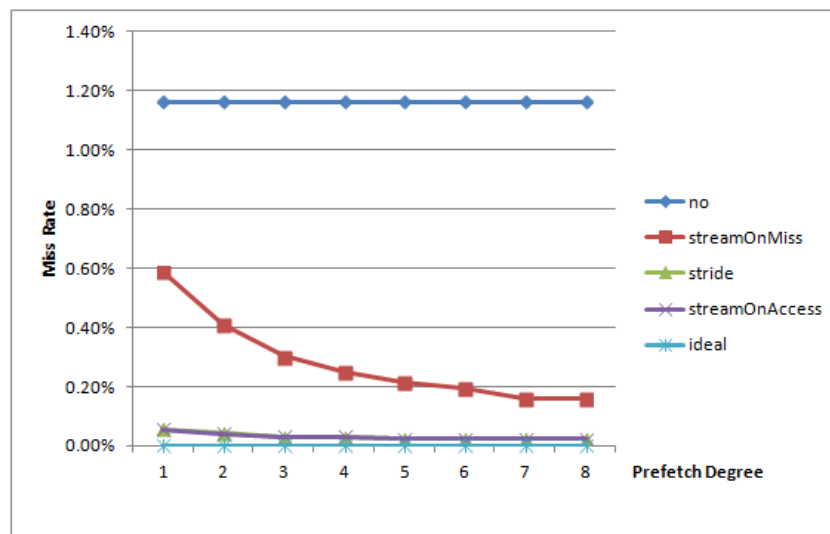


Figure A.31: Cache miss rate of systems featuring the tested prefetchers when a single iteration of the Moore Neighbourhood stencil is executed on the In-Order core as the degree increases

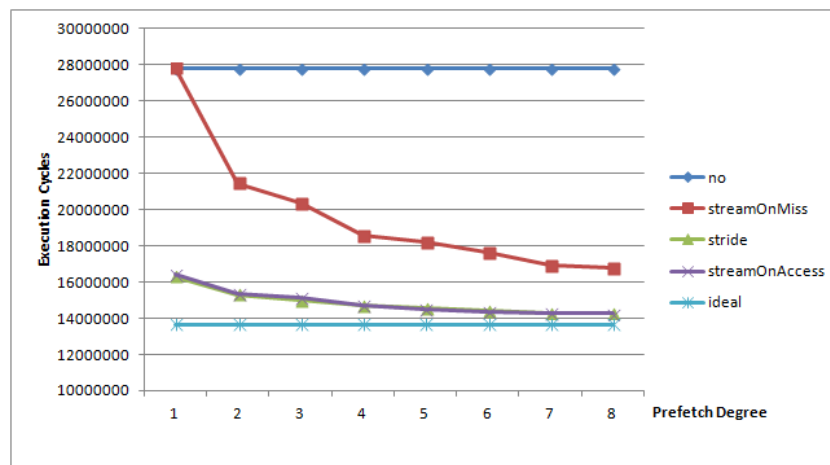


Figure A.32: Execution time of systems featuring the tested prefetchers when a single iteration of the Moore Neighbourhood stencil is executed on the OOO core as the degree increases

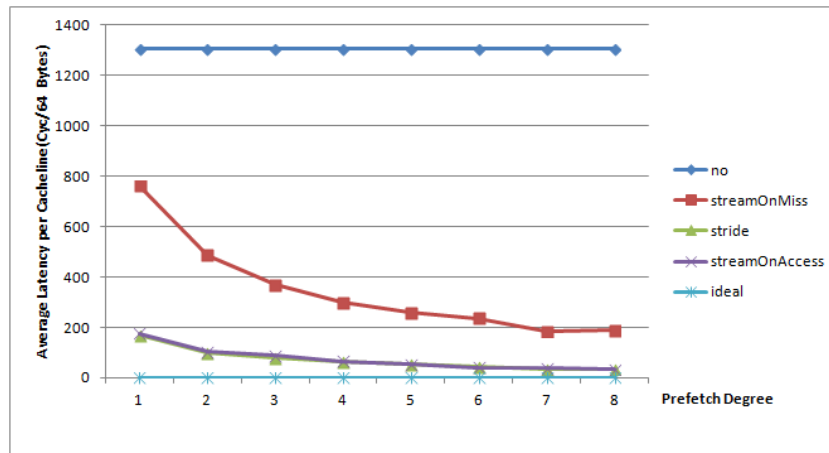


Figure A.33: Latency per cache block of systems featuring the tested prefetchers when a single iteration of the Moore Neighbourhood stencil is executed on the OOO core as the degree increases

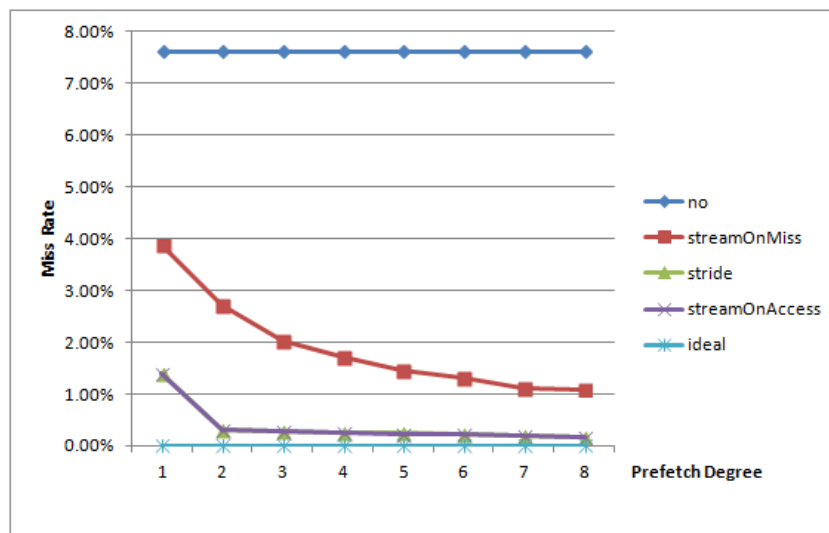


Figure A.34: Cache miss rate of systems featuring the tested prefetchers when a single iteration of the Moore Neighbourhood stencil is executed on the OOO core as the degree increases

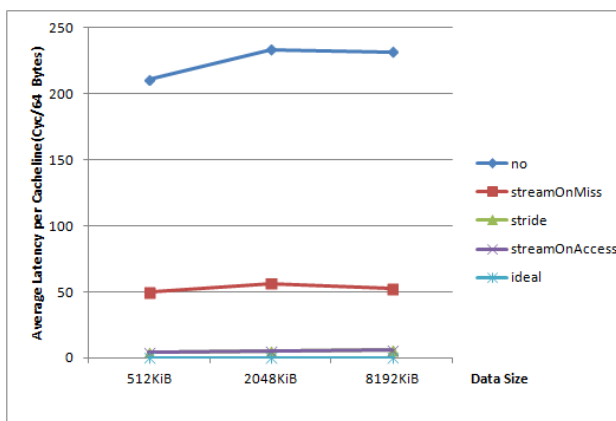


Figure A.35: Latency per cache block of systems featuring the reference prefetchers when a single iteration of the Moore Neighbourhood stencil is executed on the In-Order core as the input size increases

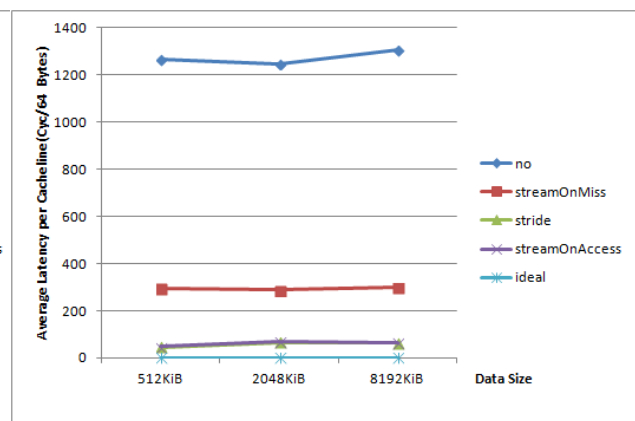


Figure A.36: Latency per cache block of systems featuring the reference prefetchers when a single iteration of the Moore Neighbourhood stencil is executed on the OOO core as the input size increases

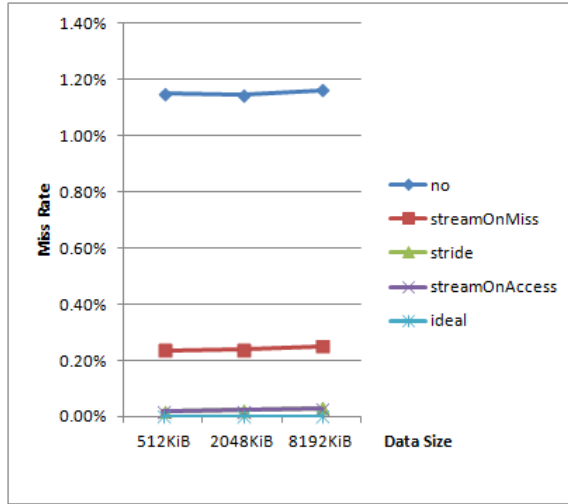


Figure A.37: Cache miss rate of systems featuring the reference prefetchers when a single iteration of the Moore Neighbourhood stencil is executed on the In-Order core as the input size increases

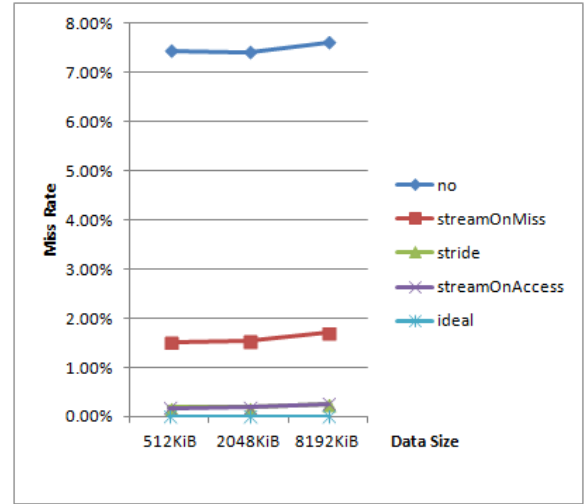


Figure A.38: Cache miss rate of systems featuring the reference prefetchers when a single iteration of the Moore Neighbourhood stencil is executed on the OOO core as the input size increases

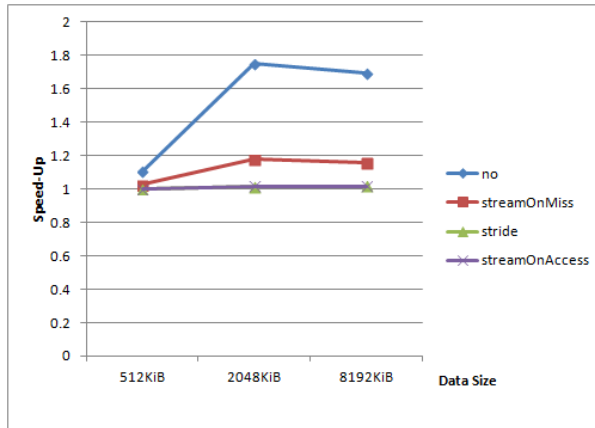


Figure A.39: Speed-up of the Ideal Prefetcher when 10 iterations of the Moore Neighbourhood stencil are executed on the In-Order core over the reference prefetchers as the input size increases

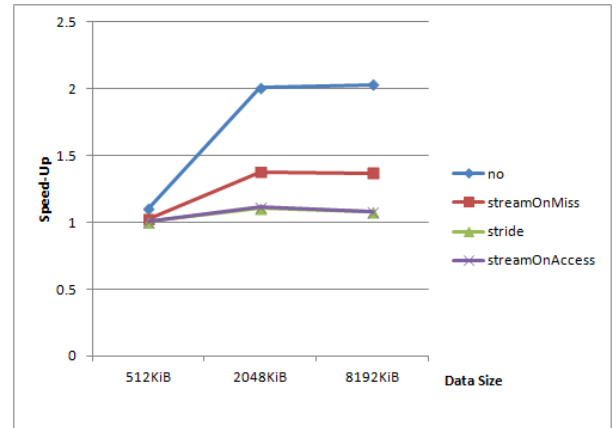


Figure A.40: Speed-up of the Ideal Prefetcher when 10 iterations of the Moore Neighbourhood stencil are executed on the OOO core over the reference prefetchers as the input size increases

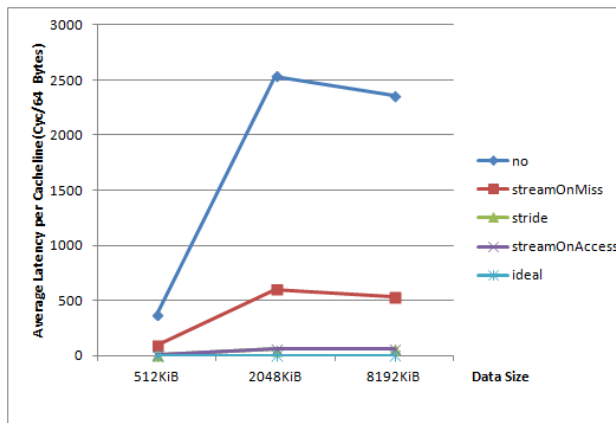


Figure A.41: Latency per cache block of systems featuring the reference prefetchers when 10 iterations of the Moore Neighbourhood stencil are executed on the In-Order core as the input size increases

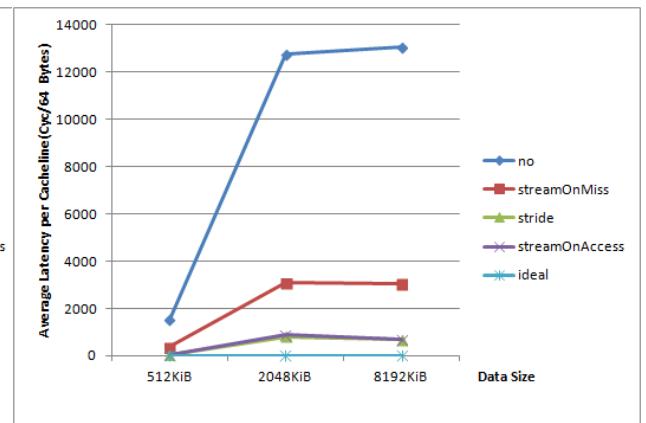


Figure A.42: Latency per cache block of systems featuring the reference prefetchers when 10 iterations of the Moore Neighbourhood stencil are executed on the OOO core as the input size increases

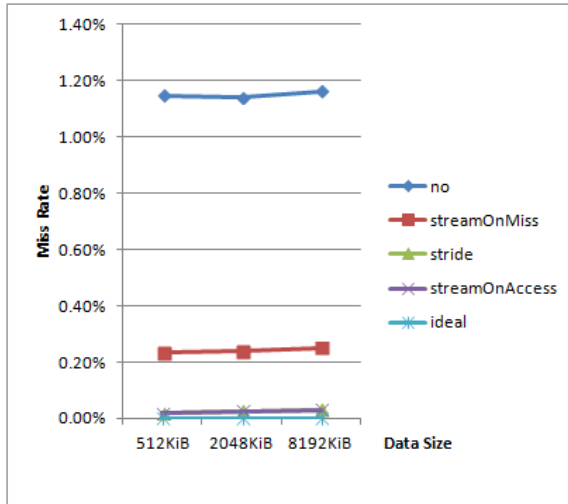


Figure A.43: Cache miss rate of systems featuring the reference prefetchers when 10 iterations of the Moore Neighbourhood stencil are executed on the In-Order core as the input size increases

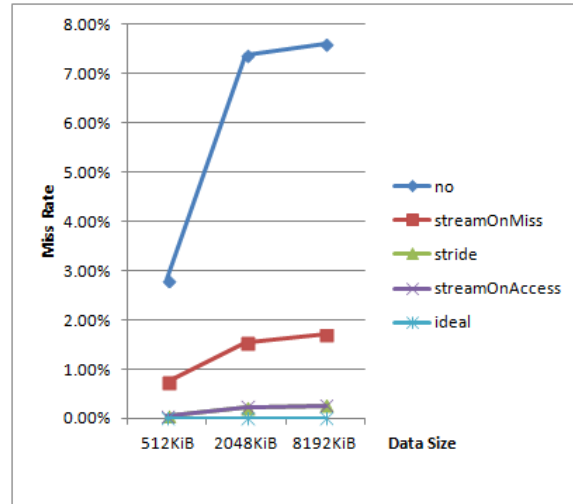


Figure A.44: Cache miss rate of systems featuring the reference prefetchers when 10 iterations of the Moore Neighbourhood stencil are executed on the OOO core as the input size increases

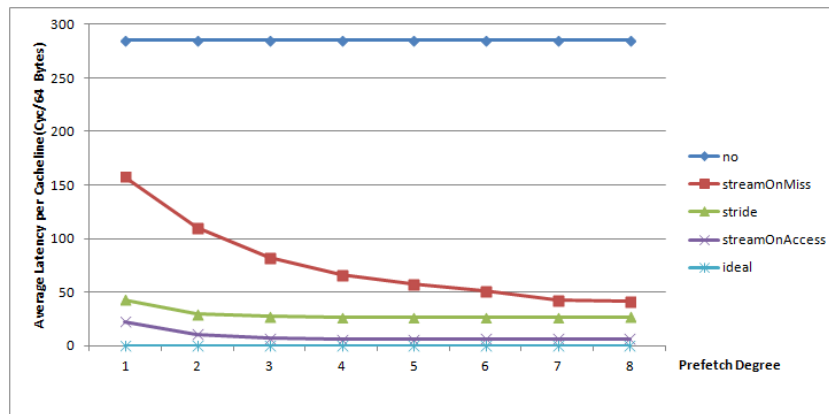


Figure A.45: Latency per cache block of systems featuring the tested prefetchers when a single iteration of 3D Von Neumann Neighbourhood stencil is executed on the In-Order core as the degree increases

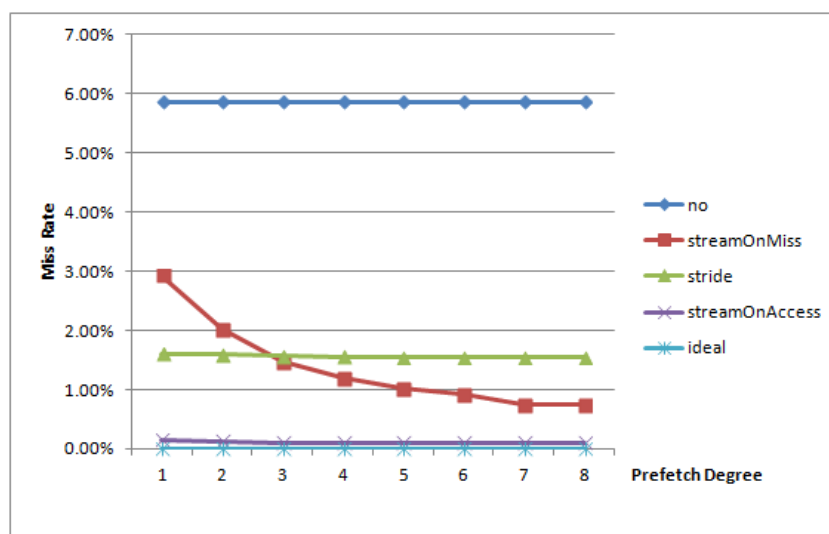


Figure A.46: Cache miss rate of systems featuring the tested prefetchers when a single iteration of 3D Von Neumann Neighbourhood stencil is executed on the In-Order core as the degree increases

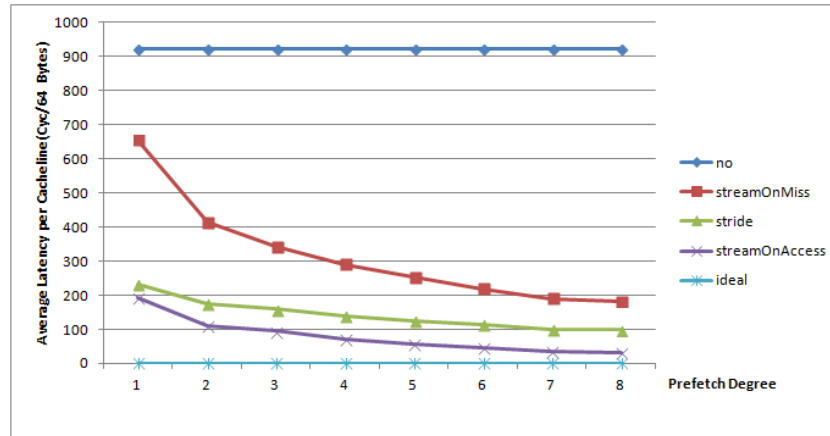


Figure A.47: Latency per cache block of systems featuring the tested prefetchers when a single iteration of 3D Von Neumann Neighbourhood stencil is executed on the OOO core as the degree increases

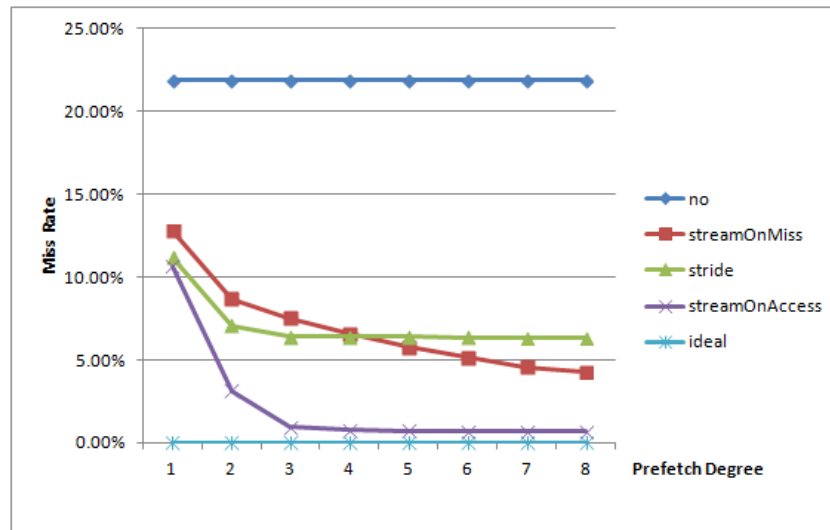


Figure A.48: Cache miss rate of systems featuring the tested prefetchers when a single iteration of 3D Von Neumann Neighbourhood stencil is executed on the OOO core as the degree increases

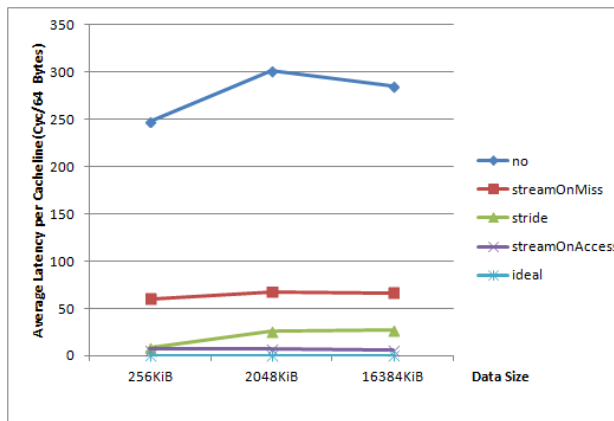


Figure A.49: Latency per cache block of systems featuring the reference prefetchers when a single iteration of the 3D Von Neumann Neighbourhood stencil is executed on the In-Order core as the input size increases

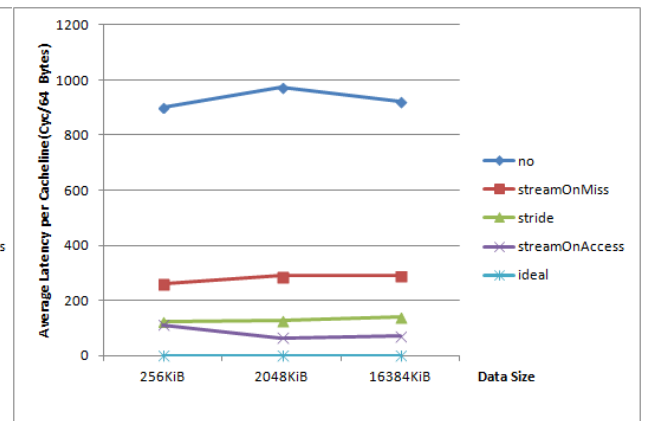


Figure A.50: Latency per cache block of systems featuring the reference prefetchers when a single iteration of the 3D Von Neumann Neighbourhood stencil is executed on the OOO core as the input size increases

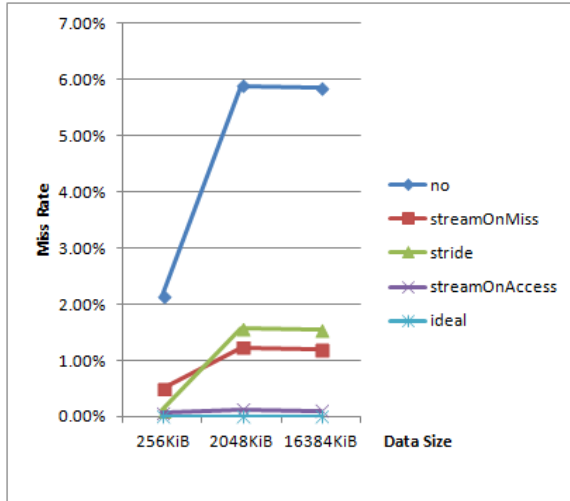


Figure A.51: Cache miss rate of systems featuring the reference prefetchers when a single iteration of the 3D Von Neumann Neighbourhood stencil is executed on the In-Order core as the input size increases

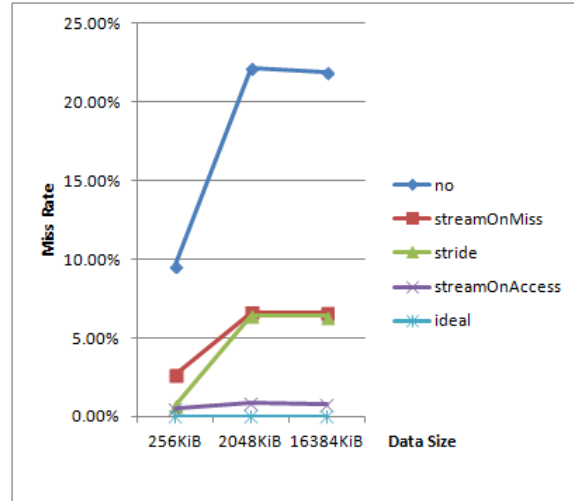


Figure A.52: Cache miss rate of systems featuring the reference prefetchers when a single iteration of the 3D Von Neumann Neighbourhood stencil is executed on the OOO core as the input size increases

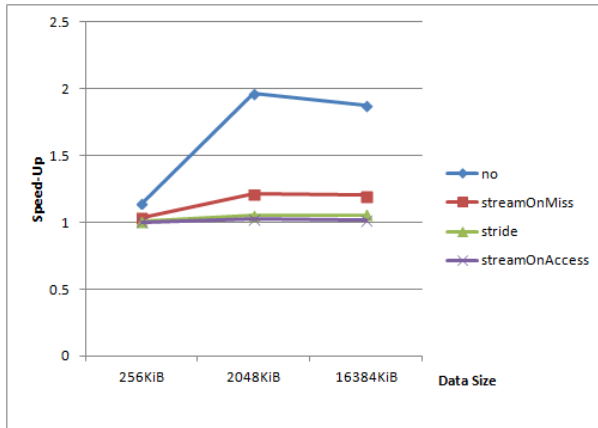


Figure A.53: Speed-up of the Ideal Prefetcher when 10 iterations of the 3D Von Neumann Neighbourhood stencil are executed on the In-Order core over the reference prefetchers as the input size increases

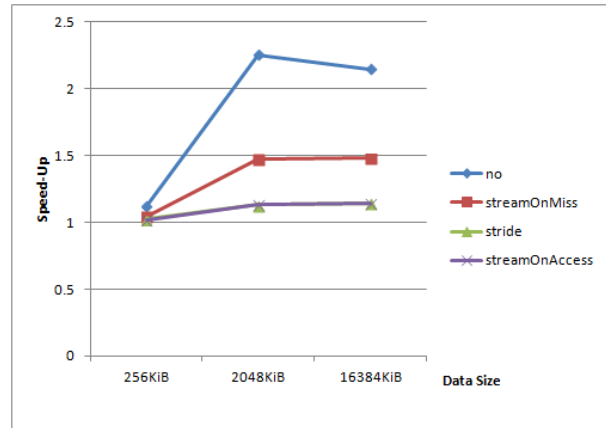


Figure A.54: Speed-up of the Ideal Prefetcher when 10 iterations of the 3D Von Neumann Neighbourhood stencil are executed on the OOO core over the reference prefetchers as the input size increases

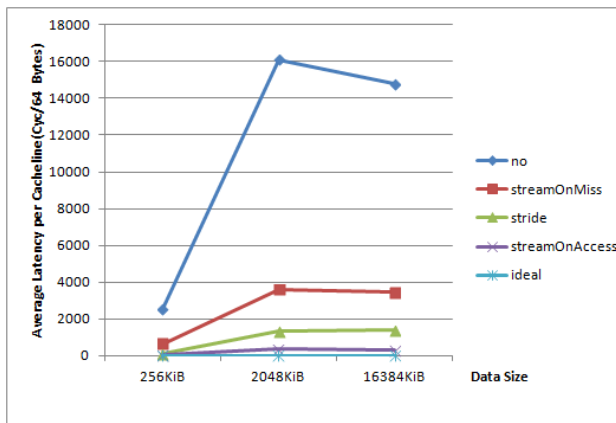


Figure A.55: Latency per cache block of systems featuring the reference prefetchers when 10 iterations of the 3D Von Neumann Neighbourhood stencil are executed on the In-Order core as the input size increases

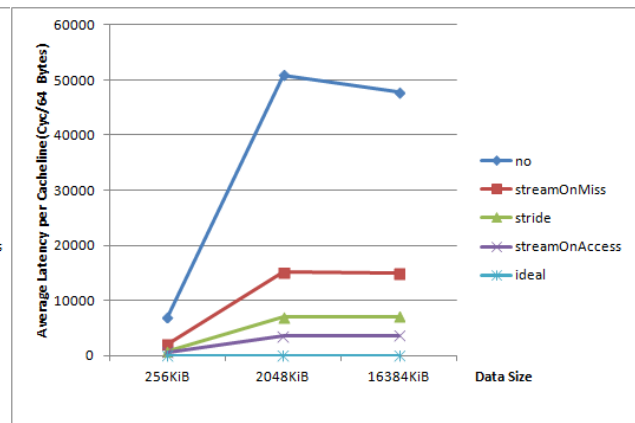


Figure A.56: Latency per cache block of systems featuring the reference prefetchers when 10 iterations of the 3D Von Neumann Neighbourhood stencil are executed on the OOO core as the input size increases

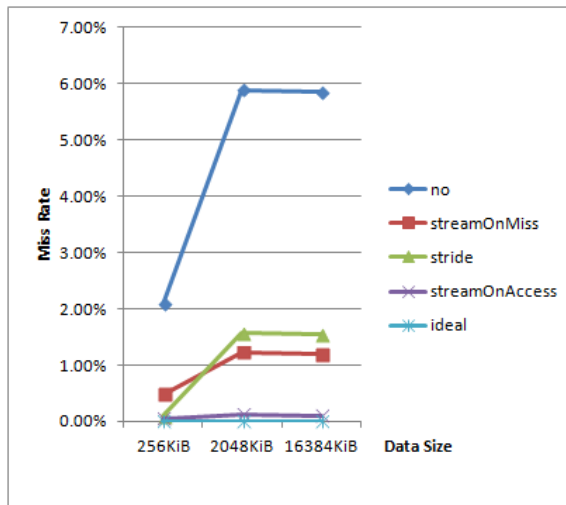


Figure A.57: Cache miss rate of systems featuring the reference prefetchers when 10 iterations of the 3D Von Neumann Neighbourhood stencil are executed on the In-Order core as the input size increases

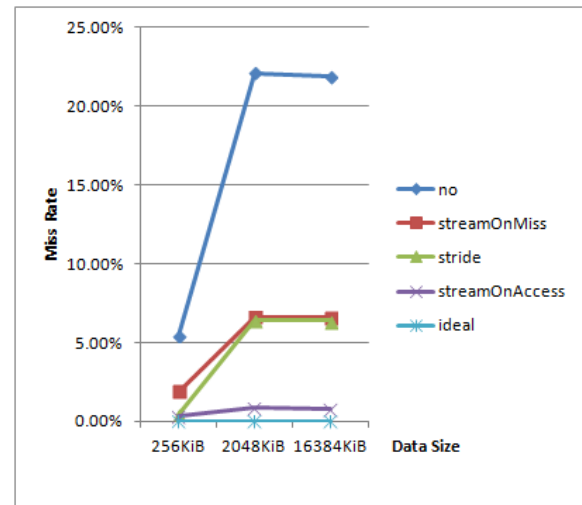


Figure A.58: Cache miss rate of systems featuring the reference prefetchers when 10 iterations of the 3D Von Neumann Neighbourhood stencil are executed on the OOO core as the input size increases

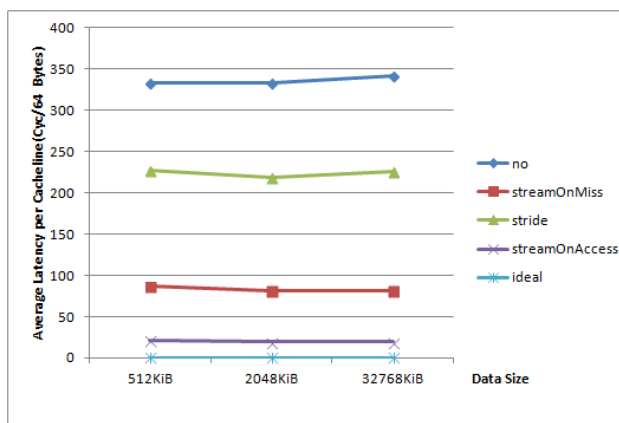


Figure A.59: Latency per cache block of systems featuring the reference prefetchers when the word's first letter capitalisation algorithm is executed on the In-Order core as the input size increases

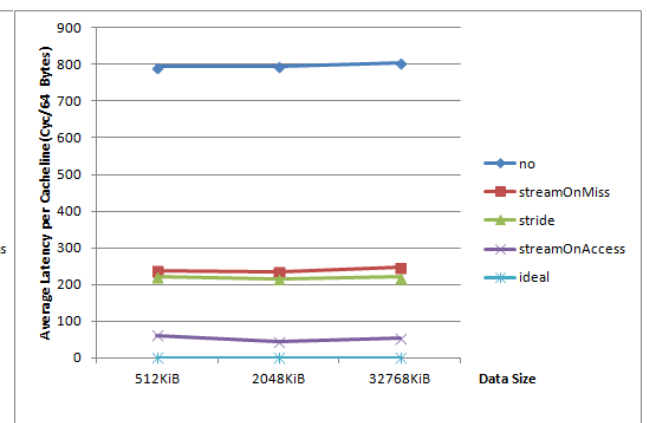


Figure A.60: Latency per cache block of systems featuring the reference prefetchers when the word's first letter capitalisation algorithm is executed on the OOO core as the input size increases

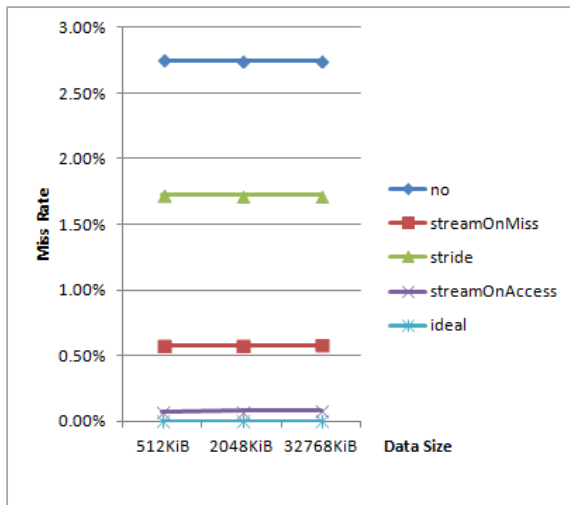


Figure A.61: Cache miss rate of systems featuring the reference prefetchers when the word's first letter capitalisation algorithm is executed on the In-Order core as the input size increases

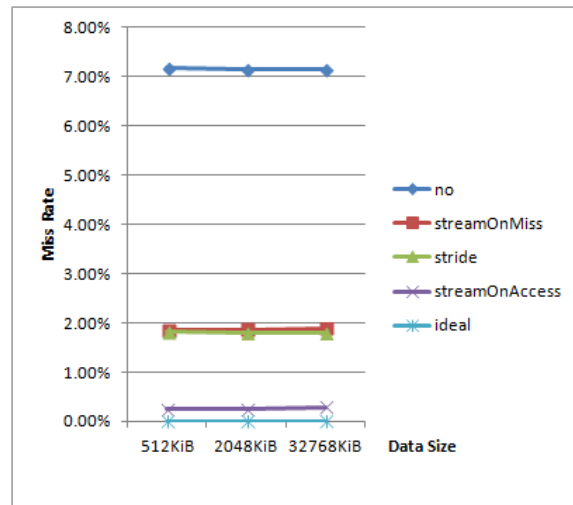


Figure A.62: Cache miss rate of systems featuring the reference prefetchers when the word's first letter capitalisation algorithm is executed on the OOO core as the input size increases

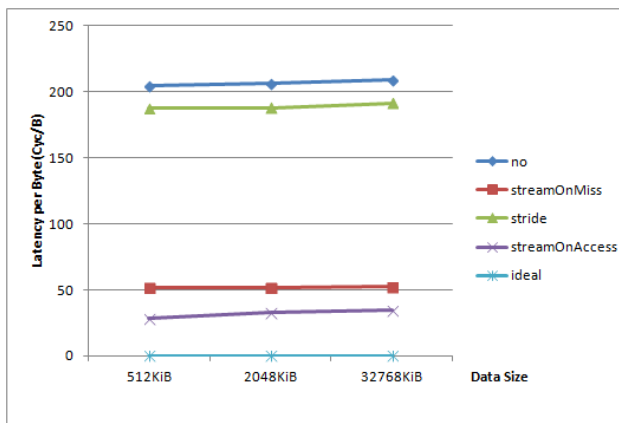


Figure A.63: Latency per cache block of systems featuring the reference prefetchers when the sentence's first letter capitalisation algorithm is executed on the In-Order core as the input size increases

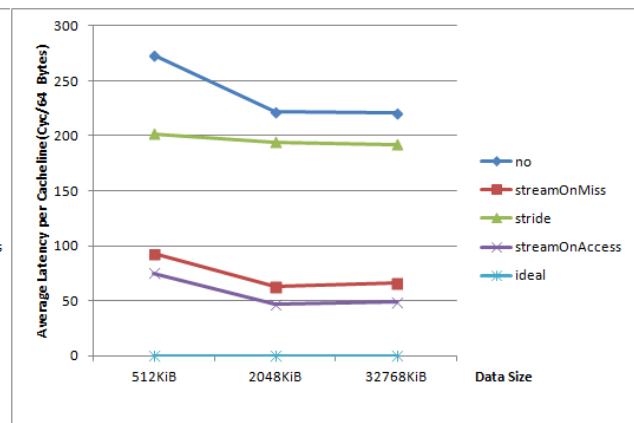


Figure A.64: Latency per cache block of systems featuring the reference prefetchers when the sentence's first letter capitalisation algorithm is executed on the OOO core as the input size increases

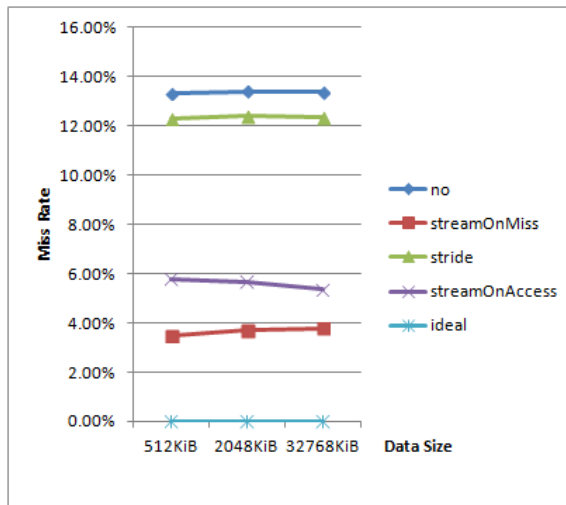


Figure A.65: Cache miss rate of systems featuring the reference prefetchers when the sentence's first letter capitalisation algorithm is executed on the In-Order core as the input size increases

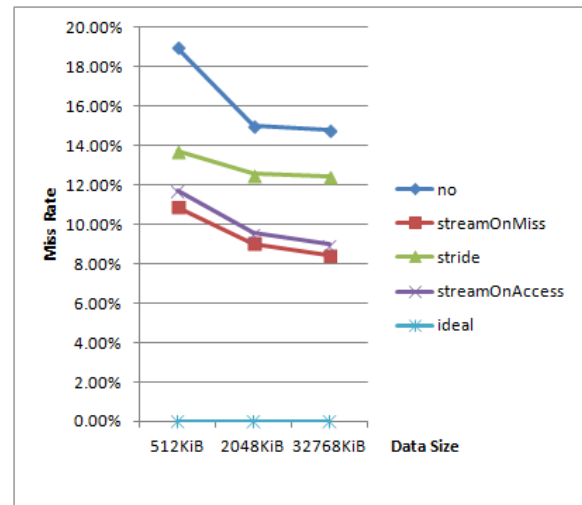


Figure A.66: Cache miss rate of systems featuring the reference prefetchers when the sentence's first letter capitalisation algorithm is executed on the OOO core as the input size increases

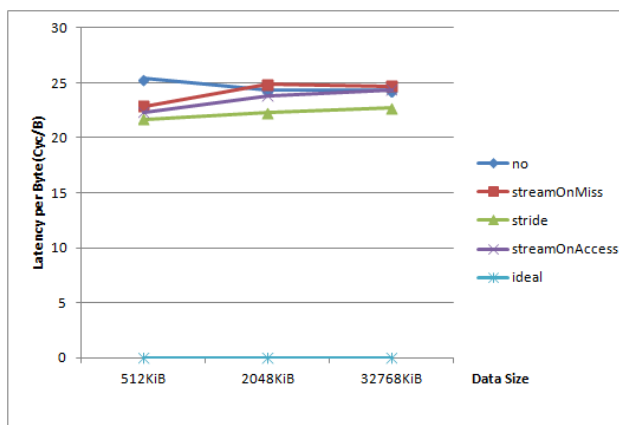


Figure A.67: Latency per cache block of systems featuring the reference prefetchers when the paragraph's first letter capitalisation algorithm is executed on the In-Order core as the input size increases

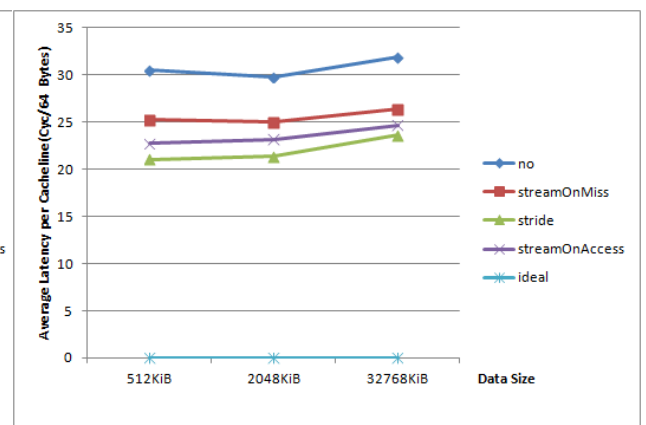


Figure A.68: Latency per cache block of systems featuring the reference prefetchers when the paragraph's first letter capitalisation algorithm is executed on the OOO core as the input size increases

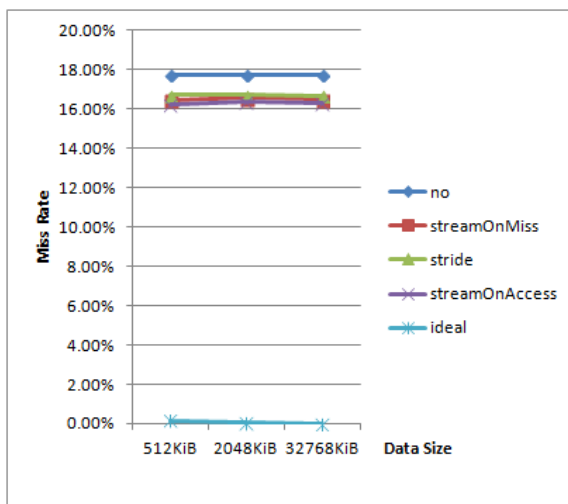


Figure A.69: Cache miss rate of systems featuring the reference prefetchers when the paragraph's first letter capitalisation algorithm is executed on the In-Order core as the input size increases

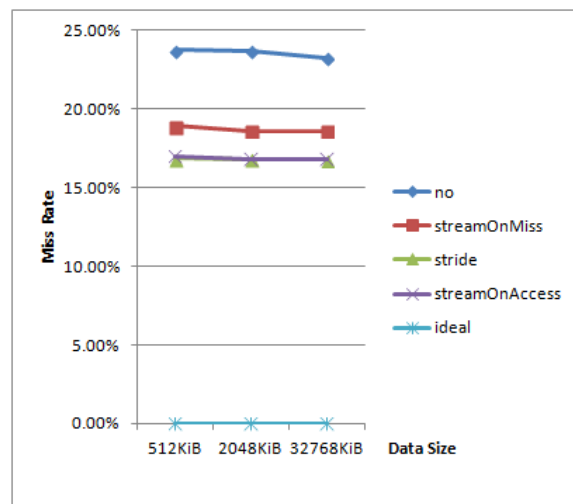


Figure A.70: Cache miss rate of systems featuring the reference prefetchers when the paragraph's first letter capitalisation algorithm is executed on the OOO core as the input size increases

A.2. Arrow Software Prefetching

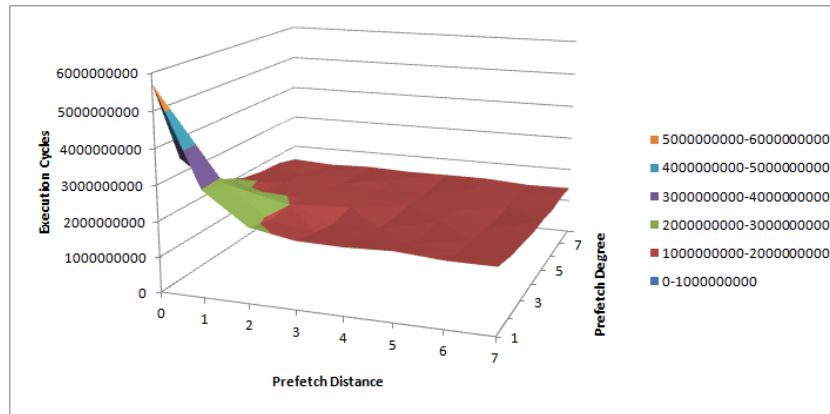


Figure A.71: Execution time of a system featuring the *stream on miss* prefetcher when the synthetic benchmarks with 4+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase

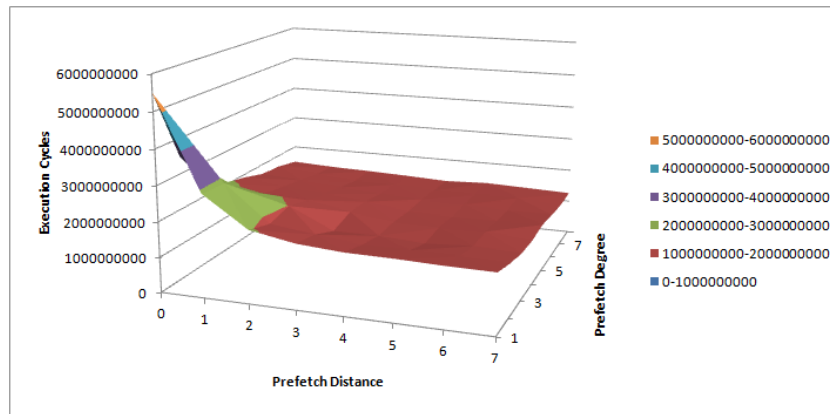


Figure A.72: Execution time of a system featuring the *stream on access* prefetcher when the synthetic benchmarks with 4+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase

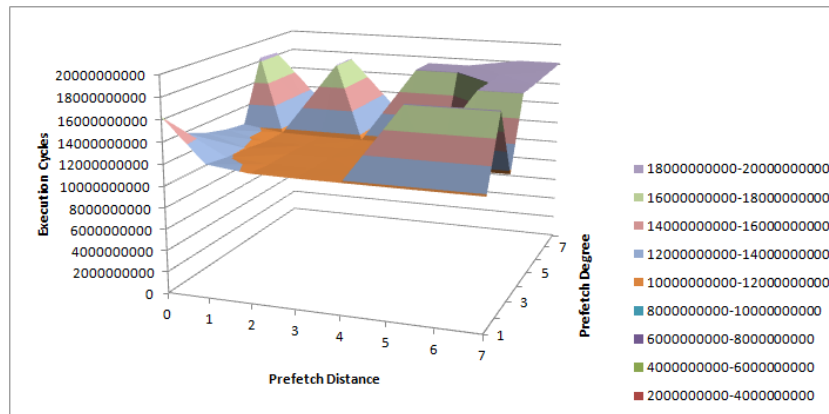


Figure A.73: Execution time of a system featuring the *stride* prefetcher when the synthetic benchmarks with 240+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase

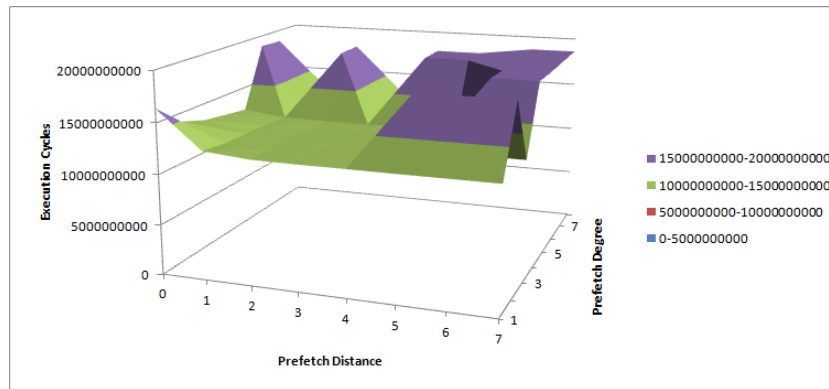


Figure A.74: Execution time of a system featuring the *stream on miss* prefetcher when the synthetic benchmarks with 240+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase

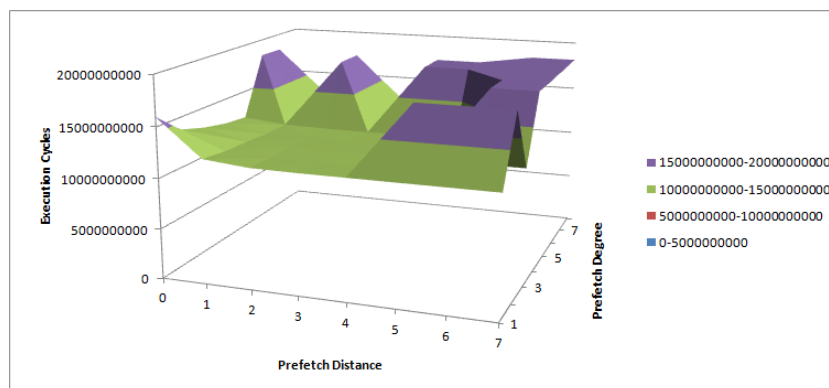


Figure A.75: Execution time of a system featuring the *stream on access* prefetcher when the synthetic benchmarks with 240+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase

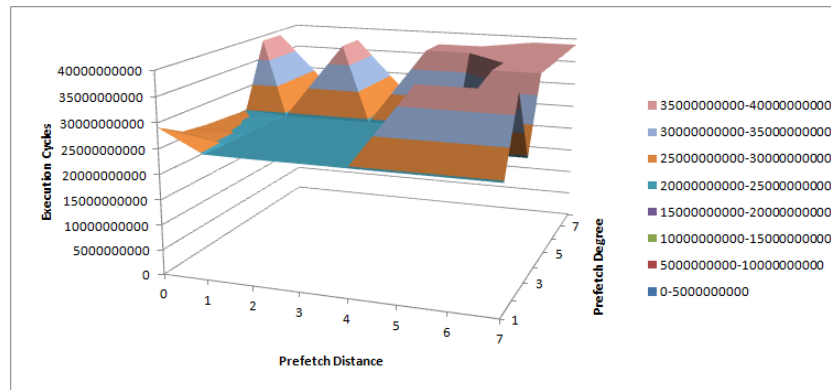


Figure A.76: Execution time of a system featuring the *stream on miss* prefetcher when the synthetic benchmarks with 520+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase

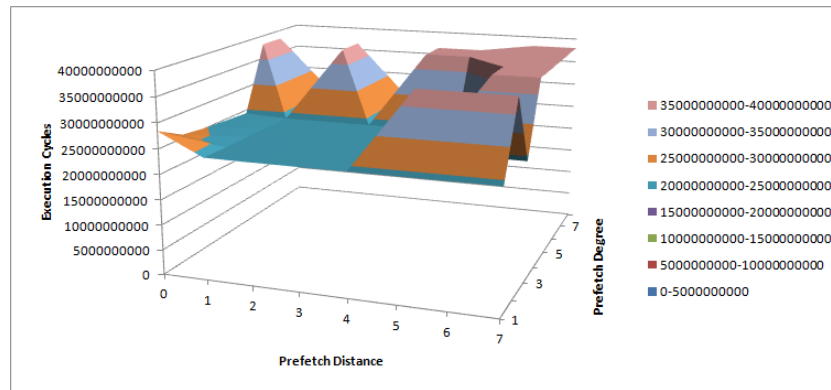


Figure A.77: Execution time of a system featuring the *stream on access* prefetcher when the synthetic benchmarks with 520+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase

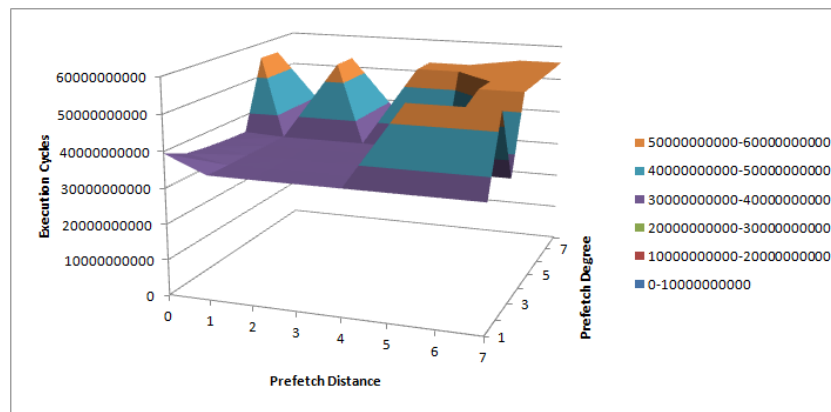


Figure A.78: Execution time of a system featuring the *stride* prefetcher when the synthetic benchmarks with 760+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase

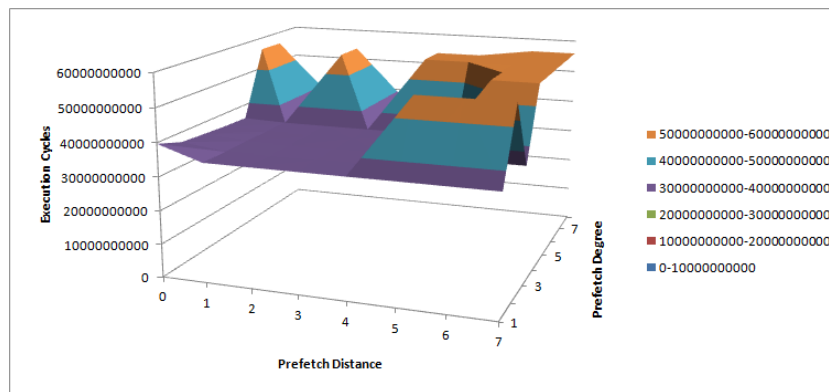


Figure A.79: Execution time of a system featuring the *stream on miss* prefetcher when the synthetic benchmarks with 760+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase

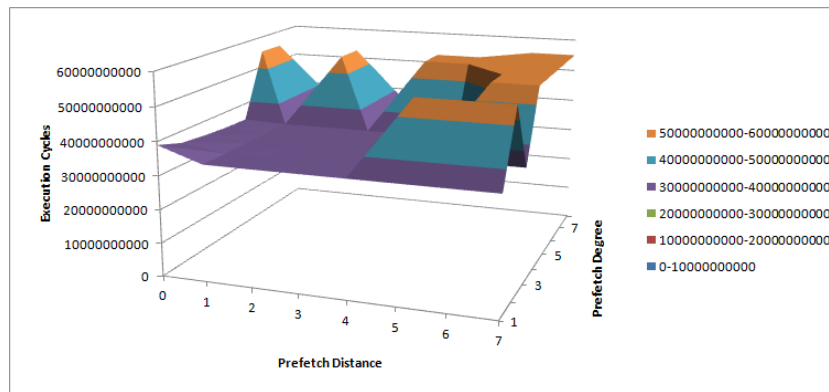


Figure A.80: Execution time of a system featuring the *stream on access* prefetcher when the synthetic benchmarks with 760+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase

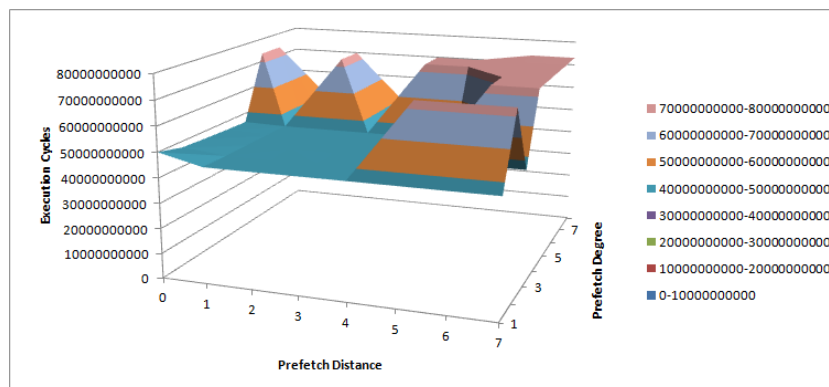


Figure A.81: Execution time of a system featuring the *stream on miss* prefetcher when the synthetic benchmarks with 1000+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase

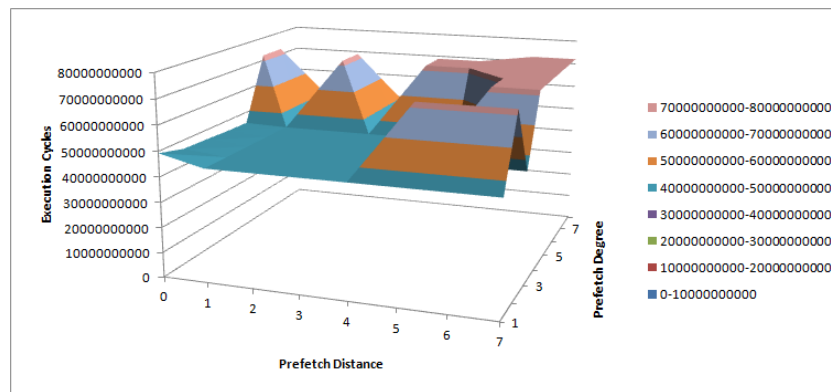


Figure A.82: Execution time of a system featuring the *stream on access* prefetcher when the synthetic benchmarks with 1000+1 operations per loop is executed on the In-Order core as the software prefetching degree and distance increase

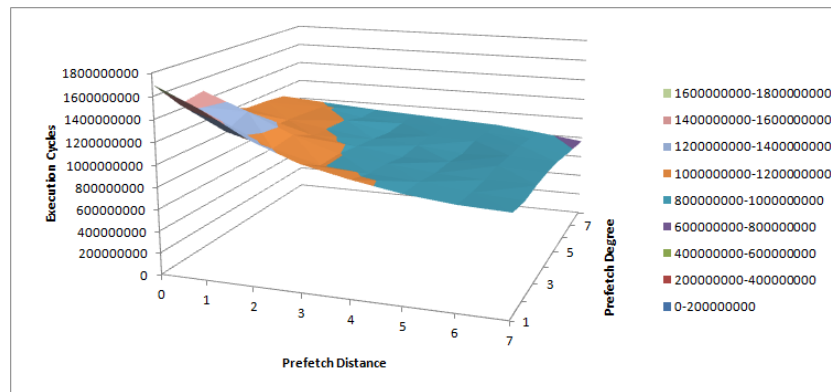


Figure A.83: Execution time of a system featuring the *stream on miss* prefetcher when the synthetic benchmarks with 4+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase

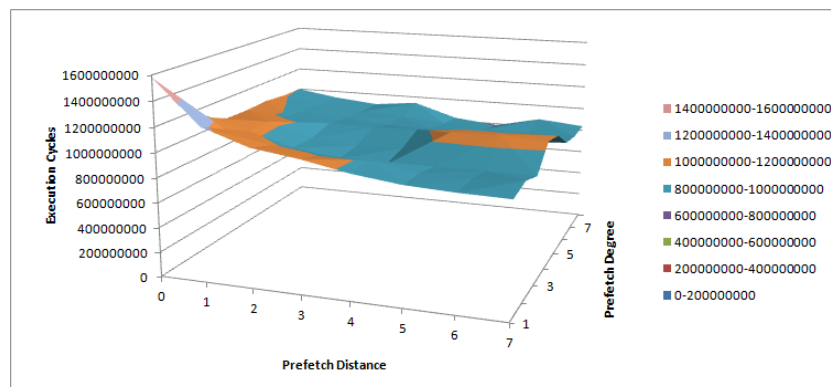


Figure A.84: Execution time of a system featuring the *stream on access* prefetcher when the synthetic benchmarks with 4+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase

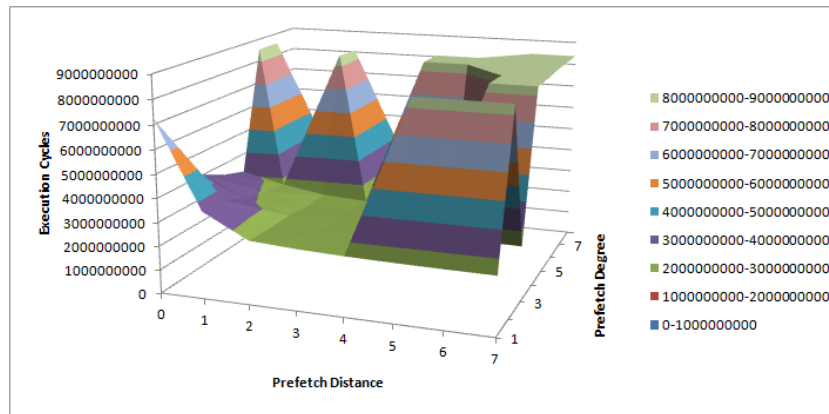


Figure A.85: Execution time of a system featuring the *stride* prefetcher when the synthetic benchmarks with 240+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase

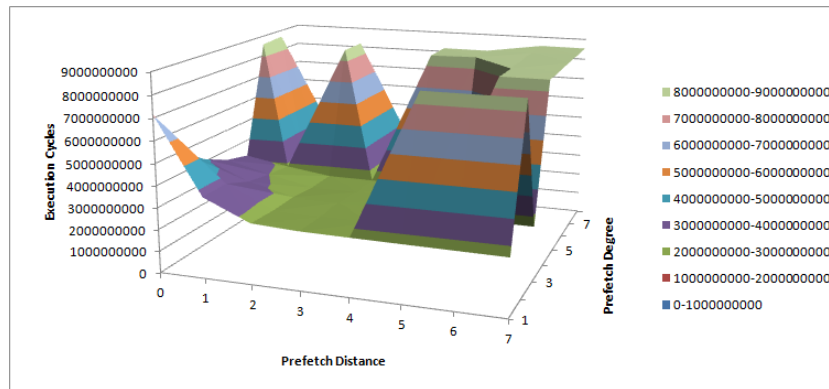


Figure A.86: Execution time of a system featuring the *streamOnMiss* prefetcher when the synthetic benchmarks with 240+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase

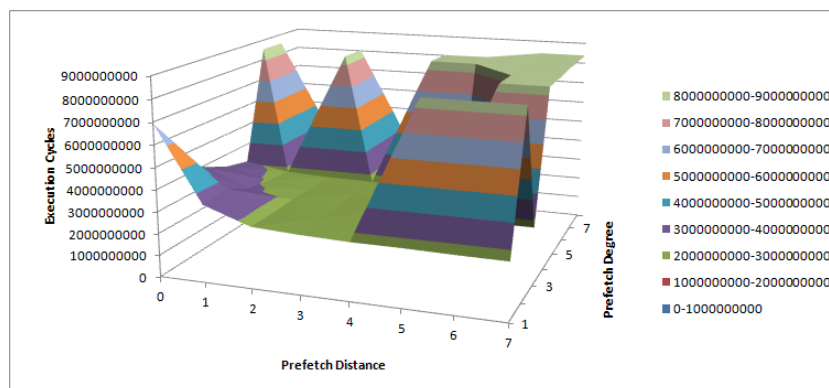


Figure A.87: Execution time of a system featuring the *streamOnAcc* prefetcher when the synthetic benchmarks with 240+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase

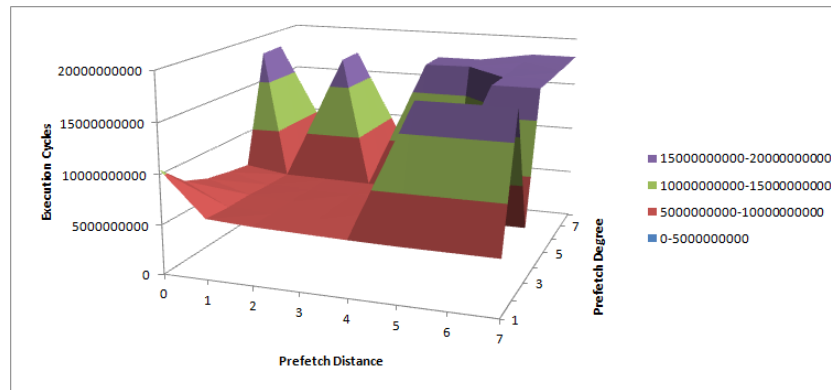


Figure A.88: Execution time of a system featuring the *streamOnMiss* prefetcher when the synthetic benchmarks with 520+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase

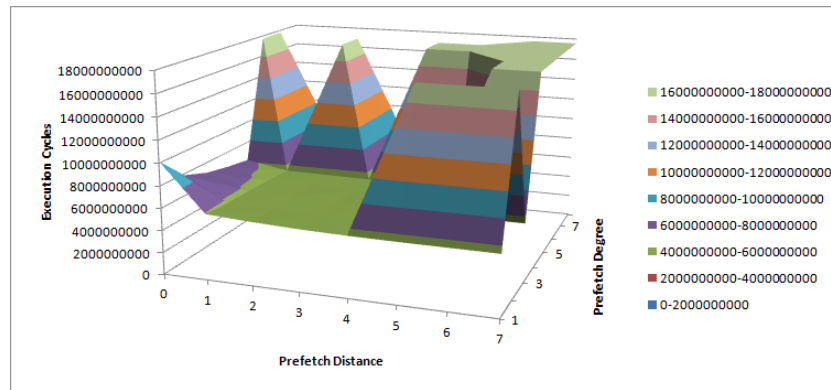


Figure A.89: Execution time of a system featuring the *streamOnAcc* prefetcher when the synthetic benchmarks with 520+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase

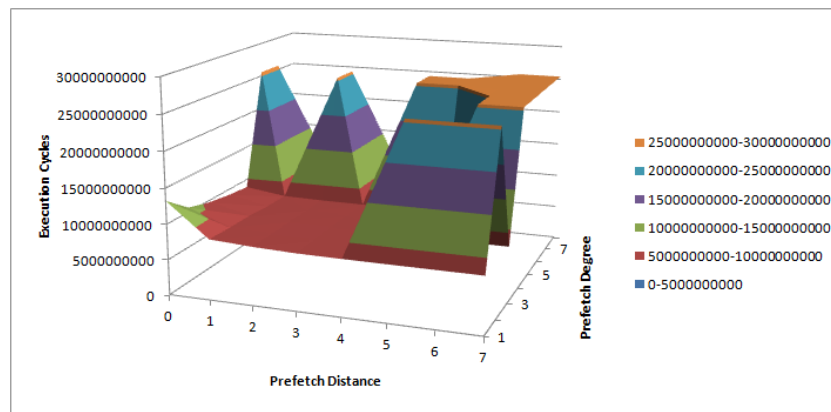


Figure A.90: Execution time of a system featuring the *stride* prefetcher when the synthetic benchmarks with 760+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase

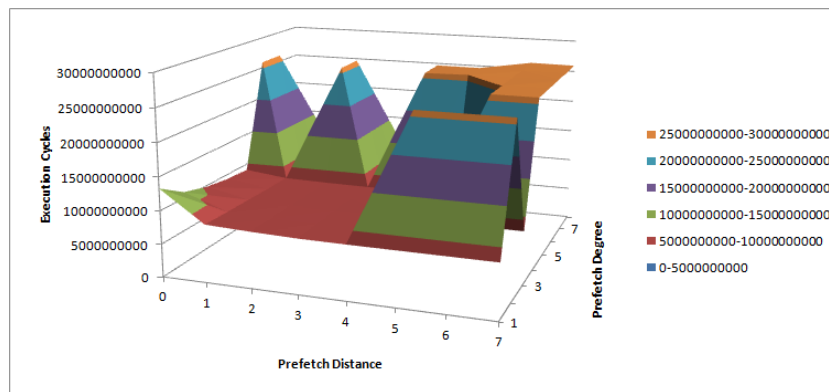


Figure A.91: Execution time of a system featuring the *streamOnMiss* prefetcher when the synthetic benchmarks with 760+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase

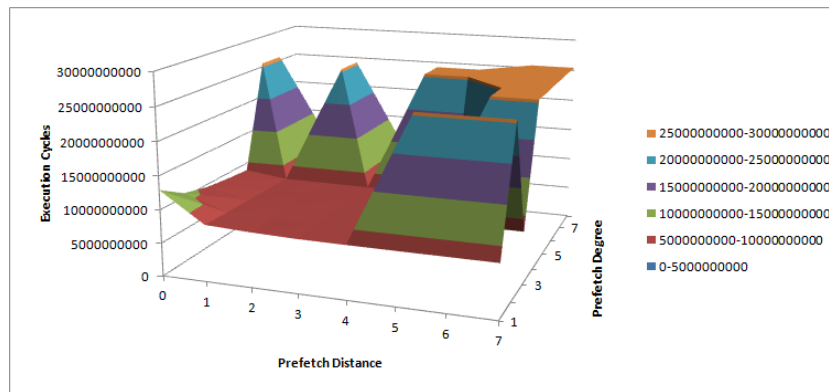


Figure A.92: Execution time of a system featuring the *streamOnAcc* prefetcher when the synthetic benchmarks with 760+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase

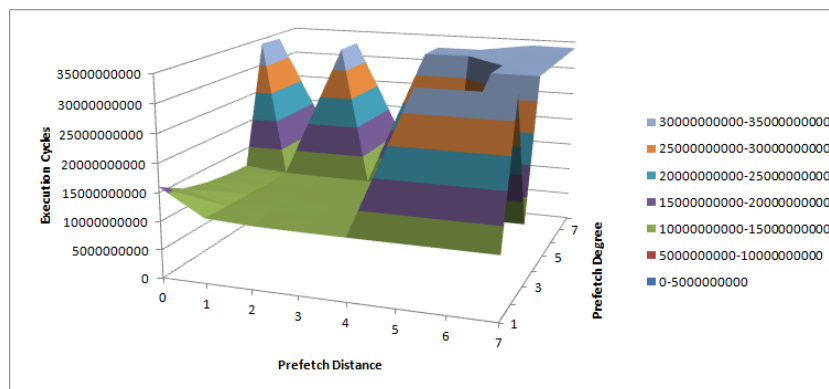


Figure A.93: Execution time of a system featuring the *streamOnMiss* prefetcher when the synthetic benchmarks with 1000+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase

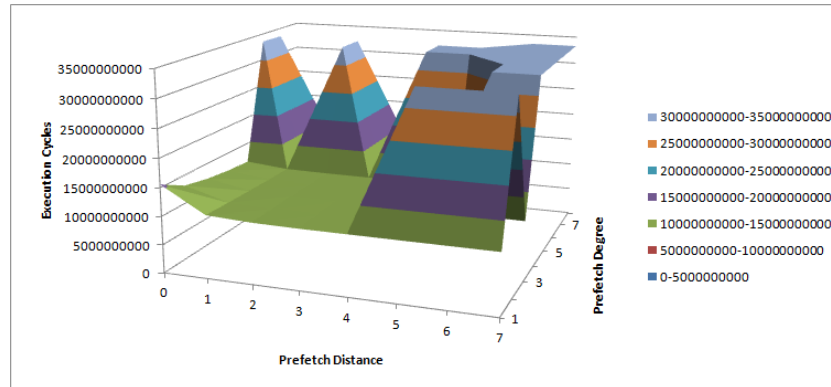


Figure A.94: Execution time of a system featuring the *streamOnAcc* prefetcher when the synthetic benchmarks with 1000+1 operations per loop is executed on the OOO core as the software prefetching degree and distance increase

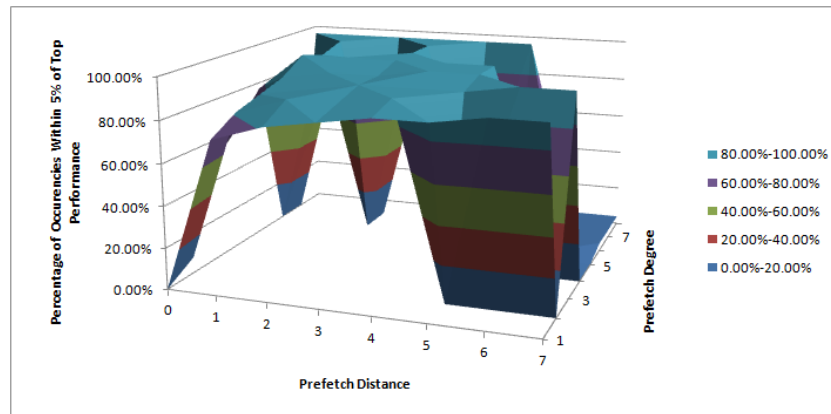


Figure A.95: Percentage of total number of computation intensity cases, each combination of SW prefetching parameters performing within a 5% range of the top performer when executed in the In-Order core of the system featuring a *stream on miss* prefetcher

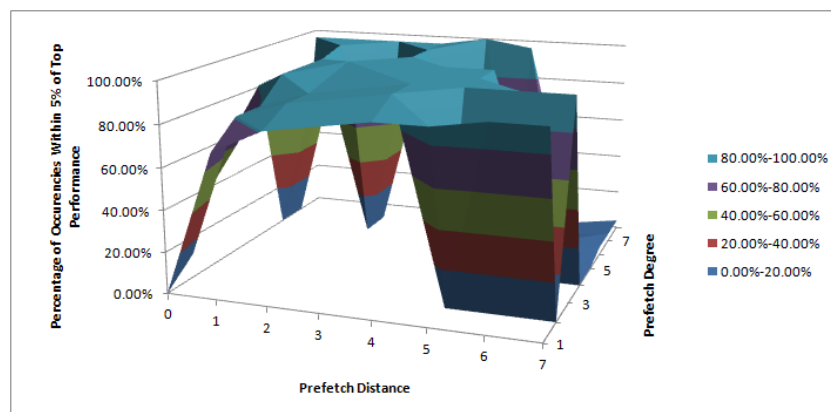


Figure A.96: Percentage of total number of computation intensity cases, each combination of SW prefetching parameters performing within a 5% range of the top performer when executed in the In-Order core of the system featuring a *stream on miss* prefetcher

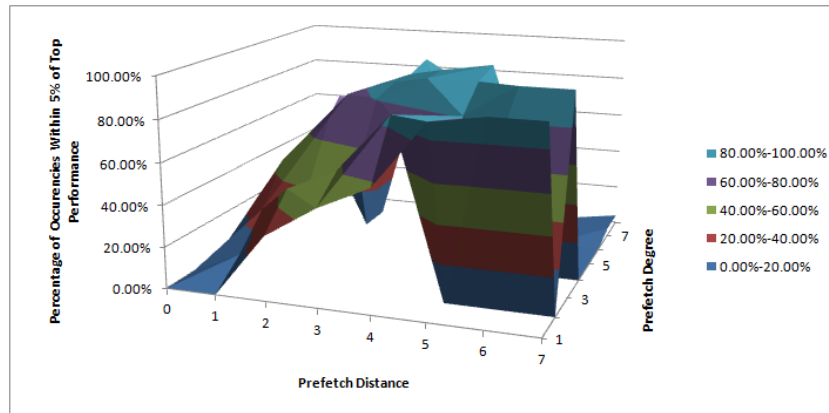


Figure A.97: Percentage of total number of computation intensity cases, each combination of SW prefetching parameters performing within a 5% range of the top performer when executed in the OOO core of the system featuring a *stream on miss* prefetcher

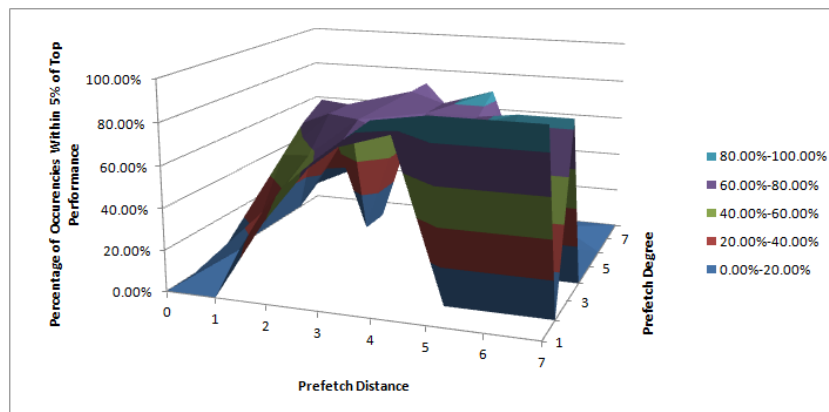


Figure A.98: Percentage of total number of computation intensity cases, each combination of SW prefetching parameters performing within a 5% range of the top performer when executed in the OOO core of the system featuring a *stream on miss* prefetcher

Bibliography

- [1] Apache Arrow Homepage, . URL <https://arrow.apache.org/>.
- [2] Data Prefetch Support, . URL <https://www.gnu.org/software/gcc/projects/prefetch.html>.
- [3] ARM Compiler armasm Reference Guide, 2014. URL http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0802b/PRFM_reg.html.
- [4] Supported Architectures - gem5, April 2017. URL http://www.gem5.org/Supported_Architectures.
- [5] C++ Implementation — Apache Arrow v0.14.2, 2018. URL <https://arrow.apache.org/docs/cpp/index.html>.
- [6] Metadata: Logical types, schemas, data headers — Apache Arrow v0.14.2, 2018. URL <https://arrow.apache.org/docs/format/Metadata.html>.
- [7] Physical memory layout — Apache Arrow v0.14.2, 2018. URL <https://arrow.apache.org/docs/format/Layout.html>.
- [8] CPU cache, June 2019. URL https://en.wikipedia.org/w/index.php?title=CPU_cache&oldid=902581544. Page Version ID: 902581544.
- [9] Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile. Reference Manual 0487E.a, Arm Limited, July 2019. URL https://static.docs.arm.com/ddi0487/ea/DDI0487E_a_armv8_arm.pdf.
- [10] EPYC 7742 - AMD - WikiChip, 2019. URL <https://en.wikichip.org/wiki/amd/epyc/7742>.
- [11] gem5 System Emulation on ARM System Configuration script, August 2019. URL https://github.com/gem5/gem5/blob/master/configs/example/arm/starter_se.py. original-date: 2014-12-02T09:46:00Z.
- [12] gem5 big.LITTLE Memory Hierarchy Configuration script, August 2019. URL <https://github.com/gem5/gem5/blob/master/configs/example/arm/devices.py>. original-date: 2014-12-02T09:46:00Z.
- [13] gem5 Main Memory Configurations, September 2019. URL <https://github.com/gem5/gem5/blob/master/src/mem/DRAMCtrl.py>. original-date: 2014-12-02T09:46:00Z.
- [14] Matrix multiplication, August 2019. URL https://en.wikipedia.org/w/index.php?title=Matrix_multiplication&oldid=912748705. Page Version ID: 912748705.
- [15] Matrix addition, July 2019. URL https://en.wikipedia.org/w/index.php?title=Matrix_addition&oldid=906489363. Page Version ID: 906489363.
- [16] MT40a1g8 DDR4-2400 Datasheet Revision P. Datasheet CCMTD-1725822587-9875, Micron Technology, Inc., April 2019. URL https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/8gb_ddr4_sdram.pdf.
- [17] POWER9 - Microarchitectures - IBM - WikiChip, 2019. URL <https://en.wikichip.org/wiki/ibm/microarchitectures/power9>.
- [18] Xeon W-3275m - Intel - WikiChip, 2019. URL https://en.wikichip.org/wiki/intel/xeon_w/w-3275m.

- [19] Daniel Abadi. DBMS Musings: Apache Arrow vs. Parquet and ORC: Do we really need a third Apache project for columnar data representation?, October 2017. URL <http://dbmsmusings.blogspot.com/2017/10/apache-arrow-vs-parquet-and-orc-do-we.html>.
- [20] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends® in Databases*, 5(3):197–280, 2012. ISSN 1931-7883, 1931-7891. doi: 10.1561/19000000024. URL <http://www.nowpublishers.com/articles/foundations-and-trends-in-databases/DBS-024>.
- [21] Sam Ainsworth. Prefetching for complex memory access patterns. Technical 923, University of Cambridge, Cambridge, July 2018. URL <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-923.pdf>.
- [22] A. Akram and L. Sawalha. A Survey of Computer Architecture Simulation Techniques and Tools. *IEEE Access*, 7:78120–78145, 2019. ISSN 2169-3536. doi: 10.1109/ACCESS.2019.2917698.
- [23] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Pearson, Boston, 3 edition edition, March 2015. ISBN 978-0-13-409266-9.
- [24] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli. Accuracy evaluation of GEM5 simulator system. In *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–7, July 2012. doi: 10.1109/ReCoSoC.2012.6322869.
- [25] K. K. Chang, D. Lee, Z. Chishti, A. R. Alameldeen, C. Wilkerson, Y. Kim, and O. Mutlu. Improving DRAM performance by parallelizing refreshes with accesses. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 356–367, February 2014. doi: 10.1109/HPCA.2014.6835946.
- [26] Kattamuri Ekanadham, Jennifer A. Navarro, Il Park, and Chung-Lung Kevin Shum. Enhancing timeliness of cache prefetching, October 2012. URL <https://patents.google.com/patent/US8285941B2/en>.
- [27] Aaron Eng. Anatomy of a Memory Cell: Understanding Performance in a Distributed Computing Environment, June 2019. URL <https://mapr.com/blog/anatomy-of-a-memory-cell-understanding-performance-in-a-distributed-computing-environment/>.
- [28] John Gantz and David Reinsel. THE DIGITAL UNIVERSE IN 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East. page 16, December 2012.
- [29] Ralph E. Griswold. Drawdown Automata, Part 2: Neighborhoods and State-Transition Rules, August 2004. URL https://www2.cs.arizona.edu/patterns/weaving/webdocs/gre_dda2.pdf.
- [30] Xiaoyu Guo, Jingfu Kang, and Jinsong Zhu. 3d Cellular Automata–Based Numerical Simulation of Atmospheric Corrosion Process on Weathering Steel. *Journal of Materials in Civil Engineering*, 30(11):04018296, November 2018. ISSN 0899-1561, 1943-5533. doi: 10.1061/(ASCE)MT.1943-5533.0002502. URL <http://ascelibrary.org/doi/10.1061/%28ASCE%29MT.1943-5533.0002502>.
- [31] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, and N. Paver. Sources of error in full-system simulation. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 13–22, March 2014. doi: 10.1109/ISPASS.2014.6844457.
- [32] H. Hu, Y. Wen, T. Chua, and X. Li. Toward Scalable Systems for Big Data Analytics: A Technology Tutorial. *IEEE Access*, 2:652–687, 2014. ISSN 2169-3536. doi: 10.1109/ACCESS.2014.2332453.

- [33] Nicola Jones. How to stop data centres from gobbling up the world's electricity. *Nature*, 561:163, September 2018. doi: 10.1038/d41586-018-06610-y. URL <http://www.nature.com/articles/d41586-018-06610-y>.
- [34] J. Kalyanasundaram and Y. Simmhan. ARM Wrestling with Big Data: A Study of Commodity ARM64 Server for Big Data Workloads. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 203–212, December 2017. doi: 10.1109/HiPC.2017.00032.
- [35] N. B. Lakshminarayana and H. Kim. Spare register aware prefetching for graph algorithms on GPUs. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 614–625, February 2014. doi: 10.1109/HPCA.2014.6835970.
- [36] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. When Prefetching Works, When It Doesn'T, and Why. *ACM Trans. Archit. Code Optim.*, 9(1):2:1–2:29, March 2012. ISSN 1544-3566. doi: 10.1145/2133382.2133384. URL <http://doi.acm.org/10.1145/2133382.2133384>.
- [37] Dumitrel Loghin, Bogdan Marius Tudor, Hao Zhang, Beng Chin Ooi, and Yong Meng Teo. A Performance Study of Big Data on Small Nodes. *PVLDB*, 8:762–773, 2015. doi: 10.14778/2752939.2752945.
- [38] Steve Lohr. The Origins of 'Big Data': An Etymological Detective Story, February 2013. URL <https://bits.blogs.nytimes.com/2013/02/01/the-origins-of-big-data-an-etymological-detective-story/>.
- [39] M. Malik, S. Rafatirah, A. Sasan, and H. Homayoun. System and architecture level characterization of big data applications on big and little core server architectures. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 85–94, October 2015. doi: 10.1109/BigData.2015.7363745.
- [40] Martyn. Coding for NEON - Part 3 matrix multiplication, September 2013. URL <https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/coding-for-neon---part-3-matrix-multiplication>.
- [41] Sally A. McKee. Reflections on the memory wall. In *Proceedings of the first conference on computing frontiers on Computing frontiers - CF'04*, page 162, Ischia, Italy, 2004. ACM Press. ISBN 978-1-58113-741-5. doi: 10.1145/977091.977115. URL <http://portal.acm.org/citation.cfm?doid=977091.977115>.
- [42] Sparsh Mittal. A Survey of Recent Prefetching Techniques for Processor Caches. *ACM Computing Surveys*, 49, August 2016. doi: 10.1145/2907071.
- [43] Onur Mutlu. Lecture 2: Fundamentals, Memory Hierarchy, Caches, September 2017.
- [44] Jacques Nadeau. Apache Arrow: In Theory, In Practice, 2017. URL <https://www.dremio.com/webinars/apache-arrow-in-theory-practice/>.
- [45] Mais Haj Qasem, Alaa Abu Sarhan, Raneem Qaddoura, and Basel A. Mahafzah. Matrix multiplication of big data using MapReduce: A review. In *2017 2nd International Conference on the Applications of Information Technology in Developing Renewable Energy Processes & Systems (IT-DREPS)*, pages 1–6, Amman, December 2017. IEEE. ISBN 978-1-5386-1984-1 978-1-5386-1986-5. doi: 10.1109/IT-DREPS.2017.8277807. URL <http://ieeexplore.ieee.org/document/8277807/>.
- [46] D. Richins, T. Ahmed, R. Clapp, and V. Janapa Reddi. Amdahl's Law in Big Data Analytics: Alive and Kicking in TPCx-BB (BigBench). In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 630–642, February 2018. doi: 10.1109/HPCA.2018.00060.
- [47] Ripan Sarkar. Supercharging Analytics with Apache Arrow. *LinkedIn Pulse*, September 2018. URL <https://www.linkedin.com/pulse/supercharging-analytics-apache-arrow-ripan-sarkar/>.

- [48] Andreas Schäfer and Dietmar Fey. High Performance Stencil Code Algorithms for GPGPUs. *Procedia Computer Science*, 4:2027–2036, January 2011. ISSN 1877-0509. doi: 10.1016/j.procs.2011.04.221. URL <http://www.sciencedirect.com/science/article/pii/S1877050911002791>.
- [49] Gabriel Torres. How The Cache Memory Works - Page 3 of 10, September 2007. URL <https://www.hardwaresecrets.com/how-the-cache-memory-works/>.
- [50] Steven P. Vanderwiel and David J. Lilja. Data Prefetch Mechanisms. *ACM Comput. Surv.*, 32(2): 174–199, June 2000. ISSN 0360-0300. doi: 10.1145/358923.358939. URL <http://doi.acm.org/10.1145/358923.358939>.
- [51] M. Walker, S. Bischoff, S. Diestelhorst, G. Merrett, and B. Al-Hashimi. Hardware-Validated CPU Performance and Energy Modelling. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 44–53, April 2018. doi: 10.1109/ISPASS.2018.00013.
- [52] Sebastian Werner, Jorn Kuhlenkamp, Markus Klems, Johannes Muller, and Stefan Tai. Serverless Big Data Processing using Matrix Multiplication as Example. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 358–365, Seattle, WA, USA, December 2018. IEEE. ISBN 978-1-5386-5035-6. doi: 10.1109/BigData.2018.8622362. URL <https://ieeexplore.ieee.org/document/8622362/>.
- [53] M. V. Wilkes. Slave Memories and Dynamic Storage Allocation. *IEEE Transactions on Electronic Computers*, EC-14(2):270–271, April 1965. ISSN 0367-7508. doi: 10.1109/PGEC.1965.264263.
- [54] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. IMP: indirect memory prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture - MICRO-48*, pages 178–190, Waikiki, Hawaii, 2015. ACM Press. ISBN 978-1-4503-4034-2. doi: 10.1145/2830772.2830807. URL <http://dl.acm.org/citation.cfm?doid=2830772.2830807>.