EVALUATION OF CACHE INCLUSION POLICIES IN CACHE MANAGEMENT

A Thesis

by

LUNA BACKES DRAULT

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Daniel Jiménez
Committee Members,	Paul Gratz
	Dilma Da Silva
Head of Department,	Dilma Da Silva

August 2017

Major Subject: Computer Engineering

ABSTRACT

Processor speed has been increasing at a higher rate than the speed of memories over the last years. Caches were designed to mitigate this gap and, ever since, several cache management techniques have been designed to further improve performance.

Most techniques have been designed and evaluated on non-inclusive caches even though many modern processors implement either inclusive or exclusive policies. Exclusive caches benefit from a larger effective capacity, so they might become more popular when the number of cores per last-level cache increases.

This thesis aims to demonstrate that the best cache management techniques for exclusive caches do not necessarily have to be the same as for non-inclusive or inclusive caches. To assess this statement we evaluated several cache management techniques with different inclusion policies, number of cores and cache sizes.

We found that the configurations for inclusive and non-inclusive policies usually performed similarly, but for exclusive caches the best configurations were indeed different. Prefetchers impacted performance more than replacement policies, and determined which configurations were the best ones. Also, exclusive caches showed a higher speedup on multi-core.

The least recently used (LRU) replacement policy is among the best policies for any prefetcher combination in exclusive caches but is the one used as a baseline in most cache replacement policy research. Therefore, we conclude that the results in this thesis motivate further research on prefetchers and replacement policies targeted to exclusive caches.

DEDICATION

To Alex, for his love, patience and unconditional support.

ACKNOWLEDGMENTS

First of all, I want to thank my advisor, Daniel Jiménez, for giving me the chance of joining his lab and believing in me to be his first master's student. I appreciate the support and freedom he gave me to explore a topic I was very interested in. I also want to thank the committee members Paul V. Gratz and Dilma Da Silva for being available on their busy schedules and asking me good questions and giving feedback on this work.

I also want to thank many people that helped me directly and indirectly such as labmates, friends and family. Elvira, who I met a few years ago in Italy and I ended up in her group across the world. It was nice having a known and friendly face in the university. She made me feel welcomed and helped me during this two years, we will miss you in the lab. Sangam for her understanding and help during stressful days. And to all my other labmates for their feedback and talks, Ivan, Elba and Samira.

I also thank my previous advisor, Filippo Mantovani, for still giving me advice despite the distance and time difference. Including musical advice to help me write the last part of this thesis.

To my parents, Maria Teresa and Paulo, for supporting my decision to move in to a different country and being able to visit and attend the defense of this thesis, together with Alejandro and Emilia. I really appreciated their support and the delicious food they brought. And to all the rest of my family that are scattered around the world but still care about and support me. My 24 cousins: Ceci, Laura, Maria Emilia, July, Josefina, Tomy, Martin, Guada, Fran, Majo, Juan, Edu, Luciana, Nacho, Santy, Sebi, Juampy, Lucas, Agustín, Betty, Katia, Yaundé, Kayawe, Kitwe, and their wives/husbands and children; for cheering me up from the distance and

caring about me. My 16 aunts and uncles: *Mary*, *Lucy*, Titi, Cecilia, Miguel, Monica, Eduardo, Alicia, Emilio, José Maria, Rosi, Tito, Marcela, Ghi, Julius and Denise; for their love. My grandmother Lilia, for always sharing her knowledge and wisdom from her 101 years. My grandfathers and my vó Erica, that live in my memory. Most of them are far away but I remember them every day.

To many friends in different stages of my life, such as Alba, Laura, Ainoa, Francesc, Aleix, Khaoula, Rio, Jing, Dani, Lis, Jane, Mathi and Andrea. To follow Alba's latest success in SoundSix slightly delayed the progress of this thesis but surely made it more special.

Most special thanks to Alex, for surviving through two of my degrees and agreed to the third one. This would not have been possible without his love, support, cheers and confidence in me. The third time's the charm.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Professor Daniel A. Jiménez and Professor Dilma Da Silva of the Department of Computer Science and Engineering and Professor Paul V. Gratz of the Department of Electrical and Computer Engineering.

The traces and the simulator were provided by Jinchun Kim of the Department of Electrical and Computer Engineering. He also contributed on this thesis by upgrading the simulator and adding more features to experiment.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

This research was supported in part by a grant from the National Science Foundation, NSF CCF-1216604/1332598.

Portions of this research were conducted with the advanced computing resources provided by Texas A&M High Performance Research Computing.

NOMENCLATURE

BRRIP	bimodal RRIP
CPU	central processing unit
CSV	comma separated value
DAAMPM	DRAM-aware access map pattern matching
DRAM	dynamic RAM
DRRIP	dynamic RRIP
EAF	evicted address filter
eDRAM	embedded DRAM
FP	frequency priority
GIPPR	genetic insertion and promotion for pseudo-LRU replacement
HP	hit promotion
IPC	instructions per cycle
IPV	insertion and promotion vector
KPC	kill the program counter
KPC-P	KPC prefetching algorithm
KPC-R	KPC replacement algorithm
LLC	last-level cache

LRU	least recently used
MDPP	minimal disturbance placement and promotion
MLP	memory-level parallelism
MPKI	misses per kiloinstruction
MRU	most recently used
MSHR	miss status holding registers
OS	operating system
PC	program counter
PLRU	pseudo least recently used
RAM	random-access memory
RRIP	re-reference interval prediction
RRPV	re-reference prediction value
SDBP	sampling-based dead block prediction
SHiP	signature-based hit predictor
SMP	symmetric multiprocessor
SPEC	standard performance evaluation corporation
SRAM	static RAM
SRRIP	static RRIP
SSH	secure shell

TACO	Texas Architecture and Compiler Optimization
TLB	translation lookaside buffer

TABLE OF CONTENTS

Page

ABSTRACT ii

DEDICATION iii

ACKNOWLEDGMENTS iv

CONTRIBUTORS AND FUNDING SOURCES vi

NOMENCLATURE vii

TABLE OF CONTENTS x

LIST OF FIGURES xiv

LIST OF TABLES xvi

1. INTRODUCTION 1

    1.1 My Project 2
        1.1.1 Objectives 2
        1.1.2 Contributions 3
    1.2 Document Structure 3

2. BACKGROUND 4

    2.1 Memory Hierarchy 5
        2.1.1 Miss Types 8
        2.1.2 Load Flow 9
        2.1.3 Write Policies 11
        2.1.4 Cache Coherence 12
            2.1.4.1 MSI Protocol 14
            2.1.4.2 MSI-Like Protocols 15
    2.2 Inclusion Policies 16
        2.2.1 Inclusive 17
        2.2.2 Exclusive 19
        2.2.3 Non-Inclusive 21
        2.2.4 Summary of Inclusions 22

2.3 Replacement Policies . . . 23
2.3.1 LRU . . . 25
2.4 Prefetchers . . . 26
2.4.1 Next-Line . . . 28
2.4.2 Instruction Pointer-Based Stride . . . 28
2.4.3 Best-Offset . . . 29
2.4.4 DRAM-Aware Access Map Pattern Matching . . . 30
2.4.5 KPC . . . 31

3. RELATED WORK . . . 33

3.1 Re-Reference Interval Prediction . . . 33
3.2 Sampling Dead Block Prediction . . . 35
3.3 Signature-Based Hit Predictor . . . 37
3.4 Minimal Disturbance Placement and Promotion . . . 37
3.5 Evicted Address Filter . . . 39
3.6 Perceptron Learning for Reuse Prediction . . . 40
3.7 Hawkeye . . . 41
3.8 Bypass and Insertion . . . 42
3.9 Hierarchy-Awareness and Bypass . . . 43

4. METHODOLOGY . . . 47

4.1 Experimental Setup . . . 47
4.1.1 Host Machine . . . 47
4.1.2 Benchmarks . . . 47
4.1.3 Simulator . . . 50
4.2 Evaluation . . . 52
4.2.1 Configurations . . . 52
4.2.2 Performance Measurement . . . 53

5. IMPLEMENTATION . . . 55

5.1 ChampSim Code . . . 55
5.1.1 Cache Operation . . . 55
5.1.2 Non-Inclusive Implementation . . . 56
5.1.3 Statistics . . . 58
5.2 Modifying ChampSim . . . 58
5.2.1 Inclusive Implementation . . . 58
5.2.2 Exclusive Implementation . . . 62
5.3 Scripts . . . 65
5.3.1 Execute . . . 65
5.3.2 Plots . . . 67

6. RESULTS 69

6.1 Single-Core Results 69
6.1.1 Inclusion Results 71
6.1.2 Prefetcher Impact 71
6.1.3 Replacement Policy Impact 72
6.2 Multi-Core Results 72
6.2.1 Inclusion Results 74
6.2.2 Prefetcher Impact 75
6.2.3 Replacement Policy Impact 75
6.3 Size Sensitivity 76
6.3.1 Inclusion Results 78
6.3.2 Prefetcher Impact 78
6.3.3 Replacement Policy Impact 79
6.4 Discussion 79

7. SUMMARY 81

7.1 Conclusions 81
7.2 Future Work 83

REFERENCES 85

APPENDIX A. SINGLE-CORE RESULTS 93

A.1 Astar 94
A.2 Bwaves 95
A.3 Bzip2 96
A.4 CactusADM 97
A.5 Data_Caching 98
A.6 Gcc 99
A.7 GemsFDTD 100
A.8 Graph_Analytics 101
A.9 Gromacs 102
A.10 Lbm 103
A.11 Leslie3d 104
A.12 Libquantum 105
A.13 Mcf 106
A.14 Milc 107
A.15 Mlpack_Cf 108
A.16 Omnetpp 109
A.17 Sat_Solver 110
A.18 Soplex 111
A.19 Sphinx3 112

A.20 Wrf . . . 113
A.21 Xalancbmk . . . 114
A.22 Zeusmp . . . 115

APPENDIX B. MULTI-CORE RESULTS . . . 116

B.1 Astar-Bzip2-Sphinx3-Data_Caching . . . 117
B.2 Astar-Bzip2-Sphinx3-Data_Caching . . . 118
B.3 Astar-Leslie3d-Soplex-Zeusmp . . . 119
B.4 Bwaves-Milc-Data_Caching-Sat_Solver . . . 120
B.5 Bzip2-Omnetpp-Data_Caching-Graph_Analytics . . . 121
B.6 CactusADM-GemsFDTD-Lbm-Leslie3d . . . 122
B.7 CactusADM-Gromacs-Lbm-Milc . . . 123
B.8 CactusADM-Gromacs-Sphinx3-Sat_Solver . . . 124
B.9 Mcf-Milc-Omnetpp-Sat_Solver . . . 125

APPENDIX C. SINGLE-CORE SIZE SENSITIVITY RESULTS . . . 126

C.1 Astar . . . 127
C.2 Bwaves . . . 128
C.3 Bzip2 . . . 129
C.4 CactusADM . . . 130
C.5 Data_Caching . . . 131
C.6 Gcc . . . 132
C.7 GemsFTD . . . 133
C.8 Graph_Analytics . . . 134
C.9 Gromacs . . . 135
C.10 Lbm . . . 136
C.11 Leslie3d . . . 137
C.12 Libquantum . . . 138
C.13 Mcf . . . 139
C.14 Milc . . . 140
C.15 Mlpack_Cf . . . 141
C.16 Omnetpp . . . 142
C.17 Sat_Solver . . . 143
C.18 Soplex . . . 144
C.19 Sphinx3 . . . 145
C.20 Wrf . . . 146
C.21 Xalancbmk . . . 147
C.22 Zeusmp . . . 148

LIST OF FIGURES

FIGURE Page

2.1 Diagram of the memory hierarchy in a modern processor. The higher the capacity of the cache, the higher the latency to access a block in that cache. In a multi-core, there are replications of the "core" part inside the "SoC" part. 7

2.2 Cache coherence problem example. 13

2.3 Memory hierarchy in a multi-core processor. 13

2.4 Simplified state transition diagram of the MSI cache coherence protocol. 15

2.5 Diagram to show where the data is in an a) inclusive, b) non-inclusive, and c) exclusive cache. The intersection is data duplication. 17

5.1 Main functions on ChampSim to operate the cache. 55

5.2 Inclusive cache high-level code on top of the non-inclusive implementation. 61

5.3 Diagram with all the possible cases on an LLC miss in an inclusive cache on a single core simulation. 63

5.4 Exclusive cache high-level code on top of the non-inclusive implementation. 66

6.1 Geomean speedups to compare different configurations of L1 and L2 prefetchers, replacement policies and cache inclusions. The configurations compared are: L1 prefetcher, L2 prefetcher, replacement policy and cache inclusion. The Y-axis shows the speedup over the baseline configuration: no prefetchers, LRU replacement policy and a non-inclusive cache. The X-axis shows the different cache configurations, in order of: L1 prefetcher (l1p), L2 prefetcher (l2p) and replacement policy (repl). 70

6.2 Geomean speedups to compare different configurations of L1 and L2 prefetchers, replacement policies and cache inclusions. The configurations compared are: L1 prefetcher, L2 prefetcher, replacement policy and cache inclusion. The Y-axis shows the speedup over the baseline configuration: no prefetchers, LRU replacement policy and a non-inclusive cache. The X-axis shows the different cache configurations, in order of: L1 prefetcher (l1p), L2 prefetcher (l2p) and replacement policy (repl). 73

6.3 Geomean speedups to compare size sensitivity with different configurations of L1 and L2 prefetchers, replacement policies and cache inclusions on a large cache configuration. The size of each cache level is: 64KB L1, 512KB L2 and 2MB LLC. The configurations compared are: L1 prefetcher, L2 prefetcher, replacement policy and cache inclusion. The Y-axis shows speedup over the baseline configuration: no prefetchers, LRU replacement policy and a non-inclusive cache. The X-axis shows the different cache configurations, in order of: L1 prefetcher (l1p), L2 prefetcher (l2p) and replacement policy (repl). . . . 77

LIST OF TABLES

TABLE Page

2.1 Inclusion policy properties. 23

3.1 State-of-the-art replacement policies and the inclusion policy. 34

3.2 L2 cache block classification in the CHAR algorithm. 46

4.1 SPEC CPU2006 memory intensive benchmarks. 49

4.2 CloudSuite benchmarks. 49

4.3 Multiprogrammed workloads mixes for simulating 4 cores. 49

4.4 Simulator configuration. 50

5.1 Model of an non-inclusive two-level cache. 57

5.2 Model of an inclusive two-level cache. 59

5.3 Model of an exclusive two-level cache. 64

1. INTRODUCTION

The size of transistors has kept decreasing thanks to technology improvements and therefore increasing the number of transistors per chip as stated by Moore's Law [1, 2]. This led to add more complexity to the compute core in the chip to improve performance (e.g. out-of-order execution) and include multiple levels of cache memory to reduce the gap with memory latency by exploiting data locality.

Due to power density constraints, computer architects changed the way of designing chips over a decade ago: increasing the number of cores per chip instead of building more complex single-core chips [3]. The last-level cache (LLC), i.e., the level closer to memory and further away from cores, is typically shared among all cores in the chip. When shared, it stores blocks from all cores and is typically sized at about 1-2 megabytes (MB) per core. Increasing the number of cores requires a larger cache to maintain high core performance.

There are several ways to manage how data is allocated in the multiple levels of cache depending on whether a higher level (closer to memory) includes data resident in lower levels (closer to cores). In *inclusive* caches, a data block present in a cache, must also be present in all of its corresponding higher levels. To accomplish this, every cache miss will allocate the data block read from memory in all cache levels, including the LLC. At the same time, when a data block is evicted from a cache, the block is invalidated in all of its corresponding lower levels. The result of this policy is a lower effective cache capacity due to the data replication across cache levels, and the potential performance and energy impact of inclusiveness-induced invalidations.

Non-inclusive caches attempt to reduce the limitations of inclusive accesses by not enforcing inclusivity in higher cache levels. When a data block is accessed,

it is still allocated in all cache levels. However, an eviction on a cache does not trigger invalidation in lower levels. There is still data replication, but there are not inclusiveness-induced invalidations affecting performance and energy.

Exclusive caches go one step further by enforcing that a data block present in a cache cannot be also present in a corresponding higher-level cache. To enforce this, on a cache miss the block is sent to the lower levels and not allocated in the exclusive cache. On a LLC hit, the block would be sent to the lower level and invalidated in the exclusive cache. Only data evicted from lower levels is present in the exclusive cache, a design known as *victim* cache [4]. The result is that there is no data replication and, as a consequence, there cannot be inclusiveness-induced invalidations.

1.1 My Project

In this thesis we evaluate different prefetchers and replacement policies for the three cache inclusion types: inclusive, non-inclusive and exclusive. We use single-threaded applications for single- and multi-core (multiprogrammed workloads).

We use the ChampSim simulator, used in the 2nd Cache Replacement Championship [5], to model the different cache configurations. The benchmarks will be several traces from SPEC CPU2006 [6], CloudSuite [7] and one machine learning workload trace from mlpack [8].

1.1.1 Objectives

The main objectives of this project are:

- Quantify the correlation between cache replacement, prefetching and cache inclusion policies.
- Prove the need of having a different cache management technique depending on the cache inclusion type.

1.1.2 Contributions

The contributions of this project to fulfill the aforementioned objectives are:

- A comprehensive evaluation of multiple cache configurations including multiple replacement policies and prefetchers for all three inclusion policies: inclusive, non-inclusive and exclusive caches.

- A discussion on the results targeting to understand the gaps in the design of cache replacement policies to improve performance and reduce energy consumption in the presence of a given inclusion policy.

1.2 Document Structure

Chapter 2 introduces the basics on cache hierarchy that are necessary to understand on the following chapters.. Chapter 3 explains the state of the art on cache replacement policies. Chapter 5 explains the implementation details of this project and discusses the challenges that came up. Chapter 4 describes the methodology used in the project. This includes the tools and the evaluation methodology. Finally, we conclude showing and discussing the results, the lessons learned and the conclusions of this work. At the end, we propose different lines of future work.

2. BACKGROUND

The improvement rate of processor speed has been higher than that of memory speed for a few decades, the so called *memory wall* [9]. It is necessary to develop techniques to mitigate this performance gap. One solution was introducing several levels of memory, also known as the *memory hierarchy*, to bring data closer to the processor. There has been extensive research on improving memory management. Specifically for the cache hierarchy, the memory between the processor and main memory, the two most important topics have been prefetching and cache replacement policies.

Prefetching aims to bring data to a cache level closer to the processor before the data is requested. This reduces the latency on accessing the block if the prefetcher was accurate (brought the data that was to be requested) and timely (the time the data arrived to the cache made the accesses hit). However, prefetching can also pollute the cache with blocks that are evicted before being used if it brings data that will not be used or it is brought too late or too early, thus interfering with actual useful data.

Replacement policies improve the management of cache contents to evict first the blocks that are not likely to be used again to make space to the newly requested blocks. They can also hurt performance if the block removed was still in use and is requested shortly after.

Cache blocks that contain the data can be in either of the cache levels. The inclusion policy decides in how many levels and where to keep each block. For example, if we put the same block in all cache levels, the effective space of the cache hierarchy would be reduced but it could be faster to access assuming the block would

still be in the next cache level.

In the following sections of this chapter, we introduce basic concepts of the memory hierarchy and techniques to improve its use.

2.1 Memory Hierarchy

Computer programs usually have memory access patterns that exhibit spatial or temporal locality. Spatial locality refers to a memory access to a location most likely will result in recurrent accesses to nearby memory locations. For example, when accessing in order all elements in an array, where all the array elements are stored consecutively in memory. Temporal locality refers to a memory access that will likely result in another reference to the same memory location again in the near future. For example, when in an array operation we need to read and then write on an array several times.

Figure 2.1 shows an example of a memory hierarchy in a modern system. To reduce the latency of bringing data from main memory to the processor, architects exploited the spatial and temporal locality of programs with faster and smaller memories between the processor and the main memory. These small memories are called caches. Typically there are two, three or four levels of cache, each one of a different size and access latency. The closest level to the processor is typically called "L1", for level 1, the next "L2", and so on. We will call the closest level to the processor the *lowest* and the last one before the main memory the *highest*, i.e. the lower level of L2 is L1. Typically, the L1 is divided in two: L1 data (L1D) and L1 instructions (L1I).

The core also contains a few registers where all data in use is stored to compute the current fragment of code (inside of CPU in Figure 2.1). This is a very small and expensive memory, and it is the fastest one.

As a rule of thumb, the closer to the processor, the smaller the cache capacity and latency. The latency of each element on the memory hierarchy depends on different properties: its technology, its capacity and its distance to the core. The different latencies at the different cache levels have to do with their implementation in terms of logic and technology. Accessing larger data arrays require larger latency because of more complex circuitry, such as large decoders/encoders, that involve longer gate nets. At the same time, bit cells in on-chip caches are implemented with 6 transistors (6T SRAM) for faster access although at higher power, while off-chip memory is implemented with 1 capacitor bit cells (DRAM [10]) that loses its charge and must be refreshed periodically for higher density (less area per bit) and lower power consumption at the expense of latency and refresh cost. Also, on-chip caches typically run at higher frequency. Some on-chip caches are implemented with embedded DRAM (eDRAM) which provide higher density and low power at the expense of latency. This eDRAM technology is applicable to LLCs because their latency would be prohibited for caches closer to the core.

The basic unit for cache storage is the cache block. A cache block contains a certain number of data bytes. Each cache level is organized in cache sets. A cache set may contain from one block to all blocks of the cache. The data is placed in the cache in a position that depends on its address. To identify which block of the set we want, a tag is used as a unique identifier. Cache memories can be mapped in different ways: direct mapped, set associative and fully associative. A *direct mapped* cache places a block in a given position indexed by some bits of its address. Several blocks can be mapped to the same position. In this mapping, each set contains one single cache block. The *fully associative* cache places the blocks in any of the positions available. In this case, one set contains all blocks. The *set associative* cache is a compromise between those two: each set contains are a small number of

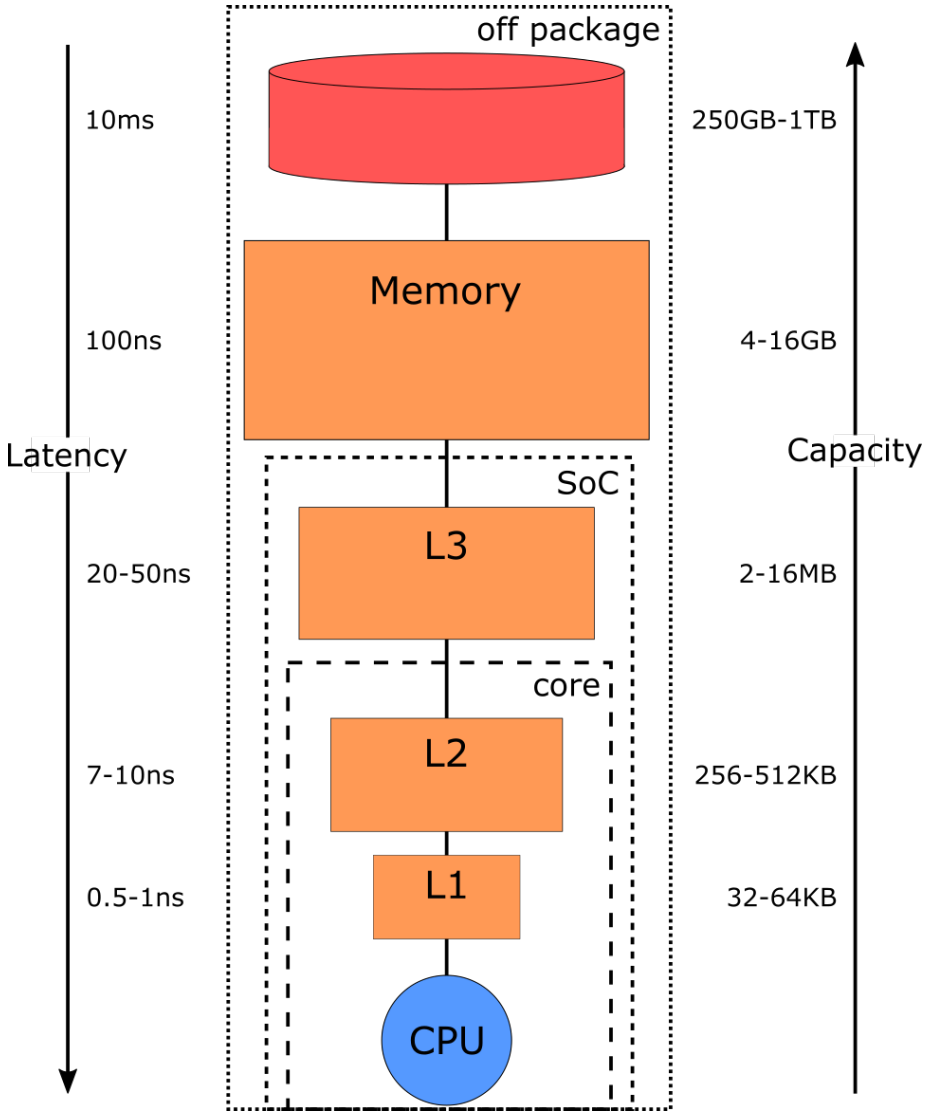


Figure 2.1: Diagram of the memory hierarchy in a modern processor. The higher the capacity of the cache, the higher the latency to access a block in that cache. In a multi-core, there are replications of the "core" part inside the "SoC" part.

ways (typically 2, 4, 8 or 16), each block being placed in the set that is chosen by its address, and at any ways within the set.

A *cache hit* happens when a block is requested and it is currently stored in one of the levels. A *cache miss* is when the block is not there on reference. On a cache miss, to improve performance, a cache in a modern processor typically has special registers called miss status holding registers (MSHR) where it stores information about the block that missed while its requests to the next cache levels and memory is resolved. A cache with MSHR is called a non-blocking cache. Keeping that information in multiple MSHRs allows to have multiple cache misses being resolved in memory and more load/store instructions in flight. This is called memory-level parallelism (MLP), and allows overlapping latencies from multiple accesses and the core to progress computation on instructions that are independent from those misses.

2.1.1 Miss Types

There are four different types of misses depending on the reasons that cause them: compulsory, capacity, conflict and coherence. Compulsory misses are the ones that are a miss because the execution of the program has just started and the caches are empty. These misses are also called cold misses. Capacity misses occur because of the limited cache size. These misses completely disappear with a sufficiently large cache. Conflict misses occur because of the data mapping in the cache. In a direct mapped cache, each block is mapped to a particular cache position. When placing a block in the cache, the block in that position is evicted. A definition by Hill, "conflict misses are misses that would not occur if the cache were fully associative with LRU replacement". Coherency misses occur when private caches invalidate other copies of their blocks in the cache hierarchy and subsequent accesses to those invalidated copies miss.

There are several techniques to reduce these four miss types. Compulsory misses can be reduced by a prefetcher that predicts which blocks are going to be used and brings them to the cache before they are requested. However, prefetchers can pollute the cache with data that is not going to be used if the predictions are wrong. Compulsory misses can also be reduced by increasing block size. A larger cache reduces capacity misses. A larger cache fits more blocks, but then latency may be higher. Conflict misses happen because of the mapping, so with a higher associativity it is less likely that a block that will be needed is going to be evicted. However, the higher the associativity, the higher the energy, the slower and more complex the cache. Coherency misses can be reduced by using a different coherence protocol, for example one that updates the block at other caches instead of invalidating the block.

2.1.2 Load Flow

At the start of a program all cache levels are empty. Below there is an order on what happens when the first load instruction executes in a 2-level inclusive cache. From the cycle the load operation is selected to access the L1 cache:

1. In parallel:
    - Decode the index of the L1 cache set with virtual bits
    - Search the translation of virtual to physical in the translation lookaside buffer (TLB)
2. There is a TLB miss, because that memory page was not accessed before
3. The page walker searches the page
4. If there is a page fault, the operating system (OS) causes an exception to bring the page from disk to memory and stores the translation in the page table and in the TLB

5. The instruction is re-executed
6. On the second access to the TLB, the physical address is found, and all tags in the set are compared to that address, and if a block matches, it checks if the block is valid
7. It is an L1 miss because the block was not accessed before and the prefetcher is not trained yet
8. Find the victim block where the data will be stored in the L1 whenever it arrives from memory (depends on the replacement policy) and update the state of the replacement policy and prefetcher (if necessary)
9. The information about the instruction is stored in an MSHR of the L1 (assuming there is one empty, otherwise we have to wait until one becomes free)
10. Decode the index of the L2 cache set
11. Check all the tags in the set and whether the block is valid
12. It is an L2 miss
13. Find the victim block where the data will be stored in the L1 whenever it arrives from memory (depends on the replacement policy)
14. The information about the instruction is stored in an available MSHR of the L2
15. Request to the memory controller to load the line that contains the required data
16. The memory controller reads the line and sends the block back to the requester L2

17. The cache controller receives the block and matches the address to the information in the L2 MSHR

18. Evict victim block and write it to memory if it was dirty

19. The block is stored in the L2 and sent to the L1 (and repeats the steps for block replacement as in the L2)

20. The data requested by the load operation is sent to the central processing unit (CPU)

If the 2-level cache was non-inclusive, it would have been the same list and order. The difference is in the exclusive cache: in the step 19, the block would not have been allocated in the L2, only in the L1.

2.1.3 Write Policies

The cache hierarchy keeps data in some or all of their levels while the data is in use. Modified data must be written back to memory at some point. There are two main moments that a block can be written to memory: immediately or before invalidation. These two write policies are called write-through and write-back respectively.

A write-through cache writes the block to main memory immediately after writing it to the cache data array. This is a simple implementation that does not require any work at invalidation, but it requires to send the block to memory on every write. In this case, cache blocks and main memory contain the latest and updated data. This simplifies coherence (see subsection 2.1.4) but increases data movement.

A write-back cache only writes the block to main memory when it is evicted from the cache. Meanwhile, the data in main memory is a stale copy. A write-back cache is more complex as it requires one bit (dirty bit) to identify whether the block has

been modified -so called dirty block. If the block is modified, it needs to be written to memory on eviction. If the block has not been modified -so called clean block, it can be simply invalidated with no further action.

In both write policies, it is not defined what to do on a write miss. There are two possibilities: either allocate the block "write allocate" or not, "write-no-allocate". Write allocate loads the block that missed to the cache and then writes it. Write-no-allocate writes the data directly to main memory bypassing the cache.

Any combination of write policies and write miss policies are possible. However, there are two that are generally more efficient: write-back with write allocate and write-through with write-no-allocate.

2.1.4 Cache Coherence

Modern microprocessors have multiple cores. Each core has at least one private cache and there generally is a cache that is shared among all cores. Figure 2.3 shows an example of a typical three-level cache hierarchy with two cores where the L3 is shared among all cores, and L1 and L2 are private for each core. The same block of data can be present in several private caches for multithreaded applications, where different threads share data. For example, in a program with two threads, each running in a different core, we can have the situation shown in Figure 2.2. If there is no coherence, the last load miss will come from memory with a stale value because the correct value is in thread 1's private cache.

The two main techniques to solve this problem are invalidation and update. On invalidation, only one copy of the block is allowed to exist at a time, whenever another core requests that block, the previous is invalidated. On update, each write to a block is also written to all other present copies of that block.

Multi-core processors implement these techniques atomically through a cache

```
Thread 1:       load A (miss)
Thread 2:       load A (miss)
Thread 1:       write A (hit)
Thread 2 L1:    load B (miss) --> invalidate A
Thread 2:       load A (miss)
```

Figure 2.2: Cache coherence problem example.

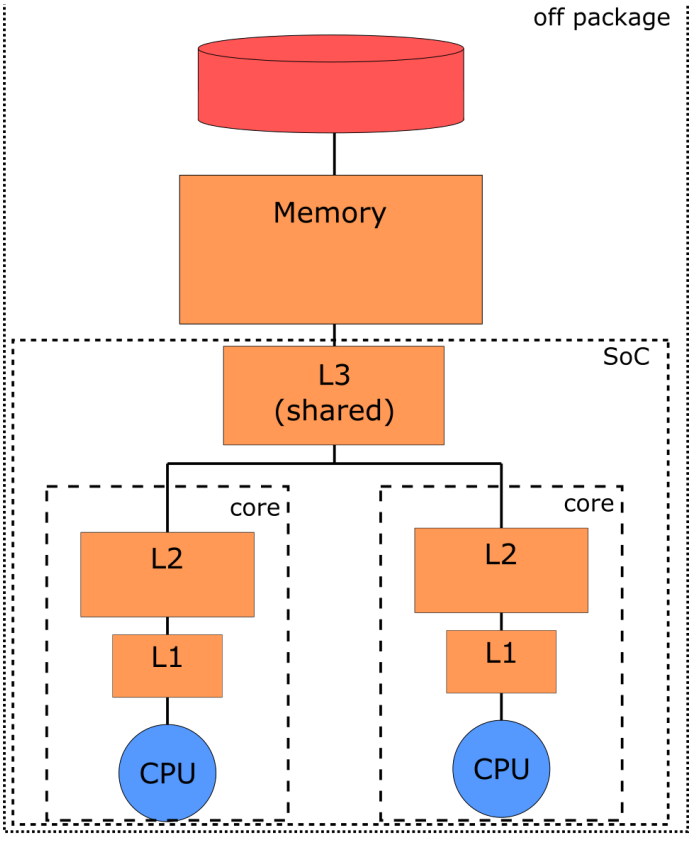


Figure 2.3: Memory hierarchy in a multi-core processor.

coherence protocol. Each cache block has a coherence state to represent information such as which and how many cores have the block and if the block has been modified. The simpler state information would be for a single-core write-through cache, where the only two states are: valid and invalid. In that example, all blocks are invalid first and are marked as valid at a fill. On a write, the block is still valid and the main memory is updated. Another simple state information would be for a single-core write-back cache, with four states: valid, invalid, modified (dirty) and clean.

2.1.4.1 MSI Protocol

A simple cache coherence protocol is the MSI protocol. This protocol implements the invalidation technique and is used in write-back caches. There are three different states in this protocol: "M", "S" and "I". The "M" state stands for *modified* and is equivalent to a dirty block. This state can only be held by one of the copies across multiple private caches. The "S" stands for *shared*, this means that the block is valid and clean. One or more copies of the block across multiple private caches can be in this state. The "I" stands for *invalid*.

Cache coherence protocols are usually represented with a state transition diagram that shows when and what triggers a transition from one state to another. Figure 2.4 shows the state transition diagram for the MSI protocol. The black arrows show the actions (read/write) initiated by the core. The red arrows represent the requests initiated by caches. There are two main transitions that cause several messages. First, a block in the shared state that is going to be written by one of the cores generates an invalidate message to invalidate all copies in other caches. Second, when a core wants to read a block that is in modified state in another core's private cache, the modified block value is written to memory and transferred to the reading core and both copies are set to shared state. This implies snoop or directory operations in

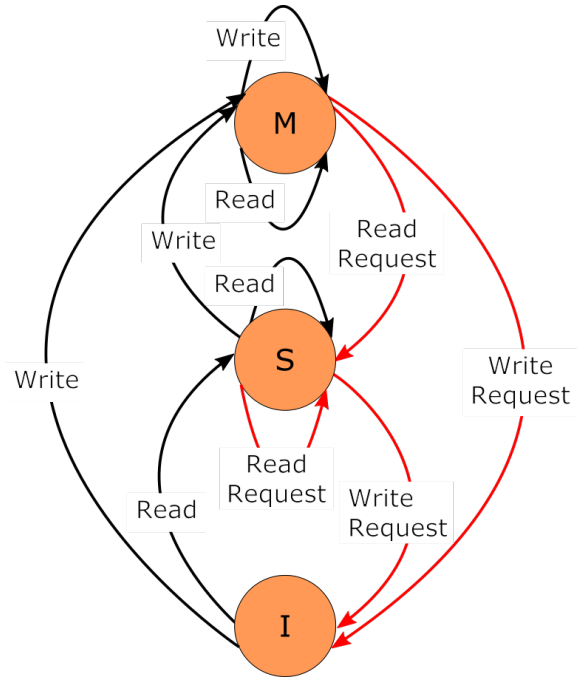


Figure 2.4: Simplified state transition diagram of the MSI cache coherence protocol.

the on-chip network to find the modified copy and transfer the modified block value to the reading core.

2.1.4.2 MSI-Like Protocols

There have been multiple efforts to improve on the MSI protocol, mainly to reduce coherence traffic. One of the issues in the MSI protocol is that every time that there is only one copy of a block in the shared state and that the core wants to modify it, it has to send unnecessary invalidation messages to the bus. The MESI protocol improves on this case. The new state "E" stands for *exclusive* and means that there is exactly one copy of the block at a time and it is clean. When a block is filled for a read the first time, it transitions from invalid to exclusive. Then, when the core wants to write the block transitions to modified without the need to send

invalidations to other because it is known that it is the only copy in the system. This protocol significantly reduces coherence traffic when a block is private, i.e., accessed by a single core.

Another issue in the MSI protocol is that every time that there is a read request from a core of a block in the modified state in another core, the modified block must be written to memory. The MOSI protocol adds the *owned* state, "O". A block arrives to this state when a modified block receives a read request from another core. The cache that had the modified block, so called the "owner", is the responsible to respond and send the block to all read requests to that block and, later, to send the value to memory before invalidation. The cores that receive that block keep the block in shared state. This avoids writing the block to memory until the latest possible moment.

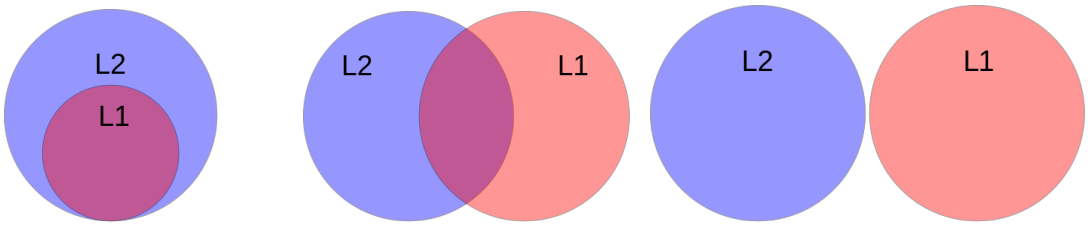
The MOESI protocol puts together all the states used in the previous protocols. This protocol improves performance by delaying the writes to main memory as much as possible and by reducing unnecessary coherence messages when a block is only referenced by a single core.

2.2 Inclusion Policies

A cache level is related to the previous or the next level (if any of those exist) depending on which data blocks each level contains. A particular cache level can contain exactly all, exactly none or some of the data blocks of the lower level.

An *inclusive* cache contains all data blocks of the lower level. An *exclusive* cache does not contain any of the data blocks of the lower level. A *non-inclusive* cache can contain some blocks from the lower level but not necessarily all or none. Figure 2.5 shows a diagram of each of the inclusion policies.

Each cache level can use a different inclusion policy. For example, the L3 can



(a) Inclusive (b) Non-inclusive (c) Exclusive

Figure 2.5: Diagram to show where the data is in an a) inclusive, b) non-inclusive, and c) exclusive cache. The intersection is data duplication.

be exclusive of L2 while the L2 is inclusive of L1. All these inclusion policies have their benefits and drawbacks, mostly related to latency, data replication and data movement.

2.2.1 Inclusive

An inclusive cache level contains all data blocks from lower levels plus some other blocks. Figure 2.5a shows a diagram of which data is in which cache, where the intersection is data duplication. That is, a data block is replicated in both cache levels.

In a 2-level cache hierarchy, the data block will be placed in both cache levels on an L2 miss. If the block is evicted from the L1 and, later, a request comes (L1 miss), the data may still be in the L2, thus avoiding accessing main memory. On an L1 eviction, only write backs of dirty blocks are required. If the block is clean, there is no need to copy it back to the L2 because it is already there as per the inclusion policy. A potential problem is on an L2 eviction: to preserve inclusivity, if the block was present in L1, it must be evicted too.

In a multi-core system, this inclusion policy simplifies the coherence protocol im-

plementation. A cache wanting to invalidate copies of a block in other caches just has to notify the LLC because it has the information of all blocks in all lower caches. With this, there is no need for coherence message broadcasts, thus reducing complexity and energy consumption. Also, coherence information (state and caches having a copy) can be encoded with the cache block so the information is available when accessing it, thus cutting latency of potentially having to access separate structures, such as a directory.

Given that an inclusive LLC knows which lower caches have a block that is going to be evicted, invalidation messages can be directed to those caches without the need of broadcasting the message.

One disadvantage is the effective cache size due to data duplication. The effective size of the cache hierarchy is the size of the LLC. For example, in a 2-level cache hierarchy, the effective size is the one from L2 because it contains all contents from L1. The L1 cache only keeps data closer but does not contribute with additional capacity.

Another disadvantage is back invalidations. An eviction from the LLC can generate an invalidation in an L1. If it was present in the L1, the block may be in use. This can be a problem if the replacement policy is not aware of the usage of a block in the lower caches. Jaleel et al. claim that the limited performance of an inclusive cache comes from back invalidations because the LLC replacement policy is not aware of the core presence of blocks and their recency [11].

A related problem is that an inclusive cache has less flexibility to improve cache management due to the impossibility of bypassing the LLC to maintain inclusivity.

$$E = c_n \tag{2.1}$$

Equation 2.1 shows the calculation of the effective cache capacity in a cache hierarchy with inclusive LLC, where E is the overall system effective cache capacity and n is the number of cache levels, thus c_n being the capacity of the LLC.

2.2.2 Exclusive

An exclusive cache does not include any replicated block from lower levels. Figure 2.5c shows a diagram of the data blocks of each cache level in an exclusive cache. In this case, there is no overlap and the intersection of the two sets is empty, so there is no data replication. This inclusion policy increases the total amount of data blocks that can fit in the whole cache hierarchy.

The exclusive inclusion policy is similar to a victim cache [4]. In a three-level cache hierarchy, the LLC would be the victim cache of a two-level cache. Victim caches contain the evicted blocks from the lower levels aiming to reduce conflict misses. This was originally introduced as a fully associative small cache to reduce conflict misses from direct-mapped caches.

However, it incurs higher complexity. In the example of two cache levels (L1 and L2), when a block that is in L1 (and not in L2) is evicted, it will be allocated in L2. When the block is accessed again, it will be invalidated in L2 and allocated in L1. This generates more work to do on an L2 hit. Also, it makes impossible to use the recency of a block to choose which block to replace when the L2 cache is full, as it only contains data that was evicted from L1 and not accessed again since that eviction.

Jouppi and Wilton identified the benefits of exclusive caching and evaluated them [12]. They found that the extra space of not duplicating the data in the two levels of cache and a higher associativity in the LLC was indeed beneficial. Ten years later, Zheng et al. evaluated the performance of exclusive cache hierarchies

with respect to inclusive caches [13]. They found that exclusive caching is beneficial for most of the benchmarks they tried (SPEC 2000), but especially for smaller lower-level caches. They suggest that exclusive caches are more suitable for server applications and embedded systems.

The main benefits when using exclusive caches are:

- Less conflict misses by behaving like a higher associativity cache, as two memory references that are mapped to the same set can reside one in each level instead of only one.
- Higher hit rate thanks to a higher effective space by avoiding the blocks duplication in different levels. This is especially relevant in caches with more than 3 levels of cache or with large lower level caches.
- Avoids premature evictions from the lower levels of cache by not requiring back invalidations, like in an inclusive cache policy.

The main drawbacks and limitations of an exclusive cache are:

- Less design flexibility because the block size of the exclusive cache has to be the same as the other cache levels.
- More control complexity and power consumption due to the higher data movement of blocks from one level to the other.
- More complex cache coherence protocols and more area required in symmetric multiprocessor (SMP). This is only important for multithreaded programs, which is not the case of this thesis work.

Equation 2.2 shows the effective capacity of an exclusive cache, being E the effective capacity, n the number of cache levels, and c the capacity of a cache level.

The higher effective capacity is one of the most attractive features of an exclusive cache. For this same reason, exclusive caches have been investigated in other fields, such as in storage to reduce the impact of the high usage of cache RAM in disk arrays [14].

$$\mathrm{E} = \sum_{i=1}^{n} c_i \ , \tag{2.2}$$

2.2.3 Non-Inclusive

A non-inclusive cache level may or may not contain blocks from lower levels. Figure 2.5b shows a diagram of one possible case in a non-inclusive cache, where just some blocks from L1 are also in L2. The data is replicated when there is a miss in a cache level, and the block is allocated in that cache level and all higher ones. For example, in an L1 miss where the block is in none of the caches, the block will be allocated in L2 and L1. The difference with an inclusive cache is that the inclusivity is not enforced. That means, when a block is evicted from a higher level, it does not generate back invalidations to the lower levels. This simplifies the implementation of this type of caches.

The main advantages that can be gained by forcing non-inclusion are a higher effective cache and lower conflict misses. The effective cache size is higher compared to an inclusive cache. In the best case scenario, the effective cache size is the sum of all caches, like in the exclusive case (see Equation 2.2). That is the case when all cache blocks present in the L1 have been replaced in the L2. However, the worst-case scenario is when none of the L1 blocks have been evicted from the L2, equivalent to an inclusive cache (see Equation 2.1). The cache hierarchy usage in a case with non-inclusive cache changes depending on the application and replacement policy.

Conflict misses will be reduced in the intermediate or last-level cache. The blocks that are referenced frequently stay in L1, therefore L2 has space for other blocks.

One disadvantage of non-inclusive caches is coherence. A non-inclusive LLC that needs to evict a block will have to ask all the lower level caches if they have the block, because that information is not present in the LLC, unless a separate directory is implemented and then it must access the directory and pay its extra latency. If the block is present in any lower-level caches, it needs to be invalidated or updated. However, there has been work to separate the cache coherence structures (i.e. directory) from the data blocks of the cache. Zhao et al. proposed a non-inclusive cache with an inclusive directory to keep the positive features of both inclusive and non-inclusive policies [15].

2.2.4 Summary of Inclusions

Table 2.1 shows a summary of all the inclusion policies explained previously in this section.

Modern processors use different types of inclusion policies in each level of cache. The most common is to either use an inclusive or an exclusive policy in the LLC and an inclusive or non-inclusive in the lower levels. Below there are a few examples of real processors with the information on the inclusion policy they use. None of the ones covered here has a non-inclusive LLC.

For example, AMD processors generally use an exclusive last-level cache and Intel, an inclusive one. The AMD Athlon (from the Thurderbird architecture) had an exclusive L2 (LLC), while its rival at the time, the Pentium 4 (from Willamette) [16] had inclusive L2 (LLC). Currently, the latest AMD Zen architecture has a (mostly) exclusive L3. Current Intel processors like Sandy Bridge, Ivy Bridge and Skylake have an inclusive L3 and a non-inclusive L2 [17]. The Intel Knights Landing has an

	Inclusive	**Non-inclusive**	**Exclusive**
Data replication	⇑	↑	None
Benefits	Simple coherence, no copy back necessary	Simple to implement	Highest effective capacity
Drawback	Wastes cache space, back invalidations	Data replication on a miss, complex coherence	More complex on an LLC hit
Replacement policies	Core-aware problem	Simple	Heuristic problem (no recency, frequency info)

Table 2.1: Inclusion policy properties.

L2 (LLC) that is inclusive of the L1D and non-inclusive of L1I [18].

The ARM Cortex-A9 can have an (optional) L2 cache(LLC). The core has support to be attached to exclusive L2 caches as long as that is properly configured both in the core and L2 controller sides [19].

The processors in the IBM POWER series had mostly L3 (LLC) exclusive caches. The POWER5 has an L3 exclusive and an L2 inclusive of both L1D and L1I [20]. The POWER6 has an L3 exclusive cache and the POWER7 has an L3 mostly exclusive cache [21]. The IBM zEC12 has an inclusive L3 (on die) and an inclusive L4 (off-die, on-package) [22].

2.3 Replacement Policies

To reduce the gap of memory and processor performance, computer architects designed caches to bring in-use data nearer the processor. One of the important design decisions in a cache is the replacement policies. Caching and replacement

policies have been researched in different fields such as systems [23, 24, 25, 26, 27] and databases. In this thesis we only consider caches in the memory system.

Cache replacement policies are algorithms to improve cache management. They are used when the cache is full and a new block has to be allocated: the algorithm chooses which block to evict from the cache to place the new one.

The best algorithm would be to evict a block that is no longer going to be used in the future. This is only possible with knowledge of the future. Belady proposed an optimal cache replacement algorithm assuming knowledge on the future [28]. As a processor does not have such knowledge, there has been plenty of work in cache replacement algorithms.

A naive algorithm would be to choose a block at random and replace it. This is easy to implement but not generally effective, as it evicts a block that me be in use. There has been plenty of work to improve this algorithm trying to evict the least useful data. One way to improve cache management would be to decide based on how recently the block has been accessed, like least recently used (LRU). Another way is using machine learning to learn from the past accesses and make a prediction of which blocks will likely be accessed again.

Cache blocks can be accessed in many different patterns. A technique that has been used to adapt cache management to different patterns depending on the application is set dueling [29]. Set dueling uses different replacement policies on a few cache sets and compares their performance. The best replacement policy across the compared sets is used for the rest of the cache.

A replacement policy can maintain information on the blocks to later decide which block to evict. There are two main cache operations that trigger actions in a replacement policy: the placement (or insertion) of a new block into the cache, and the promotion (or update) of an existing cache block. On placement, the replacement

policy can statically or dynamically decide the initial value of the information to keep with the inserted block. If the cache if full, the placement also triggers the replacement algorithm to choose which cache block to evict. A cache hit typically triggers a promotion of the accessed block. Usually, the replacement policy will mark the block to be more protected of eviction than it was before. The replacement policy can also implement a bypass policy. This means that the cache block that was going to be placed in the cache might be predicted not to be used before its eviction. In that case, the block is considered *dead* and it is not placed in the cache, but forwarded to the next level.

2.3.1 LRU

The least recently used (LRU) is a simple and intuitive replacement policy that is often used as a baseline to compare other policies. This replacement policy evicts the cache block that has not been used for the longest time. This policy has been implemented to exploit temporal locality, given that a block that has been used recently is likely to be reused soon. However, that is not always the case. Some applications do not have such high temporal locality, for example, a large data set that does not fit in the cache and it is accessed many times from the first to the last element. In that case, by the time the first element is accessed again, it would have been evicted from the cache as it was the least recently used block.

The LRU policy keeps the information per block on the order that the blocks of a set had been accessed. For example, in an 8-way set associative cache, the blocks will be ordered from zero to seven, being zero the most recently used position and seven the least. The placement of a block in the cache is set to position zero because is the block that has just been accessed. On a cache hit, for example, to a block in position four, the block is also set to position zero. All other blocks' positions are

recalculated. On a fill that triggers the replacement of a block, the victim block will be the one in the LRU position, position 7 in this example.

This replacement policy requires to store a considerable amount of bits per block and to recalculate the position and update it for each cache block in a set. This is very costly. For this reason, an approximation of this replacement policy has been proposed: the Pseudo-LRU (PLRU). PLRU is commonly implemented as a tree making the number of bits to store and modify to be small. PLRU is beneficial compared to LRU for large cache associativities. There are many other heuristics around the access recency or frequency such as *most recently used*, *not recently used*, *least frequently used*, not frequently used.

2.4 Prefetchers

Cache misses are a common reason for CPU stalls in computers. Prefetching, in conjunction with other techniques, such as replacement policies, is one effort to to reduce these stalls by predicting misses and issuing a memory request before the actual access occurs. Prefetching is a technique used to hide memory latency by bringing data that will potentially be needed by the processor to a closer level of the memory hierarchy. Prefetching can be done either for instructions or for data. Also, it can be implemented in hardware, software or a combination of both. There is, however, the risk of polluting the cache when the prefetched data is not used due its eviction before use or wrong prediction.

There are several prefetching approaches proposed in the literature contained in two categories: hardware and software. Software prefetching consists in adding instructions to the program to fetch data ahead of its use. They do not speculate on data but add instruction overhead. Contrarily, hardware schemes do not require programmer or compiler intervention and incur no instruction overhead. However, they

may mispredict the next data to be accessed causing cache pollution and additional traffic and power consumption.

Hardware prefetching can be classified in two categories: spatial and temporal. Spatial schemes use accesses to the current blocks as the basis for prefetch decisions. Temporal schemes use lookahead decoding of the instruction stream to decide what and when to prefetch [30]. Due to the greater complexity of the data access pattern compared to instructions, data prefetching techniques are more diverse and thoroughly researched than instruction prefetching techniques.

There are three main types of hardware prefetching: address correlated, spatially correlated and execution based. Address correlated prefetching relies on identifying address correlations of previous misses. These prefetchers predict that a sequence of accesses happening close in time are likely to be accessed together in the future. These prefetchers work well to exploit the patterns of algorithms that traverse data structures such as linked lists.

The spatially correlated prefetching exploits the regularity and repetition in data layout. Data structures and objects have a fixed layout in memory and is frequently aligned to cache lines. The same layout patterns are often similar for other objects in memory. The regularity of the layouts and the reusability of the patterns makes this prefetching technique effective also to reduce cold misses. One common special case is the sequential and stride prefetching. Stride prefetchers exploit the spatial locality by prefetching, for example, the next contiguous block. These prefetchers can be tuned to work for a different stride (e.g. instead the next block, the fourth next block) and to identify multiple strides.

The execution-based prefetching executes the program's memory accesses ahead of the actual execution so data is brought closer ahead of time. Execution-based prefetching seek to access earlier the exact addresses that are going to be accessed

later. It does not speculate to decide which addresses to prefetch. Prefetches are known to be useful because the program had already executed or will execute them in the future. This is achieved by using spare resources to execute future instructions or by adding extra hardware to the memory hierarchy to monitor which of the evicted addresses might be needed.

In the rest of this section, the prefetchers used in our experiments are described.

2.4.1 Next-Line

The code is stored sequentially in memory and many instructions are accessed consecutively. This prefetcher exploits these two characteristics and simply brings the cache line that is stored after the one that has been accessed. Next-line is a prefetching technique for bringing the next consecutive cache block [31]. This prefetcher works specially well for instructions, but also for applications with high spatial locality.

This prefetching technique has been extended to a variable number of cache lines to be prefetched. In terms of instructions, it can be adapted to recognize and prefetch entire basic blocks.

2.4.2 Instruction Pointer-Based Stride

Stride prefetchers aim to remove compulsory and capacity misses. These prefetchers predict that, when an access to a memory address is missed, another memory access to an address that is the same plus an offset will be likely accessed and missed in the future. Stride prefetchers generate a cache allocation of the predicted block. If the block is prefetched and accessed (hits), it is a useful prefetch. Therefore, stride prefetchers are not only configured to prefetch on miss but also to prefetch on hit.

This pattern is usually regular within the multiple execution of the same static instruction. For this reason, stride prefetchers are typically implementated to track

information per instruction pointer (PC). The tracked information includes the next *address* to be prefetched and the identified *stride* [32]. They also typically include a *degree* and *distance*. The degree is how many prefetchers an access by the instruction at the corresponding PC triggers, and distance is how far ahead it starts prefetching, i.e., a distance of one would prefetch $address + stride$, while a prefetch distance of N would prefetcher $address + (N \times stride)$.

2.4.3 Best-Offset

Offset prefetching is a generalization of the next-line prefetcher, where an offset can be specified. In next-line, the offset would be one. But bringing the immediate next line is not always the best case for all applications. The access patterns can be different and, therefore, dynamically adjusting the best offset improves the usefulness of the prefetcher. Another problem is timeliness. Prefetching a block late may still improve performance but in a smaller scale. Bringing a block with a higher offset may guarantee that the prefetch arrives on time.

Michaud proposed the Best-Offset Hardware Prefetching [33] (BOP) to solve these two problems. He implemented a learning algorithm that tries different offsets and uses the best. BOP has a table to store the base address of the recent prefetched requests that BOP tried (recent requests table or RR) plus an offset list and a score table. During the learning phase, BOP tests an offset and updates the score for some L2 read accesses depending whether it was a hit or a miss in the recent requests table. Whenever all offsets in the list have been tested, they start again until a certain number of iterations of this process. The offset with the highest score is the one chosen.

2.4.4 DRAM-Aware Access Map Pattern Matching

Dynamic RAM (DRAM) is a type of memory cheaper than the one used for caches, so it is typically used for a larger capacity. The main application of DRAM is the computer's main memory, which is the memory that is between storage and the cache hierarchy. The DRAM is divided in multiple banks, which can be accessed in parallel. Each bank is composed of many rows. Whenever a row has to be read or written, the row has to be *activated* and brought to a row buffer. The read or write operations are done to the row buffer. The latest DDR devices limit the number of activations during a time window. Optimizing the use of the row buffer will improve performance if the limit is not enough.

One problem of prefetching techniques in DRAM is that when two cores are prefetching different blocks, if those accesses go to the same DRAM bank, they will be activating and deactivating the row buffer, resulting in ping-pong effect that prevents any of the cores being able to enjoy row buffer locality. This is in contrast to the case where one of the cores sends the prefetches alone and all those prefetchers accessing the same row get row buffer hits. If those prefetches are interleaved with prefetchers from other core to the same bank but different rows, those row buffer hits will become row buffer misses given the interference between the different core's prefetchers.

Ishii et al [34] proposed a prefetching technique to exploit locality in DRAM called DRAM-aware access map pattern matching (DAAMPM). Before the time the prefetch is going to be used, they suggest to wait and reorder the prefetch requests to optimize row activation. The prefetches are reordered in a way that all blocks that need to access the same row are done together. To implement this, they maintain a memory access map data structure that tracks memory locations accessed in the

recent past.

Another problem they identify is that many replacement policies are not aware about which blocks in the cache are allocated by prefetches or by demands. Most replacement policies promote a block on a hit to protect it from being evicted. However, a prefetched block should not be considered as a hit the first time it is accessed in that context, otherwise we would be promoting all blocks that have been only used once and might be dead. A solution would be to add a *prefetch bit* per cache line that is set to one when the prefetched block is filled to the cache. Whenever there is a hit to that block, the replacement policy will not promote the block but will set the prefetch bit to zero. However, adding an additional bit to all cache blocks in the last level cache would be very costly. For this, the authors also propose a prefetch-aware cache line promotion [34] (PACP). The idea is that the core issuing a demand access after a prefetch includes a bit to specify that the accessed block should not be promoted, as it has only been accessed once after the prefetch.

2.4.5 KPC

There has been extensive research in cache management, in both cache replacement policies and prefetchers. The efforts, however, have mainly been made separately. There has been little work on studying their interaction and their effect in each of the cache levels [34, 35, 36]. Those studies show that the benefit of replacement policies can be small or negative when combined with a prefetcher.

Kim et al. proposed a holistic approach to speculatively manage all cache levels with coordinated prefetcher and replacement policies [37]. This approach aims at not only improving performance but also reduce the overall hardware budget necessary for the combination of prefetcher and replacement policy.

They found that the interference between prefetchers and replacement policies is

higher with PC-based replacement policies such as SHiP [38]. Thus, they propose kill the program counter (KPC), an integrated cache management that consists on a prefetcher and a replacement policy components. Both components learn from each other to improve their efficiency.

The prefetcher component, which they call KPC-P, is a prefetcher that decides in which level of the hierarchy to prefetch each specific block. They use a signature table to store a compressed history of past L1 misses. The history is used as a signature to index a pattern table to predict the next block. The predicted block plus the previous history generates another signature which is again used. This technique has an initial training phase that sets a confidence value that is increased as prefetchers are useful. Later, prefetchers are only triggered if confidence on the prediction is high.

The replacement policy component, which they call KPC-R, is a low-overhead replacement policy that uses two global hysteresis to predict dead blocks by tracking global reuse behavior. One hysteresis is for cache demands and, the other, for cache prefetches. KPC-R has a few sampler sets in the LLC managed with true LRU and the rest of the cache uses a similar SRRIP (explained in a following section 3.1). The hysteresis is decremented on a cache hit in one of the sampler sets. The hysteresis is incremented on a sampler miss when the victim was never used. When the hysteresis is saturated high, accessed blocks are predicted to be dead.

Single-core simulations show KPC to achieve a 9.2% geometric mean speedup over the baseline DAAMPM with LRU. That is a 5% higher than SHiP and 5.8% higher than PACMan [36] and 8.1% higher than UMO [34]. Multi-core simulations show that, on average, KPC achieves a 14.1% speedup over the baseline. That is 8.1% higher than SHiP. KPC outperforms the other cache management techniques in most of the multi-core mixes.

3. RELATED WORK

Current processors invest significant amounts of area and power on prediction mechanisms. Replacement policies is one of the most significant predictors together with cache prefetchers and branch predictors. There has been extensive research on cache replacement policies and it has mostly been focused on the last-level cache (LLC) to minimize the highest cache miss penalty when a request has to access main memory.

This chapter describes several state-of-the-art replacement policies, some of which are used in our evaluations. Research in replacement policies has been focused on non-inclusive and inclusive caches. Table 3.1 shows a list of all replacement policies that are explained in this chapter and the inclusion policy the authors used to evaluate it. There has been mostly just two efforts in exclusive caches, and both from the same authors.

3.1 Re-Reference Interval Prediction

The LRU replacement policy predicts that blocks will be referenced in the near-immediate future. However, not all applications show this behavior, some are referenced in the distant future. For example, when accessing to a large working set (thrashing) and when having bursts of references to non-temporal data (scan).

Jaleel et al. proposed two techniques for re-reference interval prediction (RRIP) [39]: a static technique that is scan-resistant —static RRIP (SRRIP)— and a dynamic technique using set dueling that deals with scan- and thrash-resistant applications —dynamic RRIP (DRRIP). To implement this and decide which block to replace, they use a re-reference prediction value (RRPV) per cache block. An RRPV of zero means that it is predicted that the block will be referenced in the near-immediate

	Inclusive	Non-Inclusive	Exclusive
RRIP	X		
SDBP		X	
SHiP		X	
GIPPR		X	
MDPP		X	
EAF		X	
Perceptron		X	
KPCR		X	
Hawkeye		X	
Bypass and Insertion			X
CHAR	X		X

Table 3.1: State-of-the-art replacement policies and the inclusion policy.

future, and the saturated value, in the distant future. We will assume here that the RRPV has two bits (values from 0 to 3), where: 0 is the near-immediate; 1 is the near; 2 is the long; and 3 is the distant future.

SRRIP inserts a new block with an RRPV of two, so the block is not immediately in danger of eviction. The victim that will be evicted is a block with an RRPV value of three. if there is none, all RRPV are incremented until it finds an RRPV of three. To promote a block on a hit, they propose two algorithms: hit promotion (HP) and frequency priority (FP). In hit promotion, the RRPV is set to zero, predicting a near-immediate reference. In the frequency priority, the RRPV is decremented, so the more hits to a block, the lower RRPV value will be.

DRRIP uses set dueling to decide among two policies using a few sampler blocks and choose the policy with fewer misses. The two policies that it compares are SRRIP

and bimodal RRIP (BRRIP). BRRIP inserts with a higher probability a block with a distant re-reference (RRPV of three) and, with a lower probability, inserts with a long re-reference (RRPV of two). BRRIP helps to keep some of the working set on the cache in thrashing applications. They also propose a thread-aware version of DRRIP which uses a set dueling monitor for each application that access the shared last-level cache.

They did not use prefetchers. They modeled a 3-level cache hierarchy with a LLC inclusive. They found that in SRRIP, the best insertion policy is an RRPV of two, to predict a long re-reference. SRRIP-FP improves 4% over LRU on average. SRRIP-HP improves 5% over LRU. DRRIP outperforms SRRIP by 5%. In a 4-core processor, SRRIP does not degrade performance of any workload and gets an 7% improvement over LRU, and Thread-Aware DRRIP improves it by 10%.

3.2 Sampling Dead Block Prediction

A cache block is defined as *live* from the moment a block is allocated in the cache until the last reference to that block before eviction. A block is considered *dead* from the last reference until its eviciton. Dead blocks unnecessarely occupy cache space. To improve cache efficiency, dead blocks should be replaced from the cache as soon as possible. The replacement policy should choose those blocks first instead of waiting for their eviction in their replacement technique. For example, in an a cache with an LRU replacement policy, imagine a read request that hits in the cache so it is the last access to that block. The block is now in the most recently used position and it will take evicting a few other blocks before it reaches the LRU position.

Khan et al. proposed sampling-based dead block prediction (SDBP) to predict the blocks that are dead in order to improve replacement policies and bypass techniques designed for LLCs [40]. They use a sampler to reduce cache metadata, instead of

adding additional state information to all blocks. They proposed to sample program counters (PCs) to predict the blocks that are likely to be dead. Their sampling predictor reduces the additional cache metadata necessary compared to other dead block predictors and outperforms them. Each access to the cache, like in similar approaches, generates an access to the predictor but the predictor is only updated on a cache operation to one of the sampler sets.

In the sampling predictor, the sampler keeps an array with partial tags of a reduced number of sets in the LLC. They implement a true LRU replacement policy and used a smaller associativity in the sampler blocks.

The predictor has three tables of 2-bit saturating counters that are indexed by the signature (a partial tag). The tables are accessed by hashing the PC of the instruction that generated the memory access. The difference on the tables is that they use a different hash function to index them. On a hit, the value of the tables corresponding to that block is decremented, and on an eviction, incremented. An access to the predictor will access the three corresponding counters of the tables and compute the sum of the values that is then compared to a threshhold. If the predicted value is higher than the threshold, the block is predicted dead. The same implementation of this sampling predictor works for single-thread and multi-core workloads.

They evaluated their dead block predictor to improve an LRU replacement policy and bypass with single-threaded benchmarks. Their approach reduces average misses by 11.7% over LRU compared to 18.6% of an optimal policy. This miss reduction translates into better performance, particularly a geometric mean speedup of 5.9%. For multi-core workloads in a shared LLC, their sampler achieves a geometric mean speedup of 12.5% compared to a 4.5% of RRIP.

3.3 Signature-Based Hit Predictor

Wu et al. propose signature-based hit predictor (SHiP), a hit predictor to improve replacement policies that use the re-reference interval such as RRIP [38]. They improve the performance of these policies by predicting which blocks are likely to be hit again before eviction (live blocks). This approach is similar to SDBP but instead of predicting dead blocks, they predict alive blocks and protect them from eviction. RRIP inserts a block with RRPV value of 2, while the ones in danger of eviction are the ones with a value of 3. SHiP sets the RRPV to 1 when a block is predicted to be alive (not dead), thus better protecting the block from eviction.

SHiP is implented similarly to SDBP: it has a table of 3-bit saturating counters indexed by a signature of the block (partial tag) instead of three tables of 2-bit saturating counters. The counters are incremented on a cache block hit and decremented on an eviction of the block. A high value of the counter predicts that the block is likely to be hit again with a higher confidence, depending on a specific threshold.

SHiP achieves a performance improvement of 9.7% on average over LRU while SDBP achieves 6.9%. For a shared LLCs with multi-core workloads, SHiP improves by an average of 11.2% over LRU while SDBP improves 5.6% and DRRIP 6.4%.

3.4 Minimal Disturbance Placement and Promotion

LRU is the widely accepted replacement policy which maintains a recency stack giving the distinct position of blocks. But it is not useful for last level caches with more associativity and worksets with low reuse.

A tree-based pseudo least recently used (PLRU) is a feasible implementation that approximates LRU. The blocks are ranked in positions from the most recently used to the *approximately* least recently used. There are multiple state-of-the-art replacement policies that improve performance over PLRU. However, the low hard-

ware budget and the low complexity for promotion required to implement it, makes PLRU useful in some cases. For example, for an L1 cache which benefits from the low complexity.

Jiménez proposed genetic insertion and promotion for pseudo-LRU replacement (GIPPR) [41]. GIPPR adapts to the best insertion and promotion policy using set dueling, replacing the static and default PLRU insertion and promotion. The algorithm finds the best insertion and promotion vector (IPV), which is a vector that specifies which position the current block should be promoted to depending on its current position. The best IPV is searched by applying a genetic algorithm over random strings of IPVs. This IPV is then used over the tree-based PLRU replacement policy. This algorithm still gets the PLRU low hardware overhead but can match state-of-the-art replacement policies' performance.

Teran et al. proposed minimal disturbance placement and promotion (MDPP), to also modify the PLRU to keep the benefits from PLRU and match state-of-the-art replacement policies' performance [42] but being simpler than GIPPR. In PLRU, whenever a block is placed or promoted, usually, many of the other block's position is going to be changed (promoting a block to MRU will demote many other positions). The main idea relies on disturbing the position of those other blocks the least. For example, a block in the third position out of 16 that hits, it is probably going to be reused again soon, so we can leave it in position three instead of changing the positions of all other blocks.

They proposed to place the block in position $3 \times n/4$, where n is the number of ways of the cache level. That is the most protected position of the pseudo least recently used part of the tree. For example, in a 16-way set associative cache, the block will be placed in position 12, only the positions 12-15 will be disturbed, which were the ones in most danger of being evicted.

To promote blocks on a hit, they propose a vector that links the old with the new position to decide what position the block should be promoted to depending on the position that the block was before. In the 16-way associative example, the first half of the positions (from 0 to 7) are not promoted, assuming that the block will be referenced again in the near future, so we avoid disturbing any other position. The second half of the positions (from 8 to 15) promotes the blocks to the first positions (0-3) predicting that the block has a long re-use distance, thus promoting the block to the most protected positions.

They also propose a dynamic version of this placement and promotion algorithms using the sampling-based dead block prediction (SDBP). They use the dead-block prediction to decide placement and bypass. For the placement, they check the confidence value returned by the dead-block predictor and place the block in a different position depending on the confidence, but always without disturbing the most-protected half of the tree.

Their static version with single-threaded workloads achieves an average speedup of 2.5% over LRU and 5.3% with multiprogram, which is comparable to SRRIP. Their dynamic version using dead block prediction achieves a 5.4% speedup over LRU for single-thread and 14.3% for multiprogram, comparable to SHiP. They show that their low-overhead implementation matches state-of-the-art cache replacement policies' performance.

3.5 Evicted Address Filter

Two main problems in cache management are: *cache pollution*, not-useful blocks that replace blocks with high locality, and *cache thrashing*, blocks with a high reuse but long reuse distance that replace each other. Previous work on replacement policies explored mechanisms to mitigate both cache pollution [43, 44, 45, 38] and

cache thrashing [29, 46, 39, 47] problems.

Seshadri et al. claimed that none of those previous efforts have been effective in preventing both cache pollution and thrashing at the same time [48]. The authors introduced a simple mechanism to predict reuse behavior of cache blocks and also prevent cache thrashing and pollution. Their solution utilizes a structure called evicted address filter (EAF) which keeps track of addresses that were recently evicted cache blocks. It predicts a high reuse when a block that misses was present in the EAF. When a block is missed in the cache and found in the EAF, then it is placed at MRU position in the cache. In case the block is missed in EAF, a bimodal insertion policy is used for the placement [29]. In order to reduce the hardware overhead, the authors proposed to implement EAF using a *Bloom* filter [49] to store recently evicted addresses.

They used a 3-level cache hierarchy, with a non-inclusive cache in all levels. For single-core, they simulated SPEC CPU2000, SPEC CPU2006, 3 TPC-H queries, TPC-C server, Apache webserver and D-EAF (EAF using Bloom filter). Their approach performs 7% better than LRU in terms of IPC. For multi-core simulation, the weighted speedup is 15% over LRU and 8% over SHiP.

3.6 Perceptron Learning for Reuse Prediction

Different applications behave differently and have different memory access patterns. Even a single application has different phases during its execution, where the patterns can also change. Machine learning has demonstrated to be useful in learning those patterns and to predict the next expected behavior, for example, in branch prediction [50]. *Perceptron* is one of the algorithms used for binary classifications to decide if something is going to happen [51].

Teran et al. proposed using the perceptron learning algorithm to predict the reuse

of a block [52]. In state-of-the-art techniques such as SDBP and SHiP, they use a single feature to make the prediction. However, the use of multiple features, such as the PC, some bits from the memory address and the trace of memory instructions, can be used to improve the accuracy of the reuse prediction. These features train a distinct table of saturating counters which is then summed. If the sum exceeds the threshold, the block is predicted not to be reused, and it is bypassed from the cache. The correlation of each feature with the reuse of the block is calculated by training the perceptron tables for each feature.

This technique used a sampler and six tables for prediction, each one with 3-bit saturating counters. The hardware overhead is 10.75 KB which is less than other state-of-the-art replacement policies: SHiP has 11.25KB of overhead and SDBP, 11.06KB. The original SDBP and SHiP studies did not include prefetching. However, this technique was evaluated using stream prefetching.

For single-thread workloads, reuse prediction achieves a geometric mean speedup of 6.1% compared to 3.8% for SHiP and 3.5% for SDBP. For multiprogrammed workloads, the geometric mean normalized weighted speedup is 7.4% compared to a 4.4% for SHiP and 4.2% for SDBP.

3.7 Hawkeye

Most of the research on cache replacement policies is based on heuristics, such as LRU. Those heuristics are used when they get good performance on most programs, but never all of them. LRU might be the best for a few benchmarks and then, MRU for others. This happens with more complex algorithms as well, each heuristic-based replacement policy works better for a set of access patterns but not in others.

Akanksha and Lin proposed *Hawkeye*, a cache replacement policy based on Belady's algorithm instead of on heuristics [53]. They proposed to use Belady's algo-

rithm [28] on past cache access to decide the future behavior of the cache blocks.

They define as *OPT* the decisions made by Belady's algorithm. The idea behind their approach is to look at a long past history of cache accesses and decide by predicting which cache lines will probably hit. They do that by checking whether in the OPT solution a load instruction brought lines that hit, and those are predicted to hit in the future too if the block stays in the cache.

They evaluated Hawkeye with the SPEC CPU2006 benchmark suite and the CMP$im simulator. Their model has a three-level cache hierarchy that is non-inclusive. For single-core simulations, Hawkeye's speedup is 8.4% over LRU, whereas SHiP and SDBP performance is 5.6% and 6.2% respectively. That translates to an average miss reduction of 17% on 20 memory intensive benchmarks, while SDBP, SHiP performs 11.4% and 11.7% respectively. In multi-core simulations with two cores and a shared LLC, Hawkeye achieves a 13.5% speedup while SHiP and SDBP only achieve 10.7% and 11.3% speedup. In multi-core simulations with four cores the advantage of Hawkeye is higer: 15% speedup compared to 11.4% and 12.1% for SHiP and SDBP respectively.

3.8 Bypass and Insertion

LRU and other replacement policies implement usage recency and usage frequency to determine the most likely dead blocks. To order the blocks by its recency, every-time there is a hit in the cache we update the state of the blocks to reflect their recency. In an exclusive last-level cache it is not possible to keep track of those properties because a hit in the last level cache causes an eviction in that level instead of promoting it. The only order we can rank the blocks of an exclusive cache would be the fill order, but that has little correlation with the recency. They assume a 3-level cache, where the L3 is exclusive from L2.

Gaur et al. propose to use an estimation of the average recall distance of L3 blocks (being the L3 the last-level cache) and their use count in the L2 cache [54]. They define the average recall distance as the mean number of L3 allocations between the allocation of a block B in the LLC and the recall of B from the L2 cache. That means, if we have a block B that is evicted from the L2, it will be filled in the LLC. All the allocations between that fill of B in the LLC and a hit of B in the LLC are going to be counted towards that average. The use count is the number of times there is a hit in a block since its fill in that cache level. They define the trip count as the number of trips the block goes from the L2 to the L3 (L3 hit) since the first hit in L3.

They found that the best way of identifying dead/live blocks was by using the information from the trip count together with the L2 use count. They decide based on a few sampler sets. They use the liveness of the block to decide whether to bypass and with what age to insert the block. They combine this technique with set dueling that always uses the bypassing.

They proposed 3 insertion techniques using the trip count (L3 hit) and use count (L2 hit). They used an aggresive multi-stream prefetcher. Their best technique improves the IPC of single-threaded applications by 3.4% compared to a baseline "not-recently-filled" replacement policy. The multiprogrammed mixes improved by 2.5%.

3.9 Hierarchy-Awareness and Bypass

Most replacement policies for LLCs are designed to use the information from that same level of cache without any knowledge of the inner levels statistics. A replacement of an LLC cache block in an inclusive cache can replace and invalidate a block that is continuosly hitting in the L1 but never in the LLC. This is not a

desirable behavior.

Chaudhuri et al. proposed a cache hierarchy-aware replacement policy (CHAR) for inclusive LLCs and bypass for exclusive LLCs [55]. They dynamically estimate the reuse probability of a block based on the L2 reuse pattern to hint the LLC replacement policy. In an inclusive cache, it will hint the LLC to mark the block as the next victim if the pattern indicates that the next reuse is beyond the LLC reach. In an exclusive cache, it will help decide whether to bypass the cache block from the LLC.

The authors implement the CHAR replacement policy in the LLC. They use a subset of L2 evictions that correspond to 16 LLC sample sets to inform the dead hint detector of CHAR. The LLC sample sets use a 2-bit SRRIP-HP replacement policy: the placement of a block sets the RRPV value to two, and the promotion on a hit to zero. To learn from the reuse behavior, the authors proposed four attributes, listed below, to classify the L2 cache blocks in five classes (described in Table 3.2). All L2 blocks contain two extra bits representing the 5 classes.

- A_0: prefetch or demand
- A_1: hit or miss
- A_2: number of demand uses in L2
- A_3: L2 coherence state (in a MESI protocol)

They included two saturating counters per class: "E" that contains the total number of evictions of that class and "L" that contains the total number of hits per class. Additionally, they have a saturating counter "N" with the total number of evictions. Every time there is an eviction from L2 of one of the blocks that map to the sample sets in the LLC, the counter "E" of the block class and "N" are incremented.

Evictions of blocks of class 0 to 3 that do not map to sample sets in the LLC, invoke the dead block detection algorithm. On an L2 fill of a block mapping to a sample set, the LLC sends which class the block belongs to and increments the hit counter of that class when it hits in the LLC. If it is a miss or the block is not in a sample set, the L2 sets the class of the block depending on whether it is a prefetch or not and if it is a hit or miss in the LLC.

Their dead block detection algorithm uses the saturating counters to dinamically choose a threshold that has to be smaller than dividing the total number of hits by the total number of evictions of each class. For example, one threshold could be the hit rate of the LLC. In other words, the block belonging to a class are considered dead when the L2 evictions exceed the number of hits in LLC for blocks of that class by a certain ratio.

An exclusive LLC design needs more interconnect bandwidth than an inclusive design due to copying all L2 evictions -clean or dirty. The authors claim that their CHAR algorithm can be used for selective bypassing blocks from the LLC that are likely to be dead. CHAR, in an exclusive LLC, every L2 eviction address is first sent to the coherence directory together with the dead hint. If the block is marked as alive, the block is then filled to the LLC. If the block is marked as dead, it will only be filled to the LLC if there is an invalid way, and it will be filled with age three (in immediate danger of eviction). Additionally, to avoid the ping-pong effect, a block that is marked as alive will be stored in the LLC in non-inclusive mode -it will not cause an eviction on LLC hit.

To evaluate the results, they used SRRIP as a baseline with a multi-stream prefetcher. They evaluated single-threaded, multiprogrammed and shared memory workloads. In an inclusive cache design, CHAR improves by 5.3% on average for 100 4-way multiprogrammed mixes over the baseline. In an exclusive design, it im-

Class	A_0	A_1	A_2	A_3
C_0	Prefetch	Miss	0	E/S
C_1	X	Miss	1	E/S
C_2	X	Miss	1	M
C_3	X	Miss	≥ 2	X
C_4	X	Hit	X	X

Table 3.2: L2 cache block classification in the CHAR algorithm.

proves 8.2% on average compared to an identical inclusive design (with about 66% bypasses).

4. METHODOLOGY

In this chapter, we explain the methodology used to evaluate our work. First, we explain what we used to perform our tests and, second, how we evaluated them.

In this thesis, we focus on single-threaded applications. How to extend the methodology to multiple threads remains as future work.

4.1 Experimental Setup

This section describes the experimental setup used in this thesis.

4.1.1 Host Machine

We ran most of our experiments in the Terra supercomputer, part of the Texas A&M University supercomputing facilities. This supercomputer uses Slurm as workload manager [56].

Part of this research was also done on the private cluster of the Texas Architecture and Compiler Optimization (TACO) research group. This is a group of heterogeneous machines with no workload manager. The cluster includes different models of Intel and AMD machines.

4.1.2 Benchmarks

We used 18 memory intensive traces from the SPEC CPU2006 benchmark suite [6] (listed in Table 4.1), three single threaded server workload traces from CloudSuite [7] (listed in Table 4.2), and a trace from a machine learning workload "mlpack_cf" [8]. In total, 22 benchmarks.

All the SPEC CPU2006 traces were collected with SimPoint [57]. The CloudSuite and ml_pack traces were collected after fast-forwarding at least 30 billion instructions.

The execution of the traces is divided in two stages: warm-up and timing modeling. The warm-up phase serves to update the machine state before starting timing simulation so that the state is similar as what it would be if we had simulated all the instructions until the point timing simulation starts. For example, in the warm-up phase, the caches and branch predictors are updated. In this thesis, the warm-up phase consists of 200 million instructions. The timing modeling phase is the detailed simulation that counts towards statistics, such as IPC. In this thesis, the simulation phase consists of one billion instructions. There are traces that are shorter than one billion instructions, for those, the trace re-starts from the beginning until completing that fixed amount of instructions.

Single-core configurations are run with a fixed instruction count of 200 million of warm-up plus one billion of timing simulation.

Multi-core simulations execute a single benchmark per core. In this case, all traces warm-up until all finish 200 million instructions. For the timing modeling phase, all cores run until all have run at least one billion instructions, also known as *last* [58]. So, some cores will run more than one billion instructions but only the first billion will count towards the statistics. This methodology is used so that all cores are running at the same time during all the execution. This is important to model the effects of a shared LLC in a more realistic environment.

For the single core simulations, we ran all cache configurations with the 22 benchmarks. For the multi-core simulations, we used multiprogrammed workloads on four cores. We used nine mixes with four traces each. The mixes were generated randomly. Table 4.3 lists all mixes.

astar	libquantum
bwaves	mcf
bzip2	milc
cactusADM	omnetpp
gcc	soplex
GemsFDTD	sphinx3
gromacs	wrf
lbm	xalancbmk
leslie3d	zeusmp

Table 4.1: SPEC CPU2006 memory intensive benchmarks.

data_caching
graph_analytics
sat_solver

Table 4.2: CloudSuite benchmarks.

Mix 1	astar	leslie3d	soplex	zeusmp
Mix 2	mcf	milc	omnetpp	sat_solver
Mix 3	astar	cactusADM	leslie3d	zeusmp
Mix 4	cactusADM	gromacs	sphinx3	sat_solver
Mix 5	cactusADM	gromacs	lbm	milc
Mix 6	cactusADM	GemsFDTD	lbm	leslie3d
Mix 7	astar	bzip2	sphinx3	data_caching
Mix 8	bwaves	milc	data_caching	sat_solver
Mix 9	bzip2	omnetpp	data_caching	graph_analytics

Table 4.3: Multiprogrammed workloads mixes for simulating 4 cores.

4.1.3 Simulator

We used the ChampSim simulator, which is an extended version of the simulator used in the 2^{nd} Data Prefetching Championship [59] and recently used in the 2nd Cache Replacement Championship [5]. This simulator models a simple multi-core out-of-order.

The configuration to model the multi-core we used is described in Table 4.4. The baseline consists of a 3-level cache hierarchy with all levels following a non-inclusive policy. All caches use copy back with write allocate write policies.

Parameter	**Configuration**
L1 I-cache (private)	32KB, 64B blocks, 8-way, 8 MSHRs, 1 cycle latency, 64 read/write/prefetch queue size
L1 D-cache (private)	32KB, 64B blocks, 8-way, 8 MSHRs, 4 cycles latency, 64 read/write/prefetch queue size
L2 unified cache (private)	256KB, 64B blocks, 8-way 16 MSHRs, 8 cycles latency, 32 read/write/prefetch queue size

Table 4.4: Simulator configuration.

Parameter	Configuration
	non-inclusive
L3 unified cache (shared)	1MB per core, 64B blocks, 16-way 32 MSHRs, 20 cycles latency, 16 per core read/write/prefetch queue size non-inclusive
Frequency	4GHz
Page size	4KB
Fetch, decode and retire	4 wide
Execution	6 wide
Load Queue	2 wide
Store Queue	1 wide
DRAM row precharge latency	11 cycles
DRAM row address to column address latency	11 cycles
DRAM column address strobe latency	11 cycles

Table 4.4: Continued

Parameter	Configuration
DRAM	2 channels (1 DIMM per channel), 8 banks (64MB per bank), 8 ranks (512MB per rank), 4GB per DIMM
DRAM channel width	8
DRAM I/O frequency	800MHz
Branch Predictor	Perceptron
Reorder Buffer size	256
Pipeline depth	5

Table 4.4: Continued

4.2 Evaluation

This section describes which cache configurations we simulated. The parameters we changed are: prefetcher, replacement policy, inclusion policy and number of cores.

4.2.1 Configurations

We evaluated several combinations of prefetchers, replacement policies and inclusion policies for both single-threaded and multiprogrammed workloads.

The evaluated L2 prefetchers are six: no prefetcher, ip_stride, next_line (in section 2.4.1), bop (in section 2.4.3), daampm (in section 2.4.4) and kpcp (in sec-

tion 2.4.5). The evaluated L1 prefetchers are two: no prefetcher and next_line.

The evaluated replacement policies are six: LRU (in section 2.3.1), EAF (in section 3.5), KPC-R (in section 2.4.5), SHiP (in section 3.3), SRRIP and DRRIP (in section 3.1).

The total number of simulations ($nsims$) are the product of all the prefetchers and replacement policies. Equation 4.1 shows the calculation of the total number of simulations for single-threaded simulations, which is 4752.

$$\begin{aligned} nsims &= \text{nincs} \times L1p \times L2p \times RP \times b \\ &= 3 \times 2 \times 6 \times 6 \times 22 \\ &= 4752 \end{aligned} \tag{4.1}$$

$$\begin{aligned} nsims &= \text{nincs} \times L1p \times L2p \times RP \times m \\ &= 3 \times 2 \times 6 \times 6 \times 9 \\ &= 1944 \end{aligned} \tag{4.2}$$

4.2.2 Performance Measurement

The baseline configuration used as reference in our evaluation has the following features: non-inclusive cache inclusion policy, no prefetcher in neither the L1 nor L2, and the LRU replacement policy.

For the single-core, single-thread simulations, we compute the speedup of a configuration i over the baseline running a particular benchmark by extracting the instructions per cycle (IPC) of configuration i (see section 4.2.1) and dividing it by the IPC of the baseline [60]. Equation 4.3 shows the speedup calculation used.

$$Speedup_i = \frac{IPC_i}{IPC_{baseline}} \tag{4.3}$$

For the multi-core simulations, we compute the speedup of a configuration i over the baseline running a particular mix by computing the average IPC across all threads and dividing it by the average IPC across all thread in the baseline. Equation 4.4 shows the speedup calculation used.

$$Speedup_i = \frac{AverageIPC_i}{AverageIPC_{baseline}} \tag{4.4}$$

In this chapter, we first explain the organization of the ChampSim simulator's code.

5.1 ChampSim Code

We simulate a cache hierarchy with 3 levels of cache. The cache is write back and write allocate. The traces are all single-threaded. To simulate more than one core we use workloads of several single-threaded benchmarks (multiprogrammed).

In this thesis, we only modified the cache component in ChampSim. Figure 5.1 shows the three main functions responsible to operate the cache.

```
1 cache_operate() {
    handle_fill();
3   handle_writeback();
    handle_read();
5 }
```

Figure 5.1: Main functions on ChampSim to operate the cache.

5.1.1 Cache Operation

The main operations that a cache level has to handle are: fills, reads and writes. When the cache has to handle a fill, it is a cache block that is pending to be added to that cache level from either a read that came back from the higher level cache or a write back from the lower level.

The *handle_fill()* function is the responsible of:

- Finding a victim block before allocating the new one
- Deciding whether to bypass the cache, if bypass is enabled
- Writing back the victim block, if it was dirty
- Allocating the block and freeing MSHR entry

The *handle_writeback()* is the responsible of:

- If it is a hit, writing the new data and mark the block as dirty
- If it is a miss:
  - Finding a victim block before allocating the new one
  - Writing back the block, if the victim was dirty
  - Allocating the block as dirty

The *handle_read()* function is the responsible of:

- If it is a hit, it returns the value
- If it is a miss, it adds the request to the next cache level MSHR

5.1.2 Non-Inclusive Implementation

The default implementation on this simulator is the "non-inclusive". Whenever there is a miss in a cache level, that cache level and all the higher levels are filled with the block. For example, an LLC miss will fill the block in all cache levels. When evicting a block at any cache level, there is no other effect than evicting the block from that cache level.

	L1	**L2**
Hit	- Return block to CPU - Update replacement and prefetcher	- Return block to L1 - Update replacement and prefetcher
Miss	- Request block to L2 - Evict previous block? - Allocate block - Return block to CPU - Update replacement and prefetcher	- Request block to memory - Evict previous block? - Allocate block - Return block to L1 - Update replacement and prefetcher
Evict	Clean: - Invalidate block Dirty: - Write back to L2 Invalid: –	Clean: - Invalidate block Dirty: - Write back to memory Invalid: – Always: –
Write back	–	Hit: - Write block Miss: - Evict block? - Write block

Table 5.1: Model of an non-inclusive two-level cache.

5.1.3 Statistics

The statistics used to compare our cache models are the following.

Eviction stats:

- The number of all evictions in every cache level.
- The number of evictions due to inclusion policies: for inclusive, all back invalidations; for exclusive, the invalidations on hit; for non-inclusive there are none.
- The number of dirty evictions in every cache level.

General cache stats, one per cache level:

- The number of accesses, and how many are reads and writes.
- The number of hits and misses, and how many are reads and writes.
- The misses per kiloinstruction (MPKI).

5.2 Modifying ChampSim

We modified the Champsim simulator (see section 4.1.3), which implemented a non-inclusive cache, to model inclusive and exclusive caches. This section describes all the modifications made to the code assuming an already implemented non-inclusive cache.

5.2.1 Inclusive Implementation

In this implementation, we model both the L2 and the L3 to be inclusive. The implementation was done on top of a non-inclusive default configuration. Table 5.2 shows a more detailed description with the main changes with respect to the non-inclusive model in bold.

	L1	**L2**
Hit	- Return block to CPU - Update replacement and prefetcher	- Return block to L1 - Update replacement and prefetcher
Miss	- Request block to L2 - Evict previous block? - Allocate block - Update replacement and prefetcher	- Request block to memory - Evict previous block? - Allocate block - Return block to L1 - Update replacement and prefetcher
Evict	Clean: - Invalidate block Dirty: - Write back to L2 Invalid: –	Clean: - Invalidate block Dirty: - Write back to memory Invalid: – **Always: - Invalidate request to L1**
Write back	–	Hit: write block Miss: (should not happen)

Table 5.2: Model of an inclusive two-level cache.

In an inclusive cache, the difference with the default non-inclusive implementation is that whenever there is an eviction in a cache level, all the lower levels need to be evicted as well. For example, an eviction in the L3 of a block that is present in both L1 and L2 will invalidate the block in all three levels. This back invalidation needs to be done before evicting that block.

We implemented inclusivity right after finding a victim block to be replaced. This happens when handling a fill and when handling a write back miss. However, in an inclusive cache, there should not be any write back misses because all blocks in a lower level must also be present in the higher levels. We added an assert on a write back miss to make sure this does not happen. Therefore, to implement inclusivity we only focus on the fill.

If the block was dirty in any of the levels, we also have to ensure that the block that is written to memory is the one in the lower level. If a block is dirty in L1 and L2, we need to write the block present in L1 because it is the most updated block.

We disabled the possibility of bypassing the cache. In the non-inclusive cache, bypassing was enabled in the LLC. In an inclusive cache all blocks in the lower levels must be always in the LLC to keep inclusivity so bypassing is not an option. In our simulations, bypassing was enabled only in non-inclusive and exclusive.

In a multi-core processor, a block might be in any of the core-private caches. In this case, on an LLC eviction, we need to back invalidate all lower level caches private to all cores.

Figure 5.2 shows a high-level code of this implementation.

Figure 5.3 shows what happens when for all cases on an LLC miss. This figure was created to make sure that our implementation does not leave any case out. The cases with a red cross are the ones that trigger an assert due to not complying with inclusivity. For example, a block that is valid (either clean or dirty) in the LLC and

```
back_invalidate(cache_level, addr) {
  if (victim.isValid()) {
    if (cache_level == L3) {
      for (i=0; i<NCORES; i++) {
        if (L2.present(core[i], addr)) {
          if (L1.present(core[i], addr)) {
            L1.invalidate(core[i], addr);
          }
          L2.invalidate(core[i], addr);
        } else {
          assert(not L1.present(core[i], addr));
        }
      }
      L3.invalidate(addr);
    } else if (cache_level == L2) {
      if (L1.present(addr)) {
        L1.invalidate(addr);
      }
      L2.invalidate(addr);
    }
  }
}

handle_fill() {
  ...
  victim = find_victim();
  back_invalidate(cache_level, victim.getAddress());
  ...
}
```

Figure 5.2: Inclusive cache high-level code on top of the non-inclusive implementation.

in L1 must also be valid in L2. The green-numbered cases are the valid ones where the inclusion policy has to perform an action.

5.2.2 Exclusive Implementation

In this implementation, we model the L2 to be non-inclusive of L1, and the L3 to be exclusive of L2. The implementation was done on top of a non-inclusive default configuration. Table 5.3 shows a more detailed description, with the main changes with respect to the non-inclusive model in bold.

In an exclusive cache, the main differences are that a miss to the L1 or L2 does not generate a fill in the LLC and on an LLC hit, the block is invalidated. Also, on an L2 eviction the block is written back to the LLC whether the block was dirty or clean (also known as copy back). The high level implementation of this model is as follows:

- Allocate
  - in L3 on an L2 eviction
  - in L2 on a fill or prefetch (bypass L3)
- Evict
  - from L3 on an L3 hit
  - from L2 on a replacement

Our modifications to implement the exclusive cache were on the fill, read and write back functions.

On an LLC fill, we always bypass the block. On an L2 fill, instead of just invalidating the block if it was clean, we always send it to the LLC. In the default non-inclusive cache, L2 fills write to the LLC only the victim blocks that are dirty .

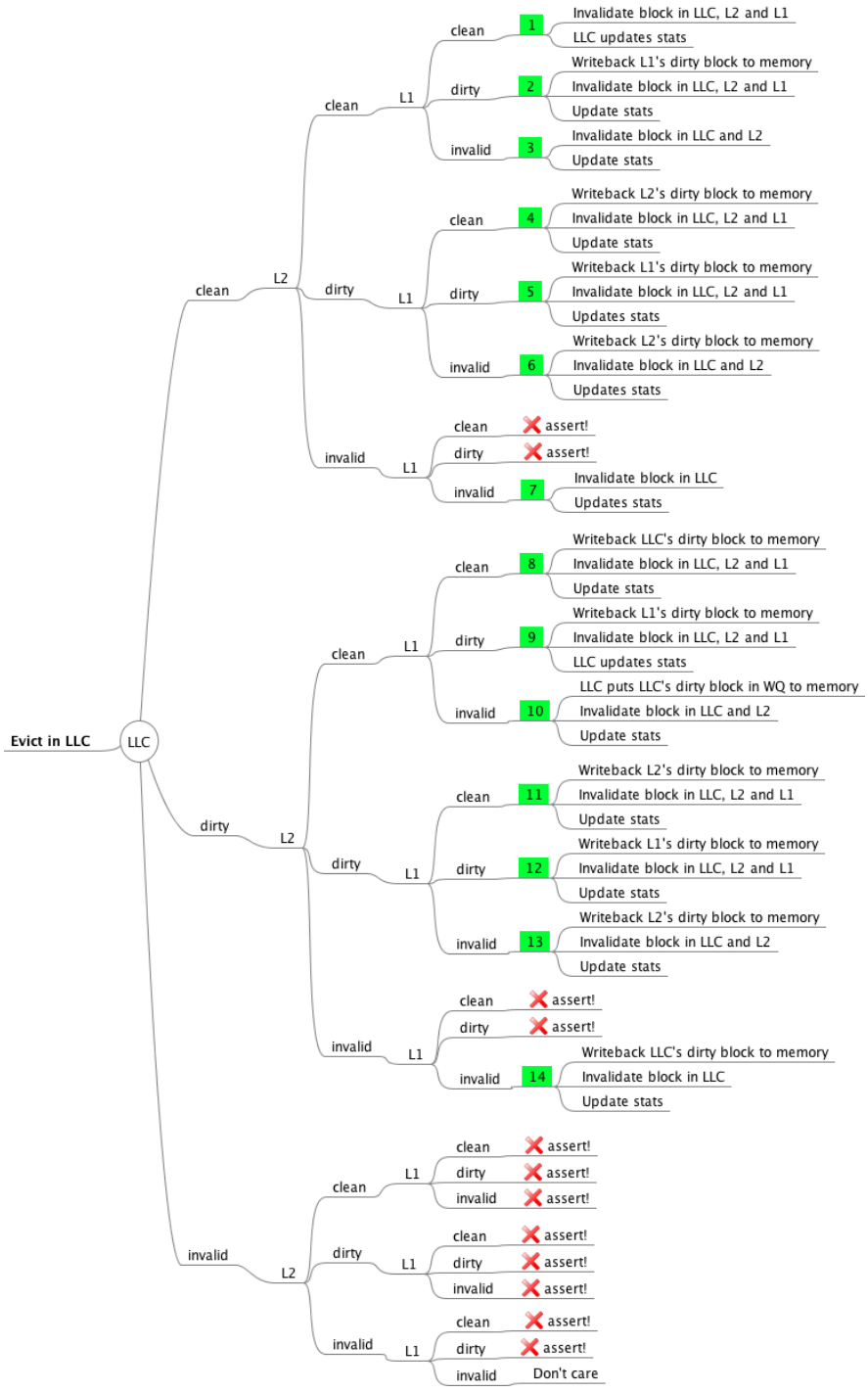


Figure 5.3: Diagram with all the possible cases on an LLC miss in an inclusive cache on a single core simulation.

	L1	**L2**
Hit	- Return block to CPU - Update replacement and prefetcher	- Return block to L1 - **Invalidate**
Miss	- Request block to L2 - Evict previous block? - Allocate block - Update replacement and prefetcher	- Request block to memory - **Return block to L1**
Evict	Clean: - **Write back to L2** Dirty: - Write back to L2 Invalid: –	Clean: - Invalidate block Dirty: - Write back to memory Invalid: – Always: –
Write back	–	Hit: – Miss: -Evict block? **-Write new block**

Table 5.3: Model of an exclusive two-level cache.

On an LLC read, we send the block to the L2 and invalidate the copy in that LLC.

On a write back, we modified the packet that is sent to the LLC to contain the information of whether the block was dirty or clean. In the default non-inclusive implementation, it assumes that, on a write back, the block was dirty in the cache were it was evicted. Then, for clean write backs (or copy backs) in the exclusive case, we implemented that on an L2 write back, the block is written back together with the corresponding dirty bit, so the exclusive cache can allocate it with the correct state: whether it is dirty or clean.

Figure 5.4 shows a high level implementation of the code, with only the modifications over the non-inclusive.

5.3 Scripts

For the work of this thesis, we prepared and used several scripts. As explained in Section 4.2.1, the numbers of simulations that we ran for one and four cores were high. On top of that, as mentioned in Section 4.1.1, we ran part of those simulations in two different clusters. One has a workload manager and the other does not.

It was necessary to have a framework of scripts to automatize all the process in all stages. All the main stages of this process are: compile the code for each configuration, run the simulations, gather all results in a parseable file, compute intermediate results (e.g. speedup, geometric mean) and plot the results.

Most of the scripts were done in bash and iterate over all possible cache configurations. We explain below the most interesting ones that are to execute the simulation in different clusters and the one that generates the plots.

5.3.1 Execute

To run the simulations we prepared two different scripts. One to run the simulations in the Terra supercomputer and another to run them in our cluster.

```
handle_fill() {
  ...
  if (L3) {
    // bypass
  }

  if (L2) {
      // copy back victim to L3
  }
  ...
}

handle_read() {
  ...
  if (hit and L3) {
    // send block to lower level
    L3.invalidate(block);
  }
  ...
}

handle_writeback() {
  ...
  if (L2) {
    // copy back victim to L3
  }
  ...
}
```

Figure 5.4: Exclusive cache high-level code on top of the non-inclusive implementation.

The Terra supercomputer uses Slurm as a workload manager. To run jobs in this machine, it is necessary to create a job script with several parameters. For example, some of these parameters are: number of cores required, time limit (after this time passes, the job will be killed), amount of memory per node required and output file name. After those parameters, we add the commands to run a simulation for a single configuration and workload. Our script iterates over all configurations and traces, generates a job script with the appropriate parameters for each configuration and submits it for execution.

The cluster of our lab does not have a workload manager, so we need to do an equivalent work in our script. We used *screen* to run this script so it keeps running after we close our connection to the cluster. We chose to use 22 machines from the cluster. The script contains the names of all those machines. As before, the script iterates over all configurations and traces, but checks the machine before executing the simulation. First, we check that the connectivity through *secure shell (SSH)* is good. Then, we check that the CPU load of the machine is under a certain threshold. This is to limit the number of simulations per machine and not to overuse a machine if other researcher in our group is using the same machine at the moment. If there is no connectivity or the CPU load is high, we try a different machine. If there was no available machine after a few tries, the script sleeps for a while, and then it tries again. Whenever the scrip finds an available machine, it runs the simulation via an *SSH* tunnel.

5.3.2 Plots

We used bash and python scripting languages to parse the result files and create the plots. The output of ChampSim includes the number of instructions and cycles executed, cache-related statistics among other information about the configuration,

progress and finalization of the simulated run. The bash script generates, for single-threaded simulations, a comma separated value (CSV) file including the number of cycles, the IPC among other statistics for each combination of benchmark, L1 prefetcher, L2 prefetcher, LLC replacement policy and inclusion policy. For multi-core simulations, the output file includes the average IPC of the mix instead of the IPC of each individual thread.

The python script reads, for single-threaded simulations, the CSV file and stores its contents in a *pandas* DataFrame for each benchmark. The IPC of each hardware configuration is divided by that of the baseline for each benchmark, resulting in a normalized set of data. The metric of interest is normalized execution time, also known as speedup. The script then computes the geometric mean across all benchmarks' speedup and plots it. For multi-core workloads, the same procedure computes the geometric mean across all mixes using the average IPC across threads instead of the IPC of a single thread as in the single-threaded runs.

6. RESULTS

In this chapter we present the results obtained during this thesis. First, we analyze single- and multi-core results of all cache configurations previously described in Section 4.2.1. Then, we analyze the same for single-core but with a larger cache capacity in all levels to compare sensitivity. Last, we compare all those results and discuss them.

6.1 Single-Core Results

This section presents the results of the single-core simulations. Appendix A contains all results per benchmark.

Figure 6.1 shows the geometric mean speedup of multiple cache configurations across all benchmarks. The baseline in this figure is: no L1 or L2 prefetcher, LRU replacement policy and non-inclusive policy.

The following subsections analyze the results from Figure 6.1 in terms of inclusion policies, prefetchers and replacement policies.

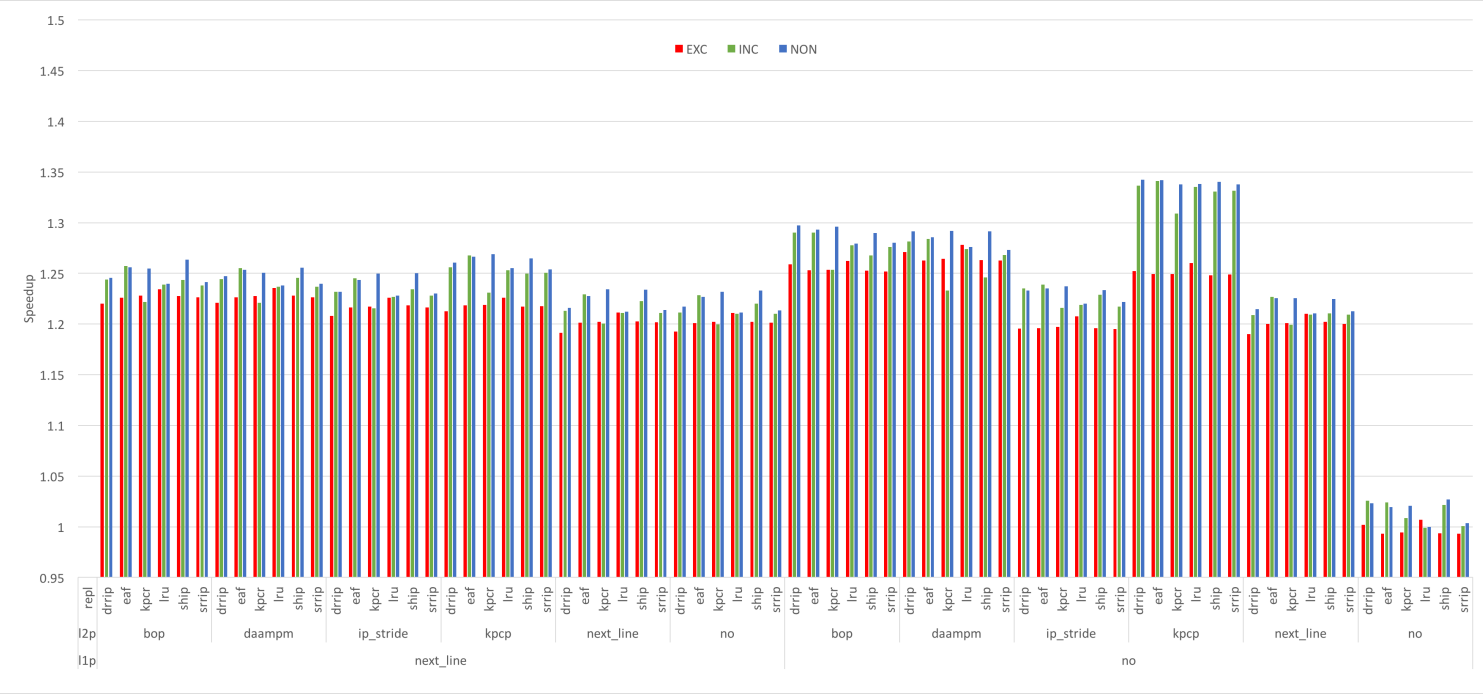


Figure 6.1: Geomean speedups to compare different configurations of L1 and L2 prefetchers, replacement policies and cache inclusions. The configurations compared are: L1 prefetcher, L2 prefetcher, replacement policy and cache inclusion. The Y-axis shows the speedup over the baseline configuration: no prefetchers, LRU replacement policy and a non-inclusive cache. The X-axis shows the different cache configurations, in order of: L1 prefetcher (l1p), L2 prefetcher (l2p) and replacement policy (repl).

6.1.1 Inclusion Results

The best configurations for non-inclusive and inclusive caches, and overall, are those without L1 prefetcher and with the KPC-P prefetcher in the L2. The six best configurations are all with that prefetcher combination. The best two configurations for both inclusive and non inclusive are using DRRIP (non-inclusive gets 34.2% speedup and inclusive an 33.6%) and EAF (both inclusive and non-inclusive get 34.1% speedup).

However, an exclusive LLC shows a different behavior. The best six configurations for an exclusive cache are those without an L1 prefetcher and with the DAAMPM prefetcher in the L2. The best one uses the LRU replacement policy, which gets a 28% speedup over the baseline. The next best configurations is with DRRIP, and the other four are about the same (KPC-R, SHiP, SRRIP and EAF). The exclusive policy, in general, performs worse than inclusive and non-inclusive policies.

6.1.2 Prefetcher Impact

Prefetching clearly has an impact on performance. The worst configurations are the ones without prefetching in any of the cache levels. The exclusive cache is the one that suffers the most of not having any prefetchers by getting a slowdown in all replacement policies except for LRU and DRRIP.

Generally, not using a prefetcher in L1 seems to get better results than using the next-line prefetcher. This might be due to interference between prefetchers. The best performing configurations for all inclusion types are all without an L1 prefetcher.

As mentioned in the previous section, the best configurations for each inclusion type are given from the prefetcher combination. This shows that many benefits of using different replacement policies are shadowed by the prefetcher.

6.1.3 Replacement Policy Impact

There is no clear winner among replacement policies, but there are a few patterns that indicate that the LRU replacement policy is among the best options to use in general for an exclusive cache independently of the prefetchers.

All inclusion policies have similar performance using the LRU replacement policy except when using the KPC-P prefetcher. This mostly comes from benchmarks such as bzip2, omnetpp, sphinx3 and xalancbmk (see A).

The KPC-R replacement policy makes the inclusive cache perform worse than the non-inclusive and, in most of cases, worse than exclusive. In other replacement policies, the inclusive and non-inclusive have more similar behavior.

The LRU replacement policy gets typically similar or worse performance for many inclusive and non-inclusive cache configurations among a combination of prefetchers. However, for exclusive caches, LRU gets generally the best performance among a prefetcher combination.

Furthermore, exclusive caches do not show much sensitivity to the replacement policy for a prefetcher combination. In contrast, inclusive and non-inclusive are more variable.

6.2 Multi-Core Results

This section presents the results of the multi-core simulations. Appendix B contains all results per benchmark.

Figure 6.2 shows the geometric mean speedup of all cache configurations in multi-core. The baseline in the figure is: no L1 and L2 prefetchers, LRU replacement policy and all non-inclusive caches.

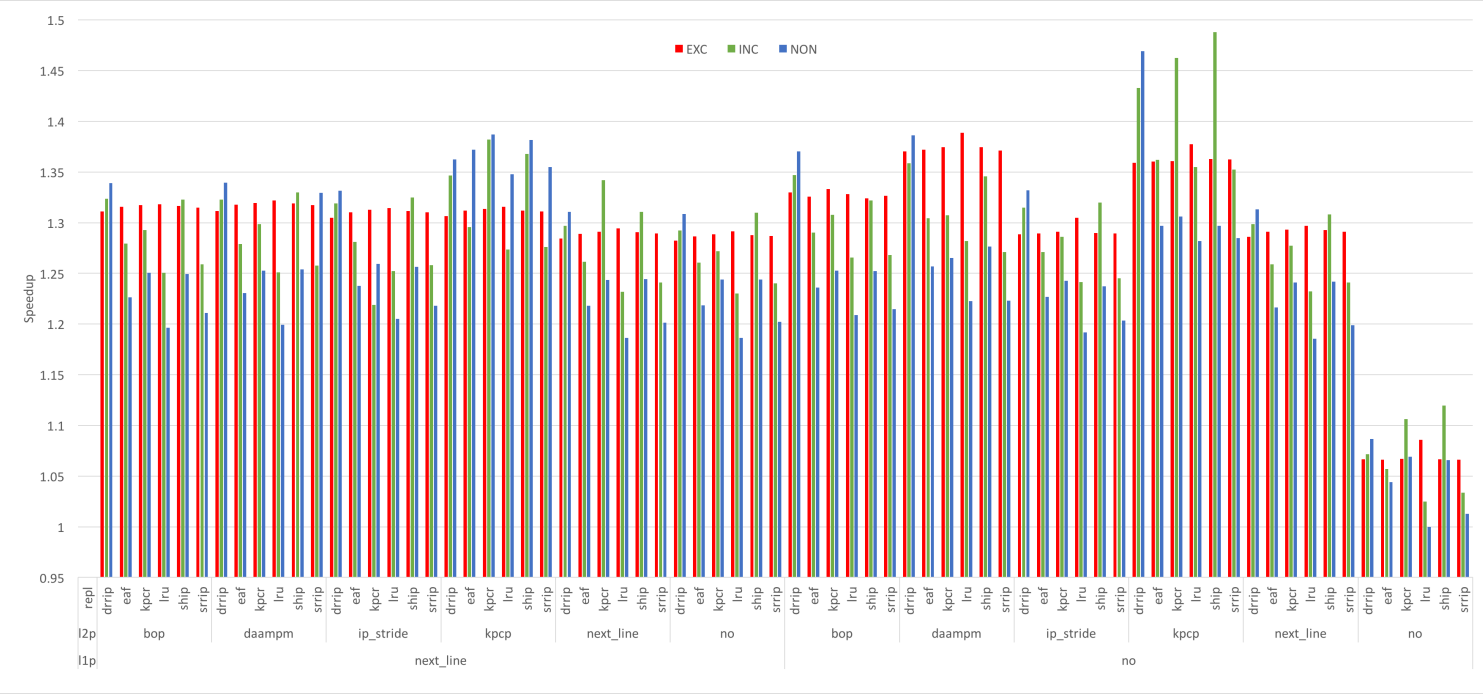


Figure 6.2: Geomean speedups to compare different configurations of L1 and L2 prefetchers, replacement policies and cache inclusions. The configurations compared are: L1 prefetcher, L2 prefetcher, replacement policy and cache inclusion. The Y-axis shows the speedup over the baseline configuration: no prefetchers, LRU replacement policy and a non-inclusive cache. The X-axis shows the different cache configurations, in order of: L1 prefetcher (l1p), L2 prefetcher (l2p) and replacement policy (repl).

6.2.1 Inclusion Results

The best configuration for non-inclusive is: no L1 prefetcher, KPC-P L2 prefetcher and DRRIP, with a 46.9% speedup over the baseline. Most of the next best configurations are with next-line L1 prefetcher, KPC-P L2 prefetcher with different replacement policies. The second best configuration is using that prefetcher configuration with KPC-R, but among the rest DRRIP stands out.

The best configuration for an inclusive cache, however, is different to the non-inclusive in contrast with single-core. The best configuration for inclusive and overall is: no L1 prefetcher, KPC-P L2 prefetcher and SHiP replacement policy, with a 48.8% speedup over the baseline. The second best is the same as the best configuration for the non-inclusive but without L1 prefetcher, achieving a 46.3% speedup. The third best is finally the same configuration as the best one for non-inclusive, achieving a 43.3% speedup.

The best configuration for an exclusive cache is, like for single-core, different from the other inclusion policies. The best configuration is exactly the same as for single-core: no L1 prefetcher, DAAMPM L2 prefetcher and LRU replacement policy, with a 38.9% speedup over the baseline. The second best configuration is the same but changing the L2 prefetcher by KPC-P, with a 37.8% speedup. The next four best configurations are all the remaining replacement policies with no L1 prefetcher and DAAMPM L2 prefetcher, in order: KPC-R, SHiP, EAF and SRRIP.

To summarize, for inclusive and non-inclusive the L2 prefetcher that works best is KPC-P. The three similar and best replacement policies are DRRIP, KPC-R and SHiP. The main difference is that for non-inclusive, using a next-line L1 prefetcher is generally better than none, and for inclusive, the opposite. For exclusive, the replacement policy that works best is, again, LRU, and generally DAAMPM L2

prefetcher and no L1 prefetcher. Again, as for single-core configurations, both the prefetcher and the replacement policies that work best for an exclusive cache are different than from the ones in non-inclusive and inclusive caches.

6.2.2 Prefetcher Impact

As shown in the previous section, all the best cache configurations generally came from specific combinations of prefetchers. The exclusive and inclusive caches do better without L1 prefetcher while the non-inclusive does better with the next-line prefetcher. This is different than from single-core, where not using an L1 prefetcher was best for any configuration.

Regarding the L2 prefetcher, the KPC-P works better for inclusive and non-inclusive while the exclusive benefits more from the DAAMPM. However, the KPC-P is the second best option as an L2 prefetcher on an exclusive cache.

All the worst configurations are definitely the ones without no prefetcher at all, like for single-core. However, the exclusive policy gets more performance than the other inclusion policies in three replacement policies (EAF, LRU and SRRIP), in contrast to single-core where the exclusive was always the worst with a slowdown.

6.2.3 Replacement Policy Impact

The DRRIP replacement policy makes the non-inclusive policy perform better in most cases compared to any other replacement policy. SRRIP, when combined with next-line L1 prefetcher and DAAMPM, makes the non-inclusive perform better and even close to DRRIP performance.

LRU makes the non-inclusive policy to consistently perform the worst among all configurations except with the next-line L1 prefetcher and KPC-P L2 prefetcher. That configuration even outperforms the other inclusion policies. On the other hand, LRU is always the best among different prefetcher combinations on an exclusive

cache. On single-core, LRU makes all inclusion policies behave similarly. On multi-core, it makes the exclusive become the best, inclusive significantly worse and non-inclusive the worst, in general.

SHiP stands out in inclusive caches. In many prefetcher configurations in an inclusive cache, SHiP is the best compared to other replacement policies. It is also typically the best, for a design with SHiP, among other inclusion policies.

On exclusive caches, the replacement policy does not seem to make much impact on any combination of prefetchers similarly to single-core simulations. Inclusive and non-inclusive caches vary more when using different replacement policies.

6.3 Size Sensitivity

This section presents the results of the single-core simulations with a larger cache as explained below. Appendix C contains all results per benchmark.

Figure 6.3 shows the geometric mean speedup of multiple cache configurations across all benchmarks. The baseline in this figure is: no L1 or L2 prefetcher, LRU replacement policy and non-inclusive policy. In these simulations we modeled a larger cache hierarchy: 64KB L1, 512KB L2 and 2MB LLC, in comparison to the configuration used in the previous analyses: 32KB L1, 512KB L2 and 2MB LLC. Appendix C contains all results per benchmark.

We discuss these larger cache size results to the baseline cache size used in the single-core results (Section 6.1) in the following sections.

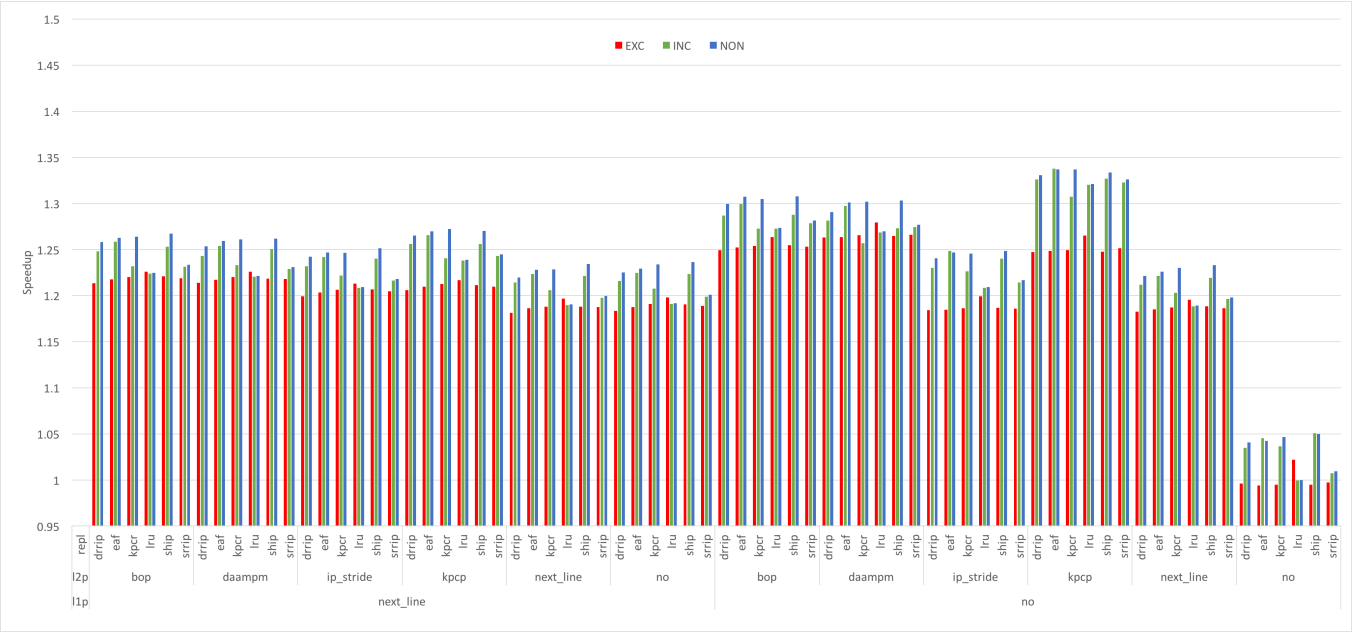


Figure 6.3: Geomean speedups to compare size sensitivity with different configurations of L1 and L2 prefetchers, replacement policies and cache inclusions on a large cache configuration. The size of each cache level is: 64KB L1, 512KB L2 and 2MB LLC. The configurations compared are: L1 prefetcher, L2 prefetcher, replacement policy and cache inclusion. The Y-axis shows speedup over the baseline configuration: no prefetchers, LRU replacement policy and a non-inclusive cache. The X-axis shows the different cache configurations, in order of: L1 prefetcher (l1p), L2 prefetcher (l2p) and replacement policy (repl).

6.3.1 Inclusion Results

One of the differences compared to the baseline cache size is that DRRIP is no longer among the best configurations for exclusive caches. In the baseline size, DRRIP with no L1 prefetcher and DAAMPM was the second best configuration. In the large size, while the others stay similar, DRRIP performance drops a little. Among the best configurations, KPC-P prefetcher and LRU is also among the best apart from the configurations with no L1 prefetcher and DAAMPM that was the preferred option for the baseline size.

Interestingly, on average, the speedups achieved by the exclusive cache with the smaller sizes are slightly better than with a larger cache. This is because a few benchmarks perform worse in the large cache. The inclusive and non-inclusive achieve similar speedups, on average, with both cache sizes.

For inclusive and non-inclusive caches, the best six configurations are the same as for the baseline size, being the prefetcher the determining component. Again, the best prefetcher combination is no L1 prefetcher plus KPC-P L2 prefetcher.

6.3.2 Prefetcher Impact

The prefetchers continue to determine the performance of all configurations. The prefetcher that dominates for inclusive and non-inclusive is KPC-P, the same as in small caches. For exclusive caches, the best one is DAAMPM. Both of them are better without L1 prefetcher.

IP-stride and *next-line* perform worse on large exclusive caches for any replacement policy and any L1 prefetcher.

For an inclusive or non-inclusive large cache, some prefetchers achieve better speedup than with a smaller cache, for example, DAAMPM. We found the large cache, for inclusive and non-inclusive to show more variability and higher perfor-

mance.

6.3.3 Replacement Policy Impact

LRU is still the best replacement policy for any combination of prefetchers for the exclusive cache. In the large cache it performs even better. For example, as aforementioned, the LRU configuration with KPC-P is among the best configurations of the large cache, instead just no L1 prefetcher plus DAAMPM. Also, even with no prefetchers at all, LRU performs better than for the small cache in comparison to the rest of replacement policies.

DRRIP provides worse speedup for all inclusion policies in the large cache, for example, see *bwaves* (see Appendix A for small cache and Appendix C for large cache). However, SRRIP performs slightly better. For example, SRRIP with no L1 prefetcher and DAAMPM L2 prefetcher matches and improves DRRIP's speedup, which was among the best configurations in exclusive caches for the baseline cache size.

KPC-R shows worse performance on the large cache for some prefetcher combinations, for example using DAAMPM L2 prefetcher and any L1 prefetcher. This happens on a few benchmarks such as *lbm* (see Appendix A for small cache and Appendix C for large cache).

6.4 Discussion

The single-core results demonstrate that the prefetchers and replacement policies that work best for non-inclusive and inclusive caches are not the best ones for an exclusive cache. This motivates further research on prefetching and replacement policies for exclusive caches and their interaction.

In single-core simulations, the best performance is never achieved using an exclusive policy. However, in multi-core simulations, exclusive caches are the ones that

get best performance except on, mostly, configurations with the DRRIP and SHiP replacement policies. The most interesting property of exclusive caches is that they increase the effective capacity of the cache. Hence, research on cache management techniques for exclusive caches should focus on multi-core.

The combinations of prefetchers have a big impact on performance while in replacement policies the impact was lower, specifically for exclusive caches. This motivates either more research for prefetching combined with simpler replacement policies or to design unified cache management techniques tuned for exclusive caches.

The replacement policy that stands out the most is LRU on exclusive caches. However, true LRU is expensive to implement. It would be interesting to explore LRU-based replacement policies to compare their performance in exclusive caches. Also, the best replacement policy for a non-inclusive is different than that for an inclusive cache for some cases. This should encourage taking the inclusion policy into account when designing a new replacement policy to decide whether it is necessary to model an inclusive cache, or the performance of both is similar enough.

Regarding cache size, we found that sometimes, for the same cache configuration, an specific replacement policy is no longer among the best to use when using a larger cache. On exclusive caches, which have a higher effective capacity, might not be necessary to have a very large cache. It might be useful to have a smaller exclusive LLC and use the extra hardware to improve performance in other system components.

7. SUMMARY

The speed of processors has been increasing at a higher rate than the speed of memories over the last years. Caches have been designed to mitigate this problem and increase overall performance. There has been extensive research on how to make caches provide higher performance, aiming to reduce the total number of misses.

Cache management techniques have been mostly designed and evaluated in the context of non-inclusive last-level caches (LLCs). However, many modern processors implement their LLC with either an inclusive or an exclusive policy. The hypothesis of this thesis is that a cache management technique (prefetcher or replacement policy) that performs well in one inclusion policy might not be the best for another inclusion policy.

In this thesis we explored the design space of cache management techniques in different cache configurations, with a focus on the cache inclusion policy. We implemented an inclusive and an exclusive policy on top of a simulator that had a non-inclusive policy by default. We evaluated different prefetchers, replacement policies, cache sizes and number of cores for each inclusion policy.

7.1 Conclusions

We found that the configurations for inclusive and non-inclusive usually performed similarly (depending on the configuration and benchmark), but for exclusive caches the best configurations were indeed different.

Prefetchers paid an important role on the performance for each cache configuration. In most of the cases, a combination of L1 and L2 prefetchers were the responsible to determine the best configurations. For example, over all cache configurations, a winning prefetcher combination had the best speedups with all replacement poli-

cies. The replacement policies then, were not as important as the prefetchers for performance. This shows that many benefits of using different replacement policies are shadowed by the prefetcher. Therefore, it is very important to implement a state-of-the-art prefetcher on cache replacement policy research. This also suggests that it might be more beneficial to use the hardware available to have a best prefetcher instead of a complicated replacement policy or to design an integrated technique.

The performance of exclusive caches in single-core simulations versus multi-core simulation was different. In single-core, the performance was generally worse than inclusive or non-inclusive caches. However, in multi-core, the performance of exclusive caches was superior to the other inclusions except for a few configurations. This is probably because the higher effective capacity of exclusive caches is better exploited on multiprogrammed workloads than in single-threaded. This suggests that studies on cache management for exclusive caches, and any other inclusion policy, should focus on multi-core processors running either on multiprogrammed or multi-threaded workloads, also because single-threaded scenarios are rare in current and future systems.

For exclusive caches, our experiments showed that the best replacement policy across different prefetcher combinations was LRU. For inclusive and non-inclusive policies, the best replacement policy was different depending on the case. This suggests that the inclusion policy can also impact performance by making a different technique behave better than another. When designing a replacement policy, the inclusion policy should be considered, mentioned and discussed.

Typical cache sizes can also vary over time and for different systems. Our experiments showed that different cache sizes affect the speedup achieved by different configurations, in our case, for exclusive caches on single core in particular. Computer architects should keep in mind that the best cache management technique can

be different depending on cache size.

7.2 Future Work

One of the insights from this thesis is that future research on multi-core should include exclusive cache designs given their better speedup. This thesis only investigated multiprogrammed workloads. In the future, we plan to run similar experiments with multi-threaded workloads. For this, we will use a different simulator that supports multithreading.

We also found that exclusive caches had slightly worse speedup in a larger cache size for single-core. In the future, we will run simulations of multiple cache sizes on multiprogrammed and multithreaded workloads to validate this result in those scenarios. If it gets worse speedup in all cases, it might be better to keep a smaller LLC per core with the increased effective capacity of exclusive caches and use the area savings to implement other optimizations.

Also, we aim to implement other prefetchers and replacement policies that use machine learning techniques to assess how those techniques learn and adapt in the context of different inclusion policies.

We plan to explore the potential of LRU-based replacement policies on exclusive caches and investigate why they work better than more advanced techniques. LRU performed better even without prefetchers, so we will initially evaluate LRU without prefetchers for simplicity to better understand the underlying reasons of its performance and assess whether replacement policies are being overdesigned or wrongly targeted for exclusive caches.

The results in this thesis motivate further research on prefetchers and replacement policies targeted to exclusive caches. Previous works use LRU as the baseline for the proposed cache management techniques evaluated in the context of non-inclusive

caches. In our work, those techniques outperform LRU for non-inclusive and inclusive caches but not for exclusive caches. This means that a different approach to prefetching and replacement is needed when caches are exclusive. We plan to learn more about the reasons behind this different behavior and propose new cache management techniques targeted to exclusive caches, given that they are being more pervasively used in modern systems.

REFERENCES

[1] G. E. Moore, "Cramming More Components onto Integrated Circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, 1965.

[2] G. E. Moore, "Progress in Digital Integrated Electronics," in *International Electron Devices Meeting*, vol. 21, pp. 11–13, 1975.

[3] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The Case for a Single-chip Multiprocessor," in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pp. 2–11, 1996.

[4] N. P. Jouppi, "Improving Direct-mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA'90, pp. 364–373, 1990.

[5] "The 2nd Cache Replacement Champsionship." `http://crc2.ece.tamu.edu`. Accessed: 2017-04-30.

[6] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[7] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the Clouds: a Study of Emerging Scale-out Workloads on Modern Hardware," in *ACM SIGPLAN Notices*, vol. 47, pp. 37–48, 2012.

[8] R. R. Curtin, J. R. Cline, N. P. Slagle, W. B. March, P. Ram, N. A. Mehta, and A. G. Gray, "MLPACK: A Scalable C++ Machine Learning Library," *Journal*

of Machine Learning Research, vol. 14, pp. 801–805, 2013.

[9] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *ACM SIGARCH Computer Architecture News*, vol. 23, no. 1, pp. 20–24, 1995.

[10] R. H. Dennard, "Field-effect transistor memory," June 4 1968. US Patent 3,387,286.

[11] A. Jaleel, E. Borch, M. Bhandaru, S. C. Steely Jr, and J. Emer, "Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (tla) cache management policies," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-43, pp. 151–162, 2010.

[12] N. P. Jouppi and S. J. Wilton, "Tradeoffs in two-level on-chip caching," in *Proceedings the 21st Annual International Symposium on Computer Architecture*, ISCA'94, pp. 34–45, 1994.

[13] Y. Zheng, B. T. Davis, and M. Jordan, "Performance Evaluation of Exclusive Cache Hierarchies," in *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS'04, pp. 89–96, 2004.

[14] T. M. Wong and J. Wilkes, "My cache or yours?: Making storage more exclusive," in *USENIX Annual Technical Conference*, USENIX'02, pp. 161–175, 2002.

[15] L. Zhao, R. Iyer, S. Makineni, D. Newell, and L. Cheng, "Ncid: a non-inclusive cache, inclusive directory architecture for flexible and efficient cache hierarchies," in *Proceedings of the 7th ACM international conference on Computing Frontiers*, CF'10, pp. 121–130, 2010.

[16] "Intel Pentium 4 Willamette." http://ark.intel.com/products/27426/Intel-Pentium-4-Processor-1_70-GHz-256K-Cache-400-MHz-FSB. Accessed: 2017-04-30.

[17] "Intel 64 and IA-32 Architectures Optimization Manual." https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf. Accessed: 2017-04-30.

[18] "Knights Landing Architecture." http://pages.cs.wisc.edu/~david/courses/cs758/Fall2016/handouts/restricted/Knights-landing.pdf. Accessed: 2017-04-30.

[19] "Cortex-A9 Technical Reference Manual." http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388i/DDI0388I_cortex_a9_r4p1_trm.pdf. Accessed: 2017-04-30.

[20] "IBM POWER5 Architecture." https://www.ibm.com/developerworks/community/wikis/home?lang=en#!/wiki/Power%20Systems/page/POWER5%20Architecture. Accessed: 2017-04-30.

[21] "IBM POWER7 Architecture." https://www.cs.rice.edu/~johnmc/comp522/lecture-notes/COMP522-2016-Lecture7-Power7.pdf. Accessed: 2017-04-30.

[22] "IBM zEC12." https://www.hotchips.org/wp-content/uploads/hc_archives/hc25/HC25.20-Processors1-epub/HC25.26.220-zEC12-Processor-Sonnelitter-IBM-v5.pdf. Accessed: 2017-04-30.

[23] P. Cao, E. W. Felten, and K. Li, "Application-controlled file caching policies.," in *USENIX Summer*, pp. 171–182, 1994.

[24] R. Karedla, J. S. Love, and B. G. Wherry, "Caching strategies to improve disk system performance," *Computer*, vol. 27, no. 3, pp. 38–46, 1994.

[25] V. Phalke and B. Gopinath, "An Inter-reference Gap Model for Temporal Locality in Program Behavior," in *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '95/PERFORMANCE '95, pp. 291–300, 1995.

[26] G. Glass and P. Cao, "Adaptive Page Replacement Based on Memory Reference Behavior," in *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '97, pp. 115–126, 1997.

[27] Y. Smaragdakis, S. Kaplan, and P. Wilson, "EELRU: Simple and Effective Adaptive Page Replacement," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 27, pp. 122–133, 1999.

[28] L. A. Belady, "A Study of Replacement Algorithms for a Virtual-storage Computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[29] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive Insertion Policies for High Performance Caching," in *ACM SIGARCH Computer Architecture News*, vol. 35, pp. 381–391, 2007.

[30] T.-F. Chen and J.-L. Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE Transactions on Computers*, vol. 44, no. 5, pp. 609–623, 1995.

[31] A. J. Smith, "Sequential Program Prefetching in Memory Hierarchies," *Computer*, vol. 11, no. 12, pp. 7–21, 1978.

[32] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, SC'91, pp. 176–186, 1991.

[33] P. Michaud, "Best-offset Hardware Prefetching," in *Proceedings of the 22nd IEEE International Symposium on High Performance Computer Architecture*, HPCA-22, pp. 469–480, 2016.

[34] Y. Ishii, M. Inaba, and K. Hiraki, "Unified Memory Optimizing Architecture: Memory Subsystem Control with a Unified Predictor," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS'12, pp. 267–278, 2012.

[35] V. Seshadri, S. Yedkar, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks," *ACM Transactions on Architecture and Code Optimization*, vol. 11, no. 4, p. 51, 2015.

[36] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely Jr, and J. Emer, "PACMan: prefetch-aware cache management for high performance caching," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pp. 442–453, 2011.

[37] J. Kim, E. Teran, P. V. Gratz, D. A. Jiménez, S. H. Pugsley, and C. Wilkerson, "Kill the program counter: Reconstructing program behavior in the processor cache hierarchy," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XXII, pp. 737–749, 2017.

[38] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr, and J. Emer, "SHiP: Signature-based Hit Predictor for High Performance Caching,"

in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pp. 430–441, 2011.

[39] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pp. 60–71, 2010.

[40] S. M. Khan, Y. Tian, and D. A. Jiménez, "Dead Block Replacement and Bypass with a Sampling Predictor," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-43, pp. 175–186, 2010.

[41] D. A. Jiménez, "Insertion and Promotion for Tree-based PseudoLRU Last-Level Caches," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pp. 284–296, 2013.

[42] E. Teran, Y. Tian, Z. Wang, D. A. Jiménez, *et al.*, "Minimal Disturbance Placement and Promotion," in *Proceedings of the 22nd IEEE International Symposium on High Performance Computer Architecture*, HPCA-22, pp. 201–211, 2016.

[43] T. L. Johnson, D. A. Connors, M. C. Merten, and W.-M. Hwu, "Run-time cache bypassing," *IEEE Transactions on Computers*, vol. 48, no. 12, pp. 1338–1354, 1999.

[44] T. Piquet, O. Rochecouste, and A. Seznec, "Exploiting single-usage for effective memory management," in *Proceedings of the 12th Asia-Pacific Conference on Advances in Computer Systems Architecture*, ACSAC '07, pp. 90–101, 2007.

[45] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun, "A modified approach to data cache management," in *Proceedings of the 28th Annual International*

Symposium on Microarchitecture, MICRO-28, pp. 93–103, 1995.

[46] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely Jr, and J. Emer, "Adaptive insertion policies for managing shared caches," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT'08, pp. 208–219, 2008.

[47] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for mlp-aware cache replacement," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, pp. 167–178, 2006.

[48] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry, "The evicted-address filter: A unified mechanism to address both cache pollution and thrashing," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT'12, pp. 355–366, 2012.

[49] B. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[50] D. A. Jiménez and C. Lin, "Perceptron Learning for Predicting the Behavior of Conditional Branches," in *Proceedings of the International Joint Conference on Neural Networks*, IJCNN'01, pp. 2122–2127, 2001.

[51] Y. Freund and R. E. Schapire, "Large margin classification using the perceptron algorithm," *Machine learning*, vol. 37, no. 3, pp. 277–296, 1999.

[52] E. Teran, Z. Wang, and D. A. Jiménez, "Perceptron Learning for Reuse Prediction," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, pp. 1–12, 2016.

[53] A. Jain and C. Lin, "Back to the future: leveraging belady's algorithm for improved cache replacement," in *Proceedings of the ACM/IEEE 43rd Annual*

International Symposium on Computer Architecture, ISCA'16, pp. 78–89, 2016.

[54] J. Gaur, M. Chaudhuri, and S. Subramoney, "Bypass and Insertion Algorithms for Exclusive Last-level Caches," in *ACM SIGARCH Computer Architecture News*, vol. 39, pp. 81–92, 2011.

[55] M. Chaudhuri, J. Gaur, N. Bashyam, S. Subramoney, and J. Nuzman, "Introducing Hierarchy-awareness in Replacement and Bypass Algorithms for Last-level Caches," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT'12, pp. 293–304, 2012.

[56] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 44–60, 2003.

[57] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, pp. 318–319, 2003.

[58] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, and M. Valero, "Fame: Fairly measuring multithreaded architectures," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT'07, pp. 305–316, 2007.

[59] "The 2nd Data Prefetching Championship." `http://comparch-conf.gatech.edu/dpc2`. Accessed: 2017-04-22.

[60] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Elsevier, 2012.

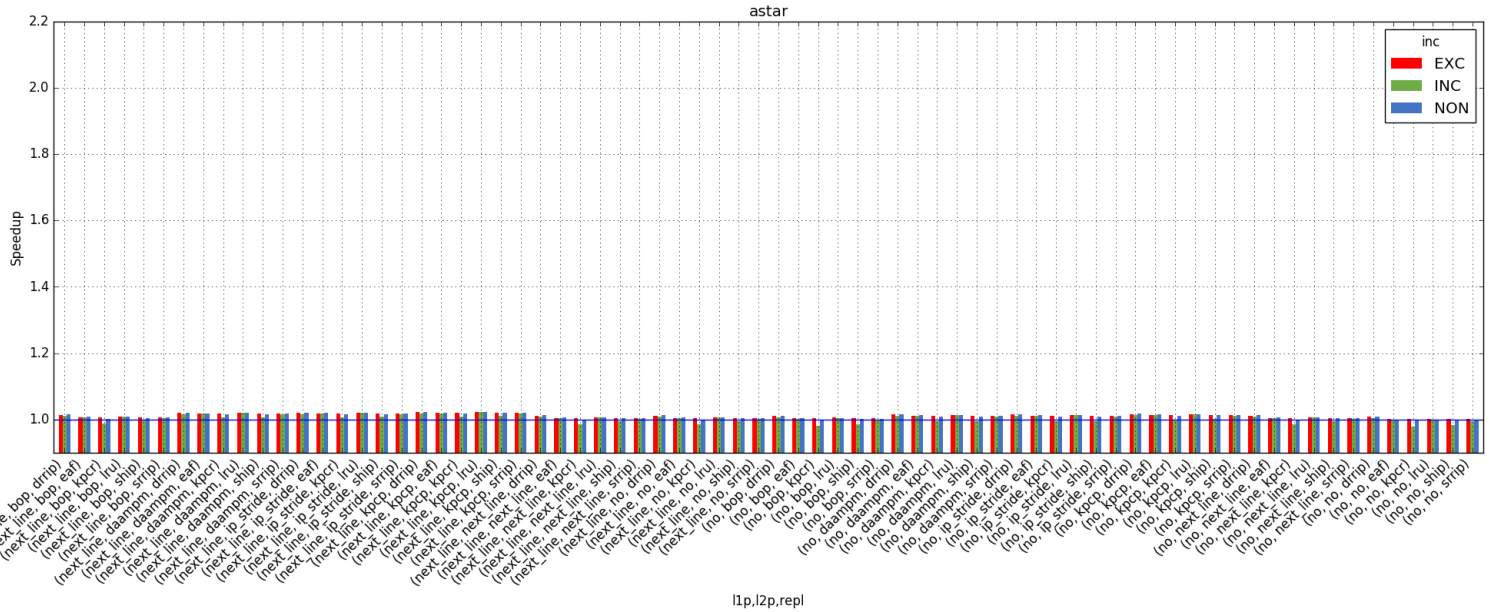
A. SINGLE-CORE RESULTS

This appendix contains the single-core plots for all the benchmarks used. In the main thesis only the geomean speedup plots are showed for readability. Below there is a description of how to read the plots.

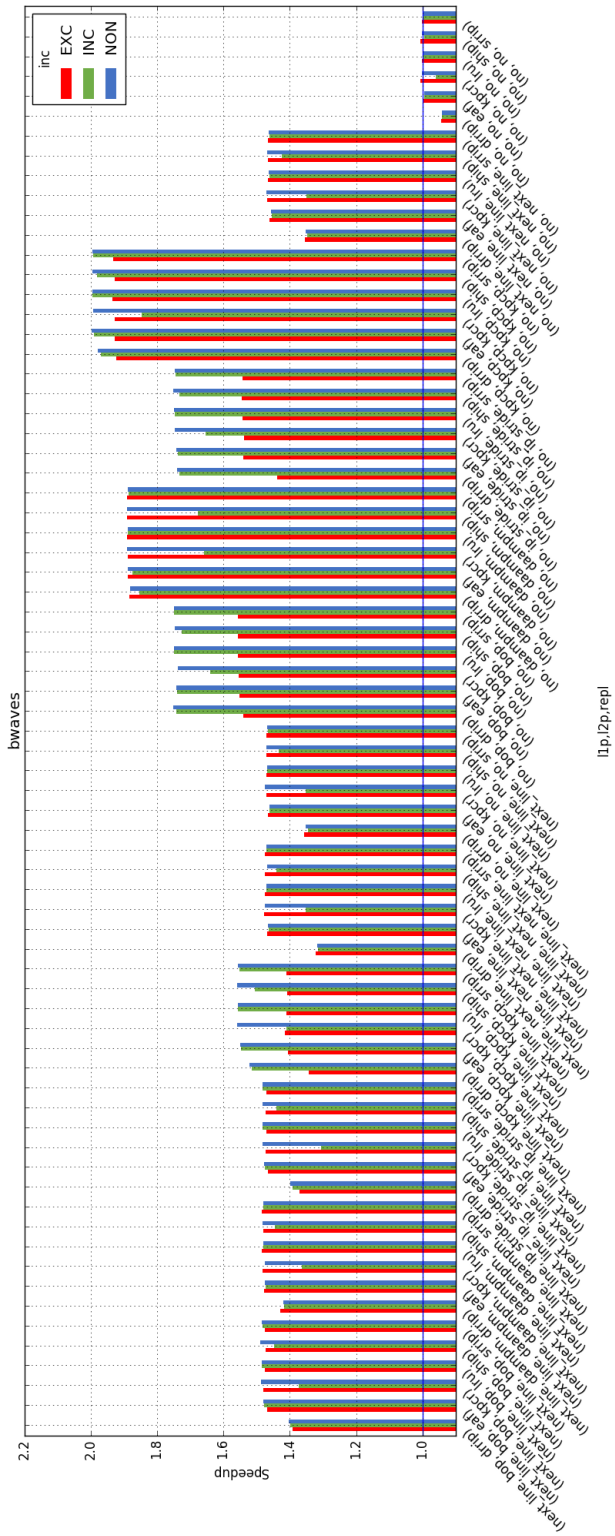
These plots compare different configurations of L1 and L2 prefetchers, replacement policies and cache inclusions. The configurations compared are: L1 prefetcher, L2 prefetcher, replacement policy and cache inclusion.

The title of the plot indicates the name of the benchmark. The Y-axis shows the speedup over the baseline configuration: no prefetchers, LRU replacement policy and a non-inclusive cache. The X-axis shows the different cache configurations, in order of: L1 prefetcher (l1p), L2 prefetcher (l2p) and replacement policy (repl). The inclusion of the cache is shown in the legend: exclusive "EXC", inclusive "INC" and non-inclusive "NON".

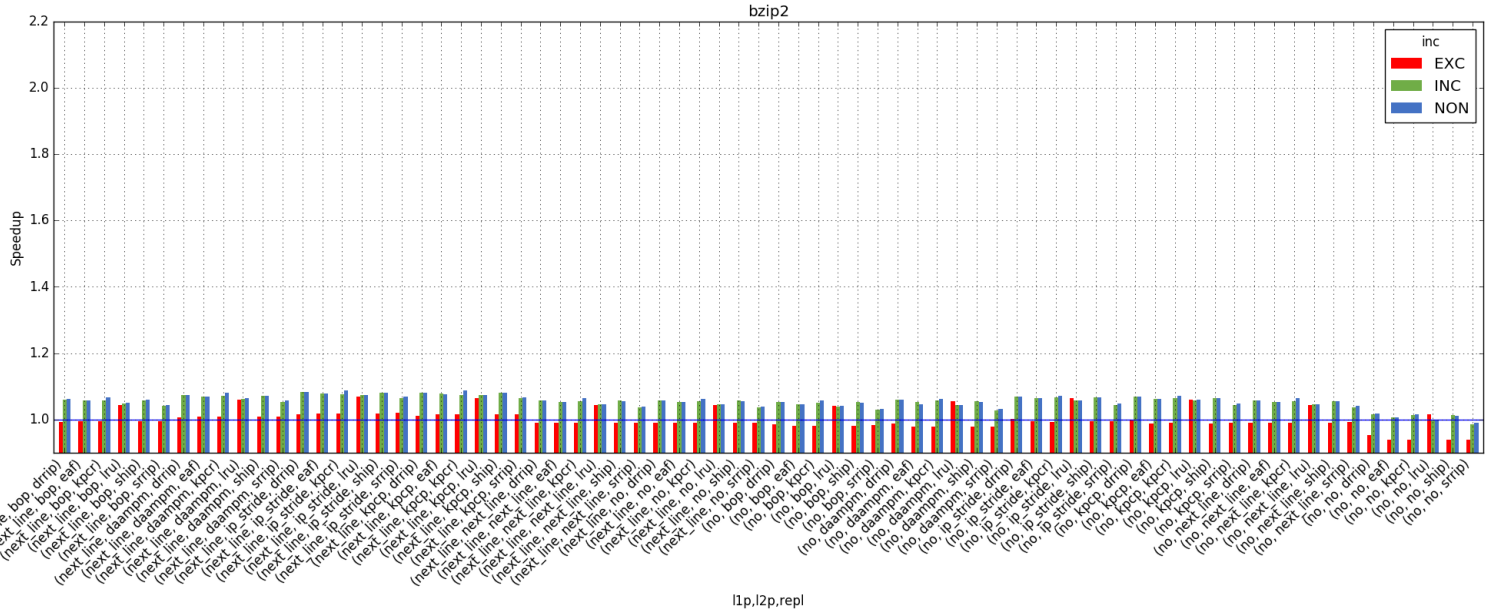
A.1 Astar



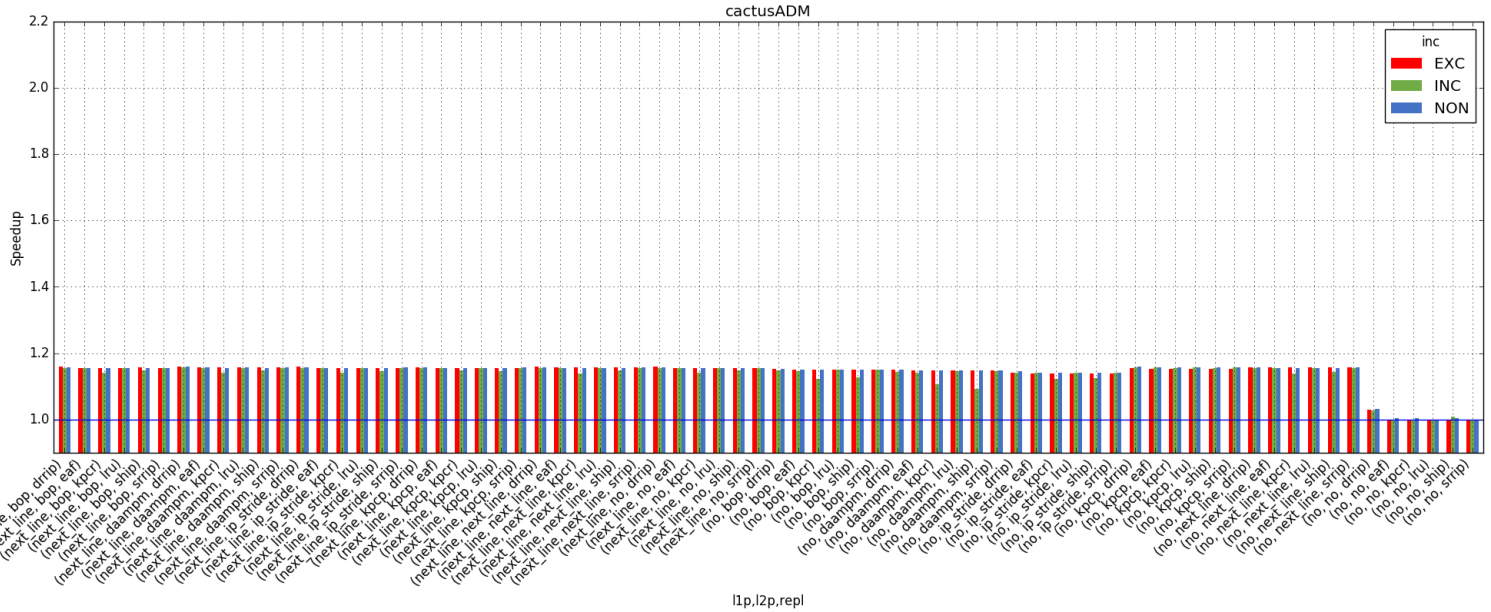
A.2 Bwaves



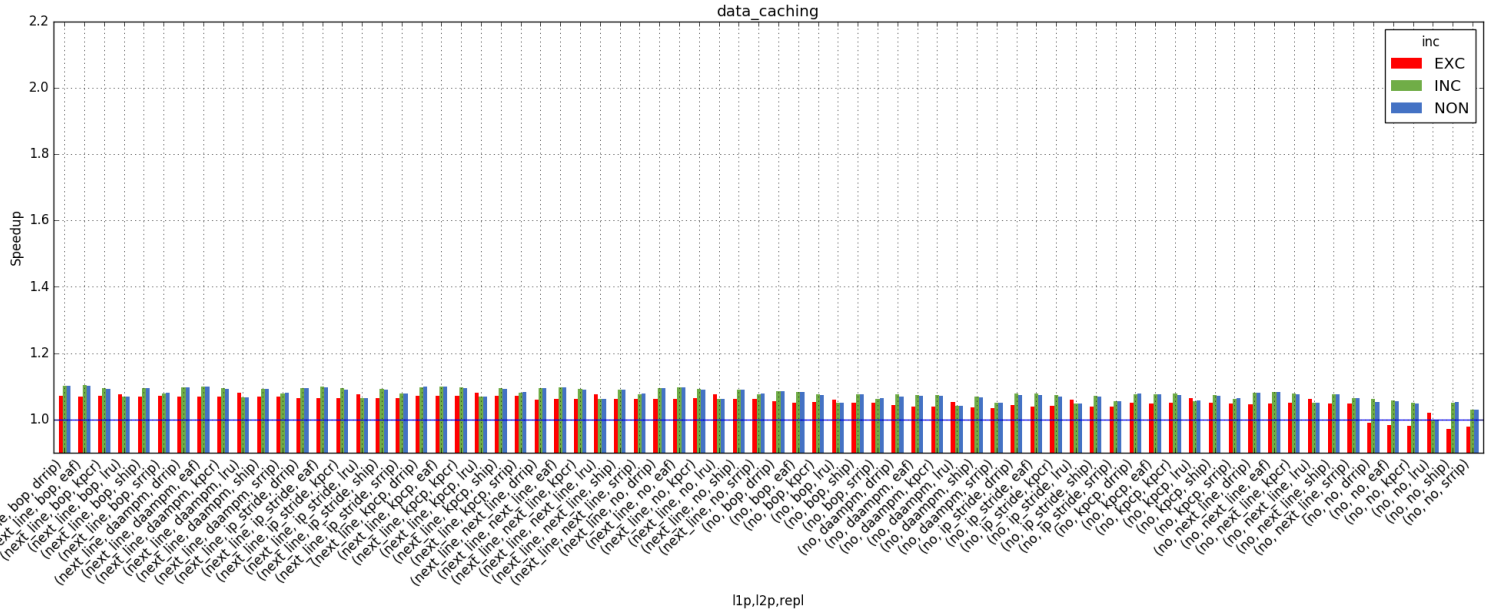
A.3 Bzip2



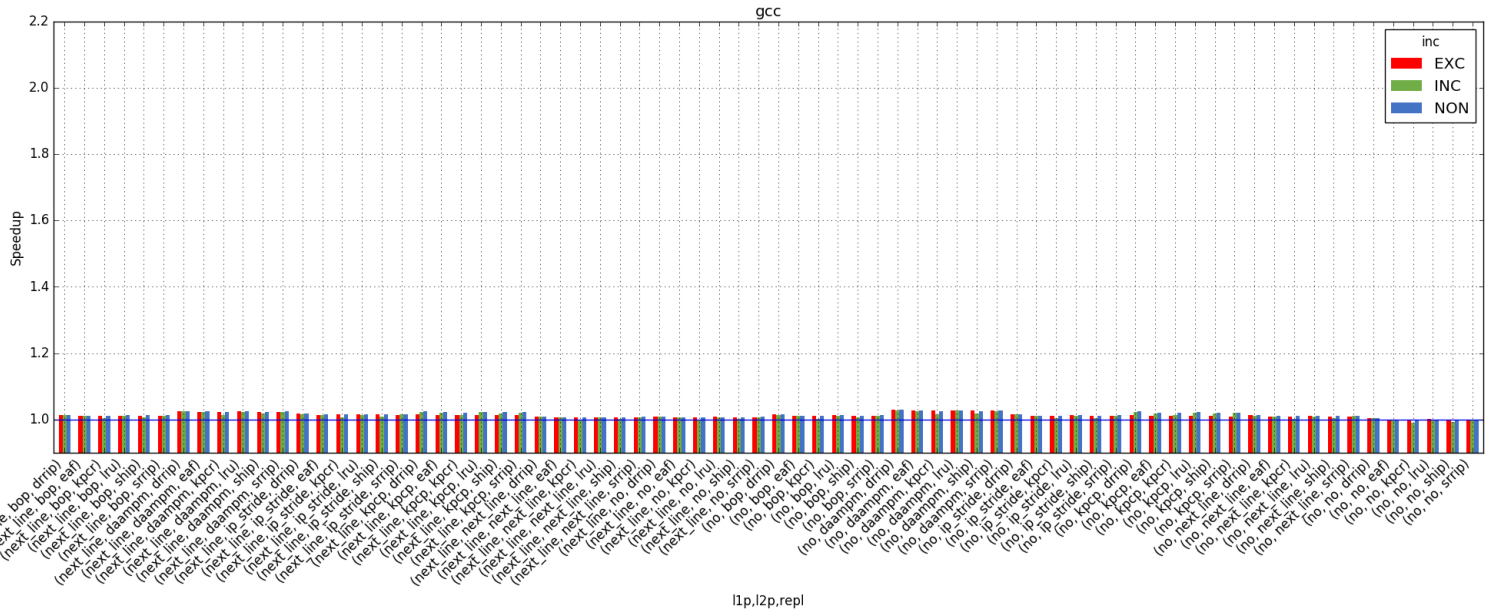
A.4 CactusADM



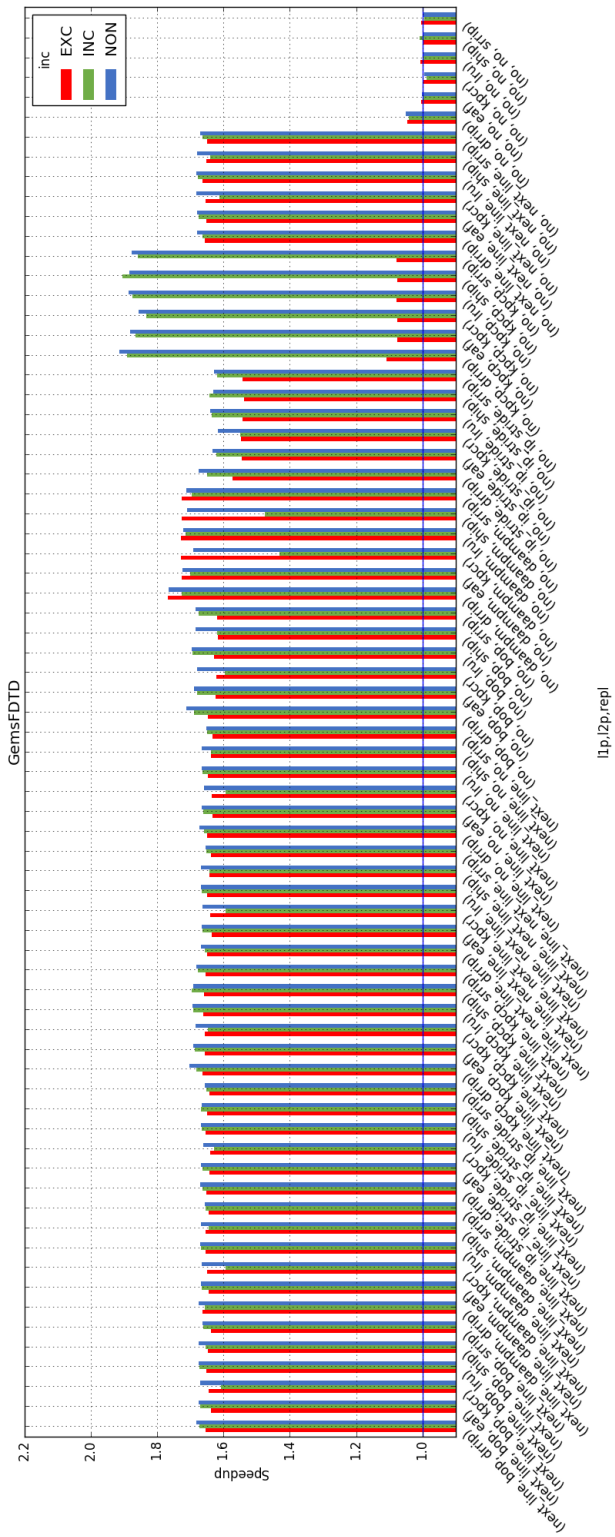
A.5 Data_Caching



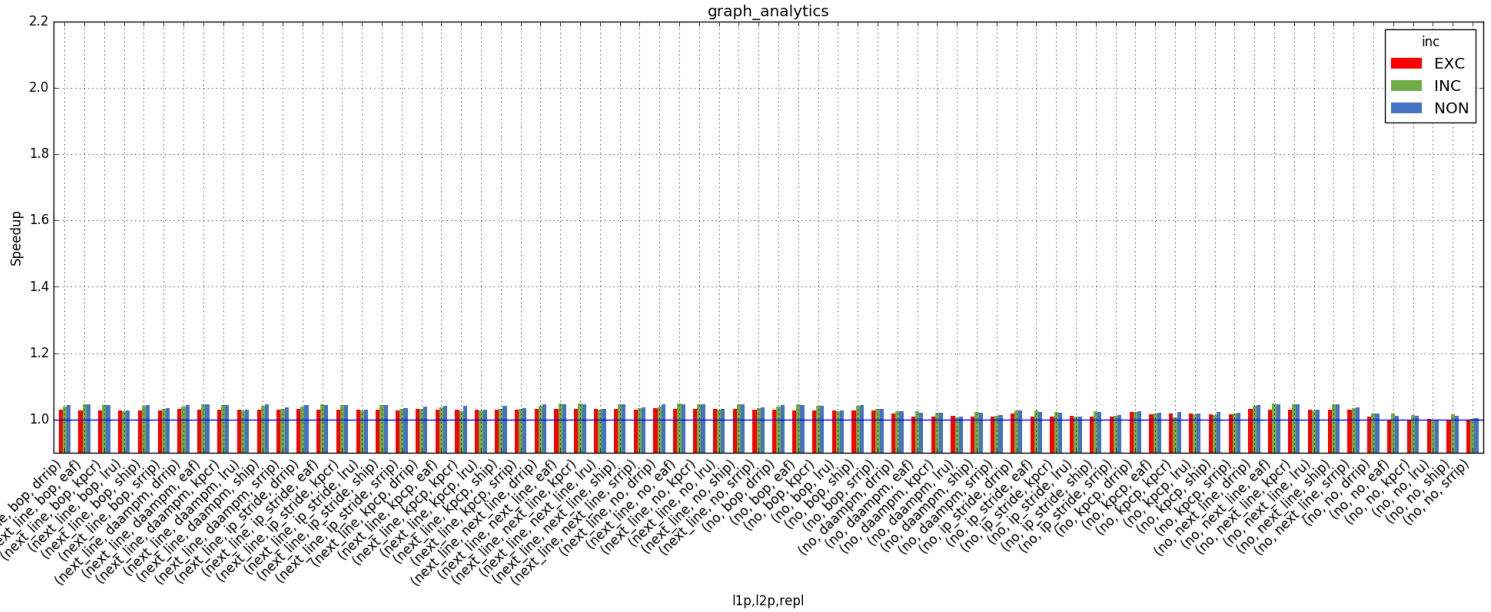
A.6 Gcc



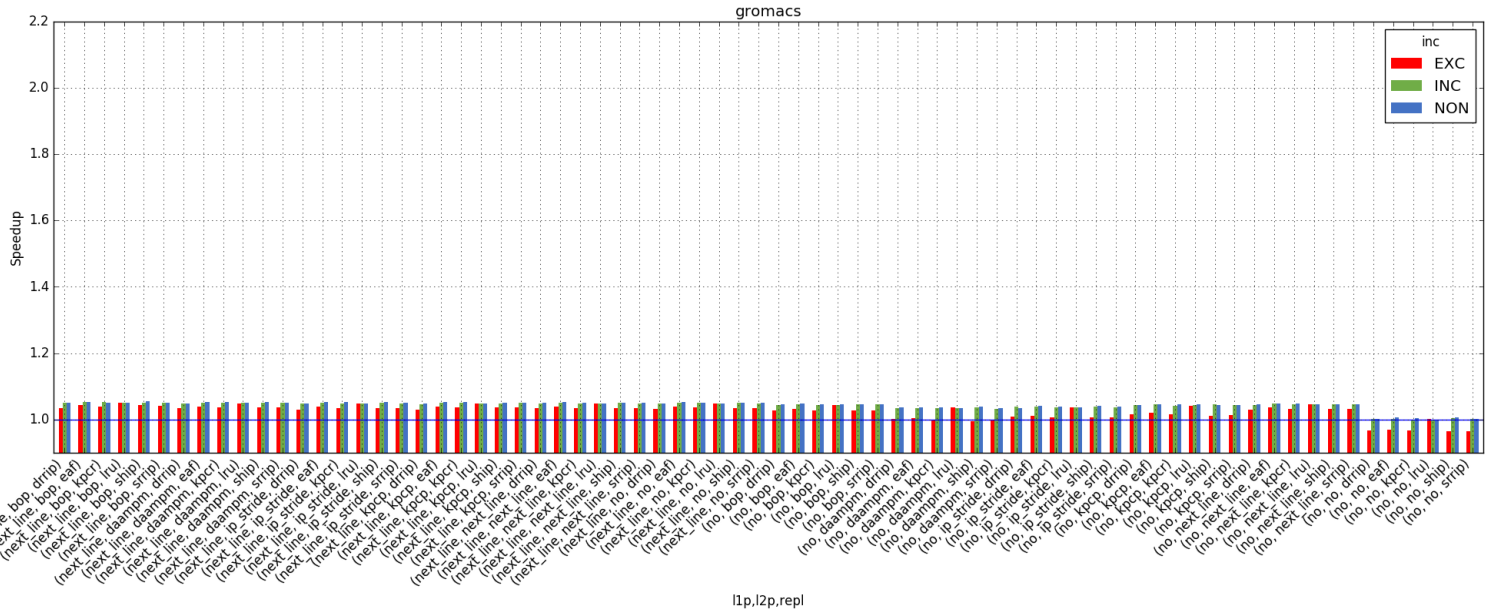
A.7 GemsFDTD


GemsFDTD
Speedup
2.2
2.0
1.8
1.6
1.4
1.2
1.0
inc
EXC
INC
NON
l1p,l2p,repl

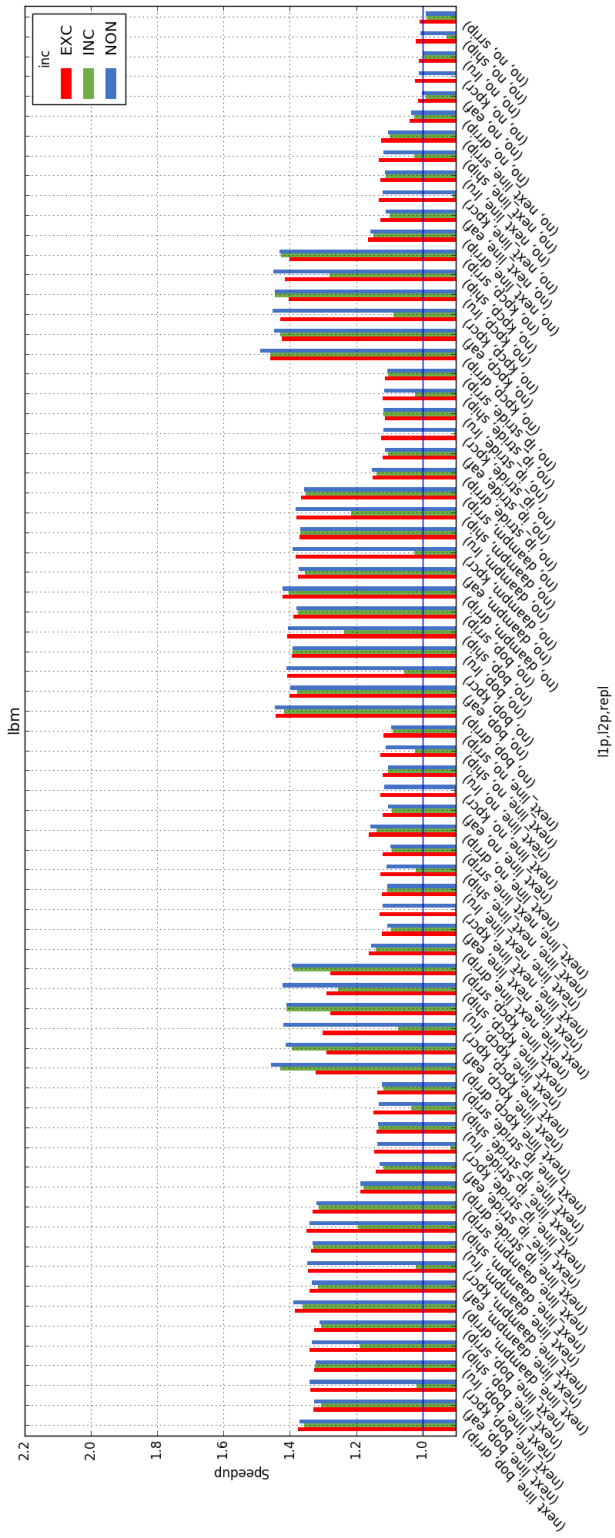
A.8 Graph_Analytics



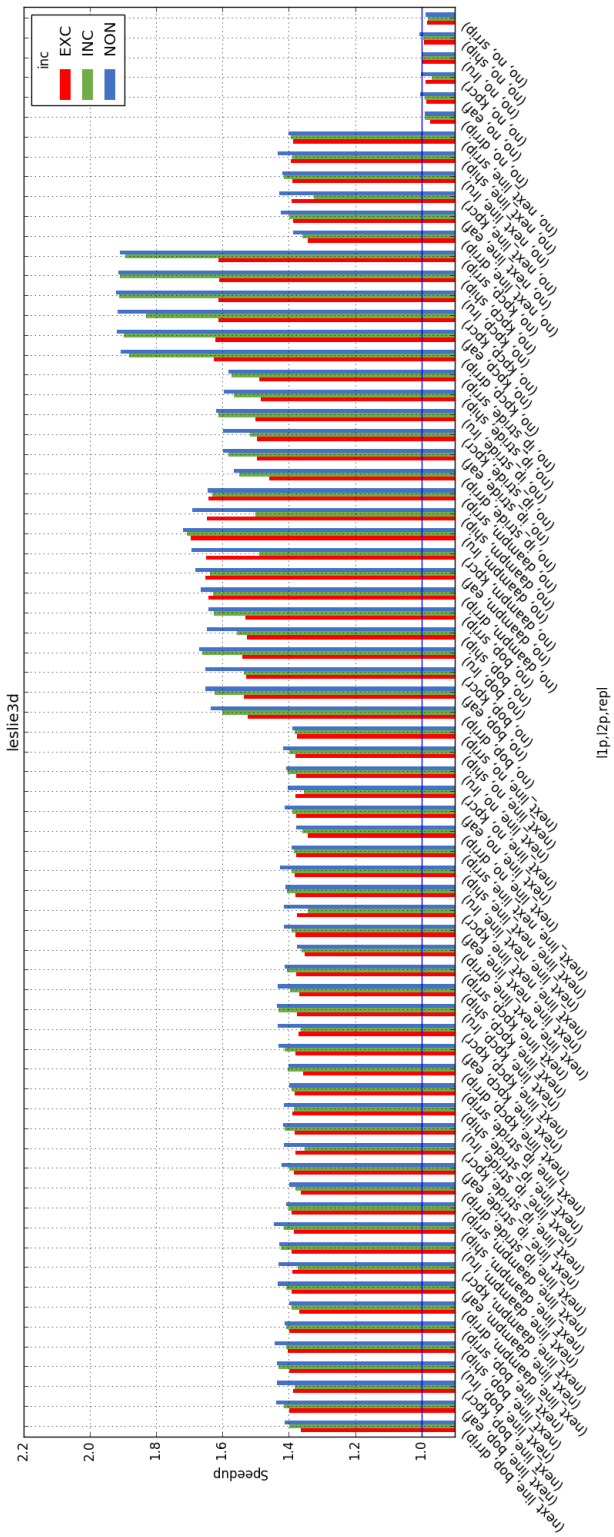
A.9 Gromacs


gromacs
Speedup
2.2
2.0
1.8
1.6
1.4
1.2
1.0
inc
EXC
INC
NON
l1p,l2p,repl

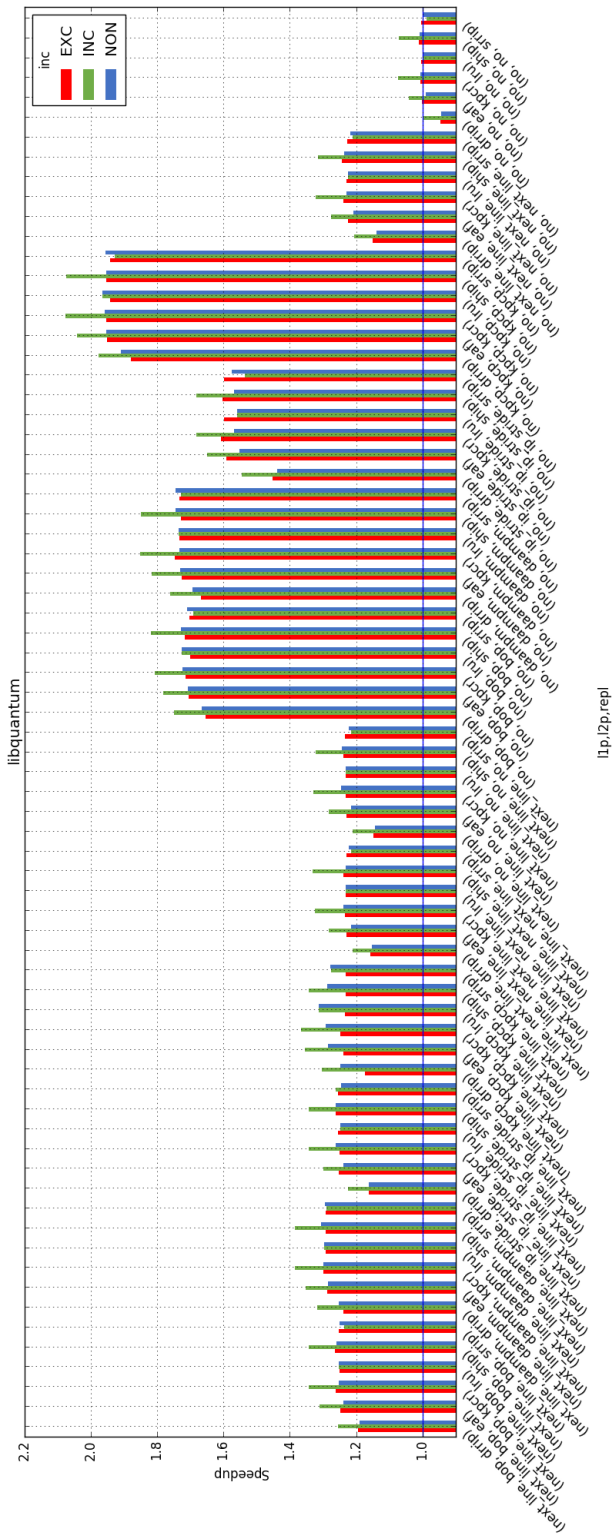
A.10 Lbm



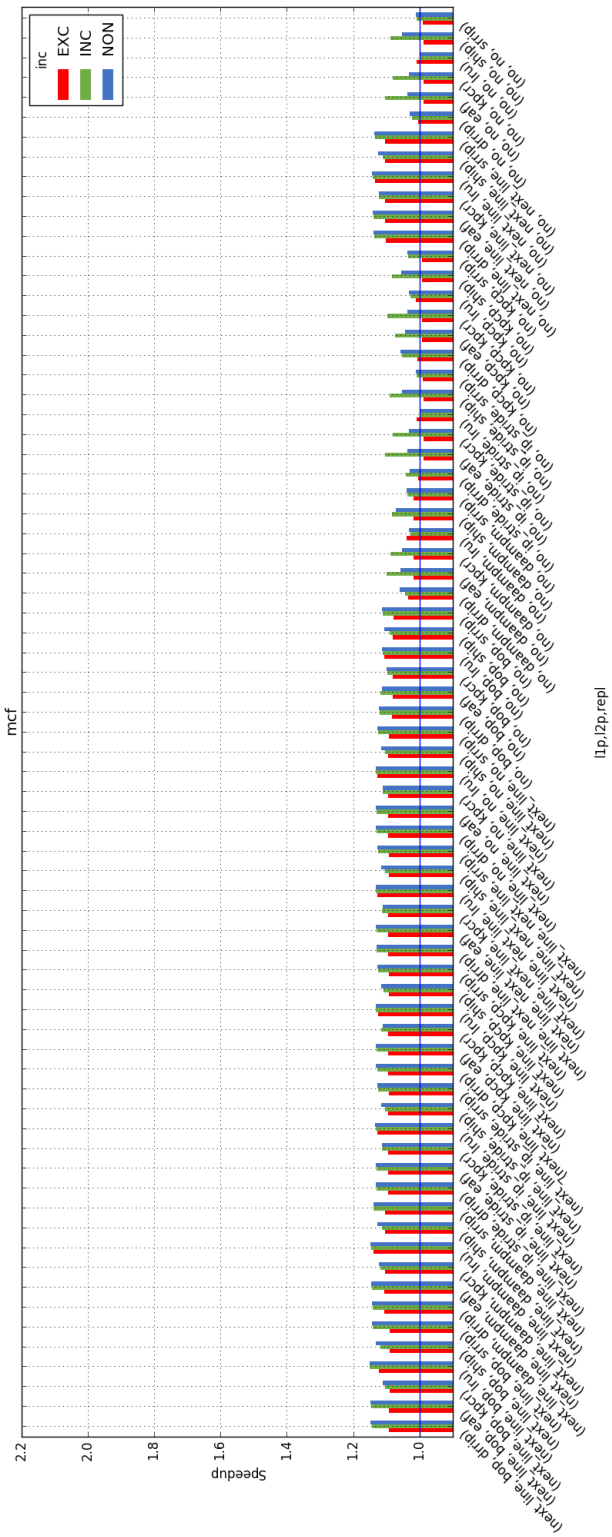
A.11 Leslie3d



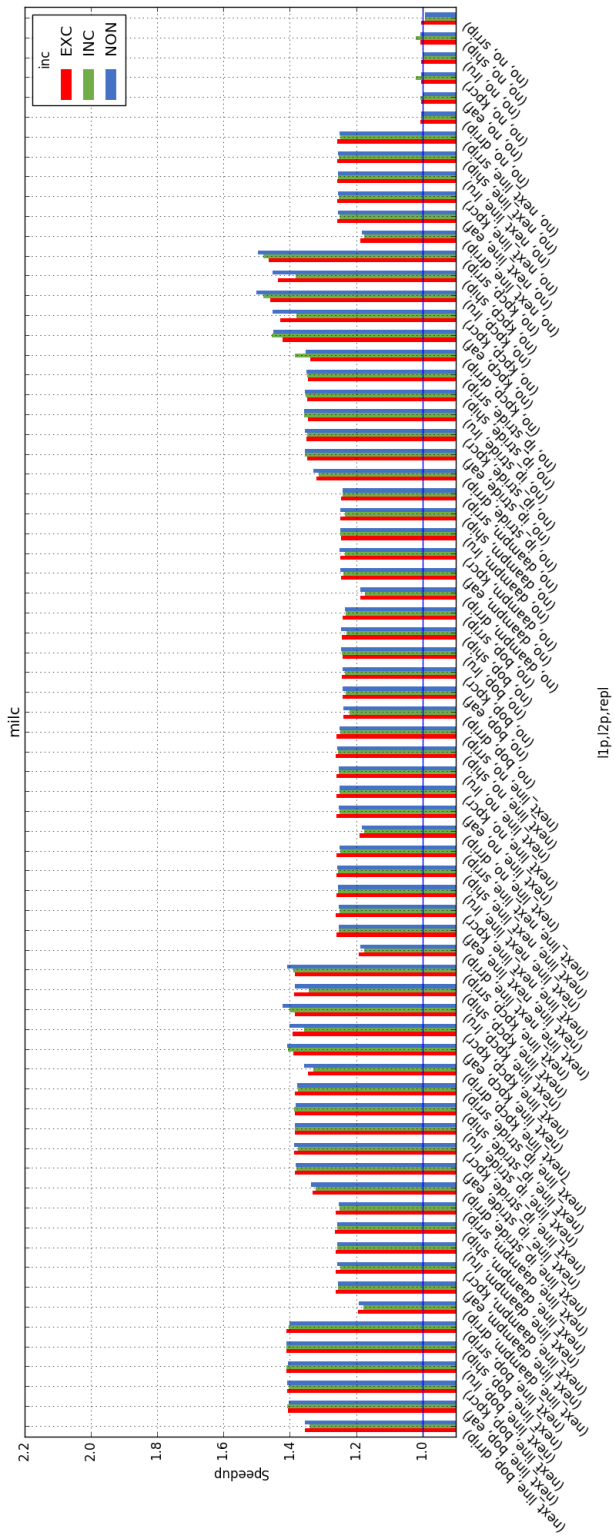
A.12 Libquantum



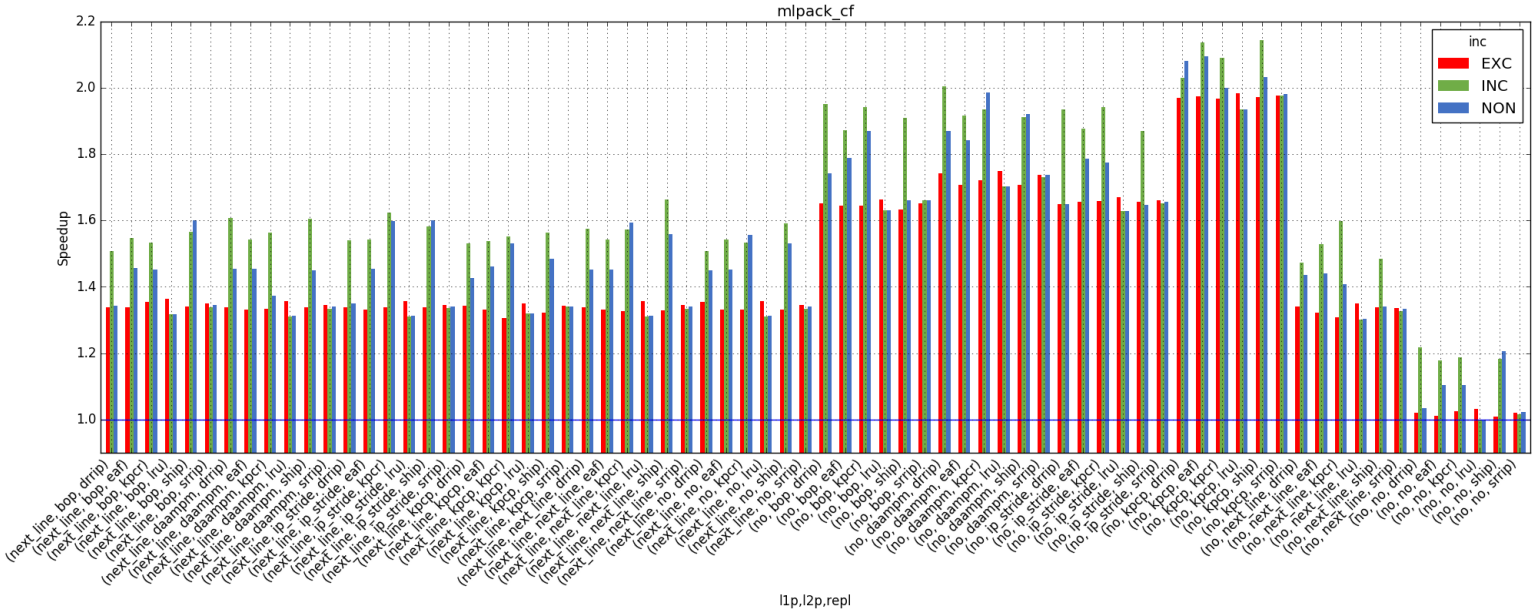
A.13 Mcf



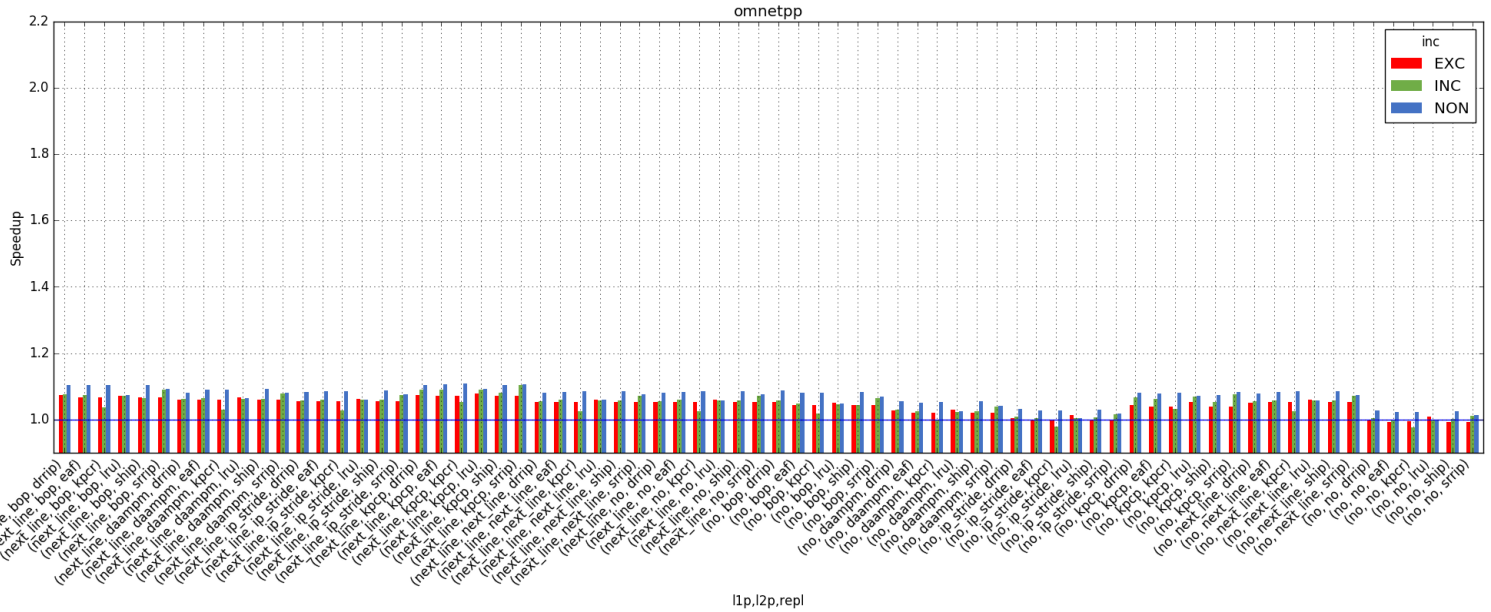
A.14 Milc


milc
inc
EXC
INC
NON
Speedup
2.2
2.0
1.8
1.6
1.4
1.2
1.0
l1p,l2p,repl
(next_line boo drip)
(next_line boo eaf)
(next_line boo kpcf)
(next_line boo liu)
(next_line boo ship)
(next_line boo strip)
(next_line dcamptm drip)
(next_line dcamptm eaf)
(next_line dcamptm kpcf)
(next_line dcamptm liu)
(next_line dcamptm ship)
(next_line dcamptm strip)
(next_line ip stride drip)
(next_line ip stride eaf)
(next_line ip stride kpcf)
(next_line ip stride liu)
(next_line ip stride ship)
(next_line ip stride strip)
(next_line kpcp drip)
(next_line kpcp eaf)
(next_line kpcp kpcf)
(next_line kpcp liu)
(next_line kpcp ship)
(next_line kpcp strip)
(next_line next_line drip)
(next_line next_line eaf)
(next_line next_line kpcf)
(next_line next_line liu)
(next_line next_line ship)
(next_line next_line strip)
(next_line no drip)
(next_line no eaf)
(next_line no kpcf)
(next_line no liu)
(next_line no ship)
(next_line no strip)
(no boo drip)
(no boo eaf)
(no boo kpcf)
(no boo liu)
(no boo ship)
(no boo strip)
(no dcamptm drip)
(no dcamptm eaf)
(no dcamptm kpcf)
(no dcamptm liu)
(no dcamptm ship)
(no dcamptm strip)
(no ip stride drip)
(no ip stride eaf)
(no ip stride kpcf)
(no ip stride liu)
(no ip stride ship)
(no ip stride strip)
(no kpcp drip)
(no kpcp eaf)
(no kpcp kpcf)
(no kpcp liu)
(no kpcp ship)
(no kpcp strip)
(no next_line drip)
(no next_line eaf)
(no next_line kpcf)
(no next_line liu)
(no next_line ship)
(no next_line strip)
(no no drip)
(no no eaf)
(no no kpcf)
(no no liu)
(no no ship)
(no no strip)

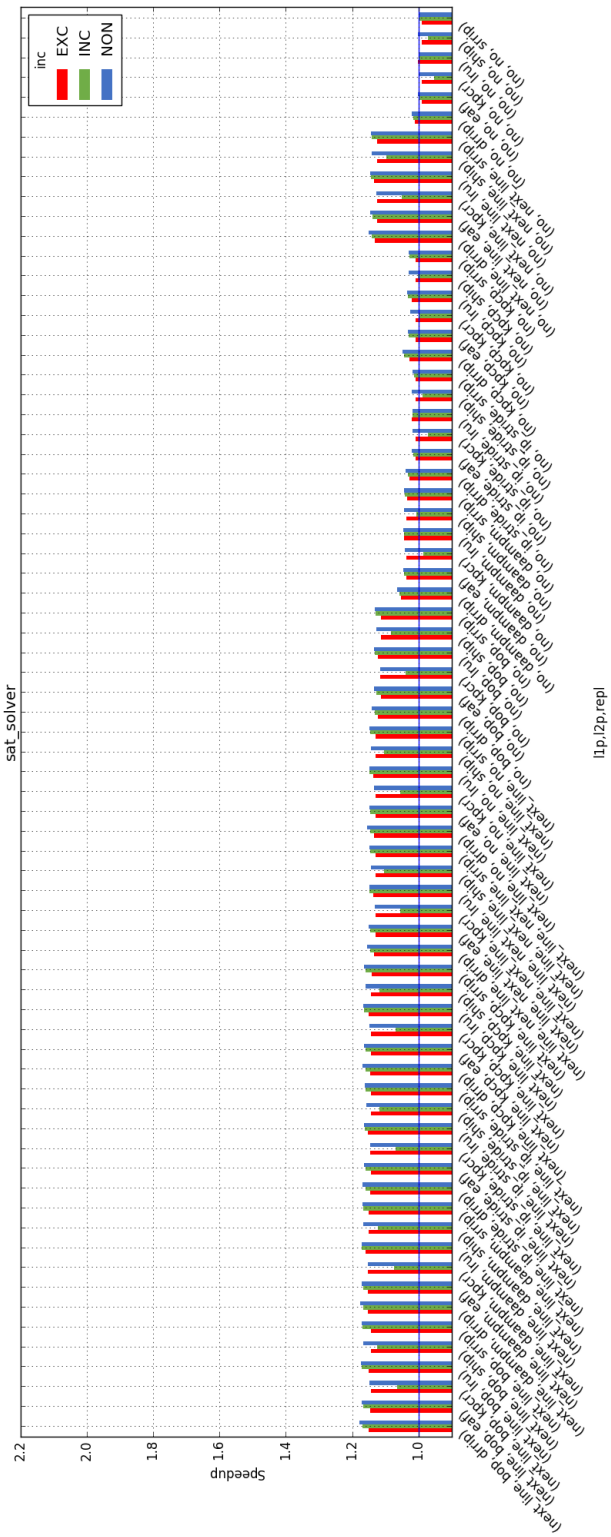
A.15 Mlpack_Cf



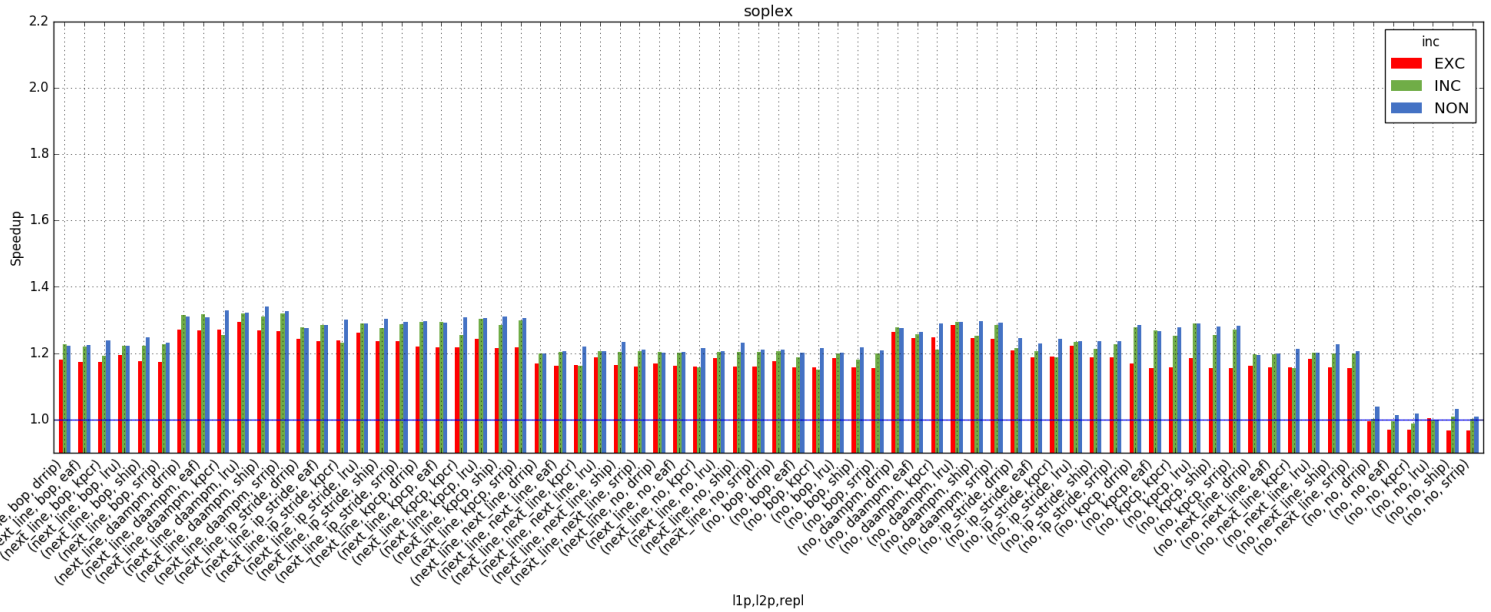
A.16 Omnetpp


omnetpp
Speedup
2.2
2.0
1.8
1.6
1.4
1.2
1.0
inc
EXC
INC
NON
l1p,l2p,repl

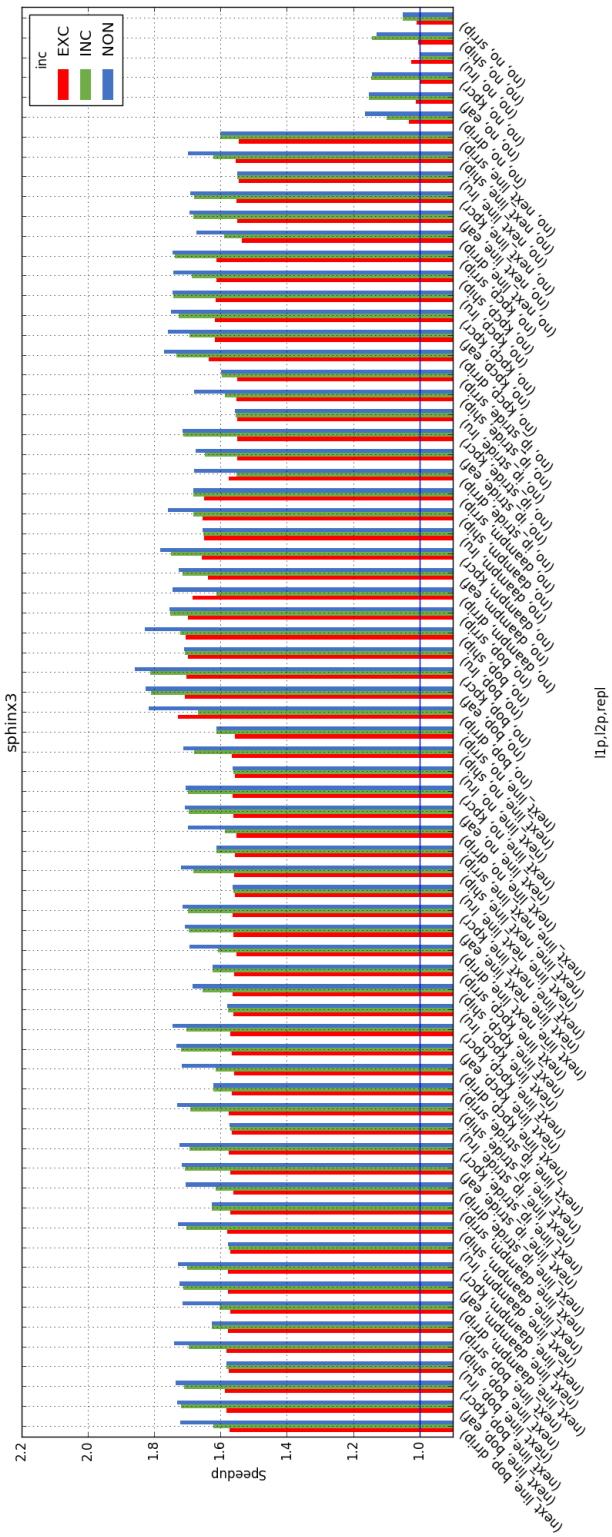
A.17 Sat_Solver



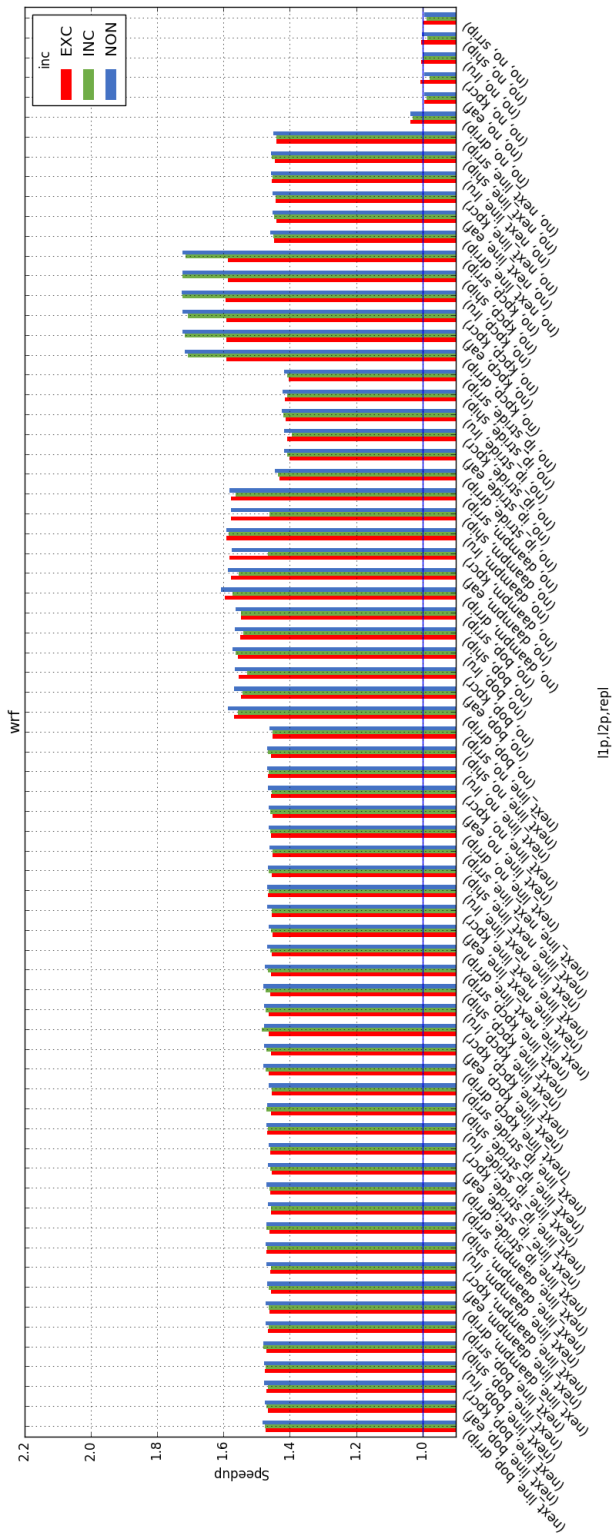
A.18 Soplex



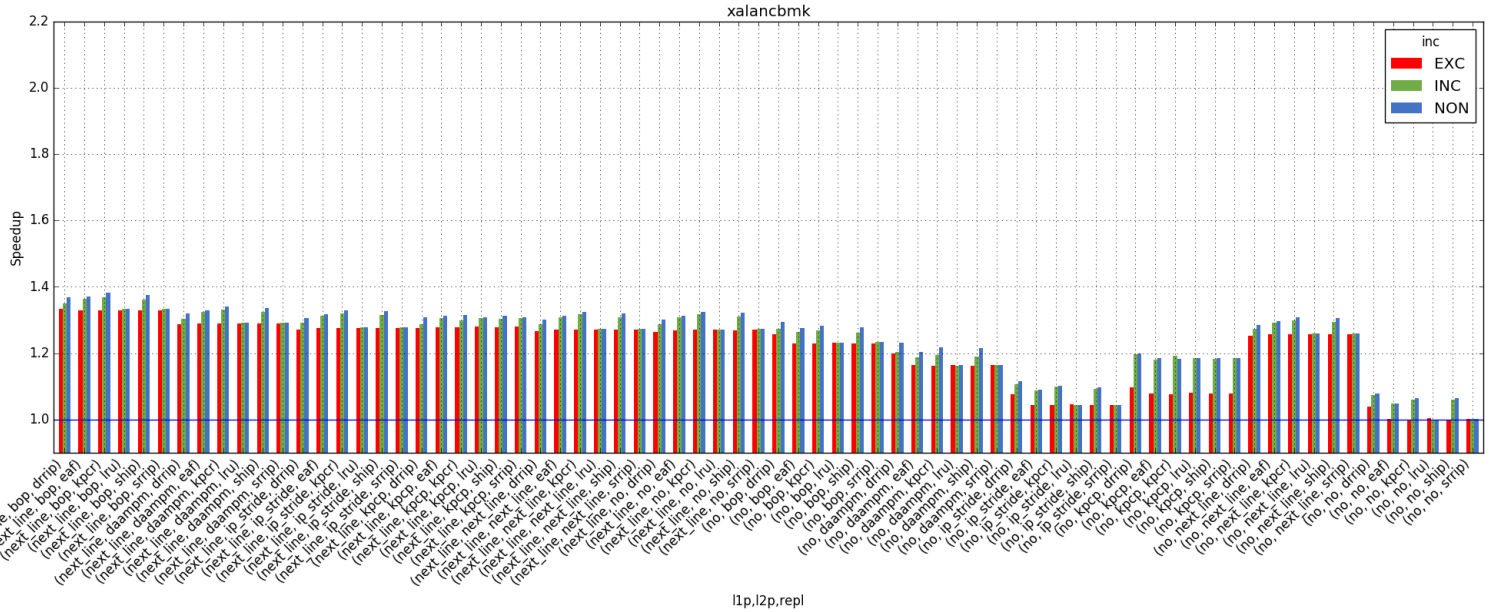
A.19 Sphinx3


sphinx3
Speedup
2.2
2.0
1.8
1.6
1.4
1.2
1.0
inc
EXC
INC
NON
l1p.l2p.repl

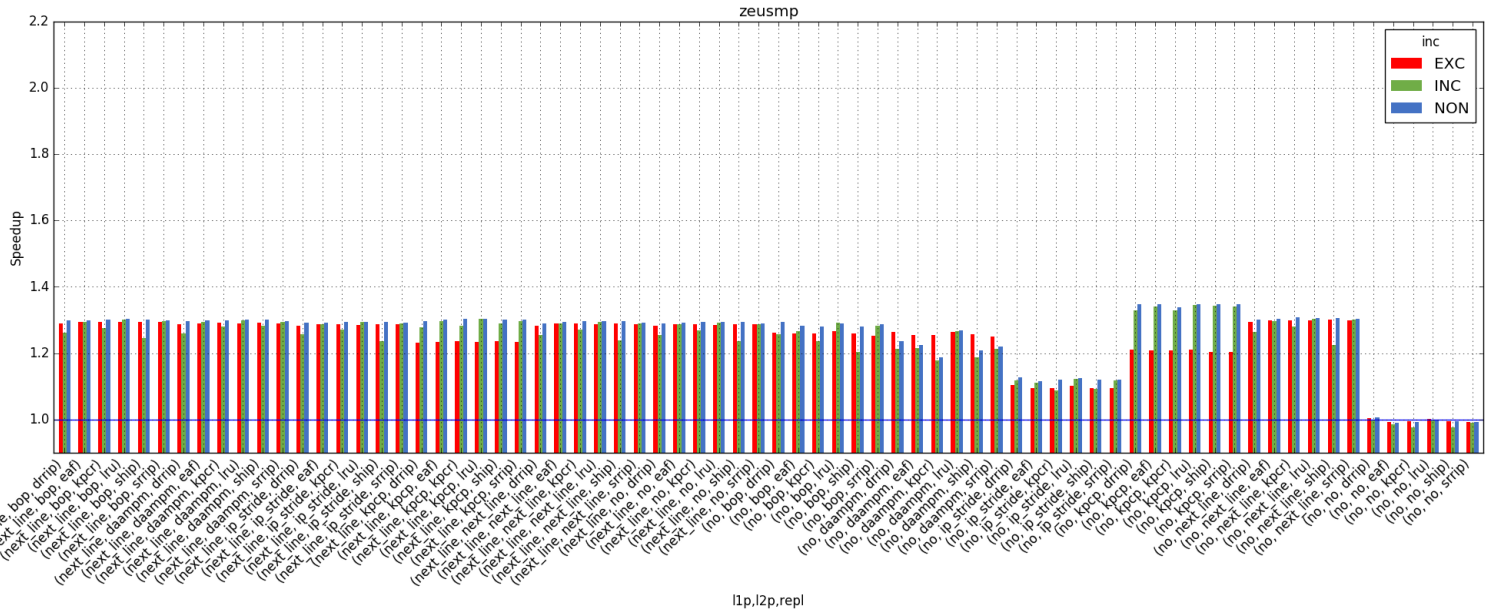
A.20 Wrf



A.21 Xalancbmk



A.22 Zeusmp



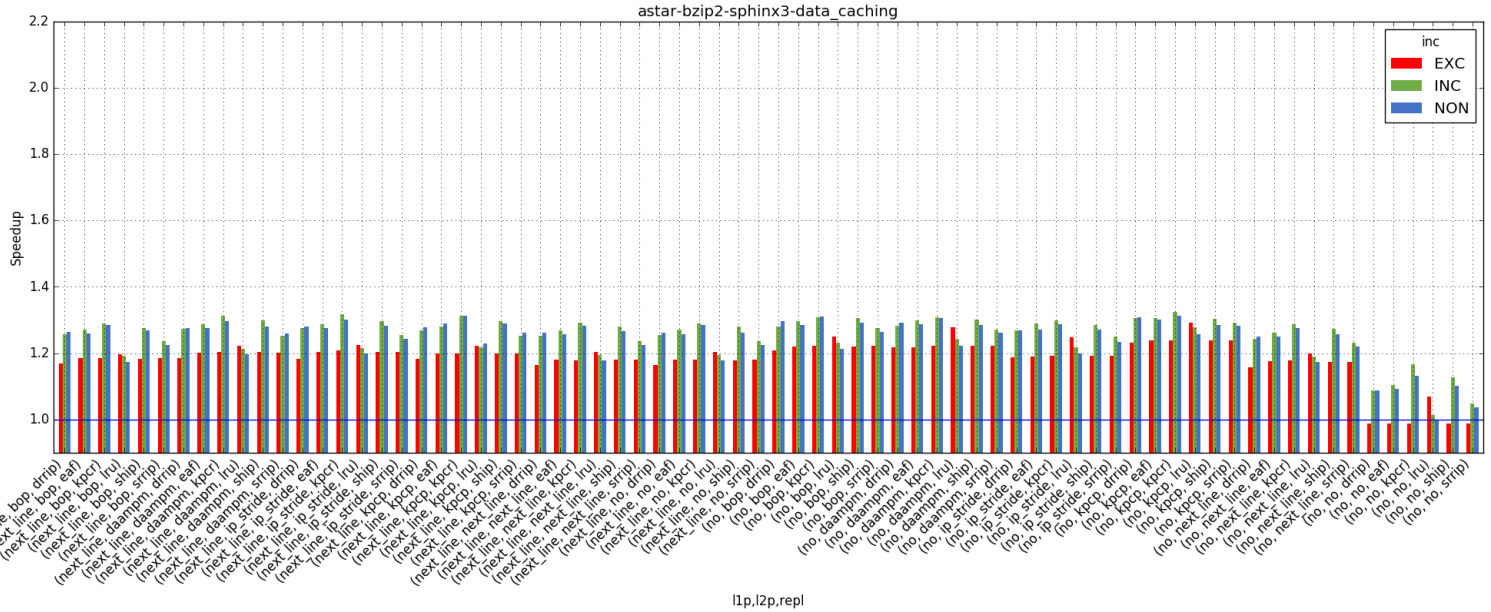
B. MULTI-CORE RESULTS

This appendix contains the multi-core plots for all the benchmarks used. In the main thesis only the geomean speedup plots are showed for readability. Below there is a description of how to read the plots.

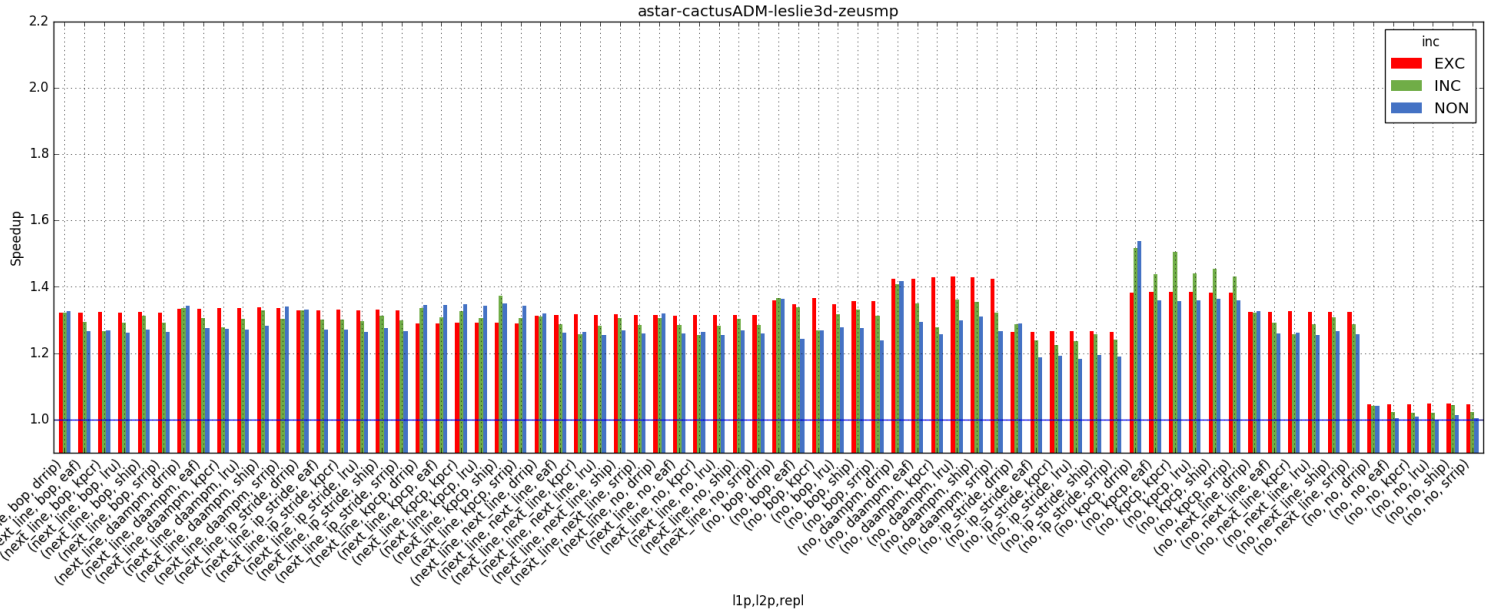
These plots compare different configurations of L1 and L2 prefetchers, replacement policies and cache inclusions. The configurations compared are: L1 prefetcher, L2 prefetcher, replacement policy and cache inclusion.

The title of the plot indicates the name of the benchmark. The Y-axis shows the speedup over the baseline configuration: no prefetchers, LRU replacement policy and a non-inclusive cache. The X-axis shows the different cache configurations, in order of: L1 prefetcher (l1p), L2 prefetcher (l2p) and replacement policy (repl). The inclusion of the cache is shown in the legend: exclusive "EXC", inclusive "INC" and non-inclusive "NON".

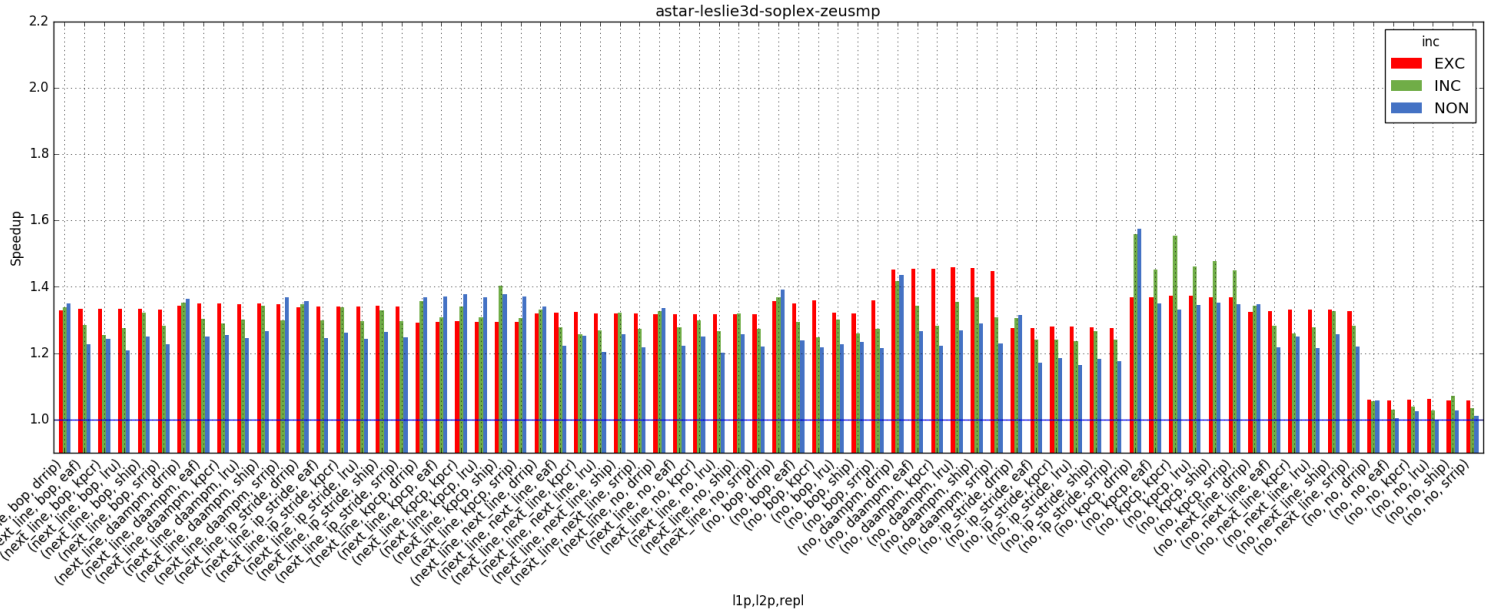
B.1 Astar-Bzip2-Sphinx3-Data_Caching



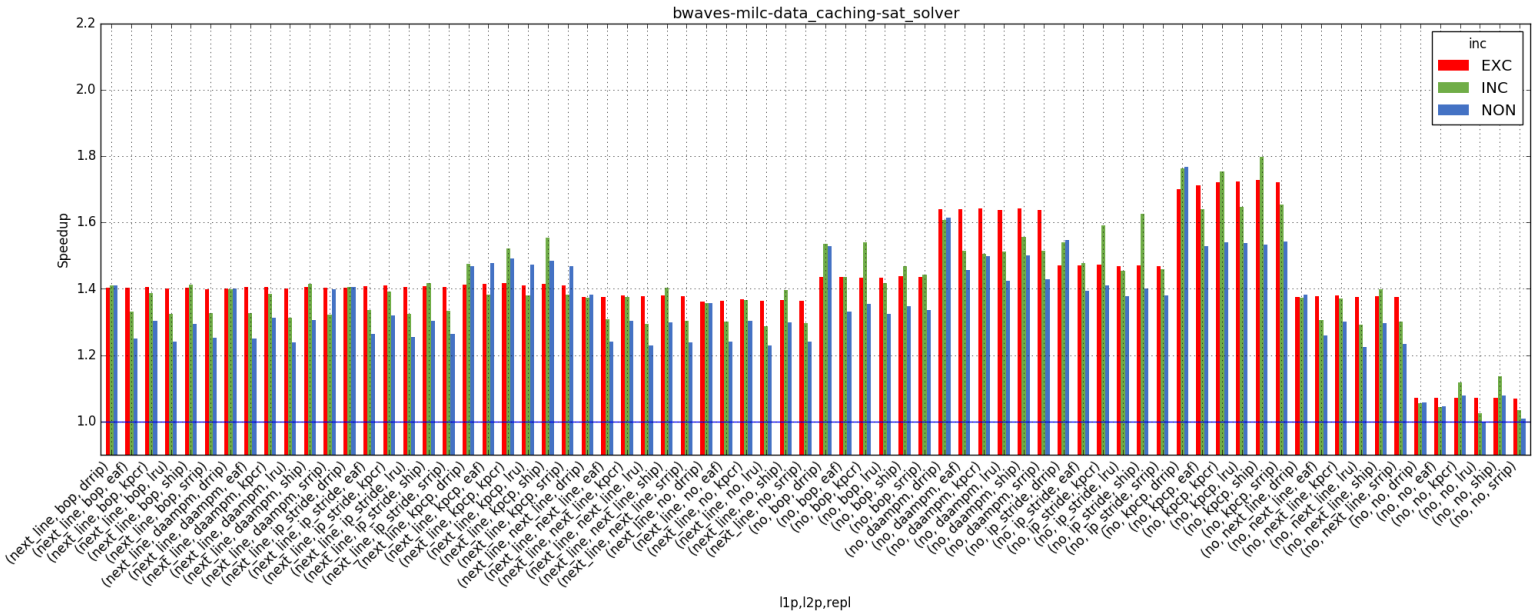
B.2 Astar-Bzip2-Sphinx3-Data_Caching



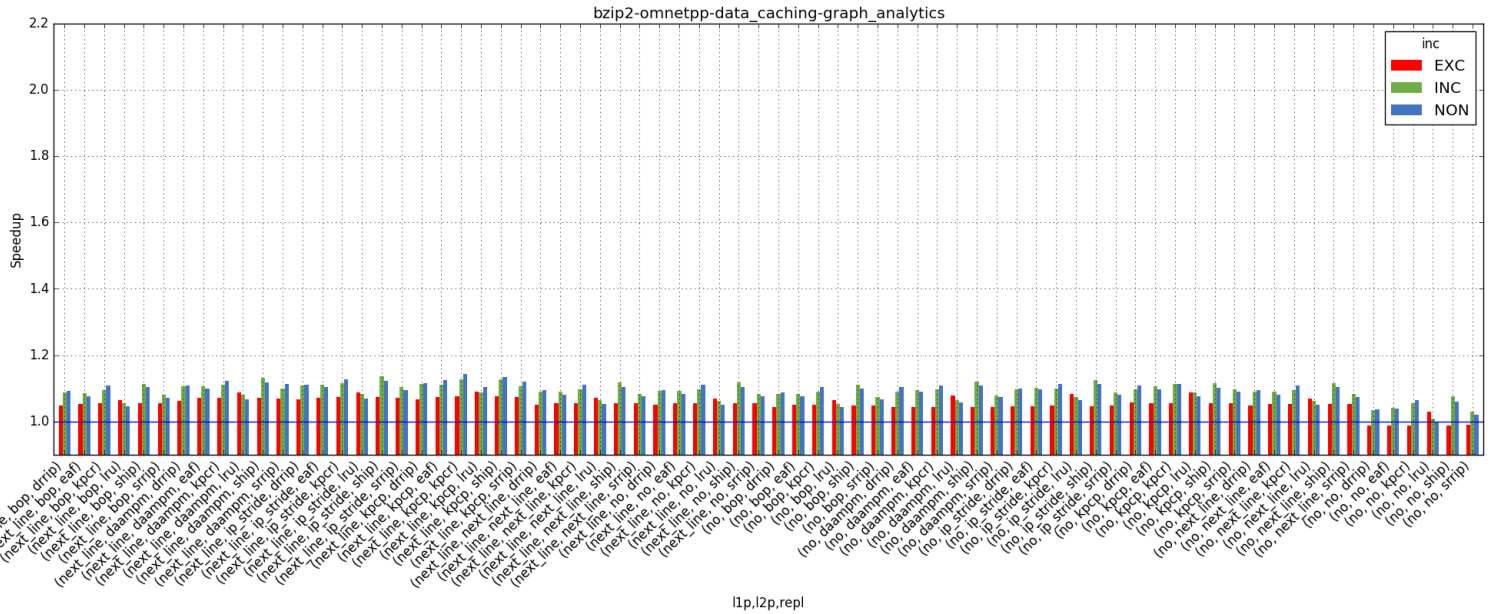
B.3 Astar-Leslie3d-Soplex-Zeusmp



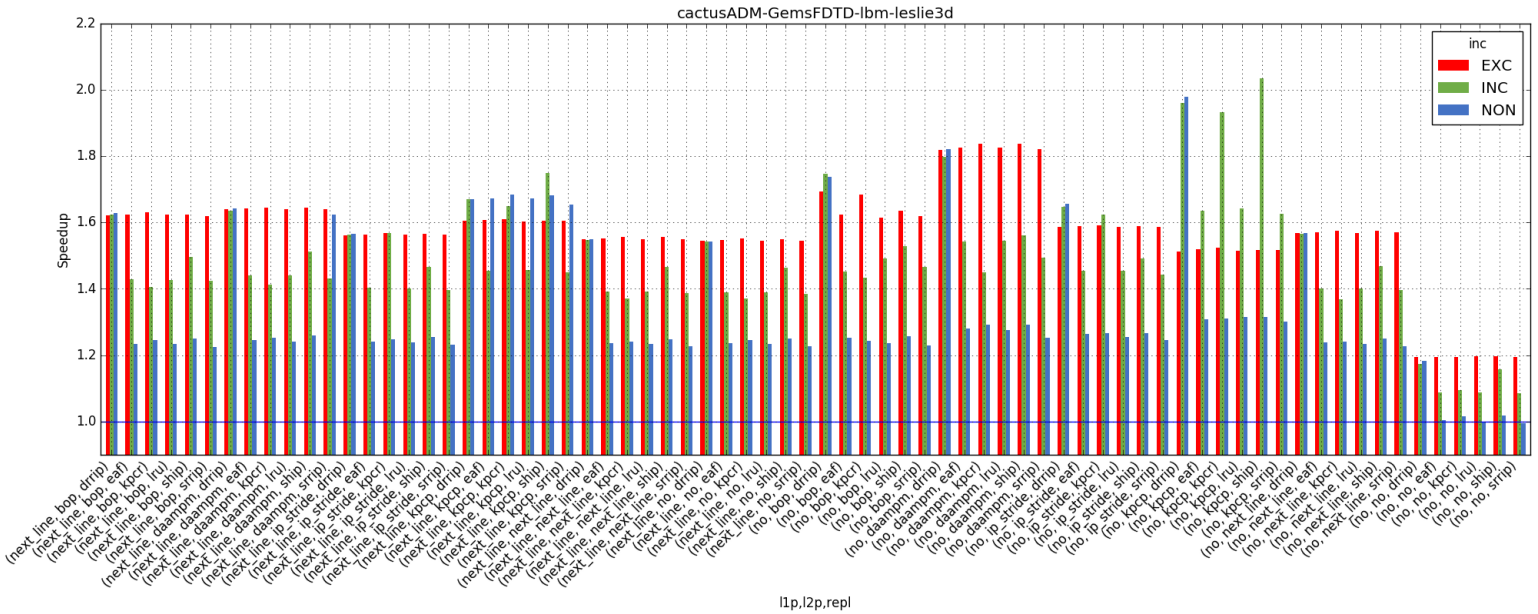
B.4 Bwaves-Milc-Data_Caching-Sat_Solver



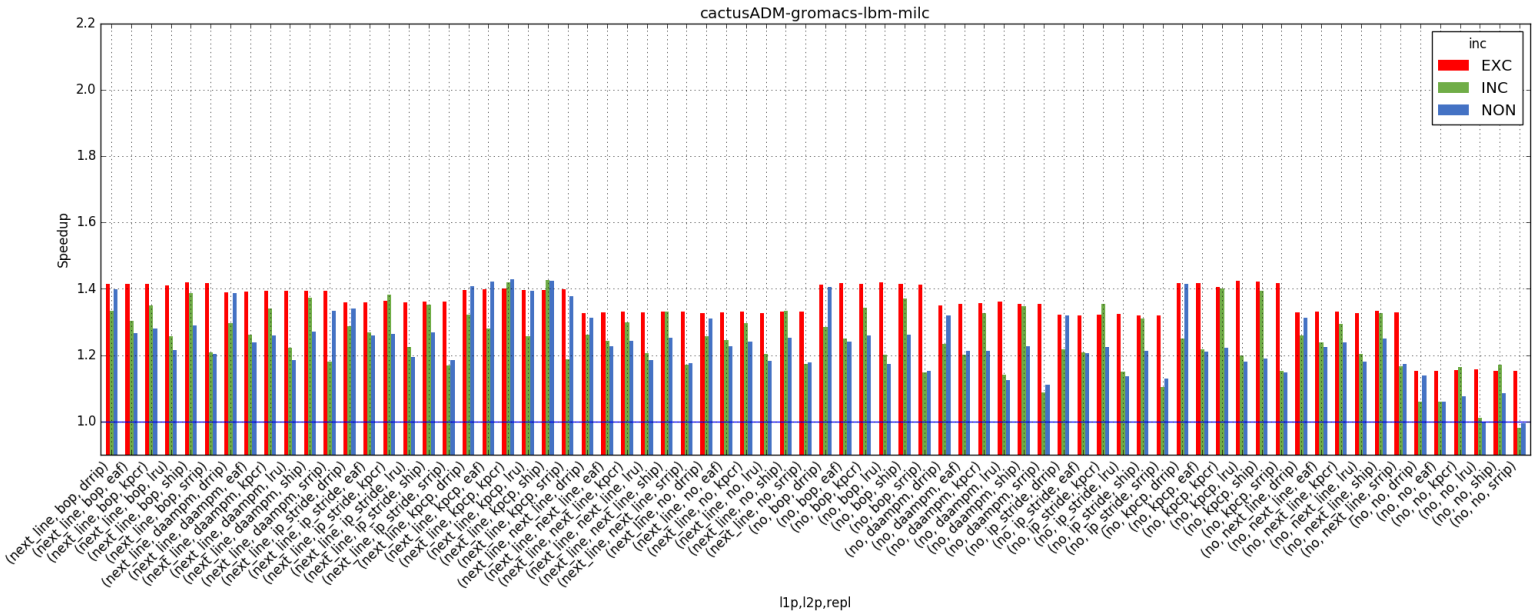
B.5 Bzip2-Omnetpp-Data_Caching-Graph_Analytics



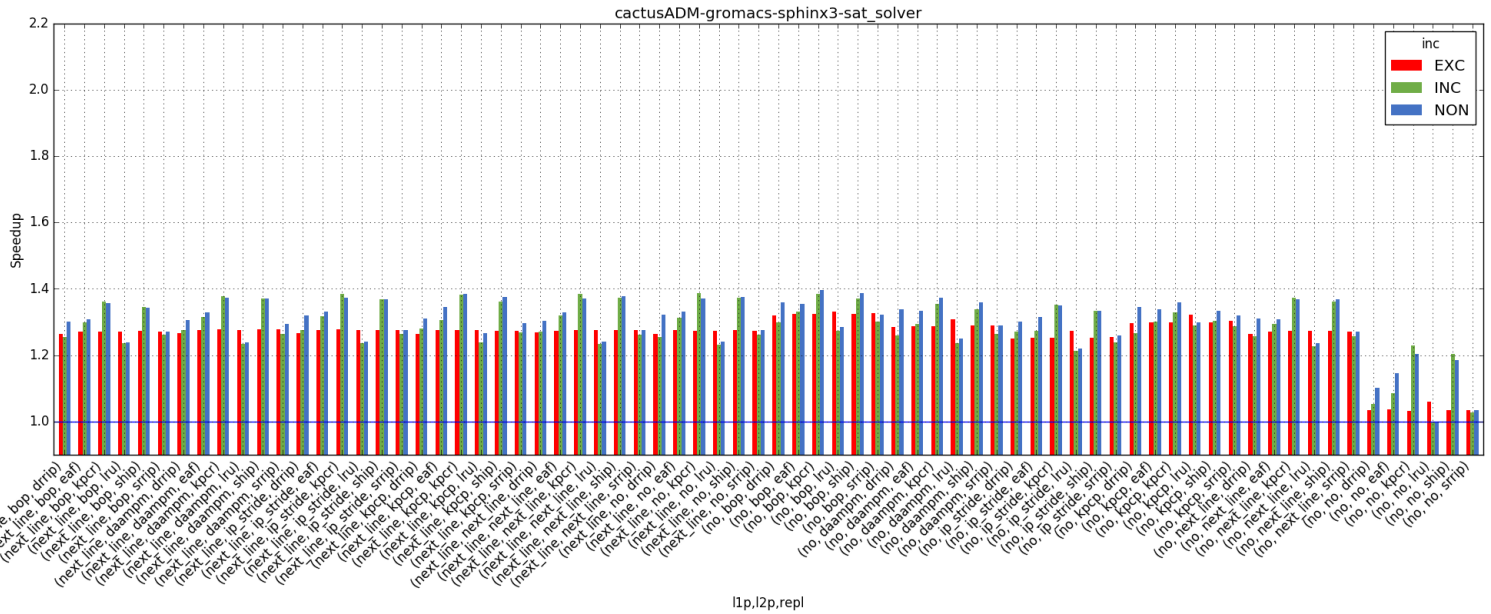
B.6 CactusADM-GemsFDTD-Lbm-Leslie3d



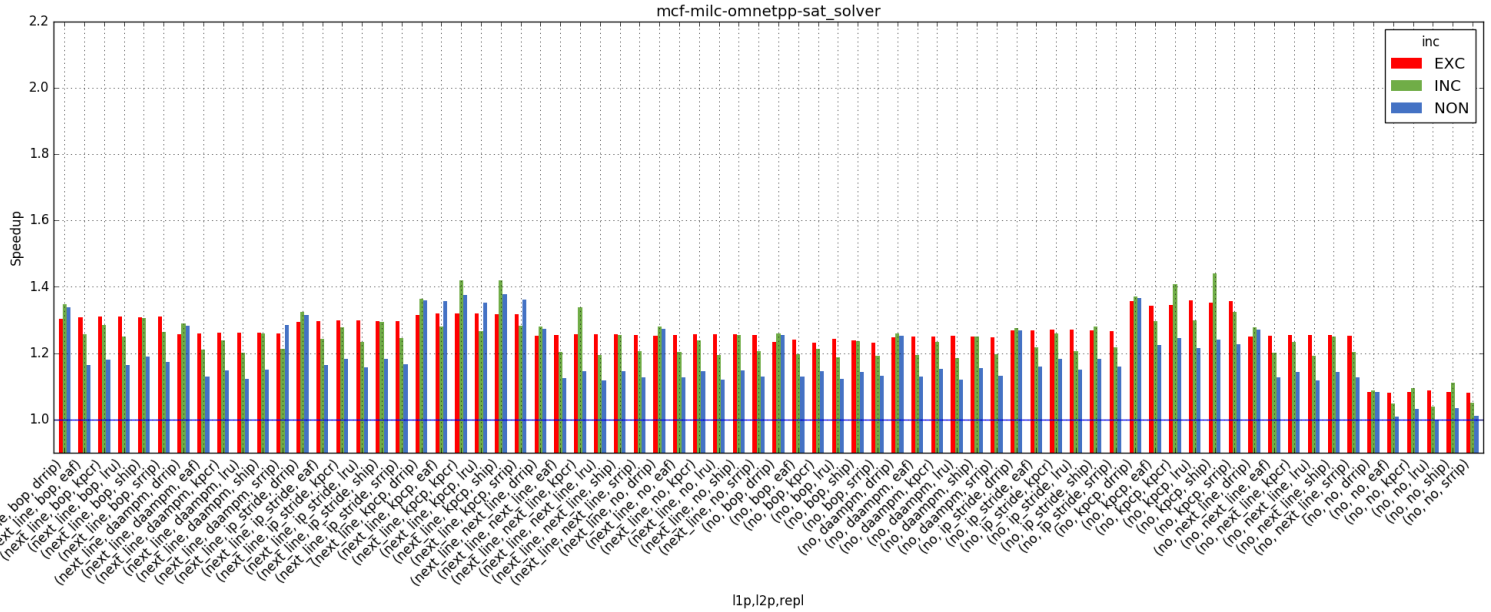
B.7 CactusADM-Gromacs-Lbm-Milc



B.8 CactusADM-Gromacs-Sphinx3-Sat_Solver



B.9 Mcf-Milc-Omnetpp-Sat_Solver



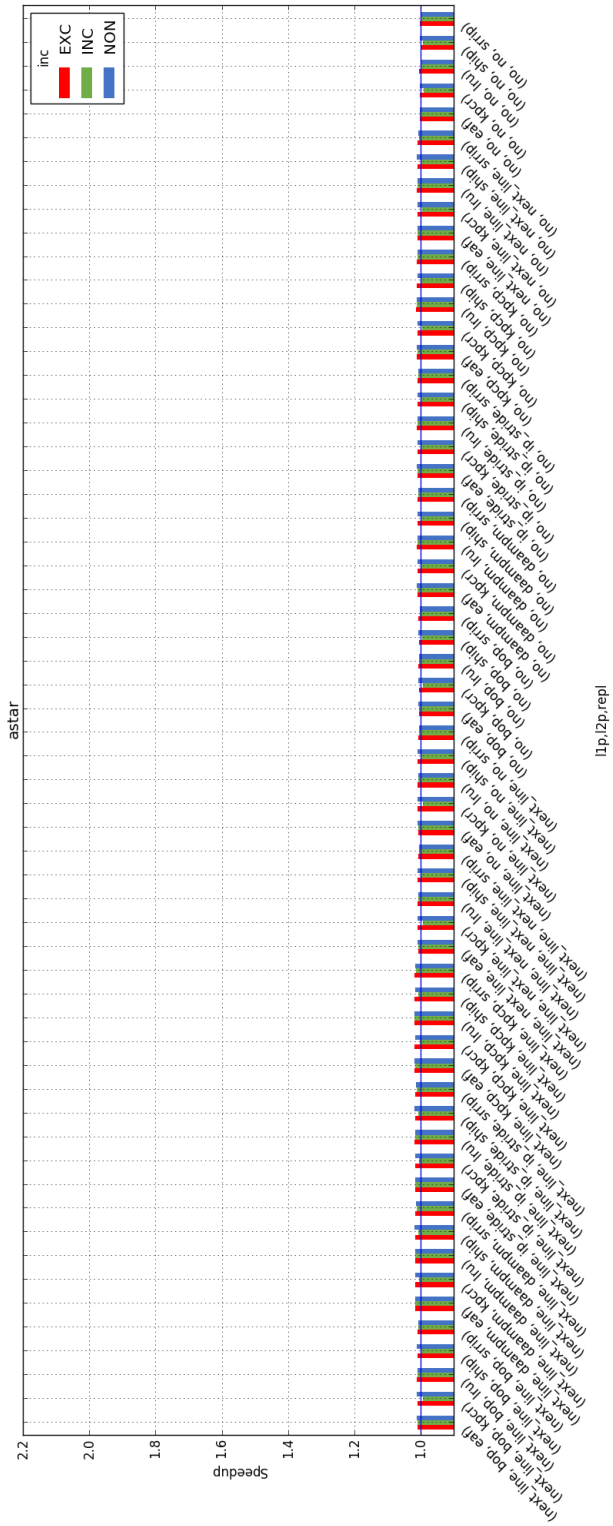
C. SINGLE-CORE SIZE SENSITIVITY RESULTS

This appendix contains the single-core with larger cache plots for all the benchmarks used. In the main thesis only the geomean speedup plots are showed for readability. Below there is a description of how to read the plots.

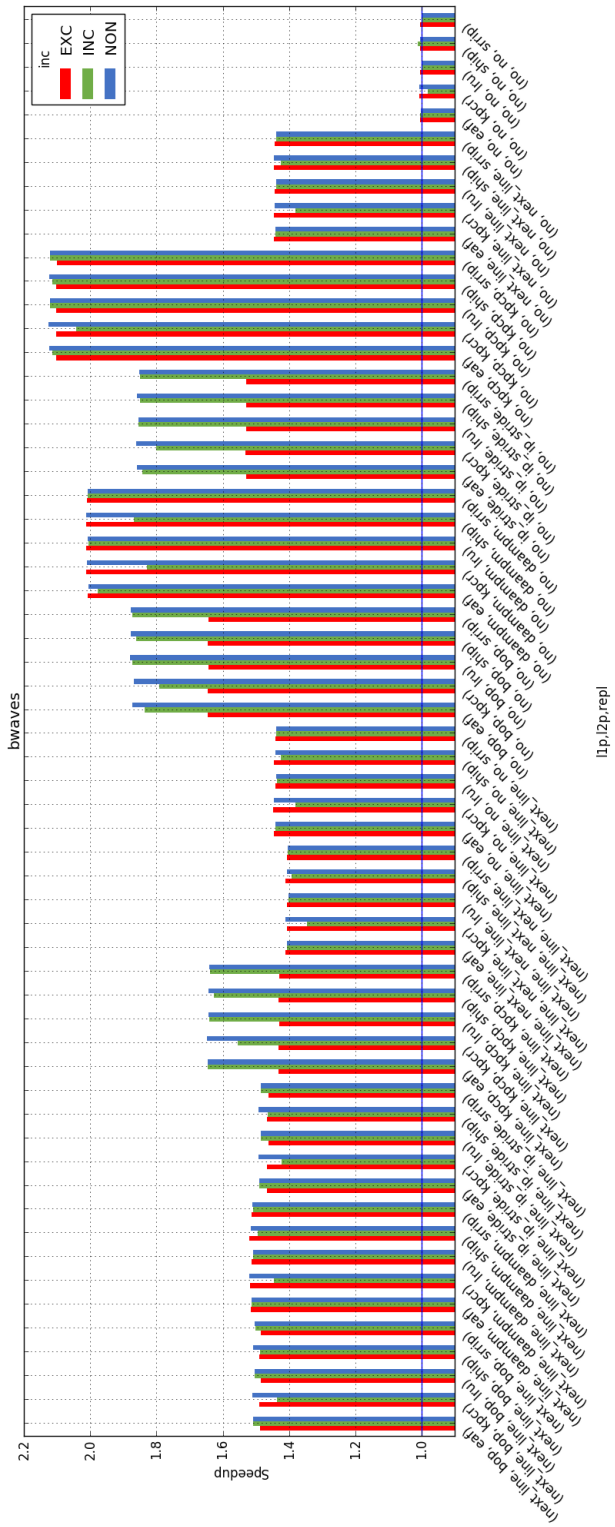
These plots compare different configurations of L1 and L2 prefetchers, replacement policies and cache inclusions. The configurations compared are: L1 prefetcher, L2 prefetcher, replacement policy and cache inclusion.

The title of the plot indicates the name of the benchmark. The Y-axis shows the speedup over the baseline configuration: no prefetchers, LRU replacement policy and a non-inclusive cache. The X-axis shows the different cache configurations, in order of: L1 prefetcher (l1p), L2 prefetcher (l2p) and replacement policy (repl). The inclusion of the cache is shown in the legend: exclusive "EXC", inclusive "INC" and non-inclusive "NON".

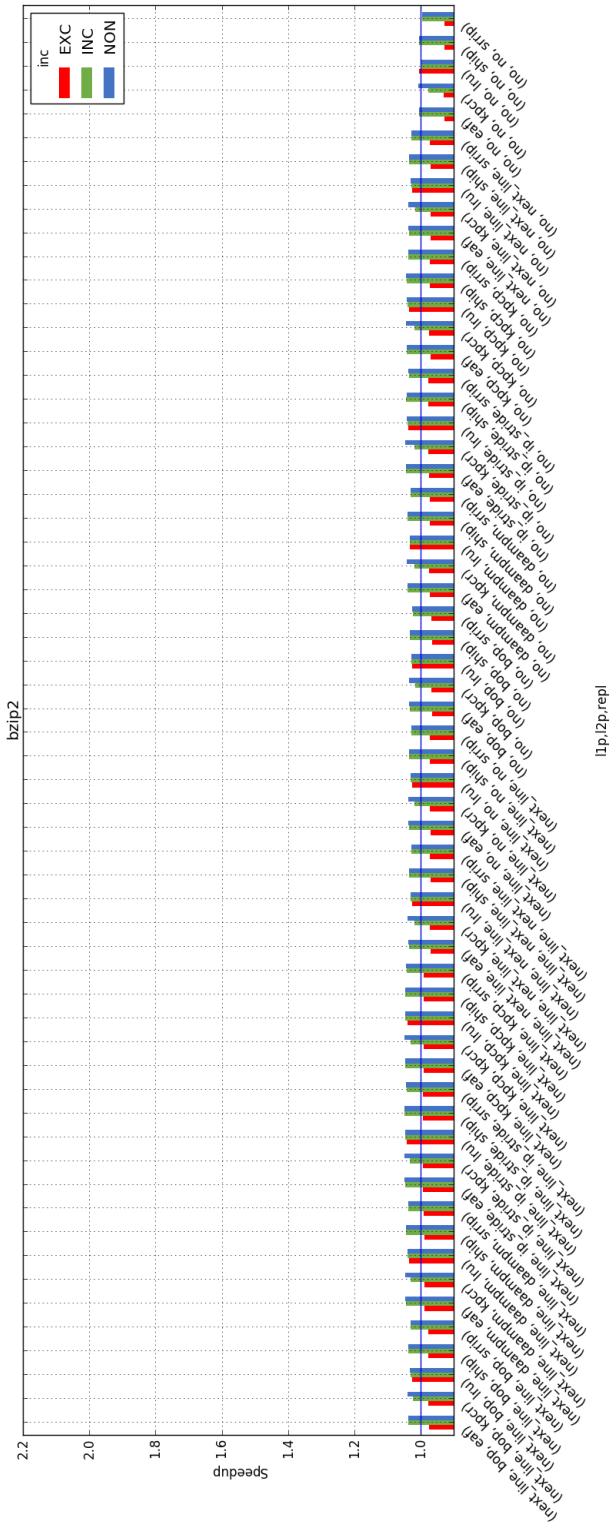
C.1 Astar


astar
Speedup
2.2
2.0
1.8
1.6
1.4
1.2
1.0
inc
EXC
INC
NON
l1p,l2p,repl
(no, no, no)
(no, no, lru)
(no, no, ship)
(no, no, srrip)
(no, no, kpcr)
(no, no, eaf)
(no, next_line, kpcr)
(no, next_line, lru)
(no, next_line, ship)
(no, next_line, srrip)
(no, next_line, eaf)
(no, kpcp, kpcr)
(no, kpcp, lru)
(no, kpcp, ship)
(no, kpcp, srrip)
(no, kpcp, eaf)
(no, ip_stride, kpcr)
(no, ip_stride, lru)
(no, ip_stride, ship)
(no, ip_stride, srrip)
(no, ip_stride, eaf)
(no, daampm, kpcr)
(no, daampm, lru)
(no, daampm, ship)
(no, daampm, srrip)
(no, daampm, eaf)
(no, bop, kpcr)
(no, bop, lru)
(no, bop, ship)
(no, bop, srrip)
(no, bop, eaf)
(next_line, no, kpcr)
(next_line, no, lru)
(next_line, no, ship)
(next_line, no, srrip)
(next_line, no, eaf)
(next_line, next_line, kpcr)
(next_line, next_line, lru)
(next_line, next_line, ship)
(next_line, next_line, srrip)
(next_line, next_line, eaf)
(next_line, kpcp, kpcr)
(next_line, kpcp, lru)
(next_line, kpcp, ship)
(next_line, kpcp, srrip)
(next_line, kpcp, eaf)
(next_line, ip_stride, kpcr)
(next_line, ip_stride, lru)
(next_line, ip_stride, ship)
(next_line, ip_stride, srrip)
(next_line, ip_stride, eaf)
(next_line, daampm, kpcr)
(next_line, daampm, lru)
(next_line, daampm, ship)
(next_line, daampm, srrip)
(next_line, daampm, eaf)
(next_line, bop, kpcr)
(next_line, bop, lru)
(next_line, bop, ship)
(next_line, bop, srrip)
(next_line, bop, eaf)

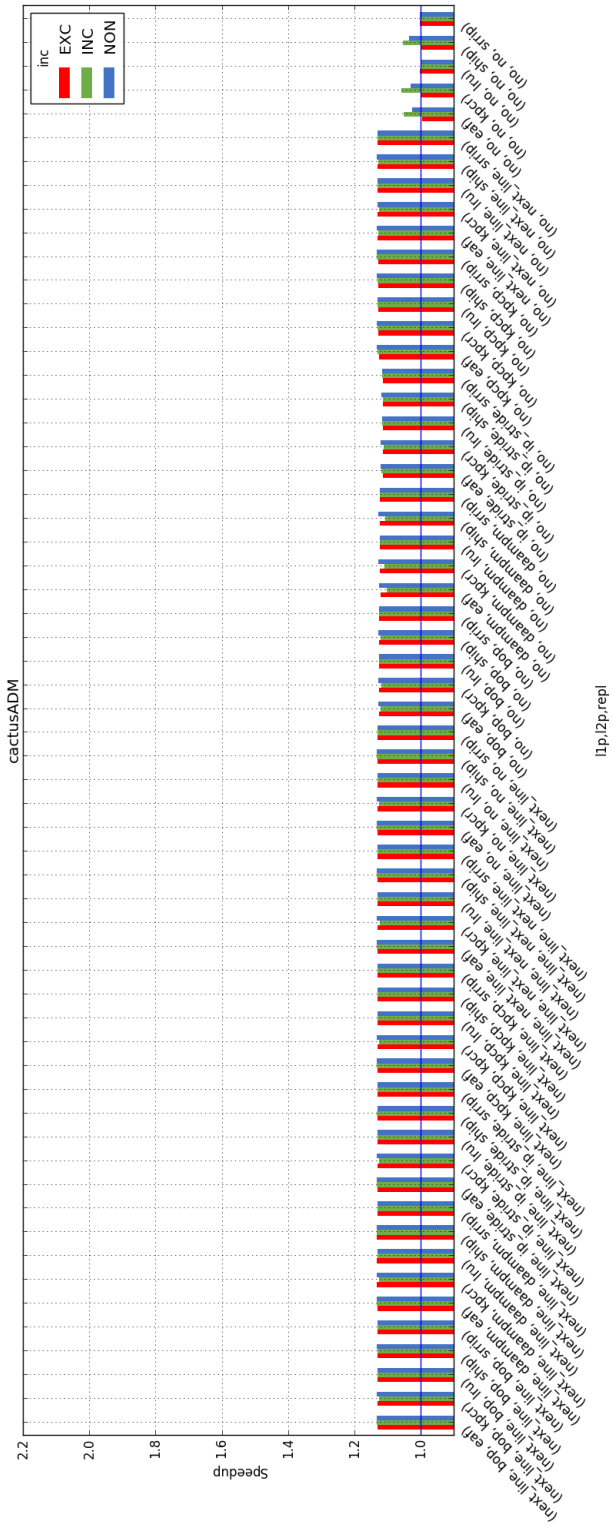
C.2 Bwaves


bwaves
Speedup
2.2
2.0
1.8
1.6
1.4
1.2
1.0
inc
EXC
INC
NON
l1p,l2p,repl
(no, no, no, strip)
(no, no, no, ship)
(no, no, no, lru)
(no, no, no, kpcr)
(no, no, no, eaf)
(no, next_line, no, strip)
(no, next_line, no, ship)
(no, next_line, no, lru)
(no, next_line, no, kpcr)
(no, next_line, no, eaf)
(no, kpcp, no, strip)
(no, kpcp, no, ship)
(no, kpcp, no, lru)
(no, kpcp, no, kpcr)
(no, kpcp, no, eaf)
(no, ip_stride, no, strip)
(no, ip_stride, no, ship)
(no, ip_stride, no, lru)
(no, ip_stride, no, kpcr)
(no, ip_stride, no, eaf)
(no, daampm, no, strip)
(no, daampm, no, ship)
(no, daampm, no, lru)
(no, daampm, no, kpcr)
(no, daampm, no, eaf)
(no, bop, no, strip)
(no, bop, no, ship)
(no, bop, no, lru)
(no, bop, no, kpcr)
(no, bop, no, eaf)
(next_line, no, no, strip)
(next_line, no, no, ship)
(next_line, no, no, lru)
(next_line, no, no, kpcr)
(next_line, no, no, eaf)
(next_line, next_line, no, strip)
(next_line, next_line, no, ship)
(next_line, next_line, no, lru)
(next_line, next_line, no, kpcr)
(next_line, next_line, no, eaf)
(next_line, kpcp, no, strip)
(next_line, kpcp, no, ship)
(next_line, kpcp, no, lru)
(next_line, kpcp, no, kpcr)
(next_line, kpcp, no, eaf)
(next_line, ip_stride, no, strip)
(next_line, ip_stride, no, ship)
(next_line, ip_stride, no, lru)
(next_line, ip_stride, no, kpcr)
(next_line, ip_stride, no, eaf)
(next_line, daampm, no, strip)
(next_line, daampm, no, ship)
(next_line, daampm, no, lru)
(next_line, daampm, no, kpcr)
(next_line, daampm, no, eaf)
(next_line, bop, no, strip)
(next_line, bop, no, ship)
(next_line, bop, no, lru)
(next_line, bop, no, kpcr)
(next_line, bop, no, eaf)

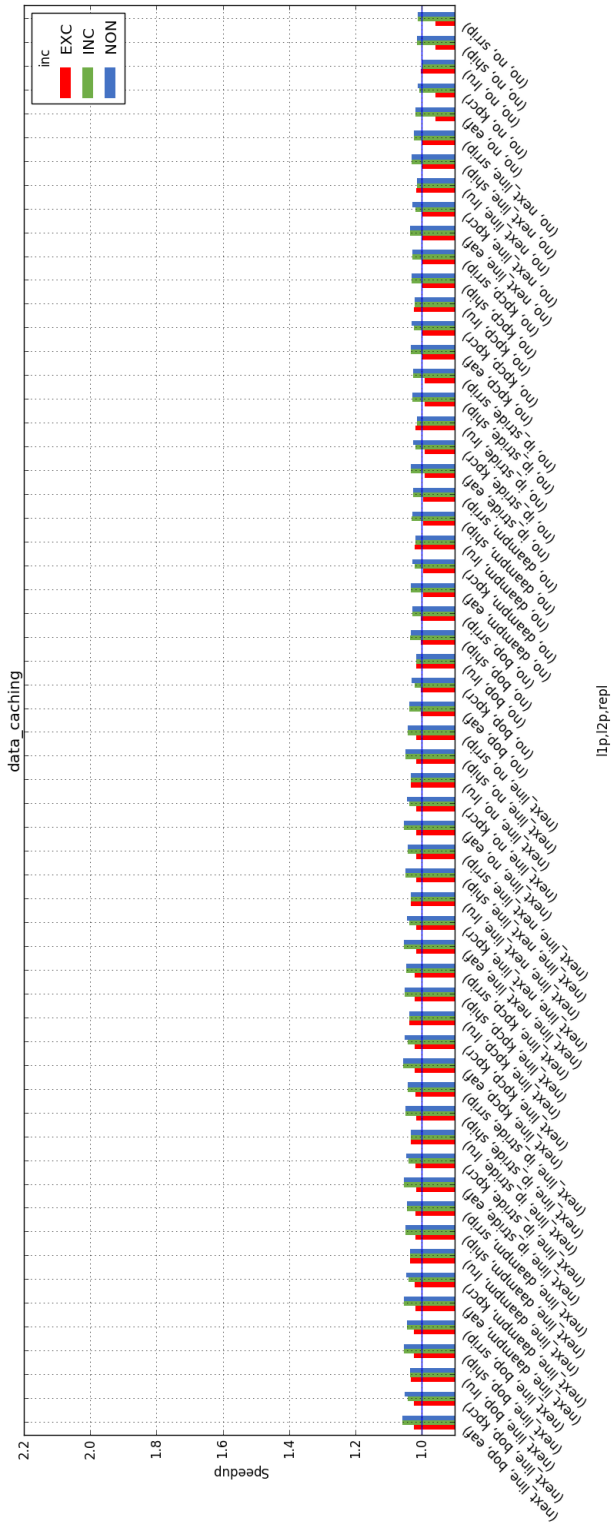
C.3 Bzip2


bzip2
Speedup
2.2
2.0
1.8
1.6
1.4
1.2
1.0
inc
EXC
INC
NON
l1p,l2p,repl
(no, no, no)
(no, no, lru)
(no, no, kpcr)
(no, no, eelf)
(no, no, srrip)
(no, next_line, no)
(no, next_line, lru)
(no, next_line, kpcr)
(no, next_line, eelf)
(no, next_line, srrip)
(no, kpcp, no)
(no, kpcp, lru)
(no, kpcp, kpcr)
(no, kpcp, eelf)
(no, kpcp, srrip)
(no, ip_stride, no)
(no, ip_stride, lru)
(no, ip_stride, kpcr)
(no, ip_stride, eelf)
(no, ip_stride, srrip)
(no, daampm, no)
(no, daampm, lru)
(no, daampm, kpcr)
(no, daampm, eelf)
(no, daampm, srrip)
(no, bop, no)
(no, bop, lru)
(no, bop, kpcr)
(no, bop, eelf)
(no, bop, srrip)
(next_line, no, no)
(next_line, no, lru)
(next_line, no, kpcr)
(next_line, no, eelf)
(next_line, no, srrip)
(next_line, next_line, no)
(next_line, next_line, lru)
(next_line, next_line, kpcr)
(next_line, next_line, eelf)
(next_line, next_line, srrip)
(next_line, kpcp, no)
(next_line, kpcp, lru)
(next_line, kpcp, kpcr)
(next_line, kpcp, eelf)
(next_line, kpcp, srrip)
(next_line, ip_stride, no)
(next_line, ip_stride, lru)
(next_line, ip_stride, kpcr)
(next_line, ip_stride, eelf)
(next_line, ip_stride, srrip)
(next_line, daampm, no)
(next_line, daampm, lru)
(next_line, daampm, kpcr)
(next_line, daampm, eelf)
(next_line, daampm, srrip)
(next_line, bop, no)
(next_line, bop, lru)
(next_line, bop, kpcr)
(next_line, bop, eelf)
(next_line, bop, srrip)

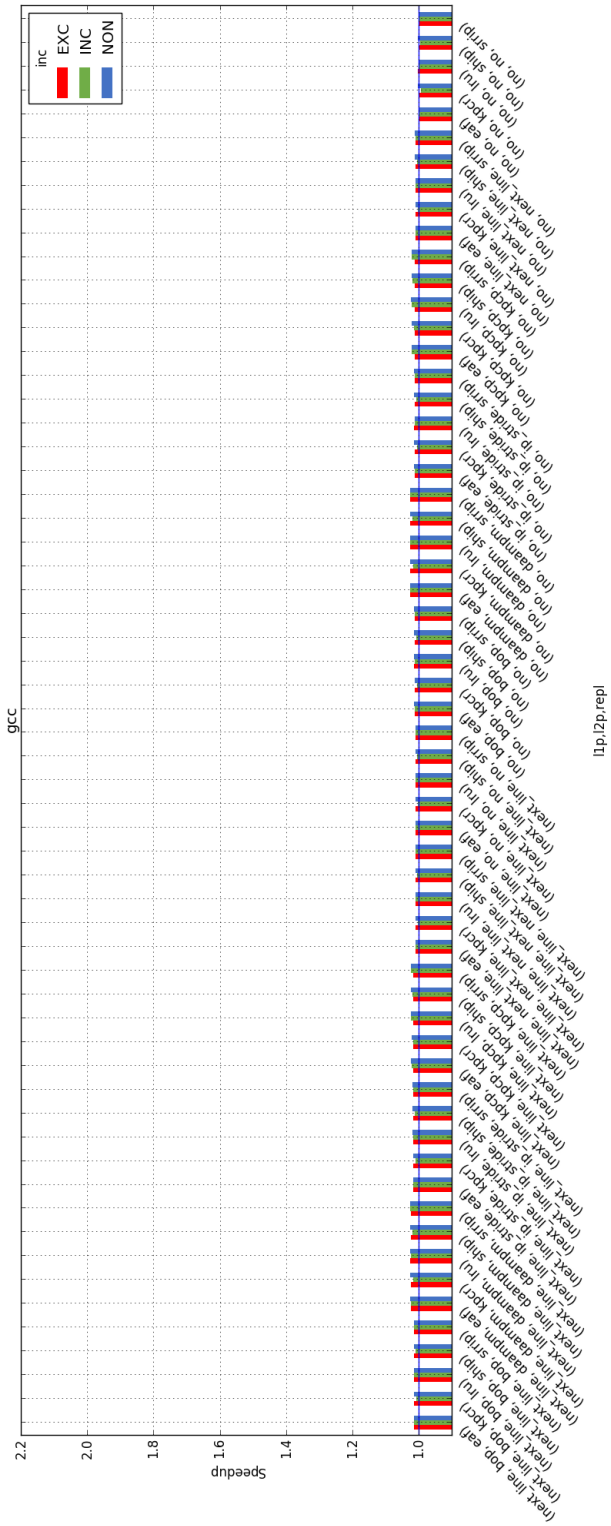
C.4 CactusADM



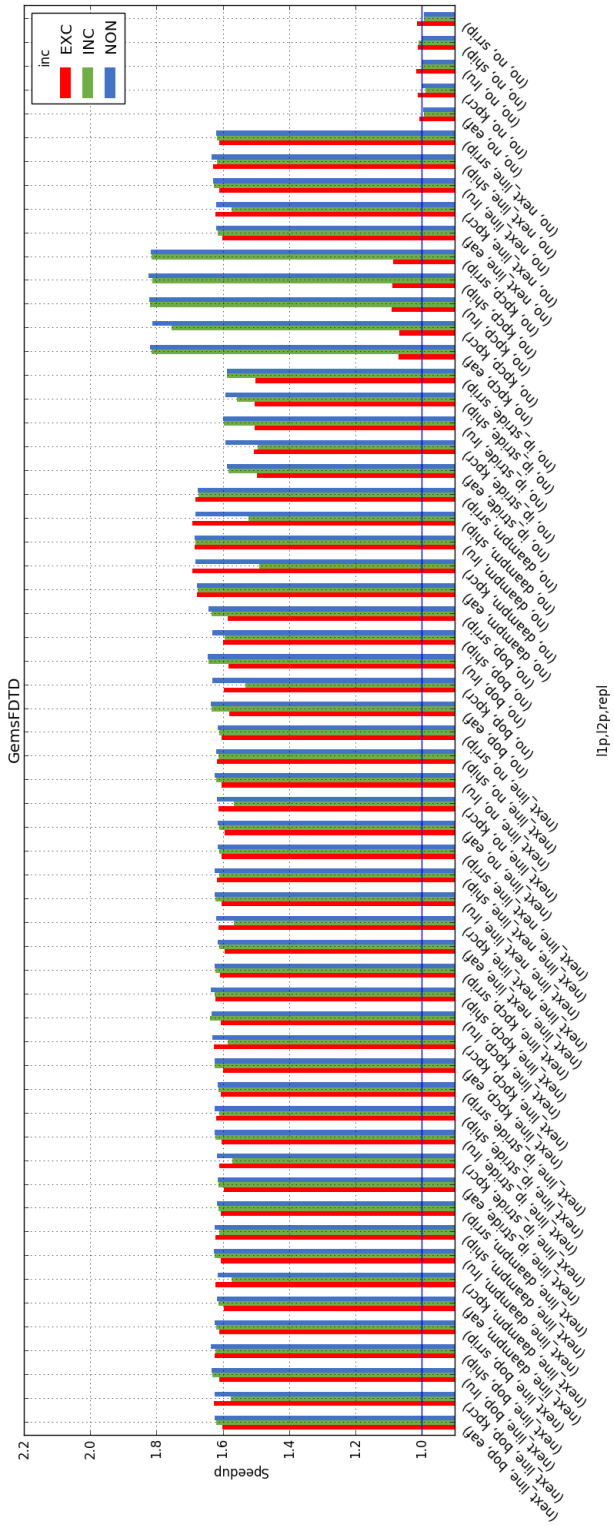
C.5 Data_Caching



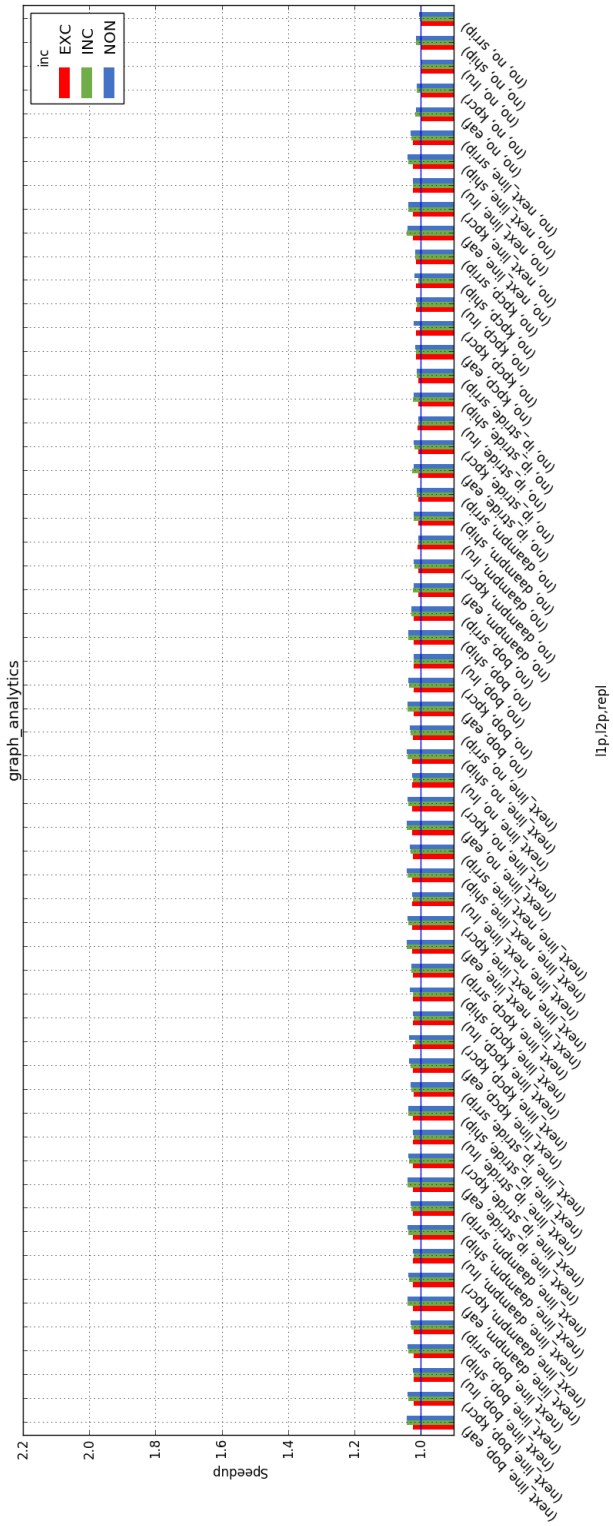
C.6 Gcc


gcc
Speedup
2.2
2.0
1.8
1.6
1.4
1.2
1.0
inc
EXC
INC
NON
l1p,l2p,repl
(next_line, bop, kcr)
(next_line, bop, lru)
(next_line, bop, srrip)
(next_line, bop, eaf)
(next_line, no, kcr)
(next_line, no, lru)
(next_line, no, srrip)
(next_line, no, eaf)
(next_line, daampm, kcr)
(next_line, daampm, lru)
(next_line, daampm, srrip)
(next_line, daampm, eaf)
(next_line, ip_stride, kcr)
(next_line, ip_stride, lru)
(next_line, ip_stride, srrip)
(next_line, ip_stride, eaf)
(next_line, kpcp, kcr)
(next_line, kpcp, lru)
(next_line, kpcp, srrip)
(next_line, kpcp, eaf)
(no, bop, kcr)
(no, bop, lru)
(no, bop, srrip)
(no, bop, eaf)
(no, daampm, kcr)
(no, daampm, lru)
(no, daampm, srrip)
(no, daampm, eaf)
(no, ip_stride, kcr)
(no, ip_stride, lru)
(no, ip_stride, srrip)
(no, ip_stride, eaf)
(no, kpcp, kcr)
(no, kpcp, lru)
(no, kpcp, srrip)
(no, kpcp, eaf)
(no, next_line, kcr)
(no, next_line, lru)
(no, next_line, srrip)
(no, next_line, eaf)
(no, no, kcr)
(no, no, lru)
(no, no, srrip)

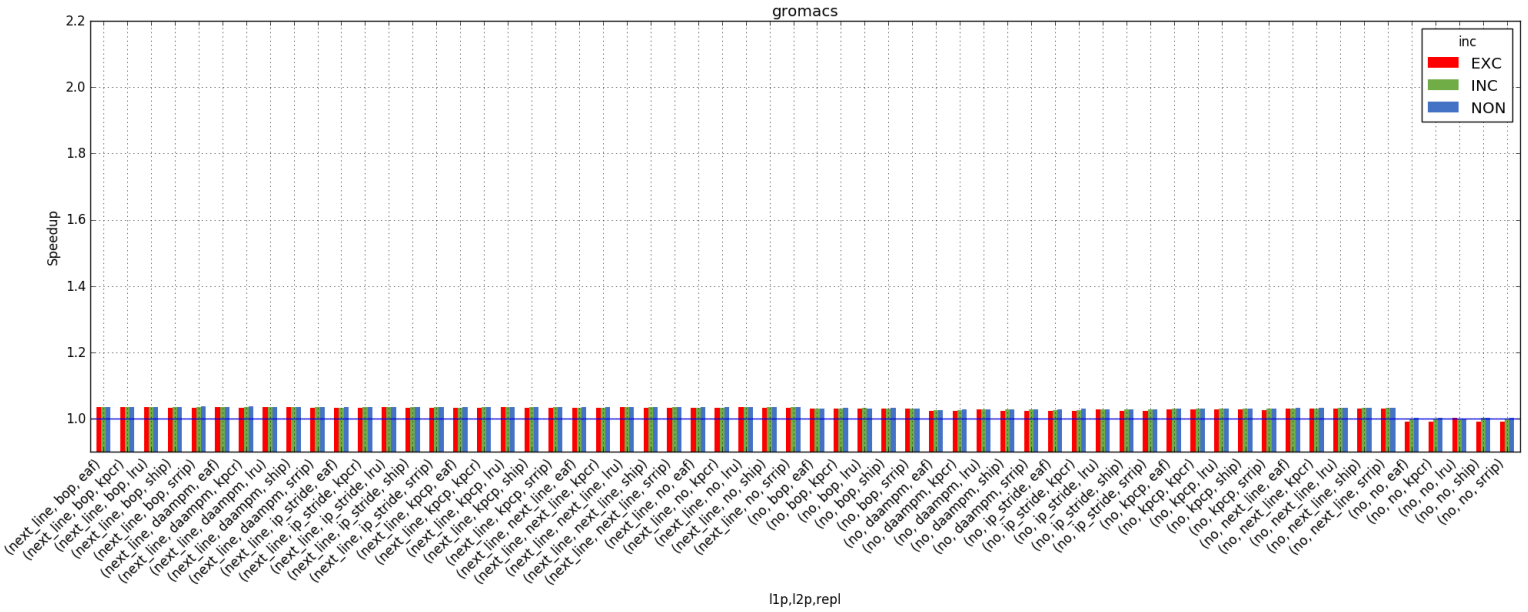
C.7 GemsFTD


GemsFDTD
Speedup
2.2
2.0
1.8
1.6
1.4
1.2
1.0
inc
EXC
INC
NON
l1p,l2p,repl
(next_line, bop, eaf)
(next_line, bop, kpcr)
(next_line, bop, lru)
(next_line, bop, ship)
(next_line, bop, srrip)
(next_line, deamnpm, eaf)
(next_line, deamnpm, kpcr)
(next_line, deamnpm, lru)
(next_line, deamnpm, ship)
(next_line, deamnpm, srrip)
(next_line, ip_stride, eaf)
(next_line, ip_stride, kpcr)
(next_line, ip_stride, lru)
(next_line, ip_stride, ship)
(next_line, ip_stride, srrip)
(next_line, kpcp, eaf)
(next_line, kpcp, kpcr)
(next_line, kpcp, lru)
(next_line, kpcp, ship)
(next_line, kpcp, srrip)
(next_line, next_line, eaf)
(next_line, next_line, kpcr)
(next_line, next_line, lru)
(next_line, next_line, ship)
(next_line, next_line, srrip)
(next_line, no, eaf)
(next_line, no, kpcr)
(next_line, no, lru)
(next_line, no, ship)
(next_line, no, srrip)
(no, bop, eaf)
(no, bop, kpcr)
(no, bop, lru)
(no, bop, ship)
(no, bop, srrip)
(no, deamnpm, eaf)
(no, deamnpm, kpcr)
(no, deamnpm, lru)
(no, deamnpm, ship)
(no, deamnpm, srrip)
(no, ip_stride, eaf)
(no, ip_stride, kpcr)
(no, ip_stride, lru)
(no, ip_stride, ship)
(no, ip_stride, srrip)
(no, kpcp, eaf)
(no, kpcp, kpcr)
(no, kpcp, lru)
(no, kpcp, ship)
(no, kpcp, srrip)
(no, next_line, eaf)
(no, next_line, kpcr)
(no, next_line, lru)
(no, next_line, ship)
(no, next_line, srrip)
(no, no, eaf)
(no, no, kpcr)
(no, no, lru)
(no, no, ship)
(no, no, srrip)

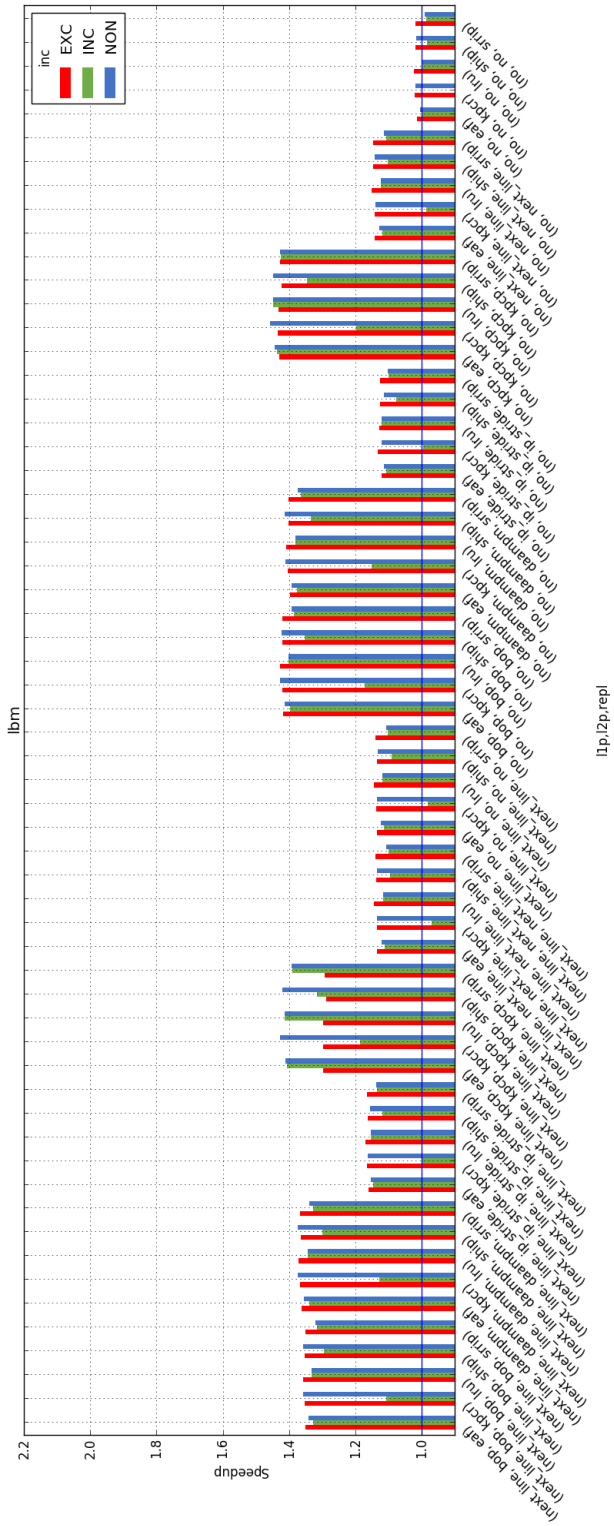
C.8 Graph_Analytics



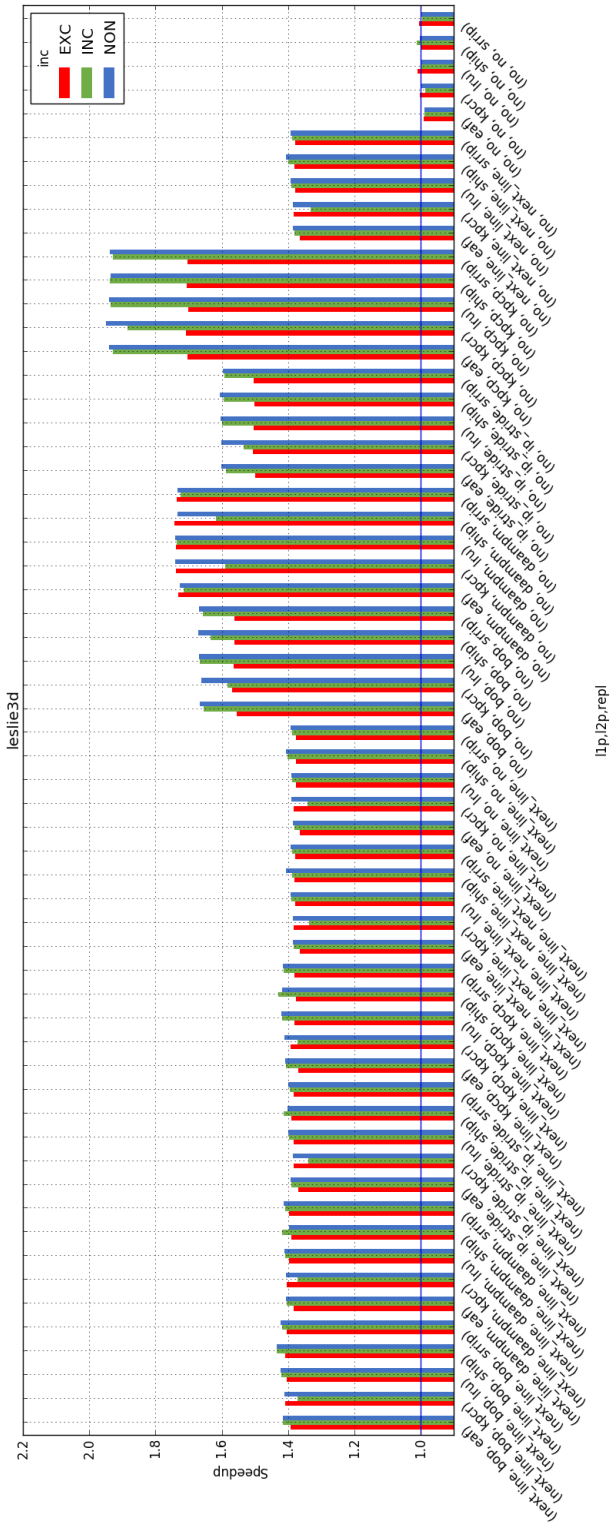
C.9 Gromacs


gromacs
Speedup
2.2
2.0
1.8
1.6
1.4
1.2
1.0
inc
EXC
INC
NON
l1p,l2p,repl

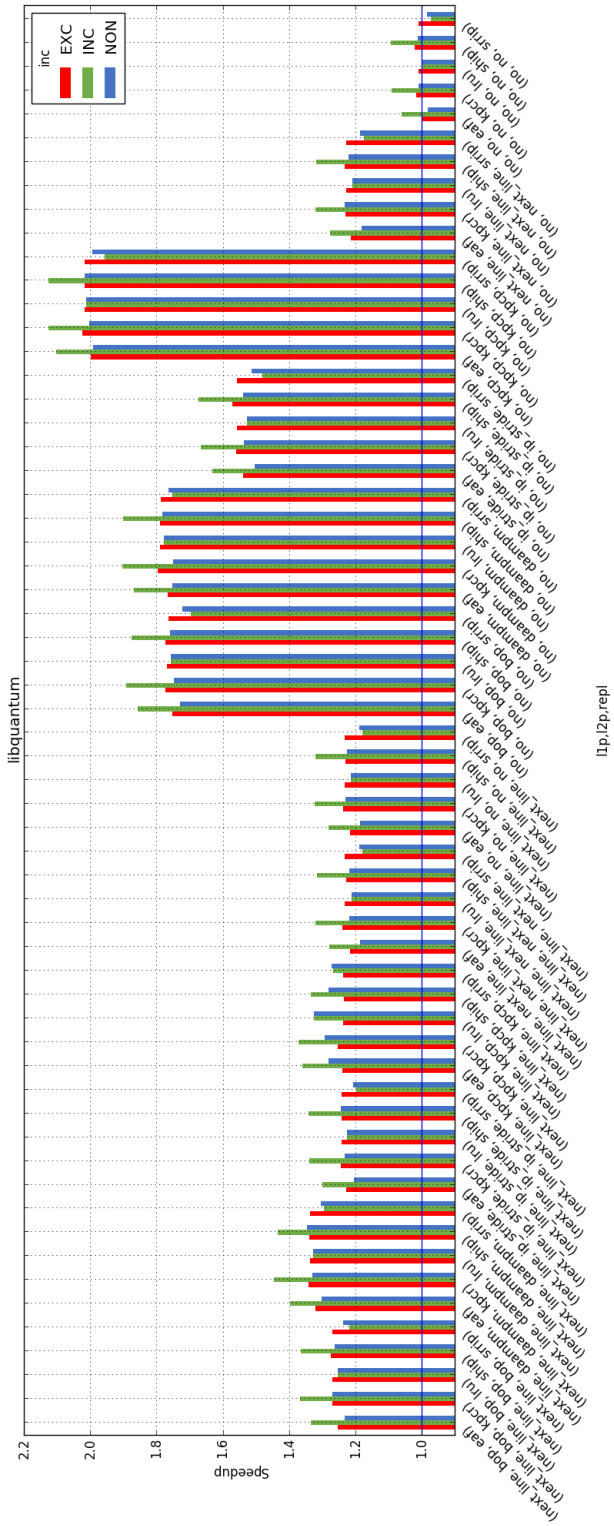
C.10 Lbm


lbm
Speedup
2.2
2.0
1.8
1.6
1.4
1.2
1.0
inc
EXC
INC
NON
l1p,l2p,repl
(no, no, no)
(no, no, shp)
(no, no, lru)
(no, no, kpcr)
(no, no, eafj)
(no, next_line, strp)
(no, next_line, shp)
(no, next_line, lru)
(no, next_line, kpcr)
(no, next_line, eafj)
(no, kpcp, strp)
(no, kpcp, shp)
(no, kpcp, lru)
(no, kpcp, kpcr)
(no, kpcp, eafj)
(no, ip_stride, strp)
(no, ip_stride, shp)
(no, ip_stride, lru)
(no, ip_stride, kpcr)
(no, ip_stride, eafj)
(no, daampm, strp)
(no, daampm, shp)
(no, daampm, lru)
(no, daampm, kpcr)
(no, daampm, eafj)
(no, bop, strp)
(no, bop, shp)
(no, bop, lru)
(no, bop, kpcr)
(no, bop, eafj)
(next_line, no, strp)
(next_line, no, shp)
(next_line, no, lru)
(next_line, no, kpcr)
(next_line, no, eafj)
(next_line, next_line, strp)
(next_line, next_line, shp)
(next_line, next_line, lru)
(next_line, next_line, kpcr)
(next_line, next_line, eafj)
(next_line, kpcp, strp)
(next_line, kpcp, shp)
(next_line, kpcp, lru)
(next_line, kpcp, kpcr)
(next_line, kpcp, eafj)
(next_line, ip_stride, strp)
(next_line, ip_stride, shp)
(next_line, ip_stride, lru)
(next_line, ip_stride, kpcr)
(next_line, ip_stride, eafj)
(next_line, daampm, strp)
(next_line, daampm, shp)
(next_line, daampm, lru)
(next_line, daampm, kpcr)
(next_line, daampm, eafj)
(next_line, bop, strp)
(next_line, bop, shp)
(next_line, bop, lru)
(next_line, bop, kpcr)
(next_line, bop, eafj)

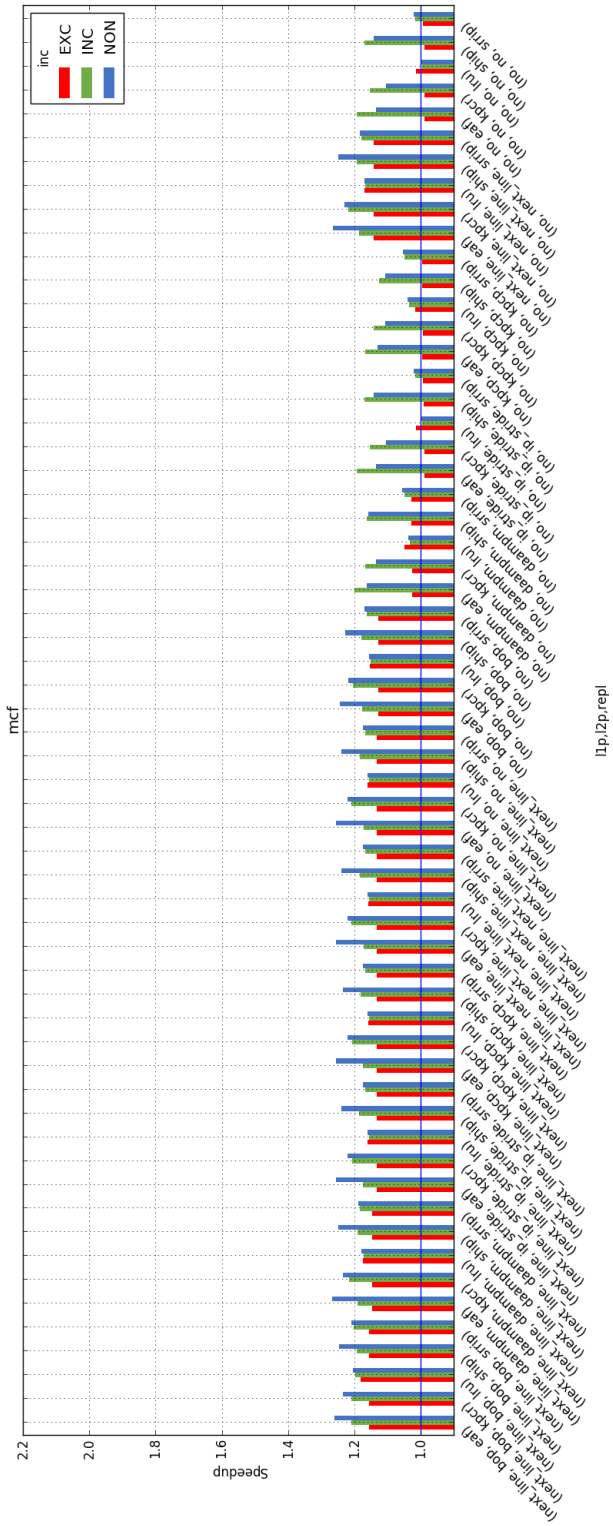
C.11 Leslie3d



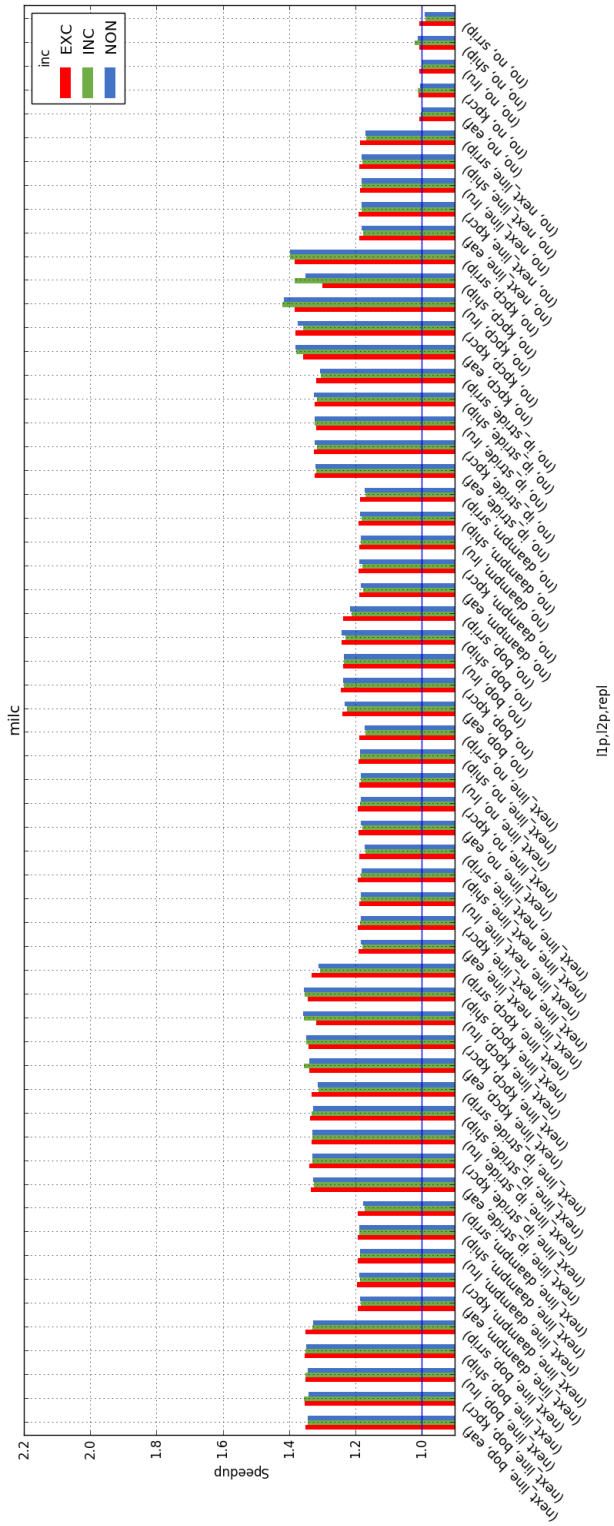
C.12 Libquantum



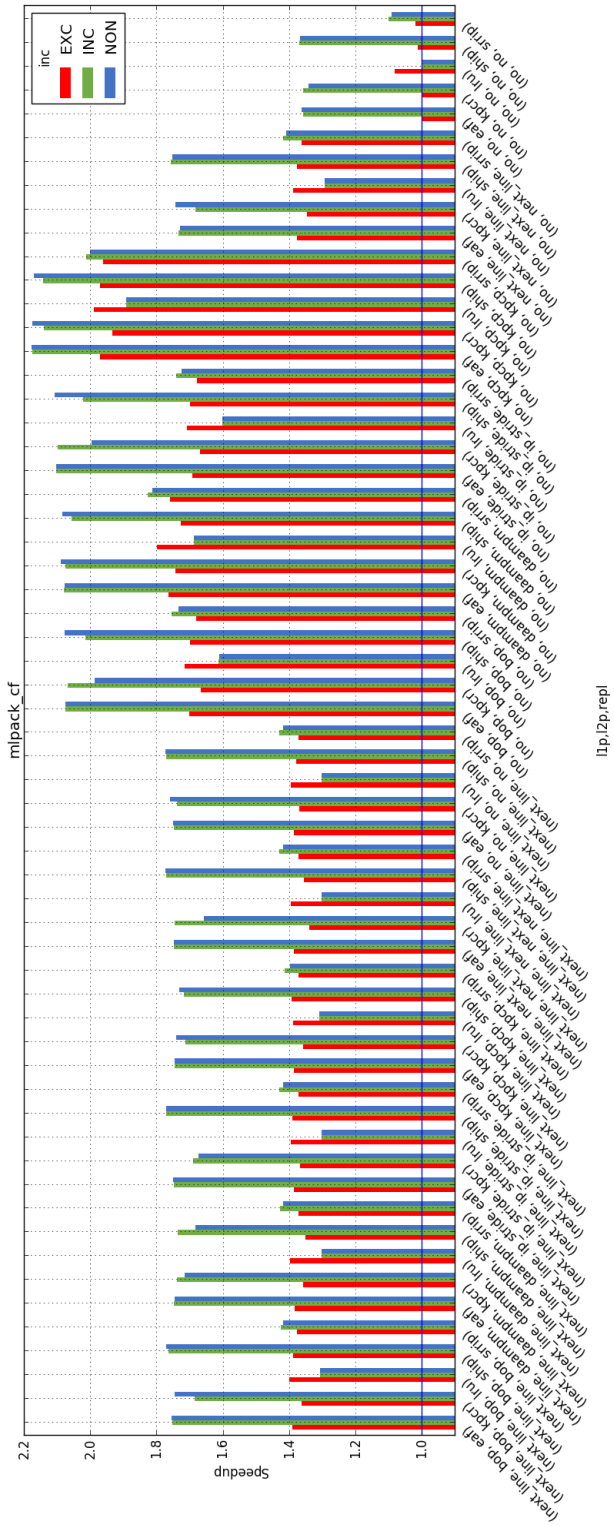
C.13 Mcf


mcf
Speedup
2.2
2.0
1.8
1.6
1.4
1.2
1.0
inc
EXC
INC
NON
l1p,l2p,repl
(next_line, bop, kpcr)
(next_line, bop, lru)
(next_line, bop, ship)
(next_line, bop, srrip)
(next_line, daampm, eaf)
(next_line, daampm, kpcr)
(next_line, daampm, lru)
(next_line, daampm, ship)
(next_line, daampm, srrip)
(next_line, ip_stride, eaf)
(next_line, ip_stride, kpcr)
(next_line, ip_stride, lru)
(next_line, ip_stride, ship)
(next_line, ip_stride, srrip)
(next_line, kpcp, eaf)
(next_line, kpcp, kpcr)
(next_line, kpcp, lru)
(next_line, kpcp, ship)
(next_line, kpcp, srrip)
(next_line, next_line, eaf)
(next_line, next_line, kpcr)
(next_line, next_line, lru)
(next_line, next_line, ship)
(next_line, next_line, srrip)
(next_line, no, eaf)
(next_line, no, kpcr)
(next_line, no, lru)
(next_line, no, ship)
(next_line, no, srrip)
(no, bop, eaf)
(no, bop, kpcr)
(no, bop, lru)
(no, bop, ship)
(no, bop, srrip)
(no, daampm, eaf)
(no, daampm, kpcr)
(no, daampm, lru)
(no, daampm, ship)
(no, daampm, srrip)
(no, ip_stride, eaf)
(no, ip_stride, kpcr)
(no, ip_stride, lru)
(no, ip_stride, ship)
(no, ip_stride, srrip)
(no, kpcp, eaf)
(no, kpcp, kpcr)
(no, kpcp, lru)
(no, kpcp, ship)
(no, kpcp, srrip)
(no, next_line, eaf)
(no, next_line, kpcr)
(no, next_line, lru)
(no, next_line, ship)
(no, next_line, srrip)
(no, no, eaf)
(no, no, kpcr)
(no, no, lru)
(no, no, ship)
(no, no, srrip)

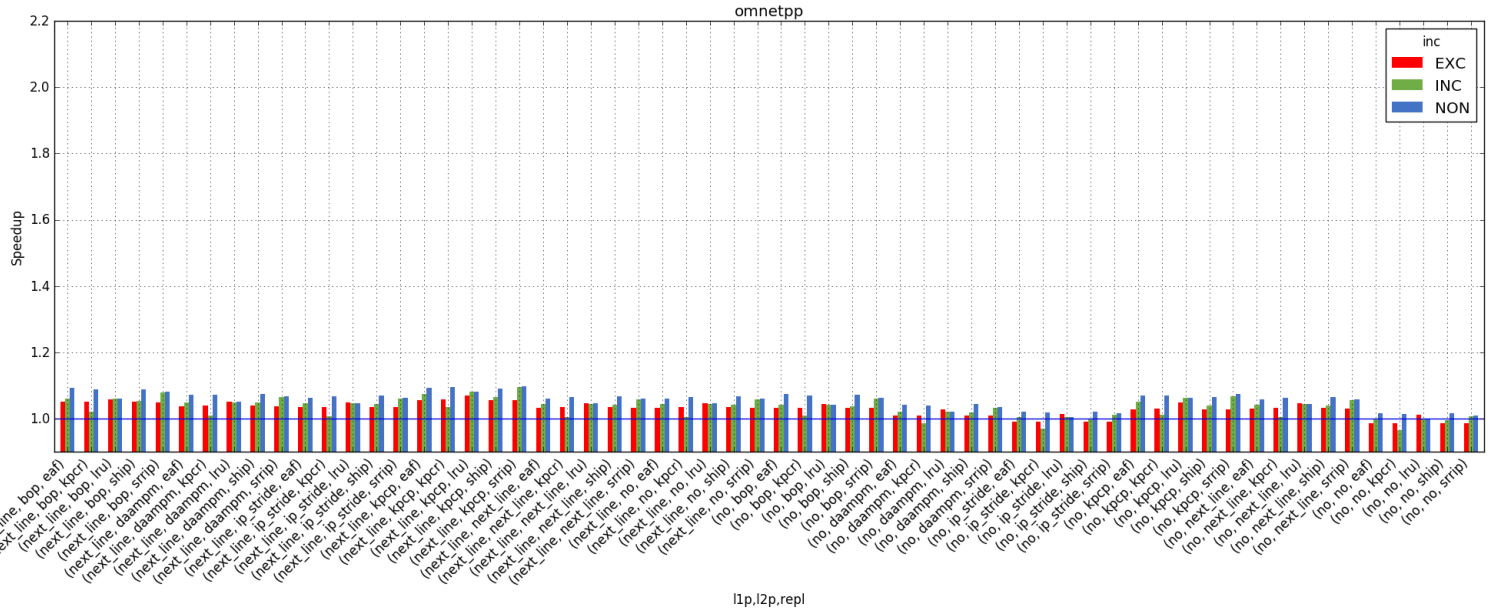
C.14 Milc


milc
Speedup
2.2
2.0
1.8
1.6
1.4
1.2
1.0
inc
EXC
INC
NON
l1p,l2p,repl
(next_line, bop, eaf)
(next_line, bop, kpcr)
(next_line, bop, lru)
(next_line, bop, shlp)
(next_line, bop, strp)
(next_line, daampm, eaf)
(next_line, daampm, kpcr)
(next_line, daampm, lru)
(next_line, daampm, shlp)
(next_line, daampm, strp)
(next_line, ip_stride, eaf)
(next_line, ip_stride, kpcr)
(next_line, ip_stride, lru)
(next_line, ip_stride, shlp)
(next_line, ip_stride, strp)
(next_line, kpcp, eaf)
(next_line, kpcp, kpcr)
(next_line, kpcp, lru)
(next_line, kpcp, shlp)
(next_line, kpcp, strp)
(next_line, next_line, eaf)
(next_line, next_line, kpcr)
(next_line, next_line, lru)
(next_line, next_line, shlp)
(next_line, next_line, strp)
(next_line, no, eaf)
(next_line, no, kpcr)
(next_line, no, lru)
(next_line, no, shlp)
(next_line, no, strp)
(no, bop, eaf)
(no, bop, kpcr)
(no, bop, lru)
(no, bop, shlp)
(no, bop, strp)
(no, daampm, eaf)
(no, daampm, kpcr)
(no, daampm, lru)
(no, daampm, shlp)
(no, daampm, strp)
(no, ip_stride, eaf)
(no, ip_stride, kpcr)
(no, ip_stride, lru)
(no, ip_stride, shlp)
(no, ip_stride, strp)
(no, kpcp, eaf)
(no, kpcp, kpcr)
(no, kpcp, lru)
(no, kpcp, shlp)
(no, kpcp, strp)
(no, next_line, eaf)
(no, next_line, kpcr)
(no, next_line, lru)
(no, next_line, shlp)
(no, next_line, strp)
(no, no, eaf)
(no, no, kpcr)
(no, no, lru)
(no, no, shlp)
(no, no, strp)

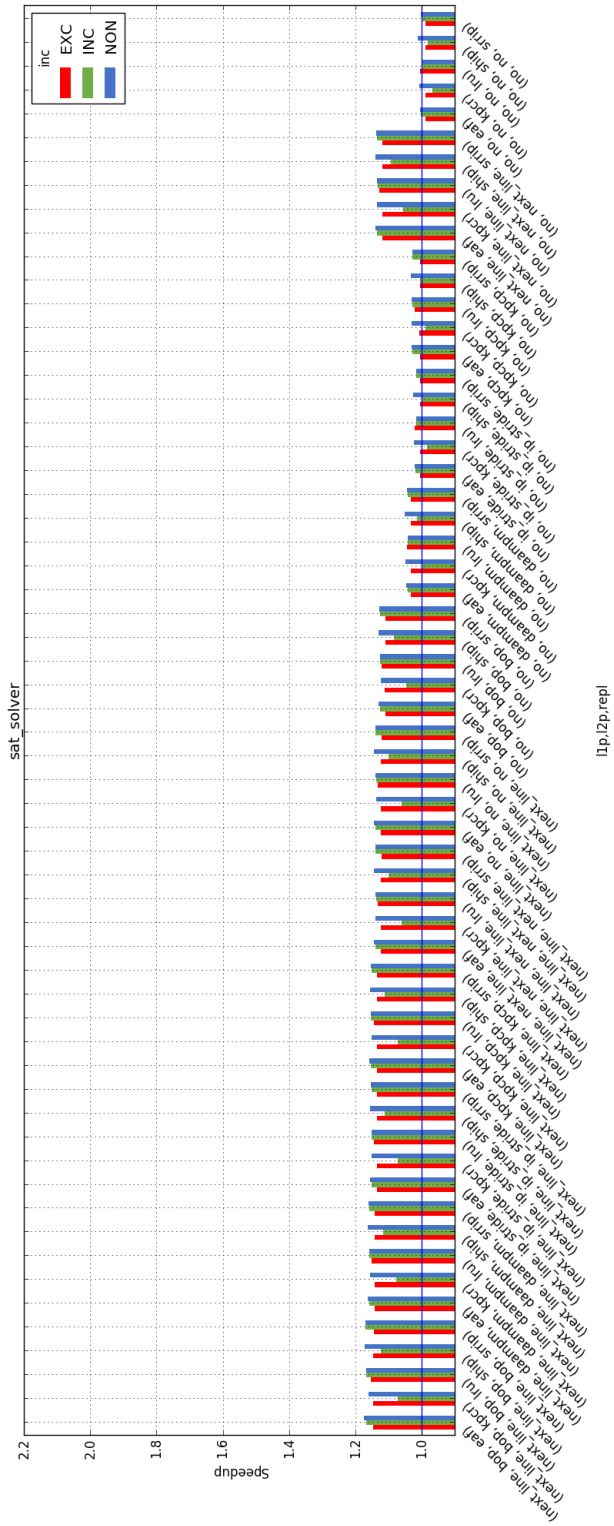
C.15 Mlpack_Cf



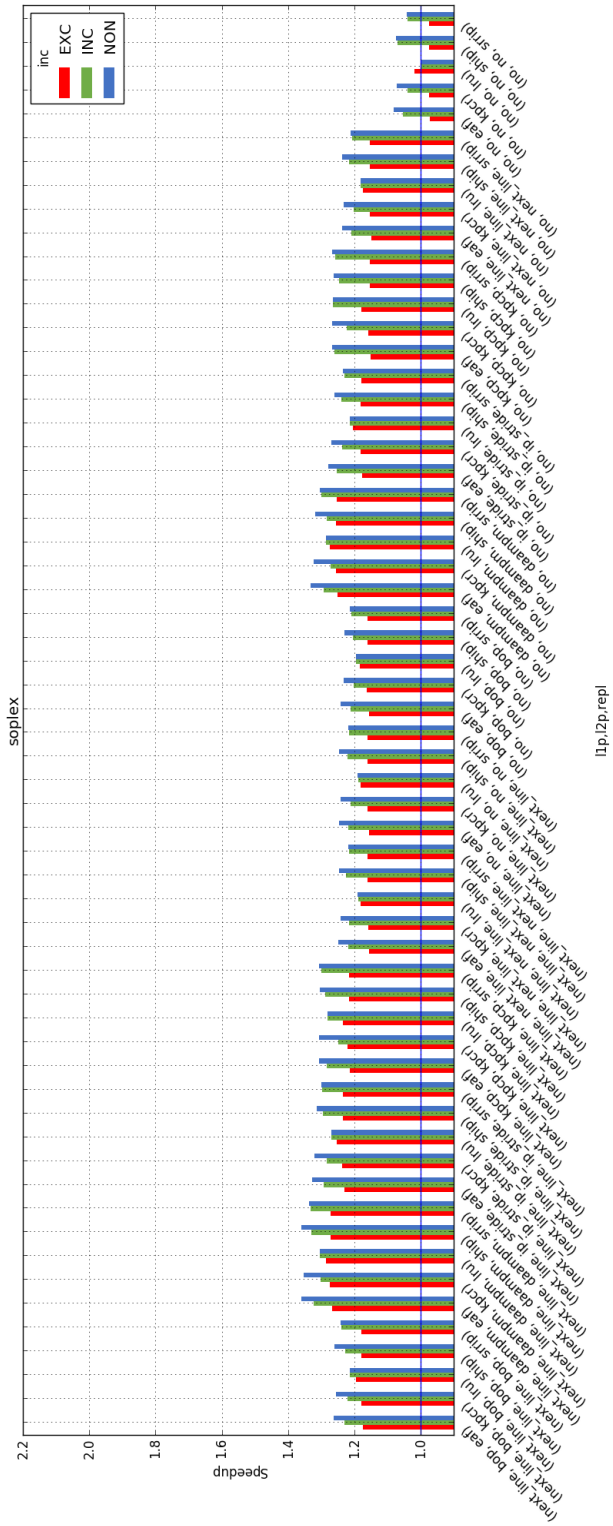
C.16 Omnetpp


omnetpp
Speedup
2.2
2.0
1.8
1.6
1.4
1.2
1.0
inc
EXC
INC
NON
l1p,l2p,repl

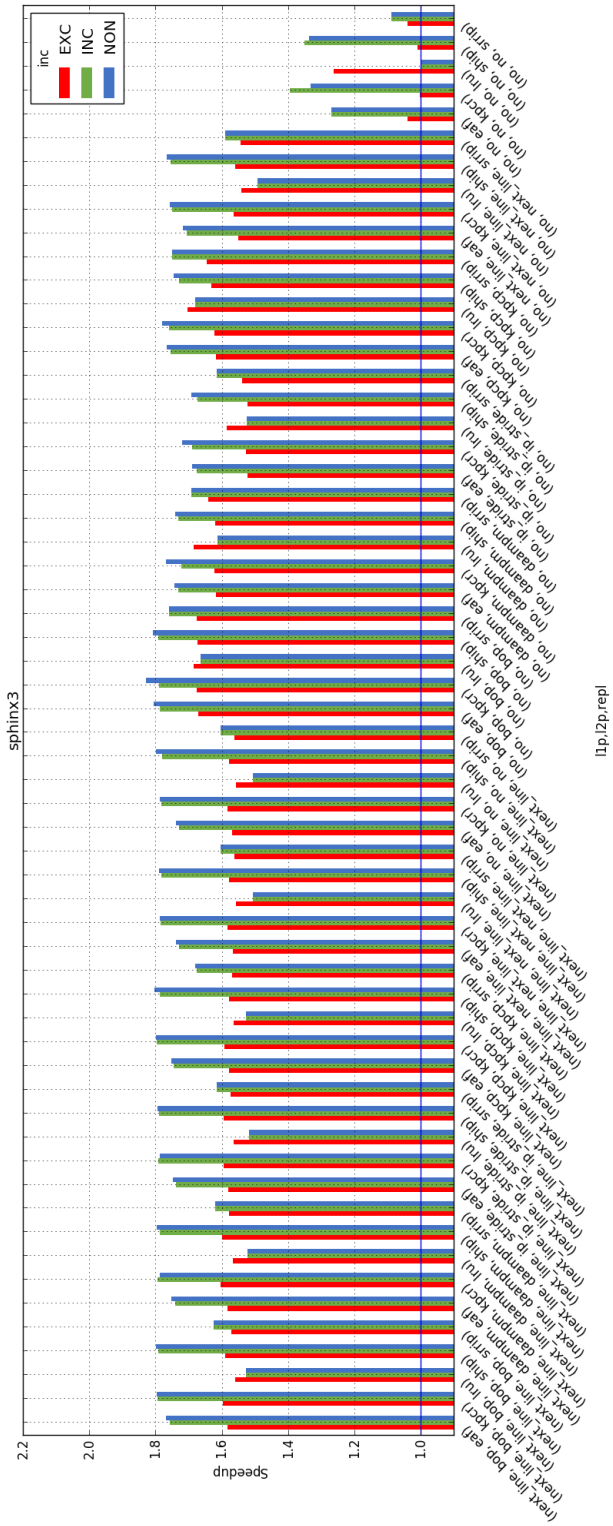
C.17 Sat_Solver



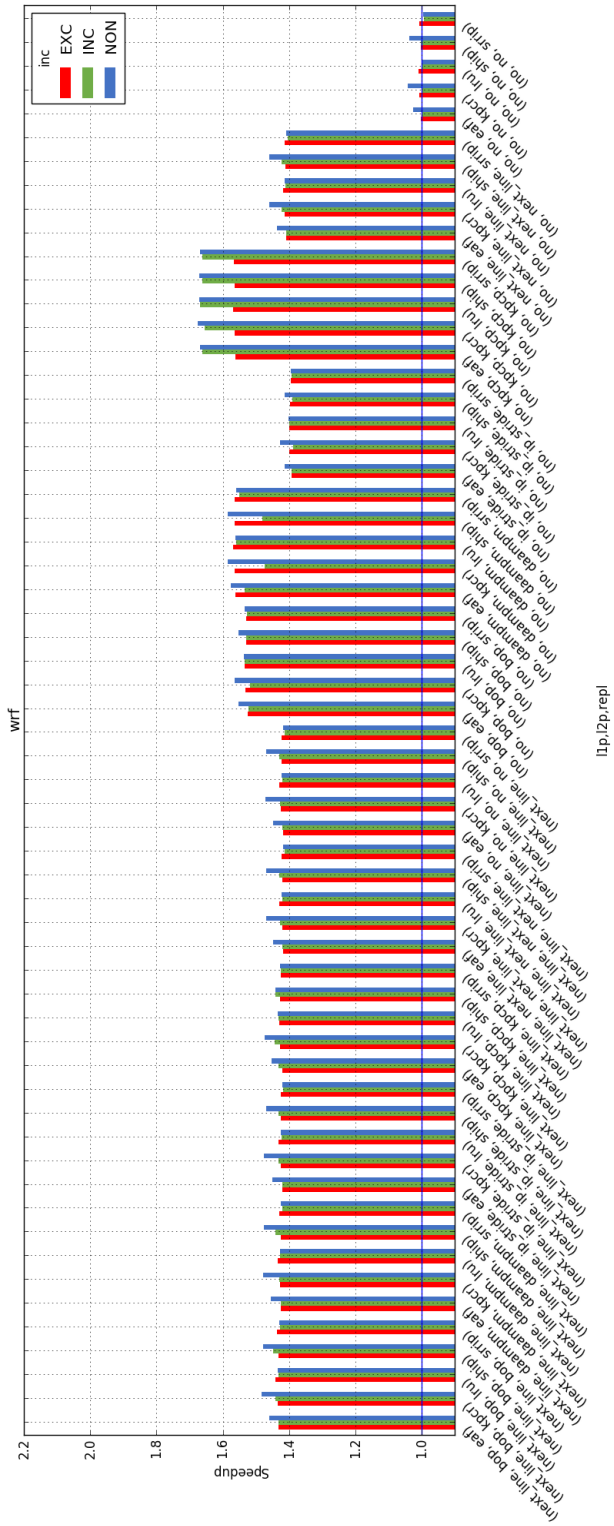
C.18 Soplex


soplex
Speedup
2.2
2.0
1.8
1.6
1.4
1.2
1.0
inc
EXC
INC
NON
(no_no_no)
(no_no_lru)
(no_no_kpcr)
(no_no_eelf)
(no_no_srrip)
(no_next_line_ship)
(no_next_line_lru)
(no_next_line_kpcr)
(no_next_line_eelf)
(no_next_line_srrip)
(no_kpcp_ship)
(no_kpcp_lru)
(no_kpcp_eelf)
(no_kpcp_kpcr)
(no_kpcp_srrip)
(no_ip_stride_ship)
(no_ip_stride_lru)
(no_ip_stride_kpcr)
(no_ip_stride_eelf)
(no_ip_stride_srrip)
(no_daampm_ship)
(no_daampm_lru)
(no_daampm_kpcr)
(no_daampm_eelf)
(no_daampm_srrip)
(no_bop_ship)
(no_bop_lru)
(no_bop_kpcr)
(no_bop_eelf)
(no_bop_srrip)
(next_line_no_ship)
(next_line_no_lru)
(next_line_no_kpcr)
(next_line_no_eelf)
(next_line_no_srrip)
(next_line_next_line_ship)
(next_line_next_line_lru)
(next_line_next_line_kpcr)
(next_line_next_line_eelf)
(next_line_next_line_srrip)
(next_line_kpcp_ship)
(next_line_kpcp_lru)
(next_line_kpcp_kpcr)
(next_line_kpcp_eelf)
(next_line_kpcp_srrip)
(next_line_ip_stride_ship)
(next_line_ip_stride_lru)
(next_line_ip_stride_kpcr)
(next_line_ip_stride_eelf)
(next_line_ip_stride_srrip)
(next_line_daampm_ship)
(next_line_daampm_lru)
(next_line_daampm_kpcr)
(next_line_daampm_eelf)
(next_line_daampm_srrip)
(next_line_bop_ship)
(next_line_bop_lru)
(next_line_bop_kpcr)
(next_line_bop_eelf)
(next_line_bop_srrip)
l1p,l2p,repl

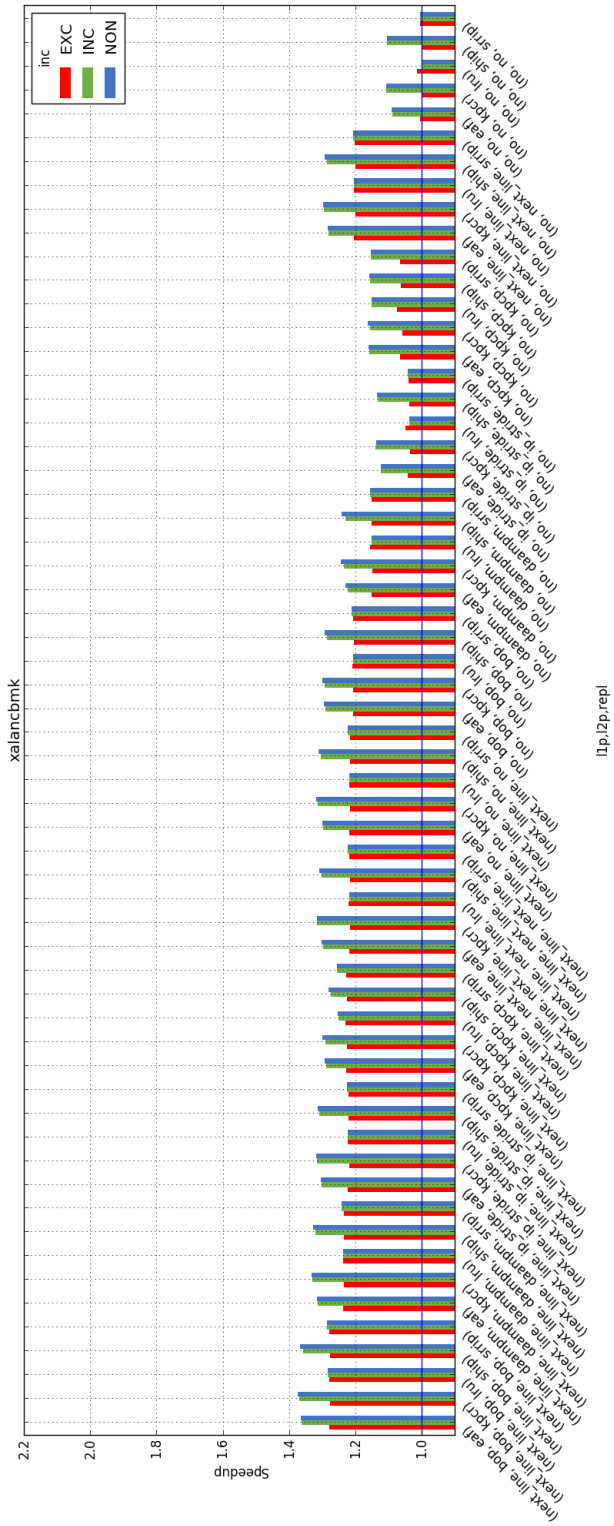
C.19 Sphinx3


inc
EXC
INC
NON
sphinx3
Speedup
2.2
2.0
1.8
1.6
1.4
1.2
1.0
l1p,l2p,repl
(next_line, bop, kcr)
(next_line, bop, lru)
(next_line, bop, ship)
(next_line, bop, srrip)
(next_line, no, eaf)
(next_line, no, kcr)
(next_line, no, lru)
(next_line, no, ship)
(next_line, no, srrip)
(next_line, next_line, eaf)
(next_line, next_line, kcr)
(next_line, next_line, lru)
(next_line, next_line, ship)
(next_line, next_line, srrip)
(next_line, daampm, eaf)
(next_line, daampm, kcr)
(next_line, daampm, lru)
(next_line, daampm, ship)
(next_line, daampm, srrip)
(next_line, ip_stride, eaf)
(next_line, ip_stride, kcr)
(next_line, ip_stride, lru)
(next_line, ip_stride, ship)
(next_line, ip_stride, srrip)
(next_line, kpcp, eaf)
(next_line, kpcp, kcr)
(next_line, kpcp, lru)
(next_line, kpcp, ship)
(next_line, kpcp, srrip)
(no, bop, eaf)
(no, bop, kcr)
(no, bop, lru)
(no, bop, ship)
(no, bop, srrip)
(no, daampm, eaf)
(no, daampm, kcr)
(no, daampm, lru)
(no, daampm, ship)
(no, daampm, srrip)
(no, ip_stride, eaf)
(no, ip_stride, kcr)
(no, ip_stride, lru)
(no, ip_stride, ship)
(no, ip_stride, srrip)
(no, kpcp, eaf)
(no, kpcp, kcr)
(no, kpcp, lru)
(no, kpcp, ship)
(no, kpcp, srrip)
(no, next_line, eaf)
(no, next_line, kcr)
(no, next_line, lru)
(no, next_line, ship)
(no, next_line, srrip)
(no, no, eaf)
(no, no, kcr)
(no, no, lru)
(no, no, ship)
(no, no, srrip)

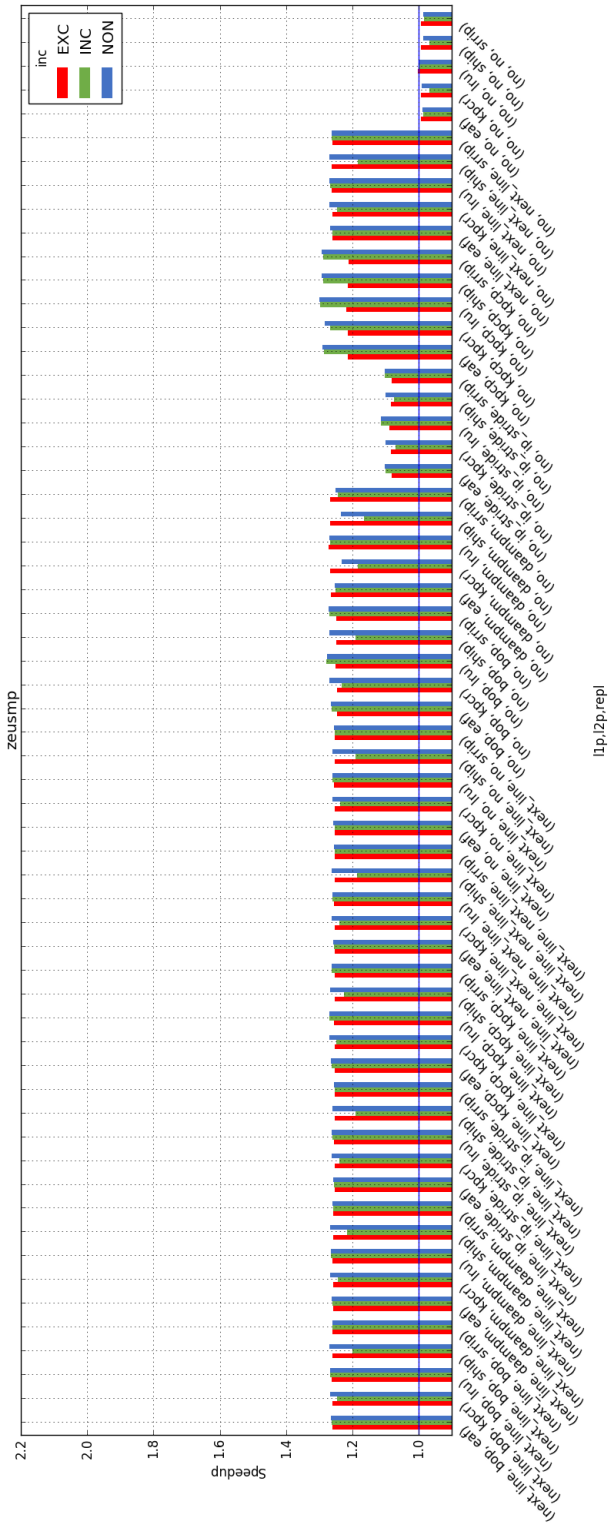
C.20 Wrf


inc
EXC
INC
NON
wrf
Speedup
2.2
2.0
1.8
1.6
1.4
1.2
1.0
(next_line bop eaf)
(next_line bop kpcr)
(next_line bop lru)
(next_line bop shlp)
(next_line bop strp)
(next_line daampm eaf)
(next_line daampm kpcr)
(next_line daampm lru)
(next_line daampm shlp)
(next_line daampm strp)
(next_line ip_stride eaf)
(next_line ip_stride kpcr)
(next_line ip_stride lru)
(next_line ip_stride shlp)
(next_line ip_stride strp)
(next_line kpcp eaf)
(next_line kpcp kpcr)
(next_line kpcp lru)
(next_line kpcp shlp)
(next_line kpcp strp)
(next_line next_line eaf)
(next_line next_line kpcr)
(next_line next_line lru)
(next_line next_line shlp)
(next_line next_line strp)
(next_line no eaf)
(next_line no kpcr)
(next_line no lru)
(next_line no shlp)
(next_line no strp)
(no bop eaf)
(no bop kpcr)
(no bop lru)
(no bop shlp)
(no bop strp)
(no daampm eaf)
(no daampm kpcr)
(no daampm lru)
(no daampm shlp)
(no daampm strp)
(no ip_stride eaf)
(no ip_stride kpcr)
(no ip_stride lru)
(no ip_stride shlp)
(no ip_stride strp)
(no kpcp eaf)
(no kpcp kpcr)
(no kpcp lru)
(no kpcp shlp)
(no kpcp strp)
(no next_line eaf)
(no next_line kpcr)
(no next_line lru)
(no next_line shlp)
(no next_line strp)
(no no eaf)
(no no kpcr)
(no no lru)
(no no shlp)
(no no strp)
l1p,l2p,repl

C.21 Xalancbmk



C.22 Zeusmp


zeusmp
Speedup
2.2
2.0
1.8
1.6
1.4
1.2
1.0
inc
EXC
INC
NON
l1p,l2p,repl
(next_line, bop, eaf)
(next_line, bop, kpcr)
(next_line, bop, lru)
(next_line, bop, ship)
(next_line, bop, srrip)
(next_line, daampm, eaf)
(next_line, daampm, kpcr)
(next_line, daampm, lru)
(next_line, daampm, ship)
(next_line, daampm, srrip)
(next_line, ip_stride, eaf)
(next_line, ip_stride, kpcr)
(next_line, ip_stride, lru)
(next_line, ip_stride, ship)
(next_line, ip_stride, srrip)
(next_line, kpcp, eaf)
(next_line, kpcp, kpcr)
(next_line, kpcp, lru)
(next_line, kpcp, ship)
(next_line, kpcp, srrip)
(next_line, next_line, eaf)
(next_line, next_line, kpcr)
(next_line, next_line, lru)
(next_line, next_line, ship)
(next_line, next_line, srrip)
(next_line, no, eaf)
(next_line, no, kpcr)
(next_line, no, lru)
(next_line, no, ship)
(next_line, no, srrip)
(no, bop, eaf)
(no, bop, kpcr)
(no, bop, lru)
(no, bop, ship)
(no, bop, srrip)
(no, daampm, eaf)
(no, daampm, kpcr)
(no, daampm, lru)
(no, daampm, ship)
(no, daampm, srrip)
(no, ip_stride, eaf)
(no, ip_stride, kpcr)
(no, ip_stride, lru)
(no, ip_stride, ship)
(no, ip_stride, srrip)
(no, kpcp, eaf)
(no, kpcp, kpcr)
(no, kpcp, lru)
(no, kpcp, ship)
(no, kpcp, srrip)
(no, next_line, eaf)
(no, next_line, kpcr)
(no, next_line, lru)
(no, next_line, ship)
(no, next_line, srrip)
(no, no, eaf)
(no, no, kpcr)
(no, no, lru)
(no, no, ship)
(no, no, srrip)