UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

ARTHUR MITTMANN KRAUSE

Efficient Caching with History-Based Preemptive Bypassing

Thesis presented in partial fulfillment of the requirements for the degree of Master of Computer Science

Advisor: Prof. Dr. Philippe O. A. Navaux
Coadvisor: Dr. Paulo Cesar Santos da Silva Junior Santos

Porto Alegre
June 2022

CIP — CATALOGING-IN-PUBLICATION

Krause, Arthur Mittmann

Efficient Caching with History-Based Preemptive Bypassing / Arthur Mittmann Krause. – Porto Alegre: PPGC da UFRGS, 2022.

80 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2022. Advisor: Philippe O. A. Navaux; Coadvisor: Paulo Cesar Santos da Silva Junior Santos.

1. Cache. 2. Memory. 3. Energy. 4. Computer Architecture. I. Navaux, Philippe O. A.. II. Santos, Paulo Cesar Santos da Silva Junior. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos André Bulhões
Vice-Reitora: Prof[a]. Patricia Pranke
Pró-Reitor de Pós-Graduação: Prof. Celso Giannetti Loureiro Chaves
Diretora do Instituto de Informática: Prof[a]. Carla Maria Dal Sasso Freitas
Coordenador do PPGC: Prof. Claudio Rosito Jung
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

“Hard work is worthless
for those who don’t believe in themselves...”

— Maito Gai

ACKNOWLEDGEMENTS

This work would not have been possible without the support of all my family and friends, especially my parents, Neuza and Erwin. Keeping motivation during the uncertainty about the future and all the other distressing feelings and thoughts provoked by the pandemic was tremendously difficult, but in the end, it was made possible by each one of the people that cared about me.

I am very thankful to have had Professor Philippe Navaux and Paulo Cesar Santos as my advisors, who frequently believed in me more than myself, and knew how to motivate me and direct my focus to the work that really mattered the most.

Many thanks to the Federal University of Rio Grande do Sul for its excellence that I enjoyed for over a third of my life; the Informatics Institute, for its great infrastructure and excellent professors; the Parallel and Distributed Processing Group, for making me into a researcher and providing me with multiple opportunities, and CAPES [1].

[1]This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001.

ABSTRACT

Cache memories can account for more than half of the area and energy consumption on modern processors, which will only increase with the current trend of bigger on-die memories. Although these components are very effective when the access pattern is cache-friendly, cache memories incur extra and unnecessary latencies when they cannot serve the data, which adds to significant energy wastes when data that is never reused is placed on them.

This work introduces HBPB, a mechanism that detects whether a memory access is cache-friendly or not, allowing the bypass of the cache for accesses that are not known to be cache-friendly. Our approach allows the processor to quickly detect when caching accesses is inadequate, improving overall access latency and reducing energy waste and cache pollution.

The presented solution achieves reductions of up to 75% in energy consumption and 35% in latency for a controlled microbenchmark and improvements in power and performance across various workloads.

Keywords: Cache. Memory. Energy. Computer Architecture.

Caches Eficientes com Bypass Preemptivo Baseado em Histórico

RESUMO

As memórias cache podem responder por mais da metade da área e do consumo de energia em processadores modernos, o que só aumentará com a tendência atual de memórias maiores dentro dos chips. Embora esses componentes sejam muito eficazes quando o padrão de acesso é amigável à cache, as memórias cache ocasionam latências extras e desnecessárias quando não podem fornecer os dados, o que aumenta o desperdício de energia significativamente quando dados que nunca são reutilizados são colocados nelas.
Este trabalho apresenta o HBPB, um mecanismo que detecta se um acesso à memória é amigável à cache ou não, permitindo ser feito bypass da cache para acessos que não são reconhecidamente como amigáveis às caches. Nossa abordagem permite que o processador detecte rapidamente quando utilizar a cache não é adequado para um acesso, melhorando de forma geral a latência de acesso à memória e reduzindo o desperdício de energia e a poluição do cache.
A solução apresentada alcança reduções de até 75% no consumo de energia e 35% na latência para um microbenchmark customizado e melhorias de energia e desempenho em uma plenitude de aplicações diferentes.

Palavras-chave: Cache, Memórias, Energia, Arquitetura de Computadores.

LIST OF ABBREVIATIONS AND ACRONYMS

DRAM	Dynamic Random Access Memory
LLC	Last-Level Cache
HBPB	History-Based Preemptive Bypassing
SIMD	Single Instruction Multiple Data
WB	Write Back
CIT	Classified Instructions Table
PC	Program Counter
AHT	Access History Table
NTB	Non-Temporal Buffer
LSQ	Load-Store Queue
MSHR	Miss Status Holding Registers
RFO	Read For Ownership
ED^2P	Energy-Delay-Squared Product
YOLO	You Only Look Once
CNN	Convolutional Neural Network
LRU	Least Recently Used
SHiP	Signature-based Hit Predictor
SRRIP	Static Re-Reference Interval Prediction
DRRIP	Dynamic Re-Reference Interval Prediction

LIST OF FIGURES

Figure 2.1 Sequence diagram of a cacheable memory access through a traditional memory hierarchy. 25
Figure 2.2 Sequence diagram of a non-temporal memory access through a traditional memory hierarchy. 25
Figure 2.3 Latency in a single-access operation through a traditional memory hierarchy. 26
Figure 2.4 Energy costs of a single-access operation through a traditional memory hierarchy. 26

Figure 4.1 HBPB architectural additions. 32
Figure 4.2 HBPB decision flowchart. 32
Figure 4.3 The path of the request through the memory subsystem. 35
Figure 4.4 The path of the data through the memory subsystem. 36

Figure 5.1 Effect of HBPB on the cycles for the kernel microbenchmarks. 41
Figure 5.2 Effect of HBPB on the cache dynamic energy for the kernel microbenchmarks. 41
Figure 5.3 Effect of HBPB on the ED^2P for the kernel microbenchmarks. 42
Figure 5.4 Frequency of each type of access for the microbenchmarks with HBPB. 46

Figure 6.1 Effect of HBPB on the cycles for the SPEC CPU 2017 benchmarks. 48
Figure 6.2 Effect of HBPB on the cache dynamic energy for the SPEC CPU 2017 benchmarks. 49
Figure 6.3 Effect of HBPB on the ED^2P for the SPEC CPU 2017 benchmarks. 50
Figure 6.4 Frequency of each type of access for the SPEC 2017 benchmarks with HBPB-RF. 51
Figure 6.5 Frequency of each type of access for the SPEC 2017 benchmarks with HBPB-R. 52
Figure 6.6 Frequency of reuse distances for the 135 most frequently occurring instructions for *cactuBSSN*. 54
Figure 6.7 Frequency of reuse distances for the 135 most frequently occurring instructions for *fotonik3d*. 54
Figure 6.8 Frequency of reuse distances for the 135 most frequently occurring instructions for *gcc*. 55
Figure 6.9 Frequency of reuse distances for the 135 most frequently occurring instructions for *lbm*. 55
Figure 6.10 Frequency of reuse distances for the 135 most frequently occurring instructions for *mcf*. 56
Figure 6.11 Frequency of reuse distances for the 135 most frequently occurring instructions for *omnetpp*. 57
Figure 6.12 Frequency of reuse distances for the 135 most frequently occurring instructions for *roms*. 57
Figure 6.13 Frequency of reuse distances for the 135 most frequently occurring instructions for *wrf*. 58
Figure 6.14 Frequency of reuse distances for the 135 most frequently occurring instructions for *xalancbmk*. 58
Figure 6.15 Effect of HBPB on the cycles for the SPEC CPU 2017 benchmarks with different cache capacities. 59

Figure 6.16 Frequency of each type of access for the SPEC 2017 benchmarks with HBPB-RF with an 8MB LLC. 60
Figure 6.17 Effect of HBPB on the cycles for the SPEC CPU 2017 benchmarks with prefetchers. 61
Figure 6.18 Effect of HBPB on the cache energy for the SPEC CPU 2017 benchmarks with prefetchers. 61
Figure 6.19 Effect of HBPB on the ED^2P for the SPEC CPU 2017 benchmarks with prefetchers. 62
Figure 6.20 Effect of HBPB on the ED^2P for the SPEC CPU 2017 benchmarks with different LLC replacement policies 63

Figure A.1 Fluxograma de decisão do HBPB. 72
Figure A.2 Efeito do HBPB no ED^2P das aplicações do SPEC CPU 2017 benchmarks. 73

Figure B.1 Frequency of reuse distances for the 50 most frequently occurring instructions for Pattern Matching. 75
Figure B.2 Frequency of reuse distances for the 50 most frequently occurring instructions for Deriche. 75
Figure B.3 Frequency of reuse distances for the 50 most frequently occurring instructions for Floyd-Warshall. 76
Figure B.4 Frequency of reuse distances for the 50 most frequently occurring instructions for Jacobi 1D 76
Figure B.5 Frequency of reuse distances for the 50 most frequently occurring instructions for Jacobi 2D 76
Figure B.6 Frequency of reuse distances for the 50 most frequently occurring instructions for *bc*. 77
Figure B.7 Frequency of reuse distances for the 50 most frequently occurring instructions for *bfs*. 77
Figure B.8 Frequency of reuse distances for the 50 most frequently occurring instructions for *cc*. 78
Figure B.9 Frequency of reuse distances for the 50 most frequently occurring instructions for *pr*. 78
Figure B.10 Frequency of reuse distances for the 50 most frequently occurring instructions for *sssp*. 78
Figure B.11 Effect of HBPB on the cycles for the GAP benchmarks. 79
Figure B.12 Effect of HBPB on the cache energy for the GAP benchmarks. 79
Figure B.13 Effect of HBPB on the ED^2P for the GAP benchmarks. 79
Figure B.14 Frequency of each type of access for the GAP benchmarks with HBPB. 80

LIST OF TABLES

Table 2.1 Memory hierarchy characteristics for a Intel Core i7 4770 22nm processor. .24

Table 5.1 Modeled system....39
Table 5.2 Energy values used on the evaluation....40
Table 5.3 Working set size from the microbenchmarks used....42

Table 6.1 Working set size from the SPEC CPU 2017 applications used....47
Table 6.2 Summary of the percentual reductions in Cycles and Cache Energy obtained with HBPB on the SPEC CPU 2017 applications....50
Table 6.3 Best configuration for each metric for each of the SPEC applications considering prefetchers. 62

Table B.1 Working set size from the GAP benchmarks used. 77

CONTENTS

1 INTRODUCTION......19
1.1 Contributions......20
1.2 Organization......21
2 CACHE MEMORY BYPASSING MOTIVATION......23
3 RELATED WORK......27
4 HISTORY BASED PREEMPTIVE BYPASSING......31
4.1 HBPB Architecture......31
4.2 HBPB Operation......32
4.2.1 Decision......32
4.2.2 Training......33
4.3 Data and Request Paths With HBPB......33
4.4 HBPB Hardware Overhead......37
5 EVALUATION......39
5.1 Implementation......39
5.2 Proof of Concept......40
5.2.1 Linked List......41
5.2.2 Pattern Matching......43
5.2.3 Jacobi 1D......43
5.2.4 Jacobi 2D......44
5.2.5 Floyd-Warshall......44
5.2.6 Deriche......44
5.2.7 Yolo CNN......45
5.2.8 Final Remarks......45
6 RESULTS AND ANALYSIS......47
6.1 Experimental Results......47
6.1.1 Performance......47
6.1.2 Cache Energy......48
6.1.3 Energy-Delay-Squared Product......49
6.1.4 Summary......50
6.2 Static Reuse Distance Analysis......52
6.2.1 *cactuBSSN*......53
6.2.2 *fotonik3d*......54
6.2.3 *gcc*......54
6.2.4 *lbm*......55
6.2.5 *mcf*......56
6.2.6 *omnetpp*......56
6.2.7 *roms*......57
6.2.8 *wrf*......57
6.2.9 *xalancbmk*......58
6.3 Sensibility to Memory Configuration......59
6.3.1 Bigger Cache Capacity......59
6.3.2 Prefetcher......60
6.3.3 Cache Replacement Policies......63
7 CONCLUSION......65
7.1 Future Work......65
REFERENCES......67
APPENDIX A — RESUMO EXPANDIDO......71
A.1 Introdução......71

A.2 HBPB **71**
A.3 Resultados **73**
A.4 Conclusão **73**
APPENDIX B — EXTRA RESULTS **75**
B.1 Microbenchmarks **75**
B.2 GAP **77**

1 INTRODUCTION

As processors keep getting faster, cache memories play a vital role in reducing average memory access latencies, becoming even more critical with each new generation of processors. The current trend in processor design is to increase the number of cores and the size of the caches. However, by increasing the number of processing cores, the competition for space and bandwidth in shared cache memories also increases, which impacts the performance and efficiency of these memories due to over-utilization, cache pollution, and trashing. With bigger caches, energy consumption and access latencies also tend to increase. Thus, the need for more intelligent management of these resources is of great and growing importance.

Not every memory access can be helped by caching. In fact, for many accesses, caching the data provokes only extra latency and energy consumption while also replacing other potentially useful data, which will then have to be fetched again from a slower level of the memory hierarchy. This results in once more increased latency and energy consumption indirectly. In such a way, when cache-unfriendly memory access are performed and passed through the cache hierarchy, not only they suffer from increased latency and unnecessary energy waste, they also pollute the caches, hampering their proper operation for cache-friendly accesses [Chi and Dietz 1989].

Current processors have limited implementations of bypassing instructions, which are not well documented for *load* operations and can have unpredictable behavior that depends on specifics of the implementation [Guide 2020]. Also, instructions of this kind cannot be used with fine granularity and require the programmer to employ them properly. Compilers do not automatically employ these instructions since they would have to know at compile time that an instruction always accesses non-temporal data. Lately, progress has been made with cache replacement policies that can tolerate streaming access patterns and mitigate the cache pollution generated by them, which are effectively implemented in current processors available in the market [Kumar and Singh 2016]. However, such methods still perform caching of cache-unfriendly data, not avoiding the massive energy waste that they represent. For current processors, the only way to effectively bypass the cache is through non-temporal instructions or by defining a whole region of memory as non-cacheable with the granularity of a page.

Many researchers have addressed this problem through varied approaches. Sandberg *et al.* developed a framework [Sandberg, Eklöv and Hagersten 2010] to define mem-

ory accessing instructions as bypassing at compile time. Multiple authors proposed augmenting the caches with logic that allows the bypassing decisions to be made in hardware at runtime [Albericio et al. 2013, Bae and Choi 2020, Wang et al. 2019], most only for the Last-Level Cache (LLC). Egawa et al. [Egawa et al. 2019] present a mechanism that can turn off specific levels of the cache hierarchy. Other works such as [Chaudhuri et al. 2012] propose a mechanism that allows victim fills to be bypassed on victim caches. Meanwhile, Kohler and Alves [Köhler and Alves 2019] propose to bypass the memory request when a miss is likely in order to anticipate the request to the Dynamic Random Access Memory (DRAM) and reduce the latency of the cache miss.

However, none of the existing works present a proposition to bypass both the memory requests as well as the data fills, on all levels of the cache hierarchy, while allowing spatial locality and short-term temporal reuse to happen still. Bypassing only for the LLC can be insufficient as unnecessary fills to the higher-level caches can cause evictions that provoke victim fills or write-backs to the lower levels of the cache. In this context, we propose a History-Based Preemptive Bypassing (HBPB) mechanism, with the goal of bypassing the cache memory for every access that is not known to be cache-friendly and learning which accesses could benefit from caching.

1.1 Contributions

In this thesis, we propose HBPB. Our mechanism accelerates not only cache-unfriendly accesses, through the reduction in latency achieved by bypassing the caches, but also the cache-friendly ones by reducing the cache pollution and increasing their hit ratio, avoiding energy waste in both cases. The mechanism preemptively bypasses every memory access from an instruction that is not known to be cache-friendly. Thanks to observations that the reuse behavior stays consistent for most instructions, we can confidently choose to decide on bypassing or not by the instruction that is provoking a memory access. By tracking past memory accesses and the instructions that referenced each address, HBPB can profile the program instructions and leave the cache only for those instructions that have shown reuse on their data.

The HBPB mechanism yields reductions in cache energy consumption of up to 75% while also achieving speedups of more than 1.5x for a controlled microbenchmark and 1.25x for a full application benchmark. In this work, we analyze HBPB in multiple scenarios, through different applications and memory configurations. We also test HBPB

with cache replacement policies different from the standard Least Recently Used (LRU) and hardware data prefetchers. On every occasion, HBPB proved to be advantageous when employed.

We also present a static analysis of the reuse distances for the data accessed by each memory-accessing instruction from the applications tested and understand the reasons behind the effects shown by HBPB for these benchmarks.

1.2 Organization

The objective of this work is to present HBPB and the context where it is inserted. This thesis is organized as follows: Chapter 2 presents the motivation for bypassing the caches. In Chapter 3, the state-of-the-art in cache bypassing is briefly described. Chapter 4 presents HBPB, our proposed implementation of cache bypassing in hardware. Chapter 5 presents our evaluation and proves that HBPB works as intended. In Chapter 6, we show that the benefits from HBPB also extend to real applications and are still valid among different hardware configurations. We also provide a static analysis of the reuse distances for each instruction of the applications tested and show on which of them HBPB acts on. Chapter 7 brings a conclusion to this work.

2 CACHE MEMORY BYPASSING MOTIVATION

Cache memories are a powerful mean to mitigate the memory-wall problem [Wulf and McKee 1995], highlighted by the performance gap between logic and main memory technologies. Usually organized in a multi-level hierarchy, with smaller and faster levels closer to the processing cores, they reduce the average memory access latency by storing data that is likely to be requested by the processor in the near future. The core functionality of the cache is exploring the spatial and temporal locality of memory accesses. The use of cache lines covers spatial locality: every time an access is made, 64 bytes of data - a cache line - are fetched. This way, if the next accesses are for neighboring data, they have already been fetched, and the latency is reduced. Temporal locality means that data that has been recently requested tends to be requested again in the near future. The caches leverage the temporal locality of memory accesses by storing data that has been requested in the near past, evicting data that has not been used in a long time when new lines need to be stored.

However, caches are not effective when the access pattern has no temporal locality in regards to cache lines. For example, when the application searches through a linked list, and nodes are not repeated, no data can be serviced by the cache. In this case, regular cache management dictates that the processor must first check the caches for the presence of the needed data and then store the data coming from main memory in the caches, in case it is needed in the future, which never happens. The actions of waiting for the cache answer before requesting the data to main memory and filling this data in the cache represent both an unnecessary overhead. If the request could bypass the cache, many cycles of waiting would be avoided, and if the data could bypass the cache, a significant amount of energy would be saved. Also, by not storing unnecessary data into the cache, cache pollution - replacing lines with reuse potential with lines with no reuse potential - could be mitigated, which in turn would reduce cache misses, saving on latency and energy again.

Table 2.1 shows the energy and latency costs for accessing different levels of the memory hierarchy on an Intel Core i7 4770 processor. Latency values have been taken from the Intel optimization manual [Intel 2018], energy values are given by Cacti [Balasubramonian et al. 2017], given a 22nm technology, and main memory energy values come from our own experimentations with DDR3 DRAM, and account only for dynamic energy. We assume the energy for a DRAM Read and Write are equal for simplicity.

Table 2.1 – Memory hierarchy characteristics for a Intel Core i7 4770 22nm processor.

Level	Size	Latency (Cycles)	Read Energy (pJ)	Write Energy (pJ)
L1	32 KB	4	199	202
L2	256 KB	11	169	175
L3	8 MB	∼34	1122	1163
DRAM	4 GB	∼100-200	6140	6140

Current processors offer the functionality of *load*, *store* and *prefetch* instructions with a *non-temporal hint*. *Non-temporal stores* are well established due to the need for offloading data to other devices with short latencies and are used mainly on drivers and kernel code. Also, the possibility of using write combining - buffering many stores to neighboring locations before issuing a single write operation to memory - makes non-temporal stores even more useful. However, for *load instructions*, functionality is much more restricted. On ARM, the `LDNP` instruction performs a load of a pair of values into two registers with a non-temporal hint. What the processor does with this hint is not well specified. On x86, non-temporal load instructions have been introduced with SSE 4.1. When these instructions are used interchangeably with regular load instructions, caching is usually done anyway, rendering their employment almost useless [Guide 2020]. For proper bypassing of the caches to occur, the memory region that is being accessed must be configured as *write combining*, reducing the freedom for the programmer or compiler to employ them.

In addition to not having a transparent and dependable implementation, these instructions only exist for Single Instruction Multiple Data (SIMD) operations and have to be used explicitly. For example, while a regular arithmetic instruction can perform a memory access, for this access to be made non-temporal, a non-temporal SIMD load has to be performed for the desired data, the value has to be copied from the SIMD registers to regular registers, and then the arithmetic operation can be done. Also, as the memory region has to be defined as *uncacheable* at the allocation, this limits the flexibility to decide between bypassing the cache or not depending on the state of the system or bypassing only some parts of the data on a data structure. Consequently, detecting which instructions could perform non-temporal accesses at run-time is of great value.

Figure 2.1 shows the sequence diagram of a memory access that misses in all levels of cache and is serviced by the main memory, considering all caches are inclusive. The request crosses all cache levels and prompts fills and evictions on every level. If the evicted line has been modified, a Write Back (WB) to the next level of the memory hierarchy must be done. The data is requested to the main memory only after it is confirmed

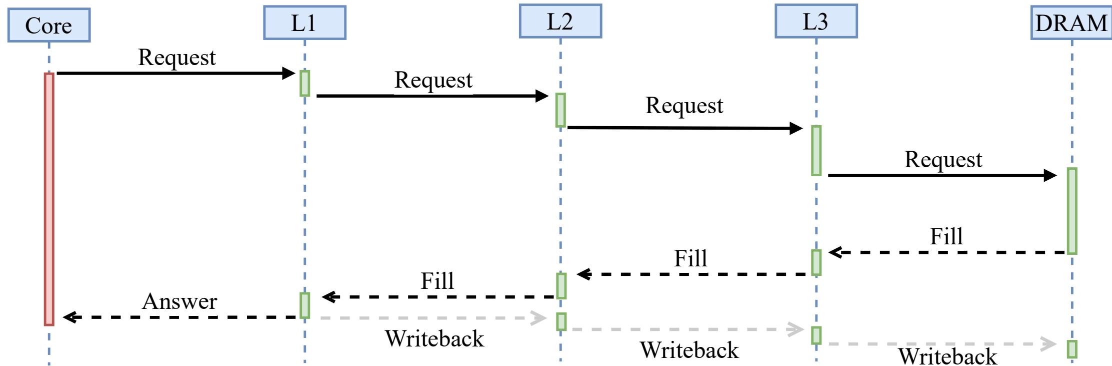

Figure 2.1 – Sequence diagram of a cacheable memory access through a traditional memory hierarchy.

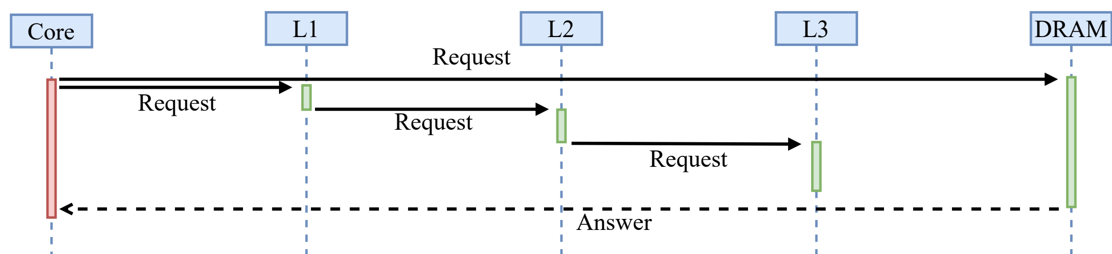

Figure 2.2 – Sequence diagram of a non-temporal memory access through a traditional memory hierarchy.

that it is not present in the caches. Figure 2.2 shows how a non-temporal memory access is performed. The request goes directly to the main memory, and the caches are checked in parallel. Since the request has arrived in the main memory much earlier, the answer arrives earlier. If the data happened to be in the caches, the data from the main memory is discarded. No fills are done, so no evictions or Write Backs are necessary.

Figure 2.3 shows the different latencies involved in a main memory access both in a regular access and in a bypassed access. Since the physical address generation is usually done in parallel with the L1 cache check, it can not be avoided by the non-temporal access. The bypassed access takes 45 fewer cycles than usual, being 43% faster, assuming the DRAM latency for a row buffer hit.

Figure 2.4 shows the energy costs of an access to main memory in the worst case, in which all cache evictions are from dirty lines and Write Backs are prompted on all levels. In the best case, no Write Backs are done, and so the only energy costs are from the DRAM read and cache fills. When the access is non-temporal, the only energy cost is from reading the DRAM. With that in mind, a non-temporal access can save up to 60% of the dynamic energy in comparison with a regular access. Also, considering the latency is reduced, it further saves on static energy. It is crucial as well to note that saving on power inside the processor package is more beneficial due to thermal dissipation factors.

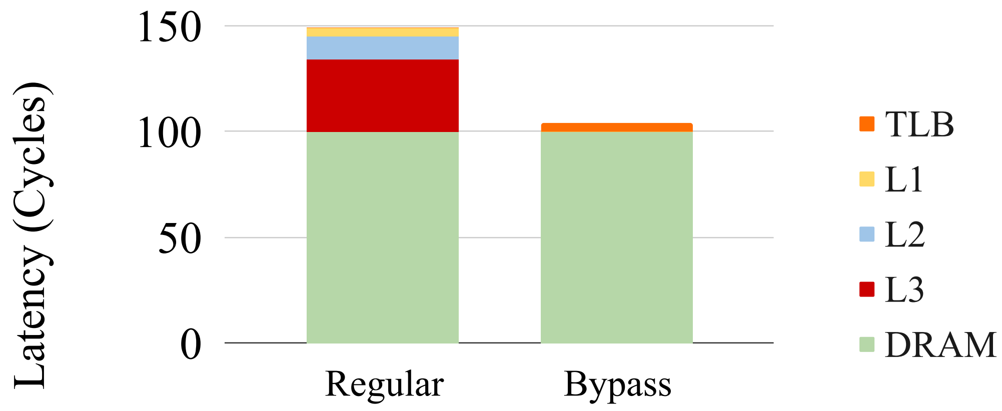

Figure 2.3 – Latency in a single-access operation through a traditional memory hierarchy.

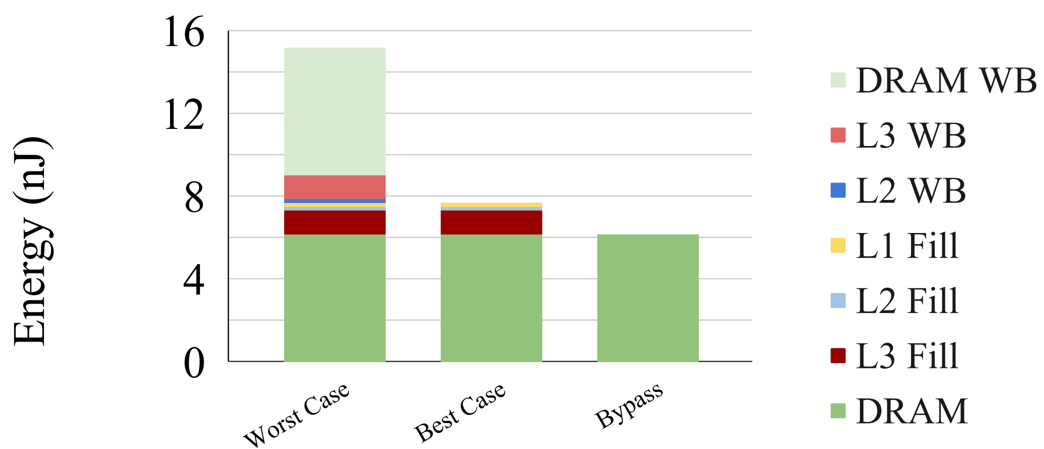

Figure 2.4 – Energy costs of a single-access operation through a traditional memory hierarchy.

3 RELATED WORK

Improving the memory system efficiency is of major importance and is the goal of a growing proportion of the research in computer architecture. From complete overhauls of their organization to minor policy tweaks, the caches are the usual target for enhancements in the memory hierarchy. The most common approach is to improve the cache replacement policy, which means changing the way that a cache block is selected for eviction when a new block must be stored [Wu et al. 2011, Jain and Lin 2016, Beckmann and Sanchez 2017, Kim et al. 2017, Jain and Lin 2018, Shi et al. 2019]. However, not as many works suggest completely skipping the caches when serving data to the processor.

Some authors have studied cache bypassing benefits and how to select accesses to be bypassed. Most approaches are done in hardware. However, Sandberg *et al.* [Sandberg, Eklöv and Hagersten 2010] suggest detecting which instructions access data with no temporal locality through profiling and analysis of the reuse distance of addresses. With this information, the authors propose using non-temporal *prefetch* instructions before the selected instructions to bypass the last level cache for the data. The lines are also evicted directly to memory.

Gupta *et al.* [Gupta, Gao and Zhou 2013] suggest the introduction of a bypass filter on the LLC that allows inclusive last-level caches to be bypassed. At the same time, data is still stored in higher levels of the cache hierarchy.

Chaudhuri *et al.* [Chaudhuri et al. 2012] present Cache Hierarchy Aware Replacement (CHAR) algorithms for inclusive LLCs that can be used on exclusive LLCs in order to bypass victim fills. They classify cache lines in the L2 cache in five classes according to four parameters: if they are prefetched or not, come from an LLC hit or miss, how many times they are reused in the L2 before being evicted, and their coherence state. The mechanism then monitors if the lines from each class are dead when filled in the LLC and uses this classification to decide whether future lines from the same class should be bypassed.

Sembrant *et al.* [Sembrant, Hagersten and Black-Schaffer 2016] propose a mechanism that tracks the reuse of cache lines brought by each instruction at each level of the cache hierarchy. The authors then propose the Reuse Aware Placement (RAP). This mechanism learns the reuse distance of data from each instruction and places this data on the most appropriate cache level on the hierarchy, bypassing data from instructions that do not fetch reused lines.

Kharbuti *et al.* [Kharbutli, Jarrah and Jararweh 2013] propose the Selective Cache Insertion and Bypassing (SCIP), a history-based mechanism that tracks the reuse of cache blocks in a counter and leverages this information to decide on the subsequent blocks' insertion location.

Sim *et al.* [Sim et al. 2012] propose FLEXclusion, a dynamic mechanism that changes the inclusion policy of a multi-layered cache between exclusive and non-inclusive, using the set dueling technique to decide between them. The authors claim that the mechanism can find the best policy at run time to improve performance and reduce energy waste. While changing the LLC to exclusive is equivalent to bypassing it for data arrivals, effectively removing them from the data path, they are still filled on evictions.

Egawa *et al.* [Egawa et al. 2019] propose a mechanism that monitors each cache level efficiency and bypasses the levels that are not energy efficient. The authors suggest using the Hits Per Kilo-Instruction of the cache level and the miss penalty of each level to achieve a metric called Hidden Access Latency Per Kilo-Instruction. This metric is used in conjunction with the power consumption of a given cache level to determine if bypassing this level would improve the system's energy efficiency. However, there is no way to keep the caches working for blocks that could eventually benefit from them. If the mechanism detects that a particular cache level is ineffective, no access can benefit from them.

Gaur *et al.* [Gaur, Chaudhuri and Subramoney 2011] observe that the LRU policy and its derivatives are ineffective when employed on an exclusive cache hierarchy and propose new insertion and bypassing policies better tailored to this set of caches, based on events that are more significant on exclusive caches, such as the number of times a cache line transits between the cache levels or the use count of a cache line during its residency on the L2 cache.

Wang *et al.* [Wang et al. 2019] introduce FILtered Multilevel caching policy (FILM). This mechanism places evicted data into the optimal cache level instead of filling them into the next level, as traditionally, bypassing the levels where it is predicted not to be reused. The mechanism uses a PC-based reuse predictor for each level. The authors also tailor the mechanism to train differently with prefetch requests, increasing its efficiency.

Bae and Choi propose the Filter Cache [Bae and Choi 2020], a small filter on the Last Level Cache that detects whether lines have spatial or temporal reuse. The filter then is used to fill on the LLC only lines with temporal reuses, because only cache lines

with temporal reuse potential need to be stored on the LLC since the private caches serve the spatial reuses. With this, the authors claim the LLC can be reduced by 75% without compromising the performance.

The Reuse Cache [Albericio et al. 2013], proposed by Albericio and Ibáñez, is a cache architecture where the tag array is decoupled from the data array on the Last Level Cache. The tag array has a greater capacity than the data array. When accesses from the private cache arrive and miss on the tag array, the data is fetched from the main memory and is filled only on the tag array and sent to the private caches. When the access hits only on the tag array, the data is then fetched again from the main memory and filled into the data array. This scheme allows the data array to be much smaller and only store cache lines that have shown reuse. The authors, however, do not suggest leveraging any information on the memory access to automatically cache a line without it needing to be accessed twice.

Teran *et al.* [Teran, Wang and Jiménez 2016] propose using a perceptron-based reuse predictor that feeds features of memory accesses such as different parts of the tag of the address and the PC of the instruction as well as recent occurring PCs into tables of weights that are changed when a block is reused or evicted. These weights are used to make predictions on the reuse of new accesses. The reuse prediction is then used to decide on the placement of the new block in the LLC. The decision can be to bypass the cache if the block is predicted not to have any reuse. On [Jiménez and Teran 2017], the authors introduce the Multiperspective Reuse Prediction, where they tweak the perceptron-based predictor and add a more extensive set of features.

Kohler and Alves [Köhler and Alves 2019] propose that requests of data to the last level cache and main memory could be made in parallel when the access is predicted to miss in the Last Level Cache (LLC) in order to reduce the request latency. The authors use the past behavior of the requests from the same instruction to predict its future results.

Most of the previous work is, to some degree, very conservative with bypassing. Most of the time, they suggest bypassing only the LLC or being exceedingly selective when deciding to bypass or not. In this context, our work aims to complement the state-of-the-art with a proposal of a mechanism that bypasses all the caches preemptively in both ways, requests and data, and show that it can be done in hardware at runtime, using a straightforward heuristic.

The main focus of this work is to analyze the impact of bypassing all the cache levels, preemptively, on both ways while offering a small buffer for spatial reuse to still

occur. Employing a separate prefetcher on this buffer to target the non-temporal accesses is also an important contribution of our work. We also propose a heuristic to select the lines to be bypassed.

4 HISTORY BASED PREEMPTIVE BYPASSING

In order to achieve an online conversion of regular memory accesses into non-temporal accesses, we propose the History-Based Preemptive Bypassing (HBPB) mechanism. The mechanism has two main goals:

- Increase performance by avoiding cache latencies for cache-unfriendly access;
- Reduce cache pollution and increase energy efficiency by not filling data from cache-unfriendly accesses into the caches.

The mechanism works by storing the history of recent memory accesses and classifying instructions into cache-friendly or not. HBPB assumes every instruction is not cache-friendly unless it has been detected as such. Thus, it aggressively tunes accesses from instructions not known to be cache-friendly into non-temporal (bypassed) memory accesses, preemptively bypassing all the cache levels.

4.1 HBPB Architecture

The mechanism requires three major additions to the architecture, as displayed in Figure 4.1. First, a table that stores identifiers of instructions is required. Each entry has the Program Counter (PC) as a key and a saturating counter as a value. This table is called the Classified Instructions Table (CIT). In order to reduce the storage needed, every PC can be hashed, and the mechanism then only sees this hash.

The mechanism also needs an Access History Table (AHT), which stores memory addresses and the identifier of the instruction that generated the last memory accesses to this address. This table needs to be as big as the number of lines in the caches and keep their ordering. It can be a replica of the LLC for better efficiency, storing the PC hash of the instruction that generated the access instead of the cache block. This is analogous to an L3 cache simulator inside the processor.

We also need a Non-Temporal Buffer (NTB) structure to hold the data requested by bypassing instructions. This buffer allows for data that have reuse only in a short distance to be serviced by this buffer, avoiding polluting the caches. The NTB allows the mechanism to sample only the L1d new misses, drastically reducing its overhead. The NTB acts as a parallel L1d, avoiding filling the L1 with non-temporal data and trickling unnecessary Write Backs through the cache hierarchy.

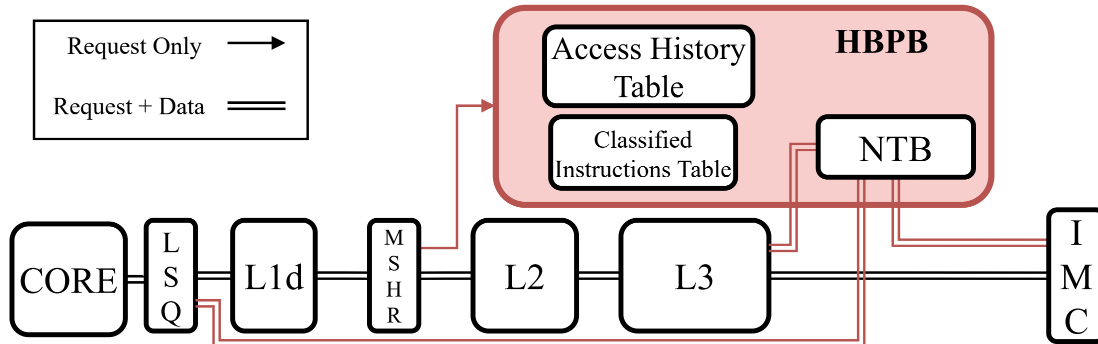

Figure 4.1 – HBPB architectural additions.

HBPB receives requests from the L1 and Last Level Cache, and serves data from the NTB to the LLC and directly to the core Load-Store Queue (LSQ). It requests, receives, and sends data to the DRAM.

4.2 HBPB Operation

The workings of the mechanism are depicted as a flowchart in Figure 4.2.

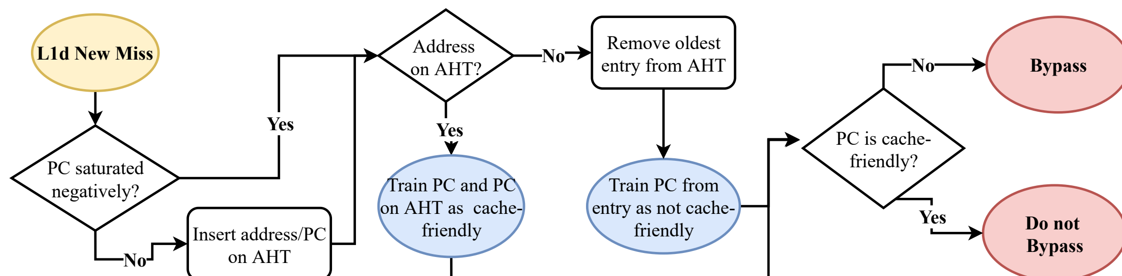

Figure 4.2 – HBPB decision flowchart.

When a memory accessing instruction is executed, and the access provokes a new miss on the data L1 cache, the processor sends the access information to HBPB. The mechanism will then decide if this memory access should be bypassed, and use the data from the access to train its future decisions.

4.2.1 Decision

When HBPB is requested to decide if an access should be bypassed or not, it checks for the presence of the instruction identifier on the CIT and the value of its counter. If the counter is above the threshold, the mechanism signals to the L1d that the access should not be bypassed. If the instruction is not present on the table or its counter is

below the threshold, the L1d is informed that this access should be bypassed.

4.2.2 Training

The mechanism checks for the presence of the memory address on the AHT. If the address is on the table, it means the data has been reused, and the instructions that requested it are cache-friendly, so the mechanism increments the counters for both the current instruction and the instruction that initially brought the data into the AHT on the CIT. If there is no entry for the instructions on the CIT, they are added with the counter set above the threshold.

If the address is not present in the AHT, it is inserted, and one element of the table is evicted, according to the AHT replacement policy, which is the same as the LLC. The instruction that brought the evicted element has accessed a dead line, and so its counter is decremented on the CIT if present. This is done to allow for instructions that bring dead data into the cache to be able to be trained back into bypassing instructions even if they rarely hit in the cache. The value decremented from the counter must be smaller than the value added on hits so that instructions are not excessively penalized for a short burst of evictions of its lines from the AHT.

If the address misses in the AHT and the instruction is present on the CIT with its counter saturated negatively, HBPB understands that this instruction is strongly cache-unfriendly and does not add the address to the AHT. This reduces the number of writes to the structure and makes it a better representation of the cache without the data that is very likely to be bypassed. In this situation, the mechanism still adds the address to the table with a 1/32 chance to allow instructions that eventually change behavior to be reclassified.

4.3 Data and Request Paths With HBPB

The paths of the memory requests through the system are depicted in Figure 4.3, and Figure 4.4 illustrates the paths of the data. When a miss occurs in the L1 data cache for an address that is not already present on the Miss Status Holding Registers (MSHR), it means that a new request needs to be propagated through the memory hierarchy, usually in the form of an L2 request. With the usual operating scheme of the caches, the requests are propagated in only one path: New L1 misses are requested to the L2; New L2 misses are

propagated to the L3, for which new misses generate a new DRAM request. The memory that has the data then sends the data back to the memory that asked for it, trickling the data back to the processor core.

We propose an alternative way for requests to arrive in the DRAM. When the HBPB orders an L1 miss to be bypassed, a new request is created and forwarded to the NTB, which will then propagate the request to the DRAM if it results in a new miss. The lower level caches still have to be checked in parallel because it is possible that the access is a cache hit. When the parallel cache query is confirmed as a miss, the NTB serves the data to the processor, and the pending misses in the cache MSHRs are deleted. If the NTB has a copy of the data, it can serve it to the processor immediately. Having a write-invalidate coherence protocol guarantees that the NTB has the most up-to-date version of the data.

Write operations that miss on the L1 cache generate a Read For Ownership (RFO) request that propagates to the other caches. Bypassed RFOs also propagate the write operation to the NTB, but they can be committed only when the miss on all the caches is confirmed. If the parallel request to the caches ends up resulting in a hit, this pending write is canceled, and the data read from memory is invalidated. When the write is committed in the NTB, all lines storing this address in the caches must be invalidated, following the write-invalidate protocol. To maintain data consistency, the LLC has to confirm that an address is not present in the NTB before accepting a new line from the DRAM, in case it has been written by a bypassed write.

Non-Temporal accesses are deployed to both the caches and the main memory. If the mechanism has mistakenly classified the instruction as not cache-friendly and the request hits in the caches, the access operates regularly, the PC is trained as cache-friendly on the CIT, and the data from the NTB is not read by the access.

The most significant benefit of inserting only data that is likely to be cache-friendly in the caches is that it does not require extensive training before deciding to bypass a cache line. On existing mechanisms, the decision to bypass data from an instruction is only made after said instruction has already flushed the entire cache. By doing this more aggressive bypassing, we can avoid pollution and energy waste from shorter bursts of accesses, leaving the cache available for data that is proven to be reusable.

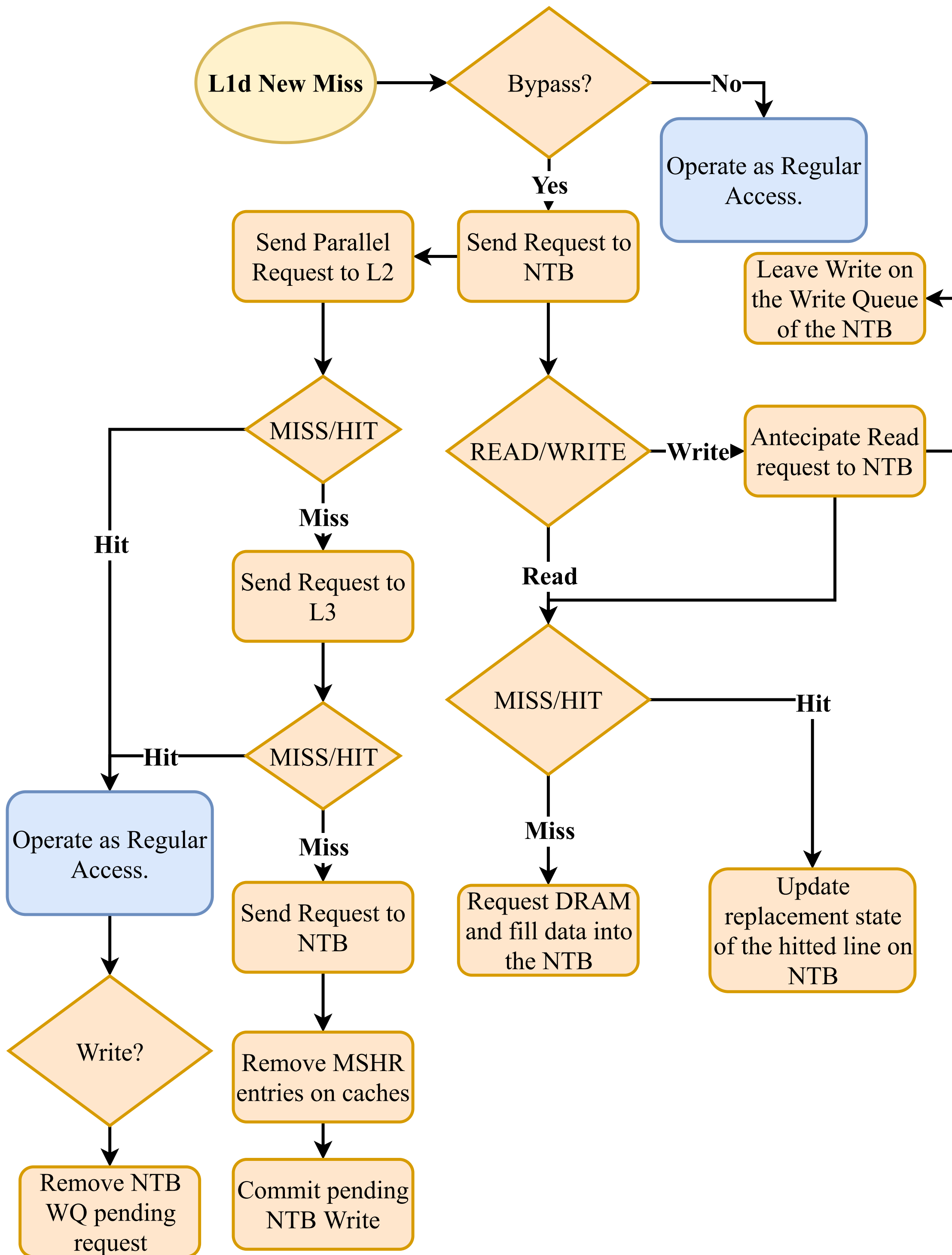

Figure 4.3 – The path of the request through the memory subsystem.

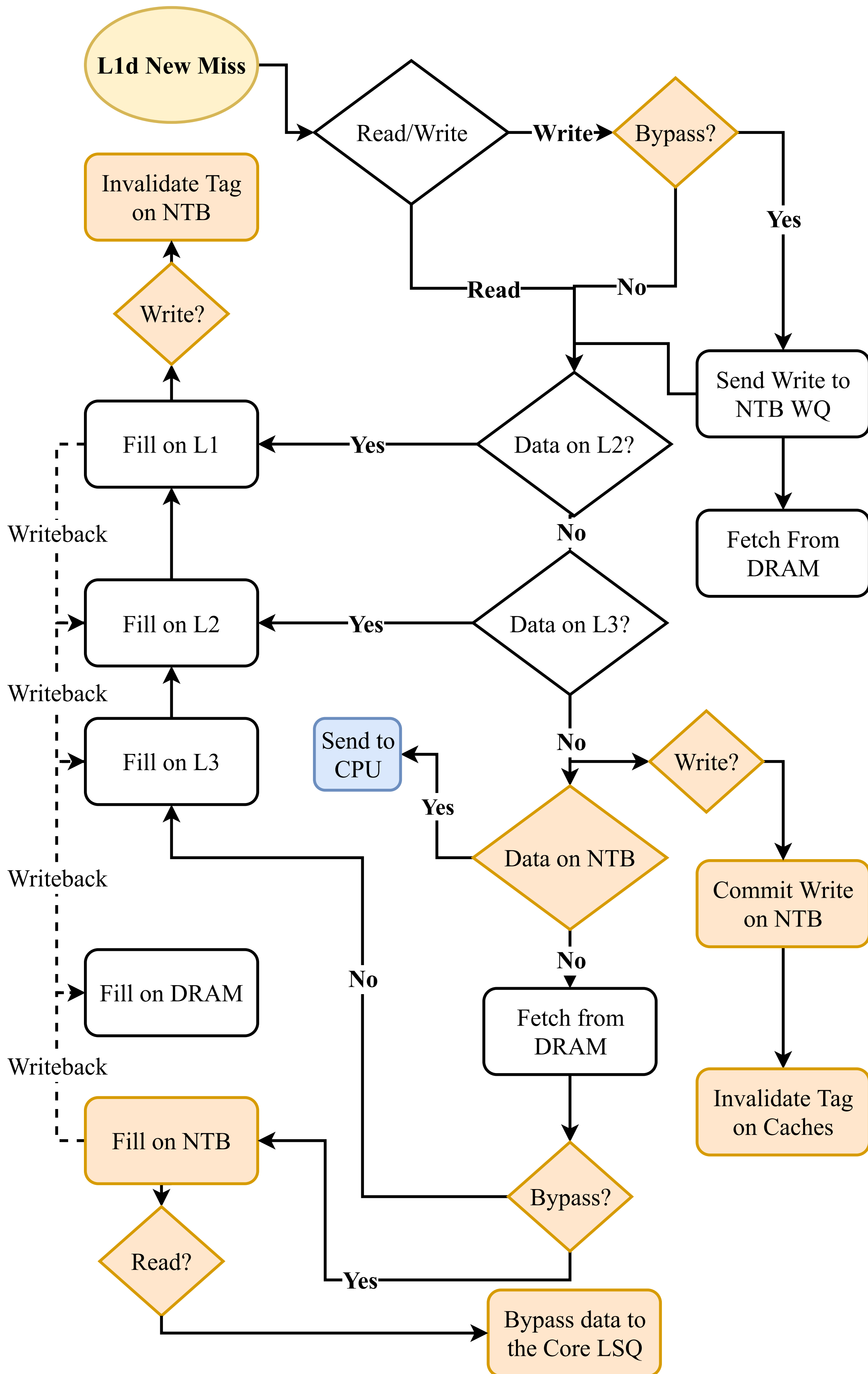

Figure 4.4 – The path of the data through the memory subsystem.

4.4 HBPB Hardware Overhead

The most significant hardware overhead incurred by HBPB is the AHT. This structure must store one entry for every cache line available for the processor. Each entry is comprised of an instruction identifier and a memory address. We do not need to store the whole 64 bits of the Program Counter to identify each instruction inside the mechanism. For better area efficiency, we identify each instruction as a 16-bit hash of its PC. Since the AHT is a clone of the Last Level Cache, the memory address is stored as a combination of index and tag, as in the caches. On a configuration with a 1 MB, 16-way set associative Last Level Cache with 64-Byte cache lines, the tag field of a cache block is 48 bits long.

This means the AHT would need to store 64 bits for every cache line available on the processor LLC, while the LLC needs 560 bits for the tag and data fields. The AHT then needs an area equal to 11.4% of the LLC. Such an implementation implicates that the AHT would require 131 KB of extra storage. Larger cache sizes would need an AHT proportionally smaller because there are fewer bits required for the tag, given that the associativity remains the same.

The other major overhead from HBPB is the NTB, which has the same size as the L1 data cache. In this case, 32 KB for the data array and 3264 Bytes for the tag array.

The CIT must store the instruction identifier with its saturated counter. The mechanism requires this structure to have low read latency since the decision to issue a regular or non-temporal access has the window of time between the instruction decode and its placement on the LSQ. Thus, a hash table of counters indexed by the hash of the Program Counter is an appropriate solution for it. We observed that storing the state of as few as 128 instructions per core yields indistinguishable results compared to an infinite table that stores all instructions.

5 EVALUATION

In this chapter, we detail an evaluation of HBPB and show the results obtained by the mechanism for a group of microbenchmakrs that prove that our proposal works as intended.

5.1 Implementation

We implemented our mechanism in ChampSim [ChampSim 2021], a cycle-accurate, trace-driven x86 simulator. We modeled a system with a cache hierarchy as in Table 2.1, with non-inclusive, non-exclusive, writeback caches.

We modeled a system inspired on the Core i7 4770 CPU, but with a smaller L3 cache to reduce the needed working set size for the benchmarks and bring the simulation time to a more sensible level. Full details of the modeled system are present in Table 5.1. The energy consumption for this cache was adjusted accordingly on Cacti. As a proof of concept, we ran simulations on several microbenchmarks in the simulator.

For fairness in the results, we compared the HBPB configurations with the LLC reduced, to offset the extra overheads from HBPB with a significant margin. The LLC was reduced from 16-way to 13-way, while keeping the same number of sets. This reduces the cache capacity by 18.75%. The latency and energy consumption of the smaller cache was kept the same in order to keep the number of changes at a minimum and avoid giving any unfair advantages to the HBPB configurations. We compare the effects of the mechanism with a baseline configuration without HBPB and a bigger LLC.

Table 5.1 – Modeled system.

Component	**Configuration**
Core	3.4 GHz, 64/36-entry Load/Store Queue, 2 Load Units, 1 Store Unit
L1 Instruction Cache	32KB, 8-way, 4 cycles latency
L1 Data Cache	32KB, 8-way, 4 cycles latency
L2 Cache	256KB, 8-way, 11 cycles latency
Baseline L3 Cache	1MB, 16-way, 34 cycles latency
Non-Temporal Buffer	32KB, 8-way, 4 cycles latency
HBPB L3 Cache	832KB, 13-way, 34 cycles latency
DRAM	DDR3, 1600 MT/s, tCAS = 10, 2 channels, 8 Banks/Channel, Dual Rank

We tested three different versions of HBPB to better understand the practical effects of each of the proposed modifications:

- **HBPB-R**: Bypasses only the requests. Cache fills are done normally;
- **HBPB-F**: Bypasses only the fills. No parallel requests are issued;
- **HBPB-RF**: Bypasses both requests and fills.

We use Cacti [Muralimanohar, Balasubramonian and Jouppi 2009] to estimate the energy consumption of the caches and, to the best the tool can provide, the energy overhead for the HBPB structures. The values used in the evaluation are shown in Table 5.2. Energy consumption values are a product of the number of reads and writes to the memories and their respective energy costs. Each access to HBPB incurs in a read and a write to the AHT, except when the information is not filled. In this case, only a read is accounted for.

Table 5.2 – Energy values used on the evaluation.

Cache	Read Energy (pJ)	Write Energy (pJ)
L1	199	202
L2	169	175
L3	925	936
NTB	199	202
AHT	50	87

We present the effect of the mechanism on the cycles, cache dynamic energy, and Energy-Delay-Squared Product (ED^2P) of the applications. The ED^2P is calculated as a product of the square of the number of cycles and the dynamic energy of the caches, including the HBPB structures. The ED^2P allows the balance between performance and power to be synthesized in a single value. We chose to have the cycles squared to give more weight to the performance part of ED^2P, which is the usual approach when employing this metric. As in the cycles and energy, the smaller the ED^2P, the better.

5.2 Proof of Concept

The mechanism should excel in applications that access large chunks of data with no reuse. Two classic examples of this behavior are linked list traversals and pattern matching. We tested both of these examples with microbenchmarks specially made for this evaluation, as well as a few kernels from the PolyBench benchmark suite [Yuki and Pouchet 2015] that exhibit streaming memory access patterns. Those kernels are often part of full-fledged applications, and if the mechanism achieves its goals on the kernels, it should also transfer to full applications.

Figure 5.1 shows the effect of HBPB on the execution cycles of the kernels compared to the baseline configuration, and Figure 5.2 displays the cache dynamic energy for the kernels, while Figure 5.3 shows the resulting ED^2P.

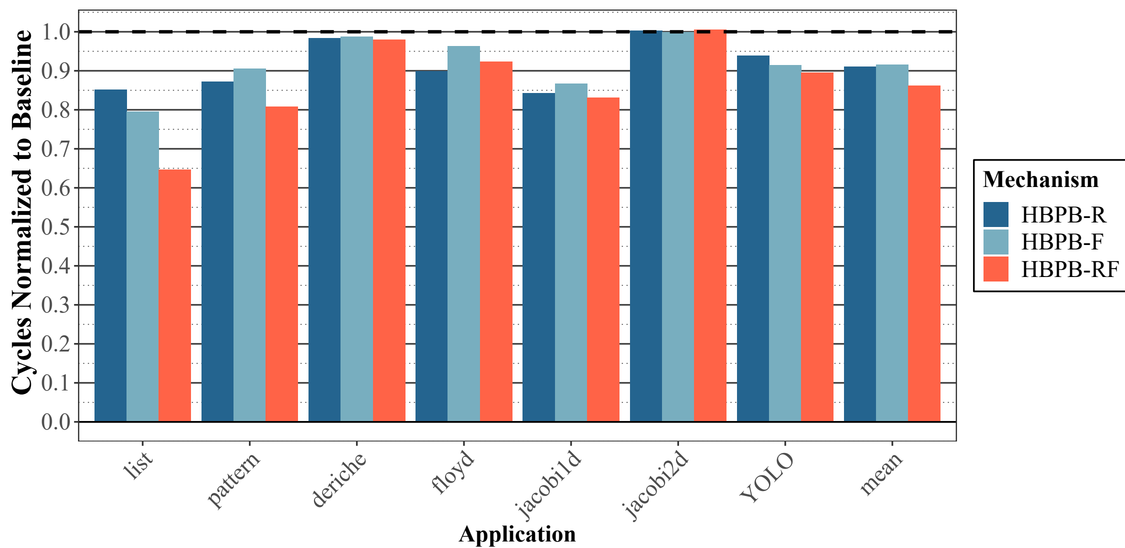

Figure 5.1 – Effect of HBPB on the cycles for the kernel microbenchmarks.

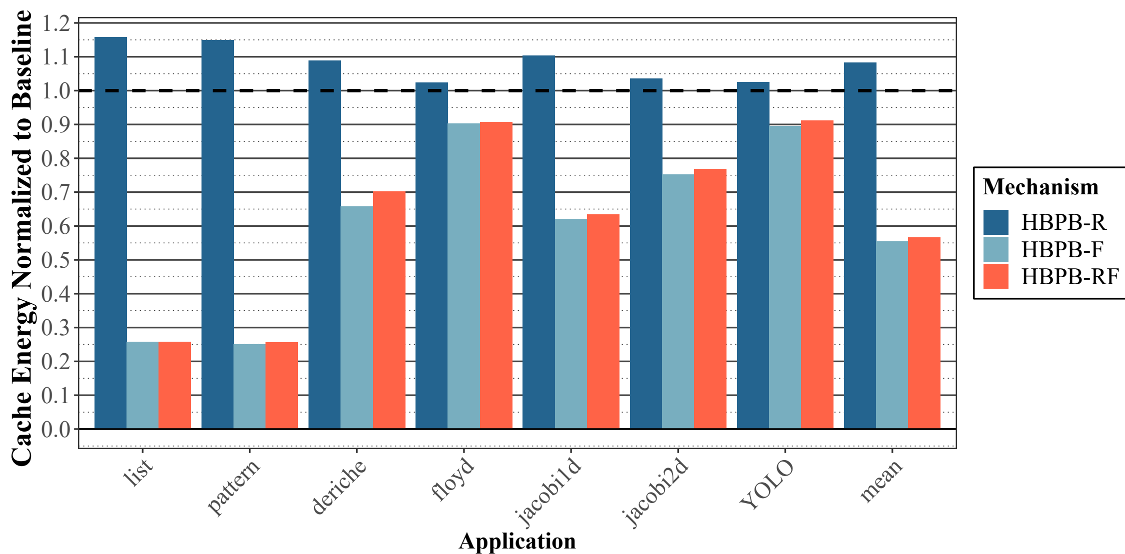

Figure 5.2 – Effect of HBPB on the cache dynamic energy for the kernel microbenchmarks.

5.2.1 Linked List

Linked lists are one of the most common data structures, so they are present in many applications of the most diverse fields. One defining characteristic of linked lists is that when it is being traversed, the position of the next node in memory is only known

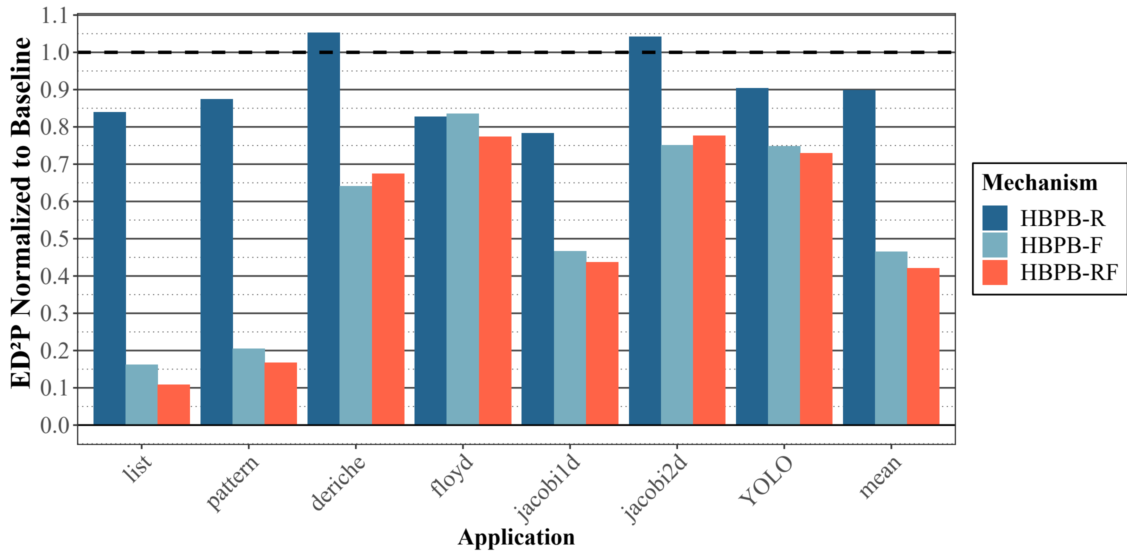

Figure 5.3 – Effect of HBPB on the ED^2P for the kernel microbenchmarks.

Table 5.3 – Working set size from the microbenchmarks used.

Application	Working Set
Linked List	8 MB
Pattern Matching	26 MB
Deriche	270 MB
Floyd-Warshall	2 MB
Jacobi 1D	764 MB
Jacobi 2D	382 MB
YOLO CNN	569 MB

after the current node is read, so it is heavily dependent on the memory access latency. In order to get to a single node, other nodes must be read, and since those nodes are needed only to get to the desired node, they have no reuse. Those factors make caching linked list accesses a poor choice, and bypassing the caches for them shows good potential for improvements.

HBPB improves this benchmark performance by anticipating the requests to the DRAM and shortening the path between memory and core. While HBPB-R and HBPB-F reduce the cycles by 14.9% and 20.5%, respectively, when combined in HBPB-RF, they achieve a 35.3% reduction. HBPB-R cannot reduce the cache fills, but since it also causes fills to the NTB, the total cache energy is increased by 16%. HBPB-F and HBPB-RF behave the same way, and both achieve a 74% reduction in total cache energy. This way, ED^2P is drastically reduced to only 10.8% of its original value.

5.2.2 Pattern Matching

Another typical display of cache unfriendly memory accesses is found in pattern matching applications (e.g., searching for matching text, matching images, DNA sequence comparison). We developed a simple implementation of text matching to benchmark HBPB. When executing *grep*, for example, an application that searches for occurrences of patterns that match given regular expressions, the processor has to load all the files to be searched, byte by byte, and check for the patterns. Most of this data is never reused, and caching is unnecessary. To verify the efficiency of the mechanism in bypassing the cache-unfriendly accesses from pattern matching applications, we developed a simple benchmark that searches for a string on a file using the `string` class from `C++` with its `find()` method, similar to an in-memory grep application. Since it does more than just memory accesses, the mechanism achieves a smaller, 19.2% reduction in cycles for it with HBPB-RF. HBPB-R is responsible for the biggest reduction in cycles, with a 12.8% reduction. Meanwhile, HBPB-F avoids most cache fills and reduces the energy by 75%. Combined in HBPB-RF, the mechanism achieves both performance and power improvements, resulting in an ED^2P of only 16.7% of the baseline configuration, which is equivalent to a 500% improvement.

5.2.3 Jacobi 1D

A 1D stencil application is a good candidate for bypassing. A large structure is traversed while only a few elements are reused in a short interval.

Listing 5.1 – Jacobi 1D kernel code

```
for (t=0; t < STEPS; t++) {
  for (i=1; i < N-1; i++)
    B[i] = 0.33 * (A[i-1] + A[i] + A[i+1]);
  for (i=1; i < N-1; i++)
    A[i] = 0.33 * (B[i-1] + B[i] + B[i+1]);
}
```

Since every iteration of the loop can be done in parallel by the out-of-order processor, the accesses end up being merged, resulting in a single request for an address that will experience a long reuse distance. The stores performed by the application also experience

no reuse since they are performed sequentially to the array. Caching the data for these stores is usually useless.

All versions of HBPB successfully speed up this application, and HBPB-F manages to reduce the cache energy by 38%. The total ED^2P is then reduced by 56.3% with HBPB-RF.

5.2.4 Jacobi 2D

The Jacobi 2D benchmark from PolyBench is similar to its 1D counterpart but operates on 2D matrices instead of 1D vectors. HBPB cannot achieve improvements as good as on the 1D version of the benchmark, resulting in a slight regression in performance, with a 0.6% increase in total cycles with HBPB-RF. However, the cache energy is dramatically reduced due to the conversion of all the stores into non-temporal stores, bypassing the caches. The cache dynamic energy is reduced by 24.8% with HBPB-F and 23.3% with HBPB-RF. Even with a performance reduction, the ED^2P is reduced by 25% for HBPB-F and 22.3% for HBPB-RF.

5.2.5 Floyd-Warshall

The Floyd-Warshall algorithm finds the shortest path between every pair of nodes in a graph [1]. The PolyBench implementation utilizes an $n \times m$ matrix where the value of each position represents the distance between the nodes n and m. The kernel accesses memory with a very stream-like behavior, in which HBPB-R manages to work well, reducing the total cycles by over 10%. HBPB-F reduces the cache energy by also 10%. When combined in HBPB-RF, the two forms of bypassing yield a reduction in ED^2P of 22.6%.

5.2.6 Deriche

The Deriche filter can be used for edge detection or smoothing of images. The kernel performs horizontal and later vertical passes through an image, with little to no

[1] Appendix B contains the results obtained for other graph applications, from the GAP benchmark suite [Beamer, Asanović and Patterson 2015]

data reuse on each pass. This application performs many mathematical computations for every memory access, so the impact of the memory access latency on the final performance is smaller than it would be otherwise. HBPB did not manage to yield significant performance improvements on this kernel, only 2%. However, the cache energy was massively reduced by nearly 34.2% with HBPB-F and 29.8% with HBPB-RF. The resulting ED^2P was then equivalent to the reductions in energy consumption.

5.2.7 Yolo CNN

The You Only Look Once (YOLO) Convolutional Neural Network (CNN) [Redmon et al. 2016] is a very fast CNN used to detect objects and their boundaries in real-time. This network is faster than others at detection because it only needs a single network evaluation, unlike other methods. We ran the CPU implementation of the network on a 16 MP image and extracted the execution trace, which was then simulated in our test environment. We used the Tiny-YOLO version for its faster execution completion.

In this application, HBPB-RF reduced total cycles by 10.5% and cache energy by 8.9%. The resulting reduction in ED^2P was 27%.

Since object recognition is one of the tasks that leverage the most benefits from computing on the edge, due to the video stream not needing to be transmitted to a different device [Abbas et al. 2017], achieving energy savings on this kind of application is essential since edge devices are frequently powered by a battery.

5.2.8 Final Remarks

On average, performance was improved by 15% with HBPB-RF on the tested microbenchmarks, while the cache energy was reduced by 44.6%. Total ED^2P was reduced by 58% on average with HBPB-RF. HBPB-R always increases the total energy because it can not avoid cache fills; it only adds fills to the NTB and AHT. HBPB-F was demonstrated to be the biggest contributor to the final effects of the mechanism.

Figure B.14 shows the total accesses to HBPB by type and the decisions made by mechanism for each of the microbenchmarks, represented as a percentage of the total instructions of the benchmark. The plot clearly shows how the linked list and pattern matching applications are similar in access types, being constituted of only loads, which

are mostly bypassed. Since the linked list is randomly shuffled across memory, though, some of the time, the reuse distance of an access will fall within the cache capacity, and the instruction will be trained as cache-friendly.

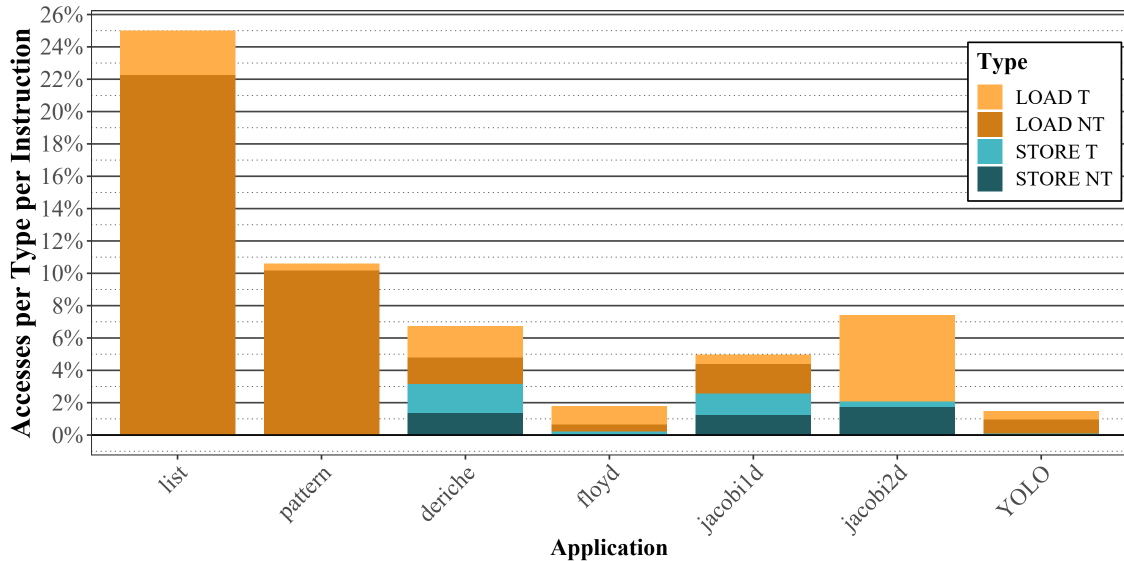

Figure 5.4 – Frequency of each type of access for the microbenchmarks with HBPB.

The deriche application shows a mix between loads and stores, and both are bypassed by HBPB half of the time. Floyd-Warshall has mostly loads, and only about 10% of them are bypassed. This correlates well with the reduction in cache energy observed. On Jacobi 1D, nearly all loads are bypassed, but only about a half of the stores are bypassed. On the two-dimensional version of the Jacobi algorithm, only the stores are bypassed. This explains why the performance for this application is not improved, since it depends on the latency of the loads, not the stores. The stores, however are all bypassed, and hence the cache energy is drastically reduced. For YOLO, most of the accesses are also bypassed, although, like Floyd-Warshall, less than 2% of the program instructions perform memory accesses that result in a new L1 miss, which reduces the ability of HBPB to improve performance even more.

6 RESULTS AND ANALYSIS

In order to verify that the benefits from HBPB on the kernels transfer to full applications, we tested the mechanism with some applications from the memory intensive benchmarks from SPEC CPU 2017 [Henning 2021], using the traces generated for the ML-Based Data Prefetching Competition [Jiménez 2019], using the biggest workload sizes for each application. Table 6.1 shows the size of the working sets for each of the applications tested, obtained by counting the unique memory addresses acessed on the traces.

Table 6.1 – Working set size from the SPEC CPU 2017 applications used.

Application	**Working Set**
cactuBSSN	273 MB
fotonik3d	1.5 GB
gcc	177 MB
lbm	2.5 GB
mcf	1.7 GB
omnetpp	47 MB
roms	378 MB
wrf	64 MB
xalancbmk	16 MB

6.1 Experimental Results

In this section, the effects of HBPB on the performance and energy consumption of the SPEC applications is presented.

6.1.1 Performance

Regarding the execution cycles, HBPB-R was the most effective on the SPEC benchmarks, reducing the total cycles by 6% on average. On *cactuBSSN* and *mcf*, the effect was nearly inexistent, while *omnetpp* was accelerated by merely 1%, and *wrf* by 2.4% with HBPB-R. Bigger improvements were seen on the execution times of *fotonik3d*, which was reduced by 7%, *lbm*, by 5.4%, *roms* by 8.3% and *xalancbmk*, by 9.1%. With a 18.1% reduction in execution cycles, *gcc* was the application that benefited the most with HBPB-R regarding performance.

HBPB-F improved the performance only on *mcf* and *xalancbmk*, by 12.8% and 6.3% respectively. When combined in HBPB-RF, the performance was a product of both previous versions. When only HBPB-R improved the performance, the gains were smaller on HBPB-RF, but when both versions showed improvement, the final gains were greater than in any of the previous versions. On *mcf*, the total cycles were reduced by 19.5% with HBPB-RF, and over 14% on *xalancbmk*. On average, the cycles were reduced by 5.1% with HBPB-RF.

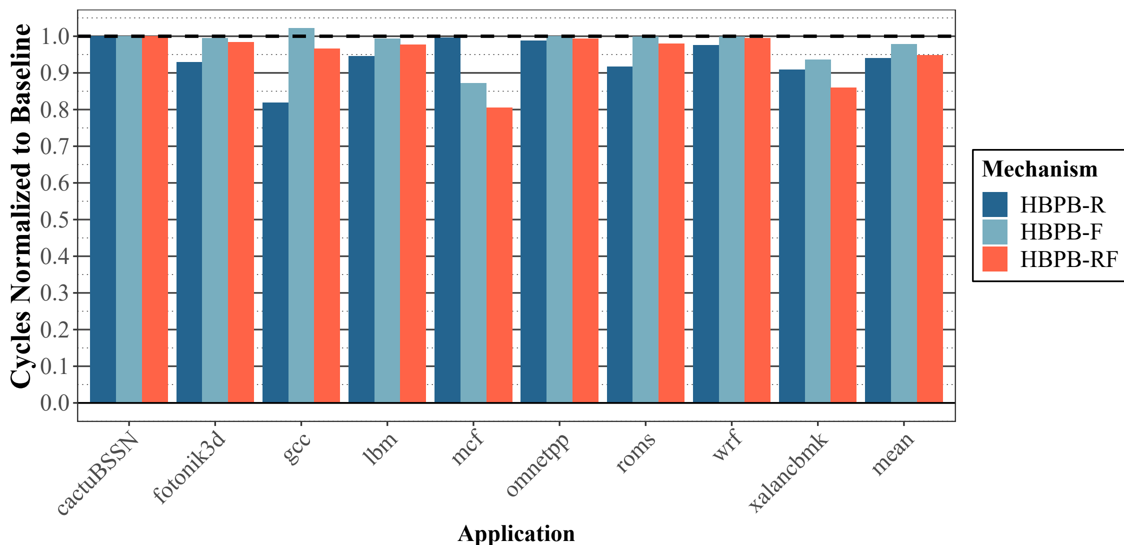

Figure 6.1 – Effect of HBPB on the cycles for the SPEC CPU 2017 benchmarks.

6.1.2 Cache Energy

When looking at the effects of the mechanism on the cache dynamic energy, shown in Figure 6.2, HBPB-R never manages to yield improvements, since it only anticipates memory requests to the DRAM. At the same time, it only adds cache fills to the NTB and AHT. This version goes as far as increasing the energy consumption by 12.7% on *gcc*, and 4% on average. Meanwhile, HBPB-F main goal is to reduce cache energy consumption, in which it succeeds. On *cactuBSSN*, *fotonik3d* and *gcc*, the changes were smaller than 1%. Small reductions are also observed on *omnetpp*, *roms* and *wrf*, with 1.2%, 2.6% and 1.6% respectively.

Massive improvements were yielded by HBPB-F on *lbm*, *mcf* and *xalancbmk*. On *lbm*, the reduction in cache energy consumption was 13%, and 17.6% on *xalancbmk*. The greatest reduction happened on *mcf*, where HBPB-F managed to reduce the energy consumption by 29.1%. On Average, energy consumption was reduced by 7.7% with HBPB-

F. The effect of HBPB-RF on the caches' energy consumption was strongly correlated to the effect of HBPB-F. On average, HBPB-RF yielded a reduction of 7.3%.

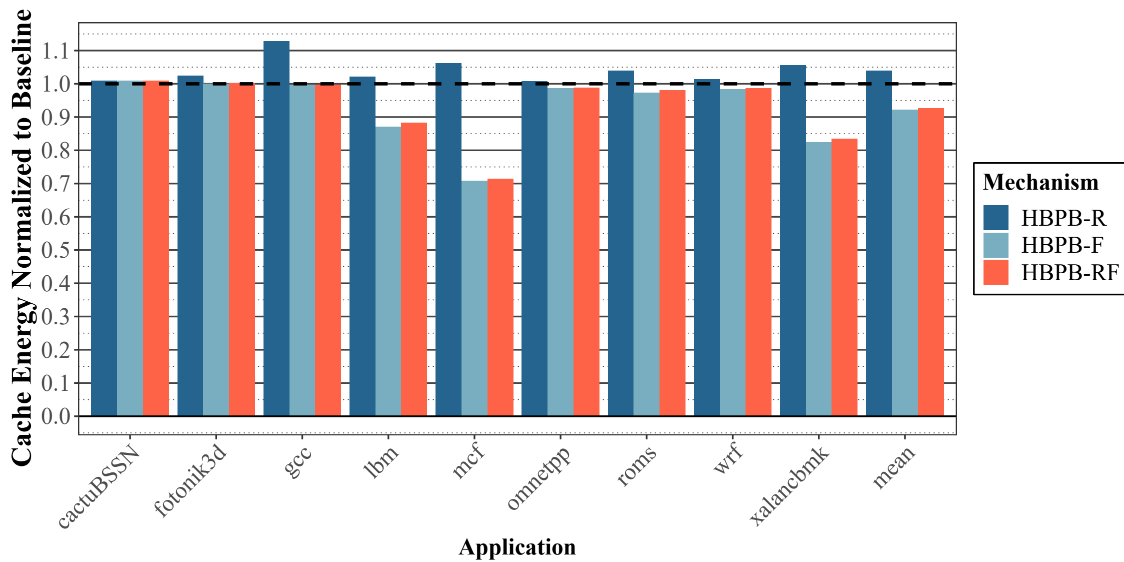

Figure 6.2 – Effect of HBPB on the cache dynamic energy for the SPEC CPU 2017 benchmarks.

6.1.3 Energy-Delay-Squared Product

The resulting effects of HBPB on the ED^{2}P of the SPEC 2017 benchmarks were positive except for two occasions, as shown in Figure 6.3. On *gcc*, since HBPB-F decreased the performance and did not improve the energy consumption significatively, the ED^{2}P was increased by 4.1%. On *mcf*, since HBPB-R increased the energy consumption and decreased the total cycles by a smaller margin, the ED^{2}P ended up being raised by 5.3%. HBPB-RF never deteriorated the ED^{2}P. Apart from *cactuBSSN*, where the mechanism had nearly to no effect, and *fotonik3d*, *wrf* and *omnetpp*, where the decreases in ED^{2}P were of 2.9%, 2.3%, and 2.4%, respectively, HBPB-RF yielded sizeable benefits when considering the ED^{2}P as the desired metric.

On *gcc*, even if not as great as the reduction by HBPB-R, of 24%, HBPB-RF reduced the ED^{2}P by nearly 7%, simply due to the speed-up achieved by bypassing the requests. On *roms*, the ED^{2}P reduction yielded by HBPB-R was also greater than the one by HBPB-RF, since the speed-up with the first was bigger. HBPB-R reduced ED^{2}P by 12.6%, while HBPB-RF reduced it by 5.8%. On *lbm*, *xalancbmk* and *mcf*, since HBPB-RF improved both performance and energy efficiency, the reductions in ED^{2}P are much more substantial, being 15.8%, 38.4% and 53.7% respectively. On average, HBPB-RF reduced ED^{2}P by 16.6%.

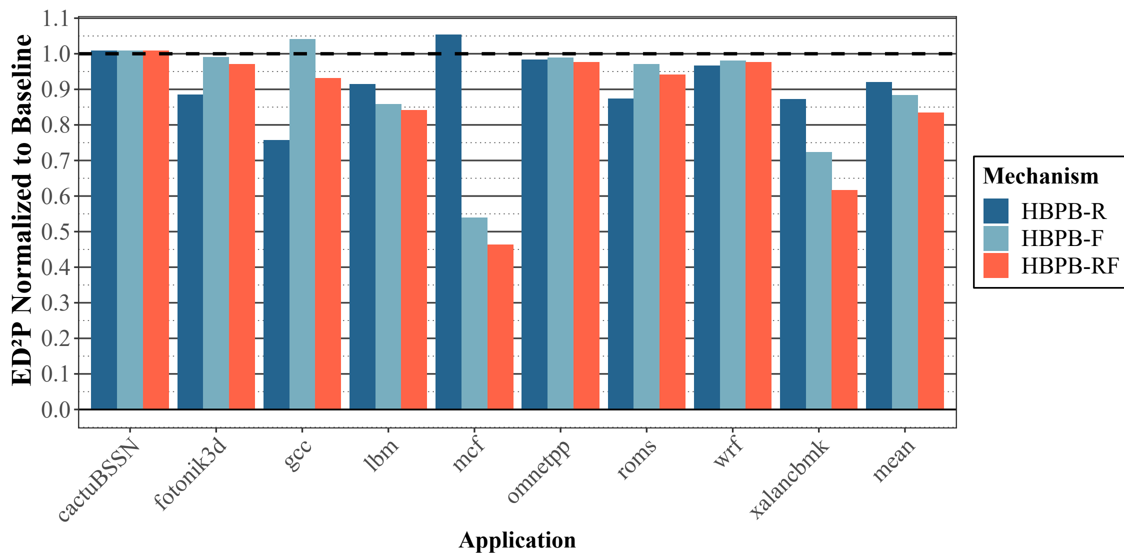

Figure 6.3 – Effect of HBPB on the ED^{2}P for the SPEC CPU 2017 benchmarks.

6.1.4 Summary

Table 6.2 summarizes the key impacts of HBPB on the SPEC applications' performance and energy efficiency. For some of the applications, HBPB-RF massively improved energy efficiency and performance while never affecting any application negatively by more than 1%.

Table 6.2 – Summary of the percentual reductions in Cycles and Cache Energy obtained with HBPB on the SPEC CPU 2017 applications.

	cactuBSSN	*fotonik3d*	*gcc*	*lbm*	*mcf*	*omnetpp*	*roms*	*wrf*	*xalancbmk*	*mean*
Cycles (R)	0.0	**7.0**	**18.1**	**5.4**	0.4	1.3	**8.4**	2.4	**9.1**	**5.9**
Cycles (F)	0.0	0.5	-2.1	0.7	**12.3**	0.0	0.1	0.1	**6.3**	2.1
Energy (F)	-0.9	0.0	0.3	**12.9**	**29.1**	1.3	2.6	1.6	**17.6**	**7.7**
Cycles (RF)	0.0	1.6	3.4	2.4	**19.5**	0.6	2.0	0.5	**14.1**	**5.1**
Energy (RF)	-0.9	-0.2	0.3	**11.7**	**28.6**	1.1	2.0	1.3	**16.6**	**7.3**

The plot in Figure 6.4 shows the decisions made by HBPB-RF on incoming requests arrived after an L1 miss. It helps in understanding the effects of the mechanism on the applications from the SPEC CPU 2017 benchmark suite. The accesses are presented as a percentage of the total program instructions. On *lbm*, *xalancbmk*, *gcc*, and *mcf*, applications that were sped-up by the mechanism, a sizeable portion of the Load instructions were bypassed, reducing their latency and improving the performance. On *lbm*, the improvements in energy efficiency were much greater than those in performance, which is explained by most of the accesses from this application that suffered interference from HBPB being Stores. Since Stores are not as relevant for program performance, the effect

of bypassing them is more perceivable on the cache energy consumption.

Some applications were not affected by the mechanism for different reasons. On *cactuBSSN* for example, the HBPB decided to make regular accesses for practically all the instructions. On *wrf*, *omnetpp* and *fotonik3d* there were simply not enough L1 misses for HBPB to act on. Meanwhile, *mcf* has more than 16 L1 misses by 100 instructions, and HBPB-RF can leverage this behavior to bypass many accesses and improve efficiency.

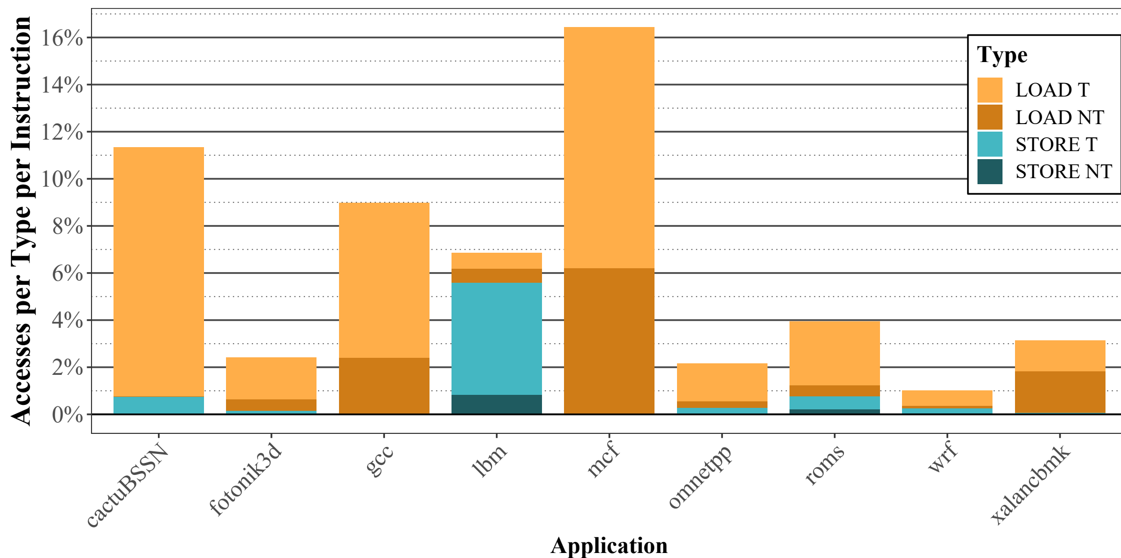

Figure 6.4 – Frequency of each type of access for the SPEC 2017 benchmarks with HBPB-RF.

Since bypassing the caches affects the L1 miss pattern, which is the information used by HBPB to perform on, it is expected that HBPB-R and the versions that bypass the fills, HBPB-F and HBPB-RF, make slightly different decisions. The plot in Figure 6.5 shows the decisions made by HBPB-R. The most significant difference to be noted is the fact that across multiple applications, the ratio of accesses that are bypassed is bigger. Most notably, nearly all the requests from *gcc* and *fotonik3d* are bypassed on this version of the mechanism, which explains the greatest speed-ups achieved by HBPB-R for these applications. Other applications, such as *lbm* and *roms*, which also have a bigger speed-up with HBPB-R than with HBPB-RF, have a significant increase in bypassed loads with the first as well.

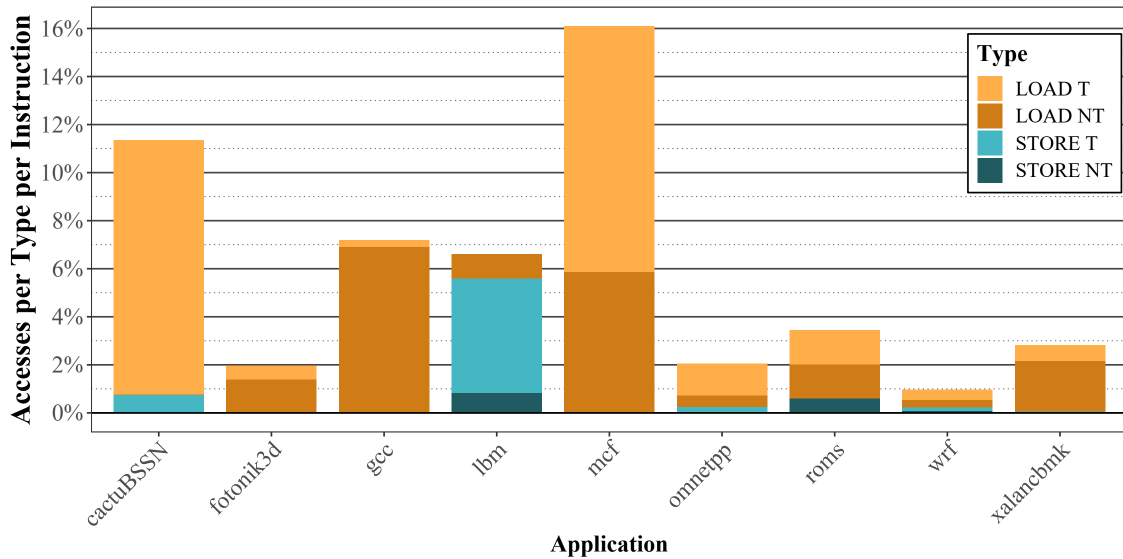

Figure 6.5 – Frequency of each type of access for the SPEC 2017 benchmarks with HBPB-R.

6.2 Static Reuse Distance Analysis

We performed an analysis of the memory access traces of the applications to better understand why HBPB was able to improve performance in some programs but not in others. ChampSim was modified to provide an output of all the memory accesses, with their instruction PC and the memory address of the data.

With this data, we calculated the reuse distance - the amount of memory accesses to unique addresses between two references to the same address - for each access and grouped them by instruction. We group the reuse distances into five buckets:

- **MSHR**: When the reuse distance is between 0 and 2. We assume that requests to the same address that are close to each other are merged in the buffers;
- **L1**: When the reuse distance is greater than the MSHR but smaller than the size of the L2 cache;
- **L2**: When the reuse distance is greater than the L1 cache but smaller than the size of the L3 cache;
- **L3**: When the reuse distance is smaller than the capacity of the caches but greater than the capacity of the L2 cache;
- **Miss**: When the reuse distance is greater than the capacity of the caches.

We calculated both the forward distance - the number of unique addresses referenced between a given reference and the next reference to the same address - and the backward distance - the number of unique addresses accessed between a given reference

and the last reference to this address. For example, if the backward distance for an access is inside the L3 bucket, we expect this reference to result in an L3 hit. If the forward distance for this access is into the Miss bucket, we expect that the next reference to this address will be a cache miss or that it is never going to be accessed again.

From the line in the middle, we plot the reuse distances for each of the 135 most frequently occurring instructions of each application, spread across the x-axis ordered by frequency. The bars are colored regarding the frequency of each reuse distance bucket. The forward reuse distances are stacked above the line, while the backward distances are stacked below the line.

We combine this data with the simulation results obtained when running the applications with HBPB-RF. The instructions that have been bypassed are annotated with a color code representing the ratio of bypasses performed for the instruction. Green lines indicate that the instruction was bypassed more than 90% of the time, yellow means the instruction was bypassed between 50% and 90% of the time, while red lines indicate that the memory accesses from the instruction were bypassed between 10% and 50% of the time. Store instructions have the colored shape above the bars, while loads are annotated at the bottom.

In this section, we show the results for all the applications from SPEC CPU 2017 that were tested. The ideal application for HBPB has many misses and instructions that are clearly offending in a manner that they are responsible for bringing dead data into the cache or always causing misses when accessing the memory.

6.2.1 *cactuBSSN*

Figure 6.6 shows the reuse distances of the most frequently occurring memory accessing instructions on the *cactuBSSN* benchmark. Due to the sheer amount of different instructions in this application alone, HBPB can not profile each instruction correctly. In fact, there are over 400 instructions that perform the highest number of memory accesses, and thousands of other instructions that also perform a very high number of accesses. As shown in the previous section, HBPB does not perform bypasses for *cactuBSSN*.

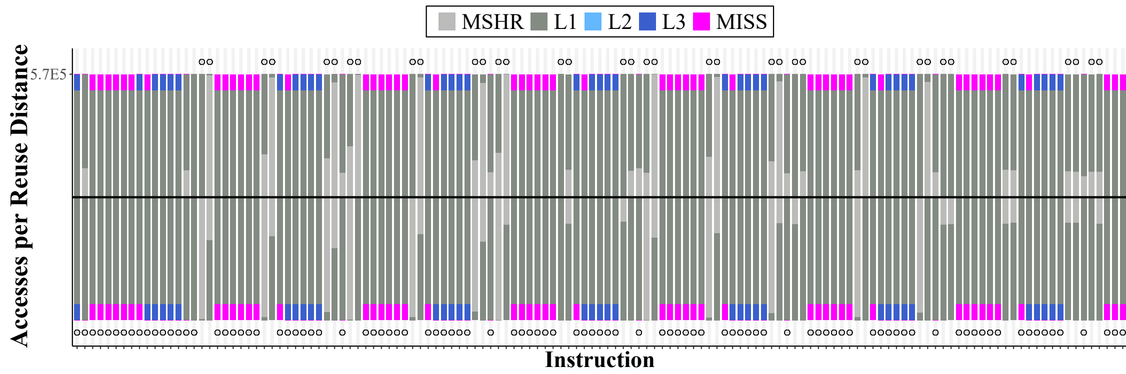

Figure 6.6 – Frequency of reuse distances for the 135 most frequently occurring instructions for *cactuBSSN*.

6.2.2 *fotonik3d*

The distribution of memory accesses between the instructions is similar on *fotonik3d*, as shown in Figure 6.7. The accesses are however more concentrated on fewer instructions, which allows HBPB to bypass some of the accesses. The reuse distances for the most frequent instructions is either too long or too short, so HBPB can bypass most of these instructions, exclusively loads. It suffers from the same problem as *cactuBSSN* of having hundreds of instructions that access the memory.

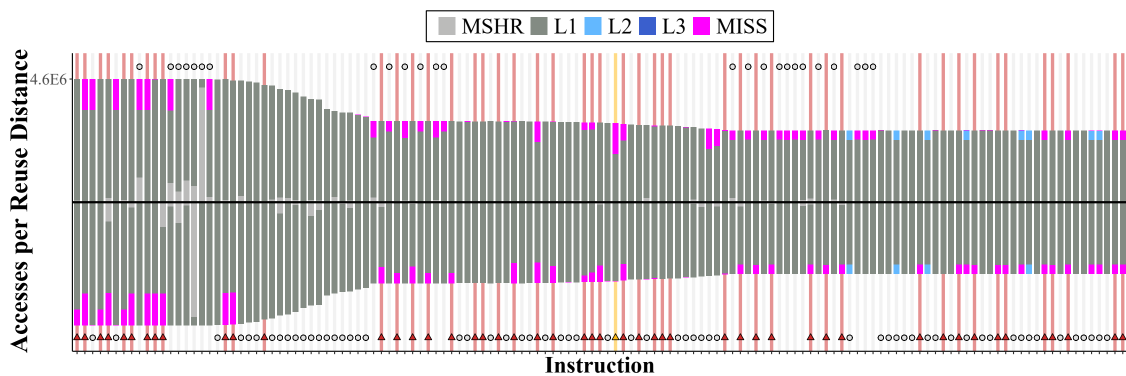

Figure 6.7 – Frequency of reuse distances for the 135 most frequently occurring instructions for *fotonik3d*.

6.2.3 *gcc*

On *gcc*, the case is exactly the opposite. A single load instruction is responsible for the majority of the accesses while the rest is almost all concentrated in the second most frequent instruction, as depicted in Figure 6.8. The reuse distances from these instructions fall in either too long or very short for the most part. HBPB-RF bypasses 30% of the

accesses from the main instruction, but HBPB-R bypasses it every time.

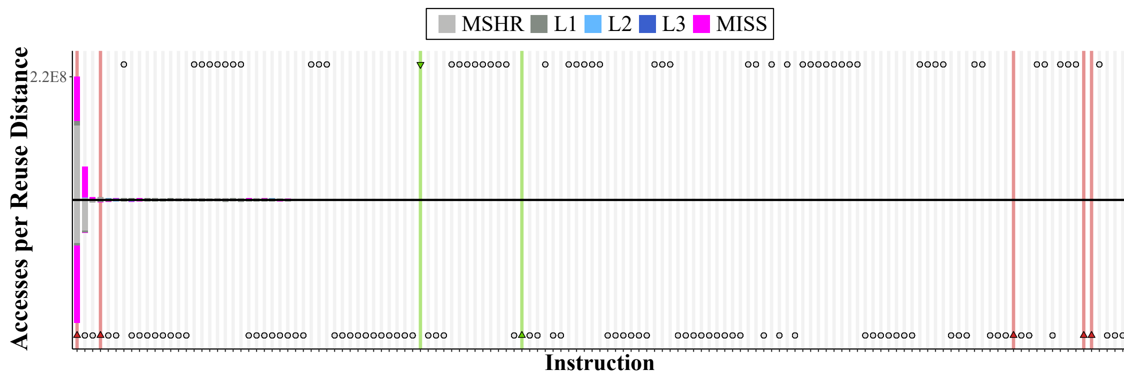

Figure 6.8 – Frequency of reuse distances for the 135 most frequently occurring instructions for *gcc*.

6.2.4 *lbm*

The distribuition of reuse distances per instruction for *lbm* is extremely suitable to HBPB. As shown in Figure 6.9, the most frequent load instruction performs more than double the accesses of the other loads, and the backwards reuse distance of the addresses it accesses is always in the Miss bucket. This allows HBPB to bypass the accesses for this instruction that always cause a cache miss, reducing the access latency. HBPB bypasses this instruction 50% of the time. Among the remaining instructions, most are stores that show reuse. HBPB bypasses some of them. After the stores, there are loads, of which a good parcel of them also display reuse distances that suit bypassing, and HBPB correctly detects and bypasses their accesses.

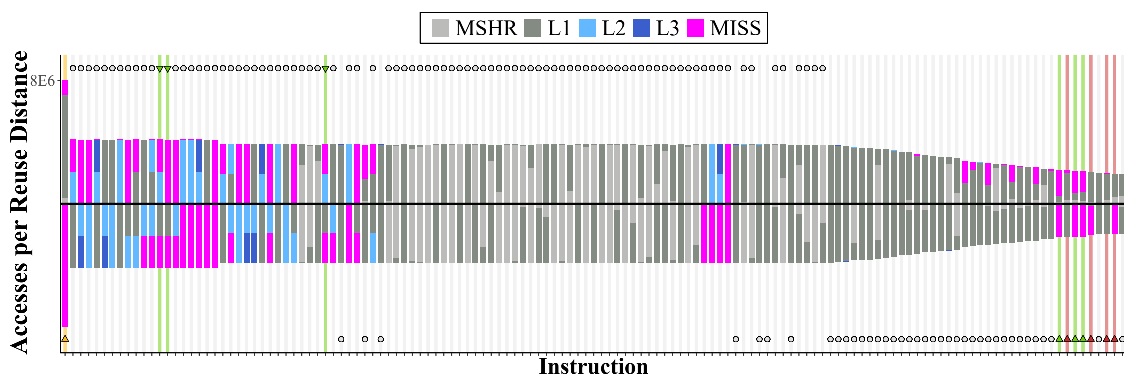

Figure 6.9 – Frequency of reuse distances for the 135 most frequently occurring instructions for *lbm*.

6.2.5 *mcf*

The distribution of the reuse distances of *mcf*, as shown in Figure 6.10 highlights the potencial of this application to benefit from reduced cache pollution. The three most frequent instructions from *mcf* are responsible for the majority of the memory accesses, with the fourth and fifth most frequent instructions also heavily access the memory. The most frequent instruction is heavily cache-unfriendly, and HBPB correctly bypasses nearly all the accesses from this instruction. The second and third instructions are deeply cache-friendly, and HBPB does not bypass their accesses, while the fourth and fifth most frequent instructions are also almost always bypassed. All of them are loads, which makes their bypassing more contributing to the final performance.

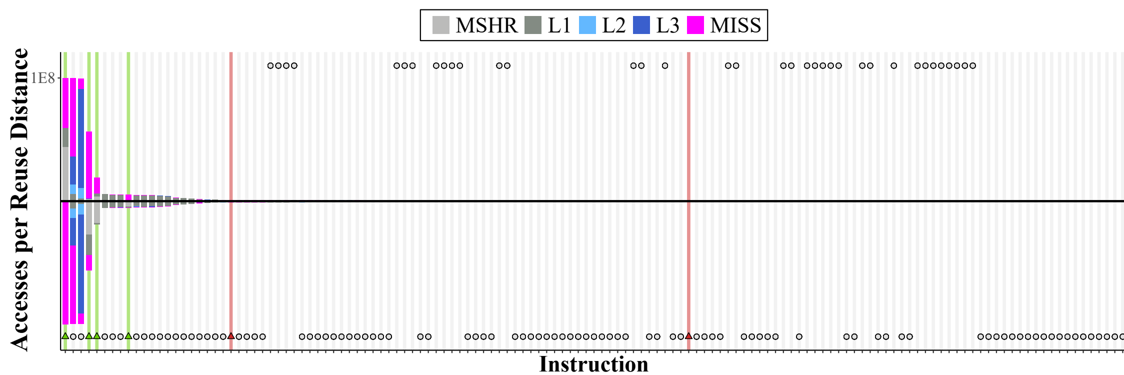

Figure 6.10 – Frequency of reuse distances for the 135 most frequently occurring instructions for *mcf*.

The bypassed instructions that would otherwise pollute the cache with their data no longer do it. This leaves more space in the caches for the also very frequent cache-friendly instructions. That is the reason for the performance being improved mainly by HBPB-F on this application. HBPB-F can display all its potential in *mcf*, where it speeds up the application by avoiding cache pollution, saves energy by not filling unnecessary data into the cache, and avoids fills by not evicting data that has reuse for data that does not.

6.2.6 *omnetpp*

On *omnetpp*, for which the reused distances are shown in Figure 6.11, the most frequently occurring memory accessing instructions display short reuse distances that frequently fall into the L1, L2 and L3 range. There are no clearly non-temporal instructions.

Only a very small amount of the least frequent instructions are bypassed.

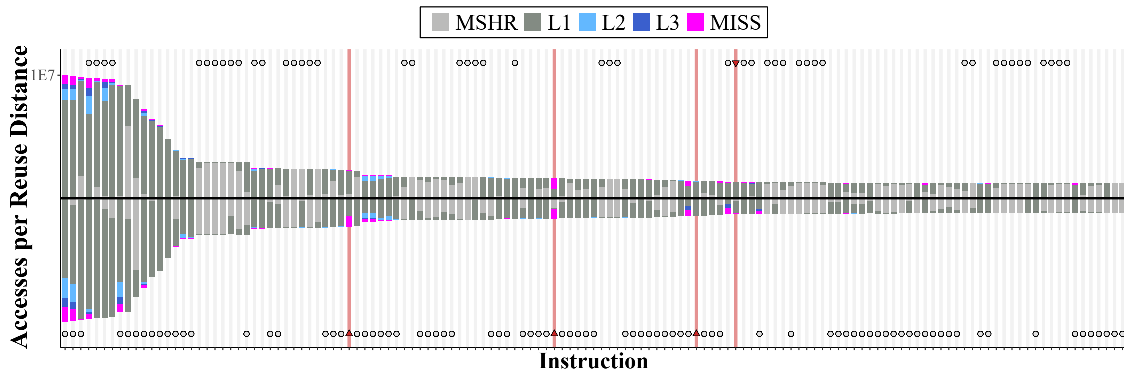

Figure 6.11 – Frequency of reuse distances for the 135 most frequently occurring instructions for *omnetpp*.

6.2.7 *roms*

Similarly to *fotonik3d*, the accesses from *roms* are well distributed among the instructions, as shown in Figure 6.12. A good parcel of the instructions are bypassed eventually, but none stand out as a clear offender that must be bypassed all the time.

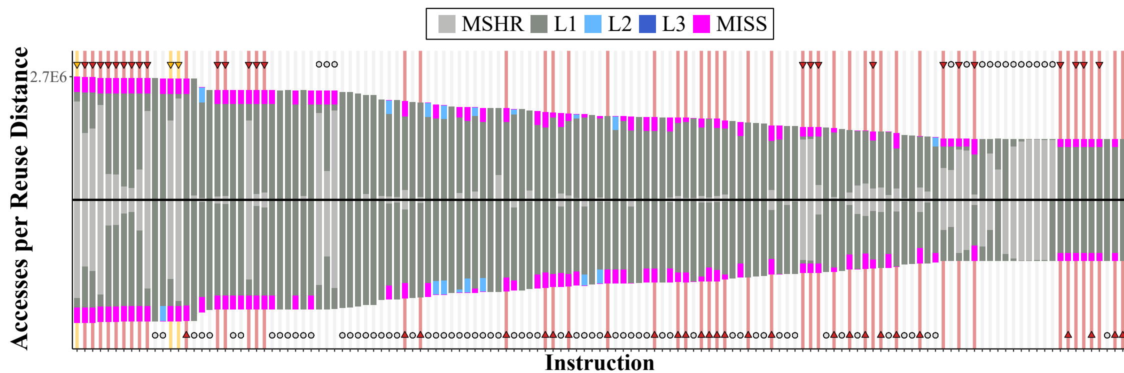

Figure 6.12 – Frequency of reuse distances for the 135 most frequently occurring instructions for *roms*.

6.2.8 *wrf*

The memory accesses of *wrf* are distributed among many instructions, with a few being more frequent than the rest, as seen in Figure 6.13. Nearly all of the instructions, however, have very short reuse distances, so they are better served by the L1 cache. As demonstrated previously on this section, only a small amount of the accesses actually miss on the L1 cache and need intervention from HBPB.

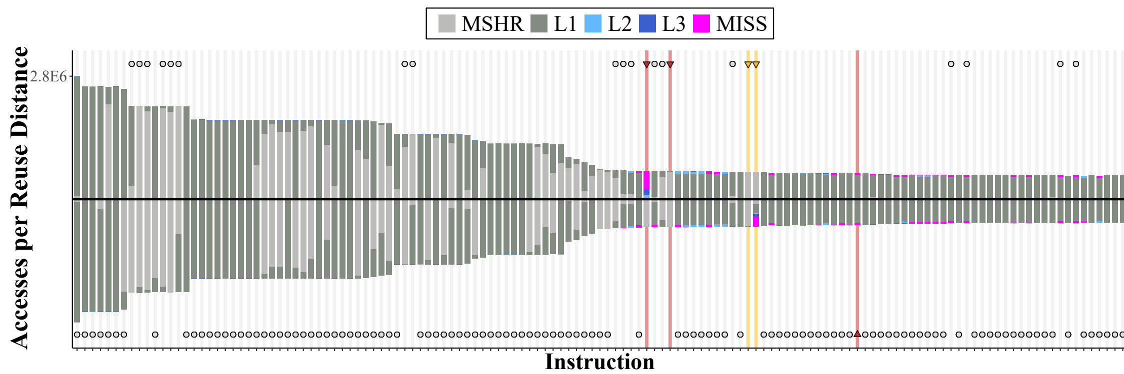

Figure 6.13 – Frequency of reuse distances for the 135 most frequently occurring instructions for *wrf*.

6.2.9 *xalancbmk*

On *xalancbmk*, the majority of the accesses is distributed among few instructions, of which a significant portion exhibits long reuse distances, as shown in Figure 6.14. These instructions, however, also show some sort of reuse inside the L2 and L3 ranges, making caching them sometimes useful. HBPB then bypasses the accesses from these instructions the majority of the time, but not always. If in a specific program phase these instructions show cache-friendliness, they are not bypassed.

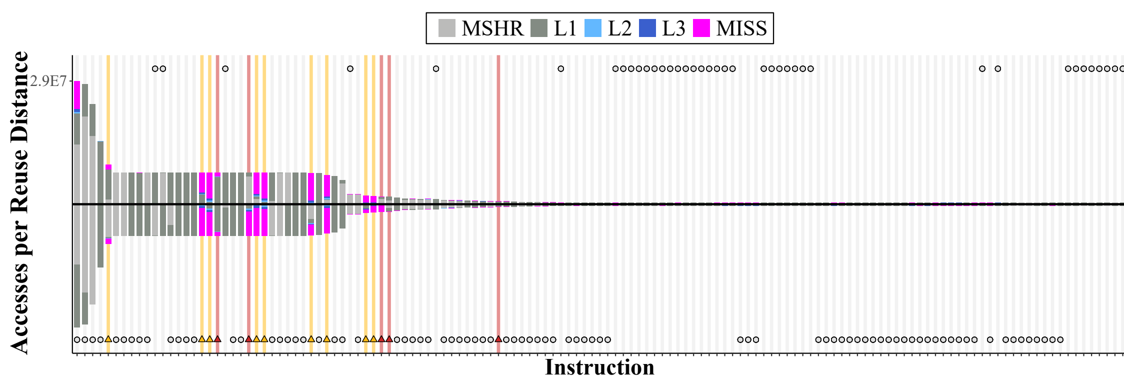

Figure 6.14 – Frequency of reuse distances for the 135 most frequently occurring instructions for *xalancbmk*.

This analysis allows us to understand more deeply how HBPB works, and proves how the instruction reuse pattern is consistent for each instruction. Specially on *mcf*, it is clear how having predominantely offending instructions coexisting with predominantely cache-friendly instructions allows HBPB-RF to show its strength at its fullest.

6.3 Sensibility to Memory Configuration

Different hardware configurations change the cache behavior and can impact the effects of HBPB. We tested HBPB with a different cache size and replacement policies and introduced hardware data prefetchers to study their impact on the mechanism.

6.3.1 Bigger Cache Capacity

With a higher cache capacity, bypassing can become less attractive since the caches have a higher probability of serving the data and being useful. As shown in Figure 6.15, HBPB is less effective if the L3 capacity is bigger. On *xalancbmk*, the mechanism has no effect with an 8 MB LLC, which is expected since this application's working set fits almost entirely into the L3. Still, HBPB-RF improved ED^{2}P even with bigger LLC capacities.

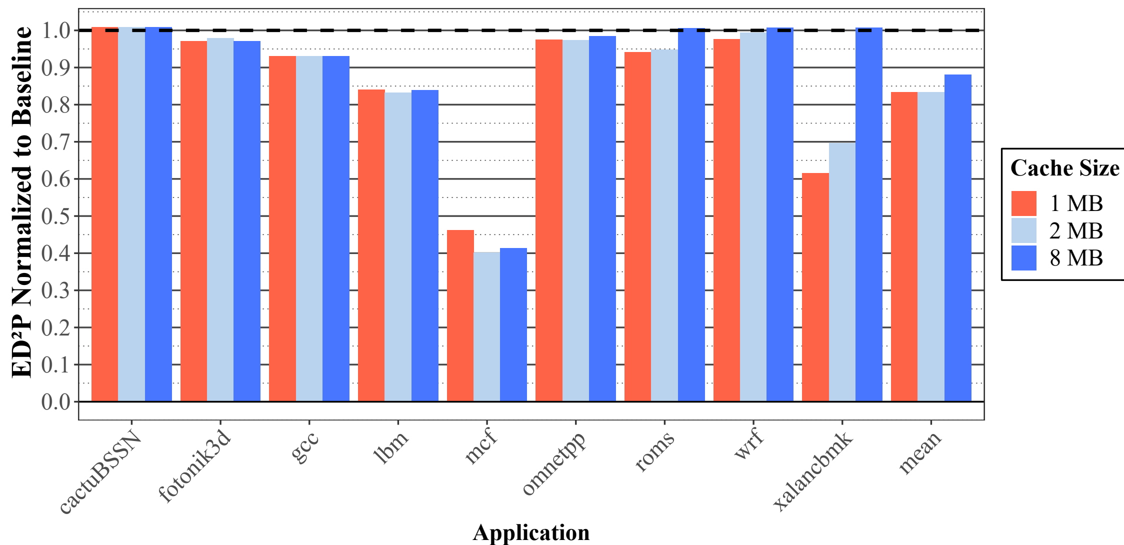

Figure 6.15 – Effect of HBPB on the cycles for the SPEC CPU 2017 benchmarks with different cache capacities.

Figure 6.16 shows the types of accesses and the decisions from HBPB-RF on an 8MB LLC. As expected, all the accesses from *xalancbmk* are cached normally in this version.

Naturally, if the latency of the caches were compensated for their increased size, the speed-up from bypassing would be magnified, as would the energy consumption reduction if the energy per access was increased accordingly. In this case, the decrease in ED^{2}P from HBPB-RF would be even more significant.

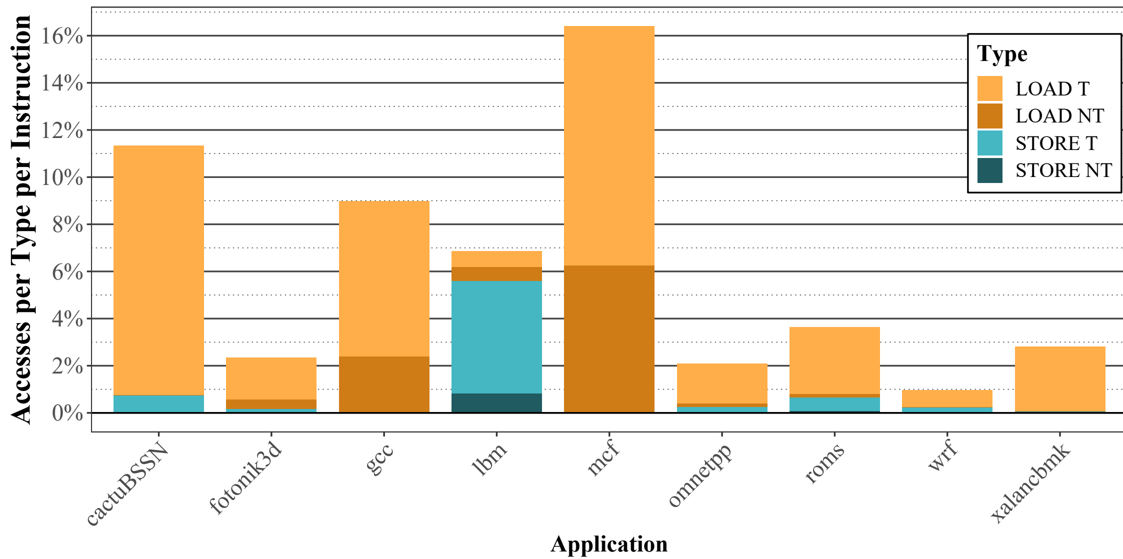

Figure 6.16 – Frequency of each type of access for the SPEC 2017 benchmarks with HBPB-RF with an 8MB LLC.

6.3.2 Prefetcher

Data prefetching is a very useful technique employed to reduce the memory access latencies. By anticipating what data the processor may need in the near future, the prefetchers can bring this data closer to the processor, into a higher level cache, for example. If a prefetch is not usefull, however, it incurs an unnecessary cache fill, eviction, and potentially the need to re-fetch the data that was evicted, resulting in reduced energy efficiency and performance.

Hardware data prefetchers completely change the pattern of requests between the levels of the memory hierarchy, including the L1 misses, and it is deemed necessary to investigate how prefetching impacts the effects of HBPB. For the cache prefetchers, we utilized a next-line prefetcher for the L1, an IP-Stride prefetcher for the L2, and a next-line prefetcher for the LLC that operates only on regular accesses, not cache checks from bypassed accesses. We also utilized an IP-Stride prefetcher on the NTB that operates only on bypassed accesses.

The plot in Figure 6.17 shows the effect of the prefetchers and HBPB-RF on the execution cycles of the applications from SPEC CPU 2017 normalized to the version without HBPB nor prefetchers. The prefetchers, when employed on the baseline version, always improved the performance when compared to the version without it, going as far as reducing the total cycles by 69% on *gcc*. This version, apart from *mcf*, consistently outperformed HBPB-RF with no prefetchers. On *mcf*, however, the access pattern is not a

stream, where the employed prefetchers can accurately work, and they end up decreasing performance by causing more cache pollution. HBPB-RF with prefetchers performed only slightly better than the baseline version with prefetchers.

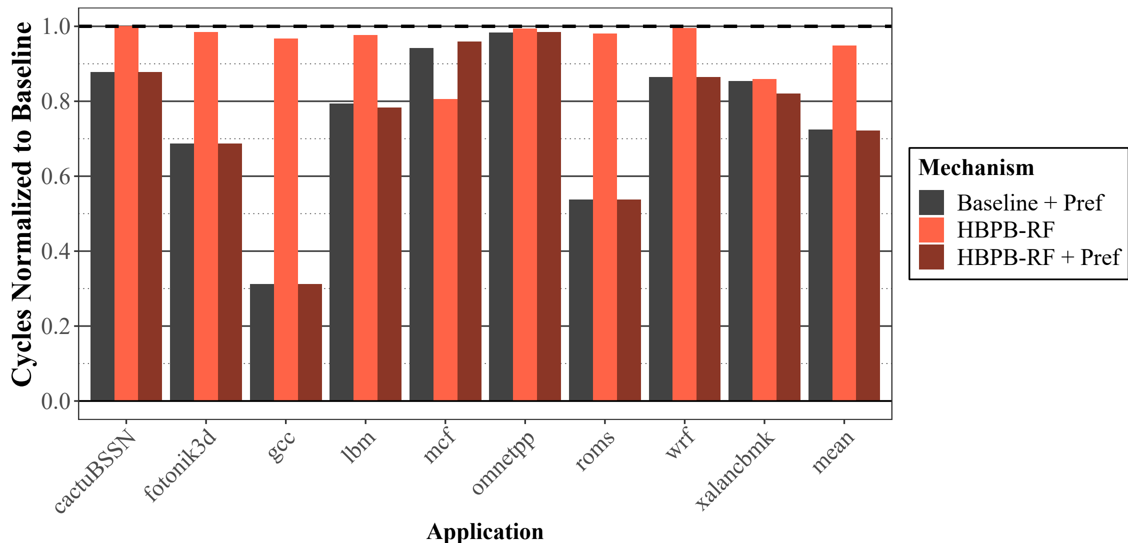

Figure 6.17 – Effect of HBPB on the cycles for the SPEC CPU 2017 benchmarks with prefetchers.

The prefetchers massively increased cache energy consumption, as shown in Figure 6.18. On *mcf*, since almost every prefetch from the LLC was inaccurate, the energy consumption was doubled. On average, prefetchers increased energy consumption by 31.5% on the baseline version and 26.9% when employed with HBPB-RF.

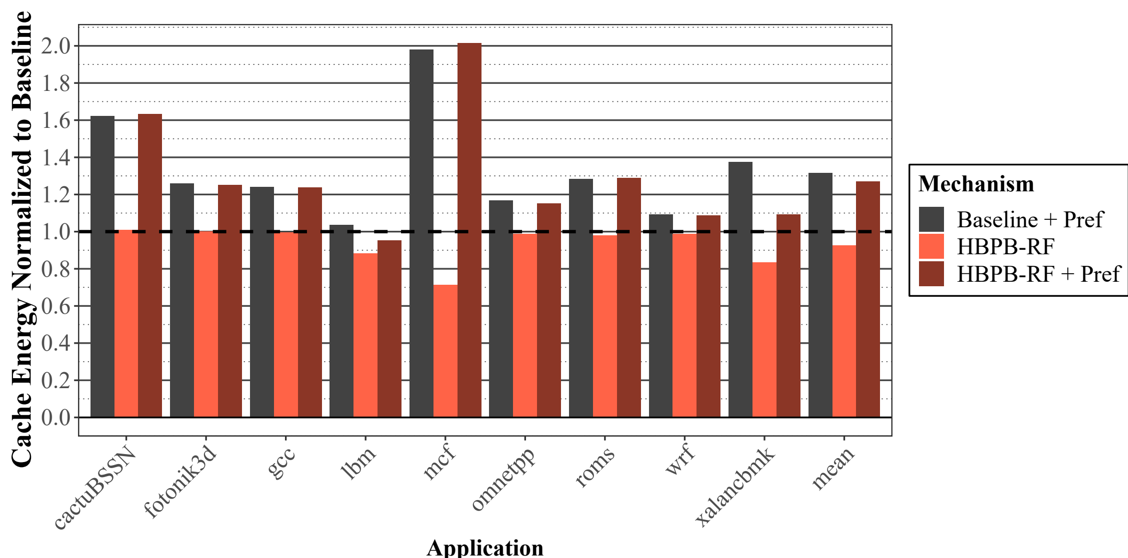

Figure 6.18 – Effect of HBPB on the cache energy for the SPEC CPU 2017 benchmarks with prefetchers.

The effect of the prefetchers on the ED^2P strongly varied across the applications, as shown in Figure 6.19. On *cactuBSSN*, *mcf*, *omnetpp*, and *xalancbmk*, prefetchers wors-

ened the final ED^2P, while on the others they yielded improvements. When compared to the baseline version with no HBPB nor prefetchers, the prefetchers alone reduced ED^2P by 31%, and when employed over HBPB-RF, ED^2P was reduced by 34%. When comparing both versions with prefetchers, HBPB reduced ED^2P by 4.5%.

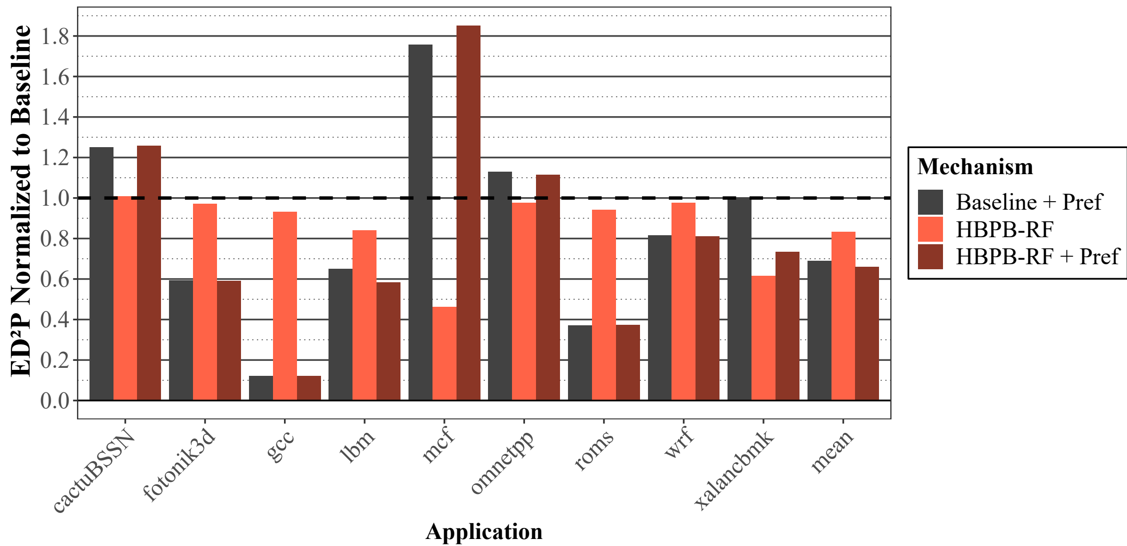

Figure 6.19 – Effect of HBPB on the ED^2P for the SPEC CPU 2017 benchmarks with prefetchers.

Across all the applications and metrics, the best configuration always included HBPB-RF. Table 6.3 shows the best version among the configurations with HBPB-RF, baseline, with and without prefetchers. Versions with results within 0.5% are considered equal. On average, HBPB overperformed the baseline configuration on all metrics, even with hardware prefetchers available. For ED^2P, HBPB improved over the baseline configuration in 5 out of 9 applications.

Table 6.3 – Best configuration for each metric for each of the SPEC applications considering prefetchers.

Application	**Best Performance**	**Best Energy**	**Best ED^2P**
cactuBSSN	HBPB/Baseline + Pref	HBPB/Baseline	HBPB/Baseline
fotonik3d	HBPB/Baseline + Pref	HBPB/Baseline	HBPB/Baseline + Pref
gcc	HBPB/Baseline + Pref	HBPB/Baseline	HBPB/Baseline + Pref
lbm	**HBPB + Pref**	**HBPB**	**HBPB + Pref**
mcf	**HBPB**	**HBPB**	**HBPB**
omnetpp	HBPB/Baseline + Pref	**HBPB**	**HBPB**
roms	HBPB/Baseline + Pref	**HBPB**	HBPB/Baseline + Pref
wrf	HBPB/Baseline + Pref	**HBPB**	**HBPB + Pref**
xalancbmk	**HBPB + Pref**	**HBPB**	**HBPB**
mean	HBPB/Baseline + Pref	**HBPB**	**HBPB+Pref**

6.3.3 Cache Replacement Policies

Different cache replacement policies have been developed, aiming to keep data that is more useful in the cache to the detriment of data that is deemed less useful. We test HBPB on three of these policies, employed on the LLC, as well as LRU, which was used previously. The AHT uses the same replacement policy as the LLC.

- **LRU**: Least Recently Used
- **SHiP**: Signature-based Hit Predictor (PC) [Wu et al. 2011]
- **SRRIP**: Static Re-Reference Interval Prediction [Jaleel et al. 2010]
- **DRRIP**: Dynamic Re-Reference Interval Prediction [Jaleel et al. 2010]

Figure 6.20 shows the impact of HBPB-RF on the ED^{2}P of the applications with the four different replacement policies. Apart from *cactuBSSN*, HBPB-RF always improves ED^{2}P. The performance of HBPB across the different policies did not vary significatively. With DRRIP, SHiP, and to a lesser extent SRRIP, HBPB had a smaller impact on *xalankbmk* due to the ability of these policies to keep a fraction of the working set on the caches since the working set from *xalancbmk* is so small, a sizeable portion of it is able to stay on the cache and improve the final performance, mitigating the gains from HBPB.

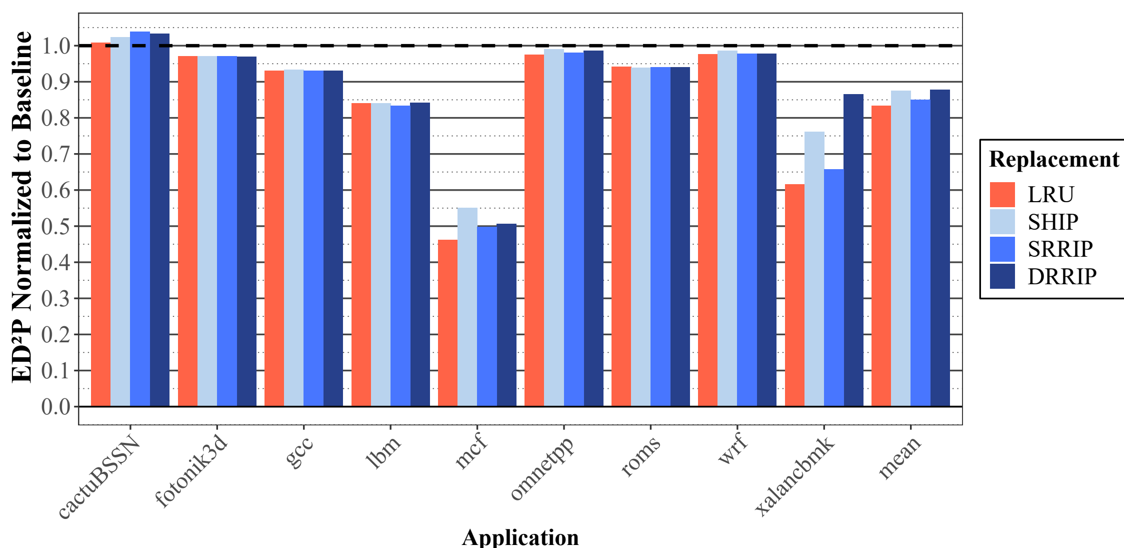

Figure 6.20 – Effect of HBPB on the ED^{2}P for the SPEC CPU 2017 benchmarks with different LLC replacement policies.

LRU is the replacement policy where HBPB yielded the biggest reductions in ED^{2}P. On average, 16.5%. With the other three policies, however, HBPB still improved

ED^2P, although on a smaller margin. On SHiP, the average reduction was 12.5%. On SRRIP, 15%, and on DRRIP, 12.1%. Although with some variance, HBPB performed similarly with different LLC replacement policies, still yielding considerable reductions in ED^2P.

7 CONCLUSION

This thesis presented History-Based Preemptive Bypassing (HBPB), a novel mechanism for bypassing the cache for all memory accesses from instructions not known to be cache-friendly based on the access history. HBPB learns which instructions are cache-friendly by keeping a copy of the tag array of the caches as if there were no bypasses and checking if the access would hit on this cache.

HBPB improves the latency of instructions that miss in the cache through the anticipation of DRAM requests and improves overall system performance by avoiding cache pollution. Avoiding unnecessary cache fills and Write Backs reduces energy waste in the caches. Improving the hit ratio reduces DRAM accesses, which further enhances energy efficiency.

Compared to a traditional cache architecture with no prefetchers, HBPB achieves reductions in execution time of up to 37% for a linked list microbenchmark and 20.5% for a full-fledged SPEC CPU application. The mechanism also yields energy savings of up to 75% for the microbenchmark and 30% for a SPEC application even when accounting for its overhead. When using ED^2P as a metric, HBPB improves nearly all the highly memory-intensive applications tested from SPEC 2017.

We performed an analysis of the reuse distance of the accesses from each instruction for every application tested to understand why HBPB had the effects observed. Applications that have a mix of strongly cache-unfriendly instructions with cache-friendly instructions benefited the most from HBPB.

We tested HBPB when employing hardware data prefetchers, and the mechanism was proven to be either advantageous or at least not detrimental on every application when considering any metric. HBPB also still proved to be useful when considering different cache replacement policies and cache capacities.

7.1 Future Work

In a future work, we intend to perform an analysis of the effects of HBPB on a multicore architecture. On such a hardware, HBPB should have its capabilities magnified due to the increased pressure on the memory subsystem and competition for the caches. Eventually, the mechanism would have to be tweaked to perform better on this condition. HBPB has the potential to mitigate cache pollution from a streaming program while

keeping the caches for cache-friendly applications.

REFERENCES

ABBAS, N. et al. Mobile edge computing: A survey. **IEEE Internet of Things Journal**, IEEE, v. 5, n. 1, p. 450–465, 2017.

ALBERICIO, J. et al. The reuse cache: Downsizing the shared last-level cache. In: IEEE. **2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)**. [S.l.], 2013. p. 310–321.

BAE, H. J.; CHOI, L. Filter cache: filtering useless cache blocks for a small but efficient shared last-level cache. **The Journal of Supercomputing**, Springer, v. 76, n. 10, p. 7521–7544, 2020.

BALASUBRAMONIAN, R. et al. Cacti 7: New tools for interconnect exploration in innovative off-chip memories. **ACM Transactions on Architecture and Code Optimization (TACO)**, ACM New York, NY, USA, v. 14, n. 2, p. 1–25, 2017.

BEAMER, S.; ASANOVIĆ, K.; PATTERSON, D. The gap benchmark suite. **arXiv preprint arXiv:1508.03619**, 2015.

BECKMANN, N.; SANCHEZ, D. Maximizing cache performance under uncertainty. In: IEEE. **2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)**. [S.l.], 2017. p. 109–120.

CHAMPSIM. **ChampSim**. [S.l.]: GitHub, 2021. Retrieved November 18, from <https://github.com/ChampSim/ChampSim>.

CHAUDHURI, M. et al. 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**. [S.l.: s.n.], 2012. p. 293–304.

CHI, C.-H.; DIETZ, H. Improving cache performance by selective cache bypass. In: IEEE COMPUTER SOCIETY. **Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences. Volume 1: Architecture Track**. [S.l.], 1989. v. 1, p. 277–278.

EGAWA, R. et al. A layer-adaptable cache hierarchy by a multiple-layer bypass mechanism. In: **Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies**. [S.l.: s.n.], 2019. p. 1–6.

GAUR, J.; CHAUDHURI, M.; SUBRAMONEY, S. Bypass and insertion algorithms for exclusive last-level caches. In: **Proceedings of the 38th annual international symposium on Computer architecture**. [S.l.: s.n.], 2011. p. 81–92.

GUIDE, P. Intel® 64 and ia-32 architectures software developer's manual. **Volume 2B: Instruction Set Reference**, 2020.

GUPTA, S.; GAO, H.; ZHOU, H. Adaptive cache bypassing for inclusive last level caches. In: IEEE. **2013 IEEE 27th International Symposium on Parallel and Distributed Processing**. [S.l.], 2013. p. 1243–1253.

HENNING, J. **SPEC CPU 2017®**. 2021. Retrieved February 12, 2021 from <https://www.spec.org/cpu2017/Docs/overview.html#benchmarks>.

INTEL. Intel 64 and IA-32 architectures optimization manual. 2018.

JAIN, A.; LIN, C. Back to the future: Leveraging belady's algorithm for improved cache replacement. In: IEEE. **2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)**. [S.l.], 2016. p. 78–89.

JAIN, A.; LIN, C. Rethinking belady's algorithm to accommodate prefetching. In: IEEE. **2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)**. [S.l.], 2018. p. 110–123.

JALEEL, A. et al. High performance cache replacement using re-reference interval prediction (rrip). **ACM SIGARCH Computer Architecture News**, ACM New York, NY, USA, v. 38, n. 3, p. 60–71, 2010.

JIMÉNEZ, D. A.; TERAN, E. Multiperspective reuse prediction. In: IEEE. **2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)**. [S.l.], 2017. p. 436–448.

JIMéNEZ, D. **Third Data Prefetching Championship ChampSim Traces**. 2019. Retrieved February 12, 2021 from <https://dpc3.compas.cs.stonybrook.edu/?SW_IS>.

KHARBUTLI, M.; JARRAH, M.; JARARWEH, Y. Scip: Selective cache insertion and bypassing to improve the performance of last-level caches. In: IEEE. **2013 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT)**. [S.l.], 2013. p. 1–6.

KIM, J. et al. Kill the program counter: Reconstructing program behavior in the processor cache hierarchy. **ACM SIGPLAN Notices**, ACM New York, NY, USA, v. 52, n. 4, p. 737–749, 2017.

KÖHLER, R.; ALVES, M. Acelerando requisições de prováveis cache misses com requisições em paralelo cache/dram. In: SBC. **Anais Estendidos do IX Simpósio Brasileiro de Engenharia de Sistemas Computacionais**. [S.l.], 2019. p. 101–106.

KUMAR, S.; SINGH, P. An overview of modern cache memory and performance analysis of replacement policies. In: IEEE. **2016 IEEE International Conference on Engineering and Technology (ICETECH)**. [S.l.], 2016. p. 210–214.

MURALIMANOHAR, N.; BALASUBRAMONIAN, R.; JOUPPI, N. P. Cacti 6.0: A tool to model large caches. **HP laboratories**, v. 27, p. 28, 2009.

REDMON, J. et al. You only look once: Unified, real-time object detection. In: **Proceedings of the IEEE conference on computer vision and pattern recognition**. [S.l.: s.n.], 2016. p. 779–788.

SANDBERG, A.; EKLÖV, D.; HAGERSTEN, E. Reducing cache pollution through detection and elimination of non-temporal memory accesses. In: IEEE. **SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis**. [S.l.], 2010. p. 1–11.

SEMBRANT, A.; HAGERSTEN, E.; BLACK-SCHAFFER, D. Data placement across the cache hierarchy: Minimizing data movement with reuse-aware placement. In: IEEE. **2016 IEEE 34th International Conference on Computer Design (ICCD)**. [S.l.], 2016. p. 117–124.

SHI, Z. et al. Applying deep learning to the cache replacement problem. In: **Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture**. [S.l.: s.n.], 2019. p. 413–425.

SIM, J. et al. Flexclusion: Balancing cache capacity and on-chip bandwidth via flexible exclusion. **ACM SIGARCH Computer Architecture News**, ACM New York, NY, USA, v. 40, n. 3, p. 321–332, 2012.

TERAN, E.; WANG, Z.; JIMÉNEZ, D. A. Perceptron learning for reuse prediction. In: IEEE. **2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)**. [S.l.], 2016. p. 1–12.

WANG, J. et al. Reducing data movement and energy in multilevel cache hierarchies without losing performance: Can you have it all? In: IEEE. **2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)**. [S.l.], 2019. p. 383–394.

WU, C.-J. et al. Ship: Signature-based hit predictor for high performance caching. In: **Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture**. [S.l.: s.n.], 2011. p. 430–441.

WULF, W. A.; MCKEE, S. A. Hitting the memory wall: implications of the obvious. **ACM SIGARCH computer architecture news**, ACM New York, NY, USA, v. 23, n. 1, p. 20–24, 1995.

YUKI, T.; POUCHET, L.-N. **Polybench 4.0**. [S.l.]: Feb, 2015.

APPENDIX A — RESUMO EXPANDIDO

A.1 Introdução

Com a crescente tendência de aceleração dos processadores em velocidade desproporcional às memórias DRAM, o emprego de memórias cache se torna fundamental para mitigar essa disparidade de desempenho que cresce a cada nova geração. As memórias cache funcionam armazenando dados que o processador requisitou recentemente, na premissa de que a tendência é que tal dado tenha maiores chances de ser requisitado novamente em um futuro próximo. Se isso ocorrer, a cache consegue fornecer o dado para o processador, reduzindo drasticamente a latência do acesso.

Entretanto, nem todo acesso à memória deve ser salvo na cache. Dados que não terão reuso em um futuro próximo, ao serem salvos nas caches, provocam apenas um gasto energético desnecessário, além de uma latência extra para verificar a presença dos dados nas caches e preenchê-la. Além disso, ao serem salvos nas caches, esses dados podem substituir dados que têm de fato potencial de reuso, forçando o processador a buscar novamente esses dados da memória principal, mais lenta e gastando novamente a energia para escrita nas caches.

Esta dissertação tem como objetivo propor um mecanismo chamado HBPB, que faz preemptivamente o bypass das caches, isto é, faz o acesso à memória principal sem utilização das caches, para todas as instruções que não são conhecidas por serem amigáveis à cache. O mecanismo mantém uma estrutura que armazena os últimos endereços acessados e qual instrução foi responsável por efetuar aquele acesso. Com isso, consegue aprender rapidamente se alguma instrução acessa dados que possuem reuso, deixando as caches para elas. Para uma benchmark de lista encadeada, o HBPB conquista reduções em Energy-Delay-Squared Product (ED^2P) de até 74% para uma lista encadeada, e 53% para um benchmark do SPEC 2017.

A.2 HBPB

Nessa dissertação, introduzimos o History-Based Preemptive Bypassing (HBPB), um mecanismo que possui o intuito de contornar as caches para todos os acessos de in-

struções que não são sabidamente amigáveis à cache. Para funcionar, o HBPB precisa de três estruturas: A Access History Table (AHT), o Non-Temporal Buffer (NTB) e a CIT (Classified Instructions Table). A AHT funciona como uma segunda cache LLC, porém em vez de armazenar uma linha de cache de 64 Bytes, armazena uma hash de 2 Bytes de um Contador de Programa que identifica uma instrução. O overhead total da estrutura é equivalente a 11.4% do tamanho da LLC para uma LLC de 1 MB. O NTB é o responsável por armazenar os dados dos acessos para os quais foi feito bypass. Ele age como uma cache paralela à própria cache principal. Seu tamanho é o equivalente a L1 normal do processador. Já a CIT é responsável por armazenar as instruções já classificadas pelo mecanismo, com um contador indicando se tal instrução acessa dados com reuso ou não.

O HBPB trabalha com os acessos entre a L1 e a L2. Ao ocorrer um novo miss na L1, o HBPB é questionado se o acesso deve ser feito de forma regular ou com bypass. O mecanismo decide o que fazer baseado no contador relativo à instrução que fez o acesso na CIT. O mecanismo usa esse acesso para treinar a CIT. Se o endereço acessado está presente na AHT, significa que houve reuso daquele endereço em um intervalo que caberia na cache, portanto esse acesso é amigável à cache. Tanto a instrução que trouxe originalmente o dado quanto a que está fazendo o acesso atual têm seu contador incrementado na CIT. Se o endereço não está presente na AHT, é inserido, e uma entrada da tabela é removida, assim como na LLC. A instrução que trouxe o dado removido é treinada na CIT como não temporal. A Figura A.1 apresenta um fluxograma do funcionamento do mecanismo.

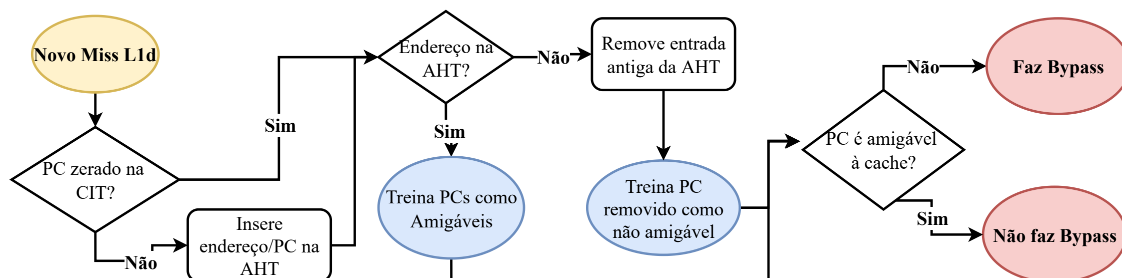

Figure A.1 – Fluxograma de decisão do HBPB.

Quando é feito o bypass para um acesso, a requisição é enviada ao NTB, que pode devolver o dado ao processador ou buscá-lo da memória principal. Se for confirmado que de fato o dado não estava presente nas caches, o dado buscado da memória é enviado ao processador, sem inseri-lo nas caches.

A.3 Resultados

O HBPB foi implementado em um simulador, e testado sob várias circunstâncias com diferentes aplicações e configurações de cache. O overhead do mecanismo foi compensado com uma redução na LLC quando empregado o HBPB. Ao testá-lo em microbenchmarks controlados, o mecanismo comportou-se como esperado, obtendo até 74% de redução do ED^{2}P para uma lista encadeada, e 53% para a aplicação *mcf* do SPEC CPU 2017.

A Figura A.2 apresenta o efeito do HBPB no ED^{2}P das aplicações do SPEC 2017. Foi testado também duas versões adicionais do mecanismo: HBPB-R, que efetua apenas o bypass das requisições, e HBPB-F, que efetua bypass apenas das escritas nas caches. A versão completa é chamada de HBPB-RF. Em média, o ED^{2}P foi reduzido em 16.6% para essas aplicações com HBPB-RF. Para nenhuma das aplicações houve piora no ED^{2}P, enquanto para 6 das 9 aplicações testadas houve melhora superior a 5%.

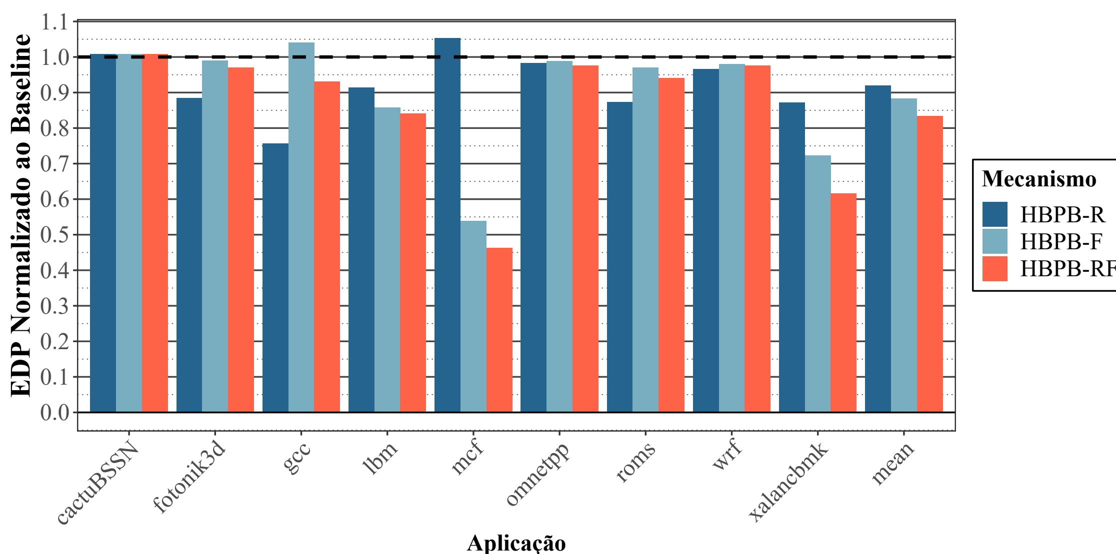

Figure A.2 – Efeito do HBPB no ED^{2}P das aplicações do SPEC CPU 2017 benchmarks.

A.4 Conclusão

Nesta dissertação foram apresentados os benefícios de não utilizar as memórias cache para determinados acessos à memória, como a economia de energia e ciclos de execução. Foi proposto um mecanismo de hardware, HBPB, que decide quando fazer bypass da cache ou não, baseado no histórico dos acessos da instrução.

O mecanismo obteve bons resultados tanto para consumo energético quanto para

desempenho. Considerando-se a métrica ED^2P, o mecanismo obteve reduções de até 53% para uma aplicação do SPEC 2017, conseguindo melhorias significativas para dois terços das aplicações testadas.

APPENDIX B — EXTRA RESULTS

B.1 Microbenchmarks

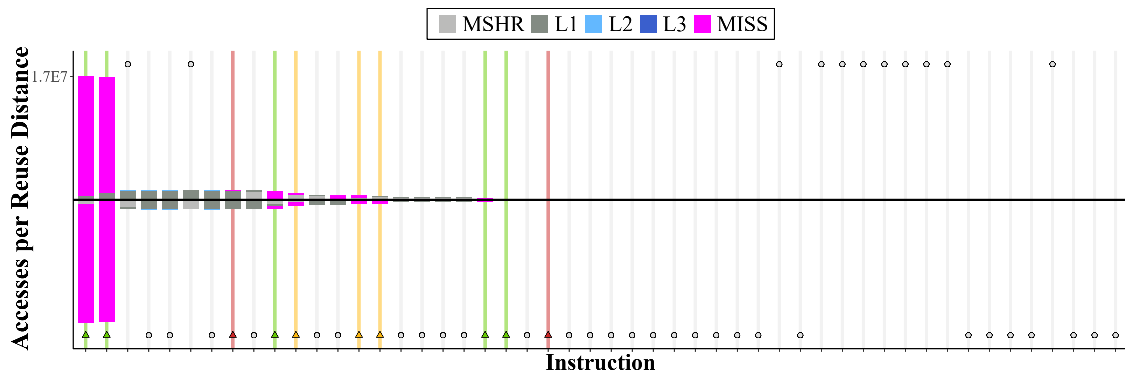

Figure B.1 – Frequency of reuse distances for the 50 most frequently occurring instructions for Pattern Matching.

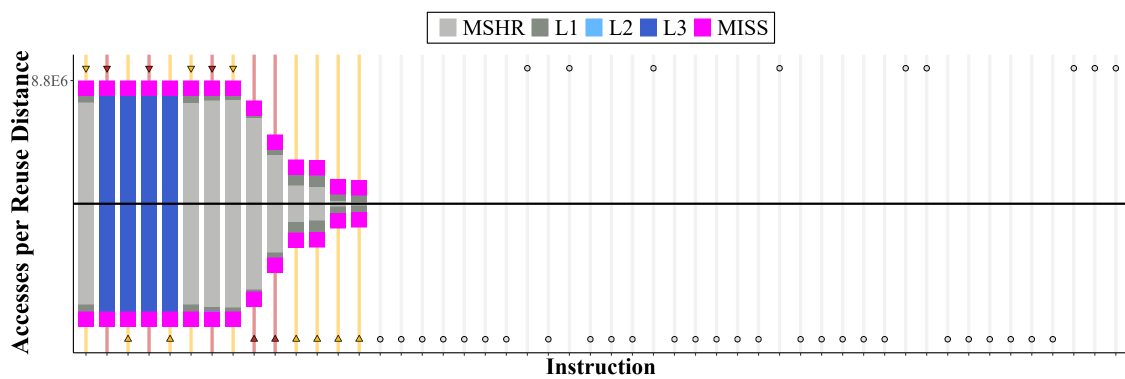

Figure B.2 – Frequency of reuse distances for the 50 most frequently occurring instructions for Deriche.

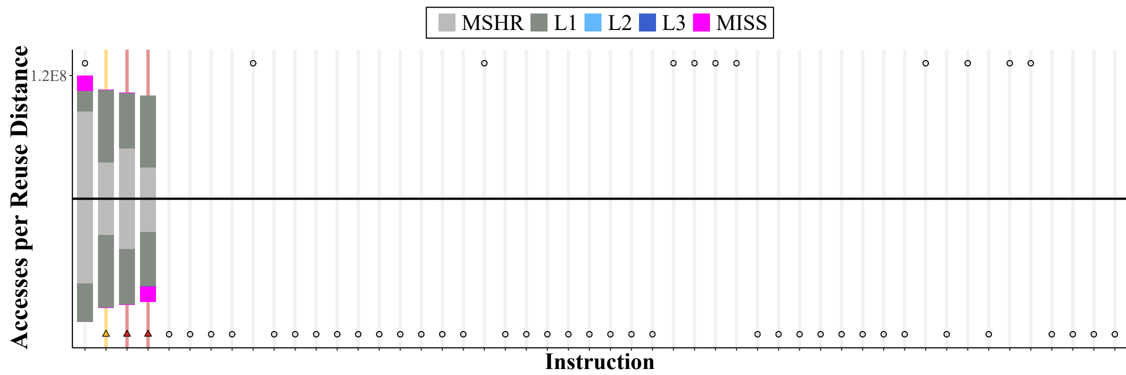

Figure B.3 – Frequency of reuse distances for the 50 most frequently occurring instructions for Floyd-Warshall.

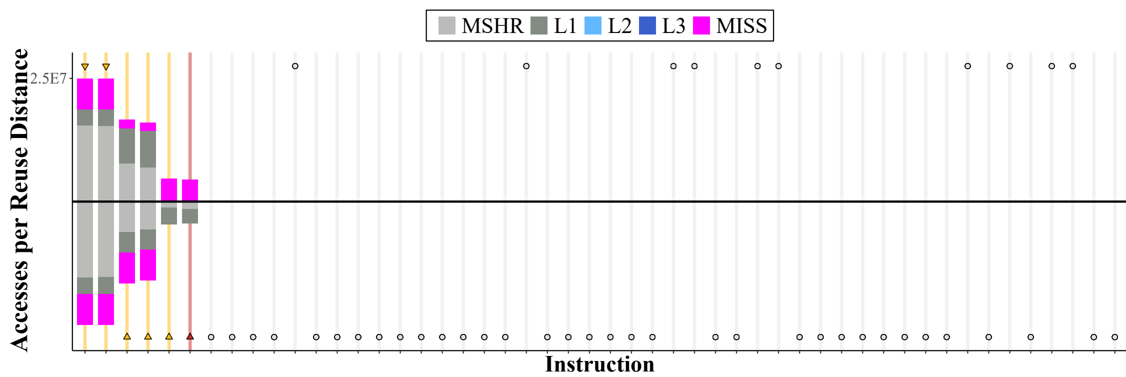

Figure B.4 – Frequency of reuse distances for the 50 most frequently occurring instructions for Jacobi 1D.

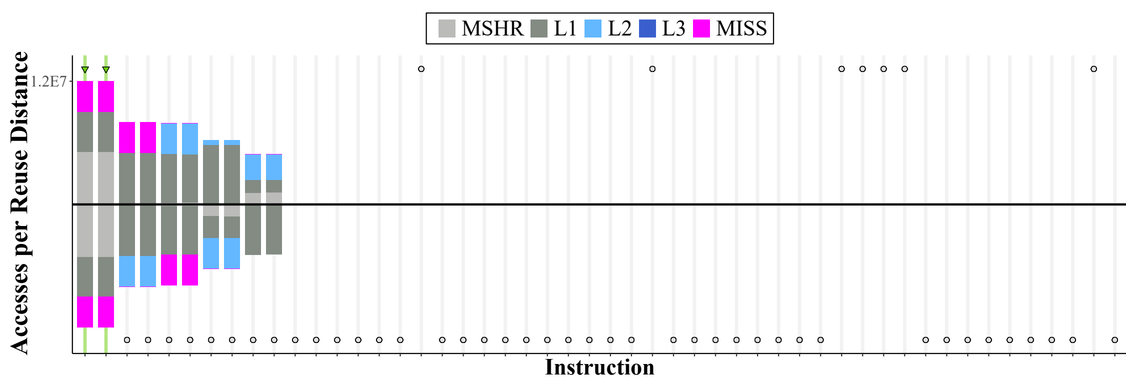

Figure B.5 – Frequency of reuse distances for the 50 most frequently occurring instructions for Jacobi 2D.

B.2 GAP

Table B.1 – Working set size from the GAP benchmarks used.

Application	Working Set
bc	146 MB
bfs	49 MB
cc	132 MB
pr	136 MB
sssp	195 MB

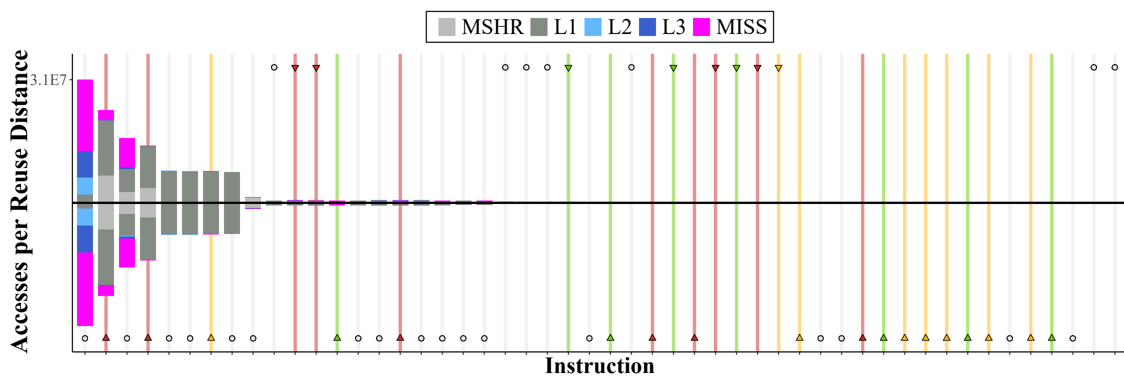

Figure B.6 – Frequency of reuse distances for the 50 most frequently occurring instructions for *bc*.

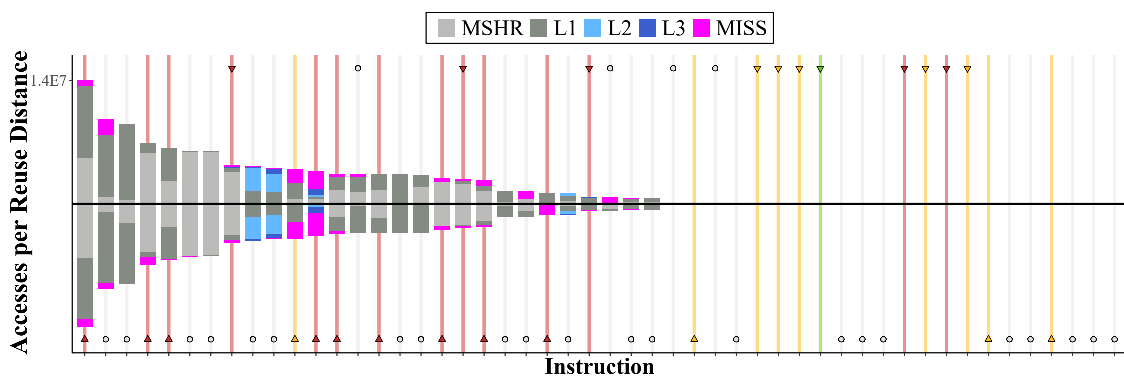

Figure B.7 – Frequency of reuse distances for the 50 most frequently occurring instructions for *bfs*.

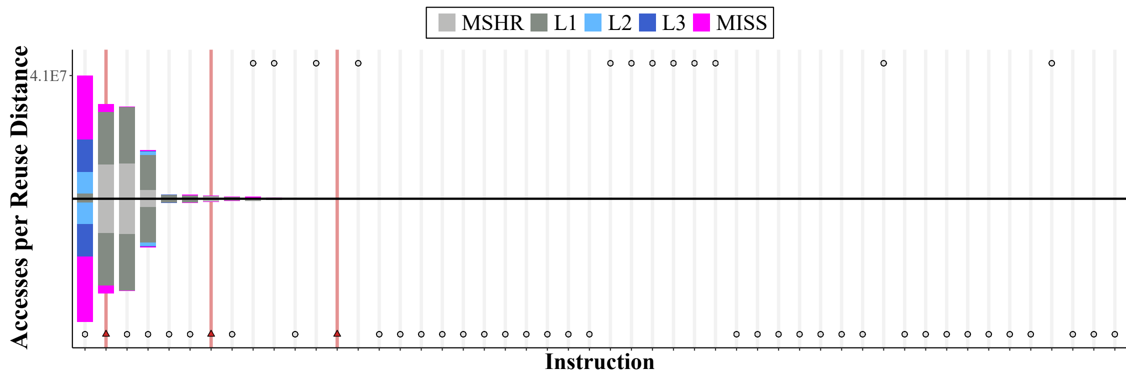

Figure B.8 – Frequency of reuse distances for the 50 most frequently occurring instructions for *cc*.

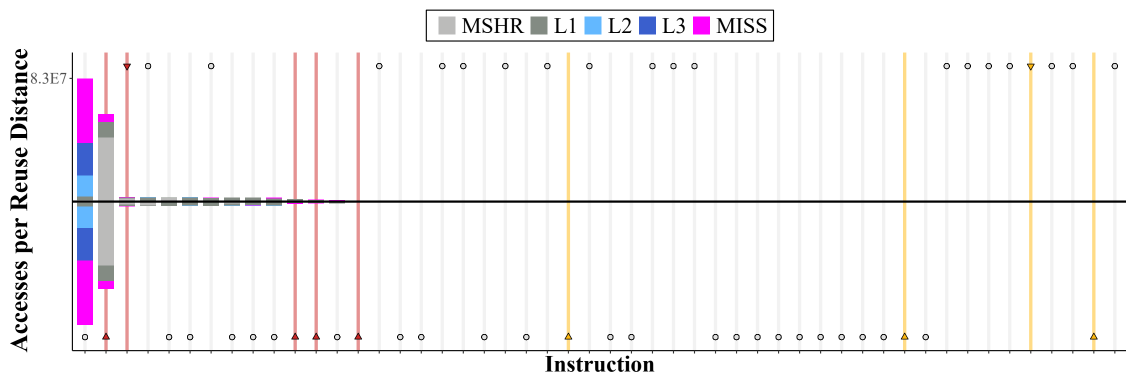

Figure B.9 – Frequency of reuse distances for the 50 most frequently occurring instructions for *pr*.

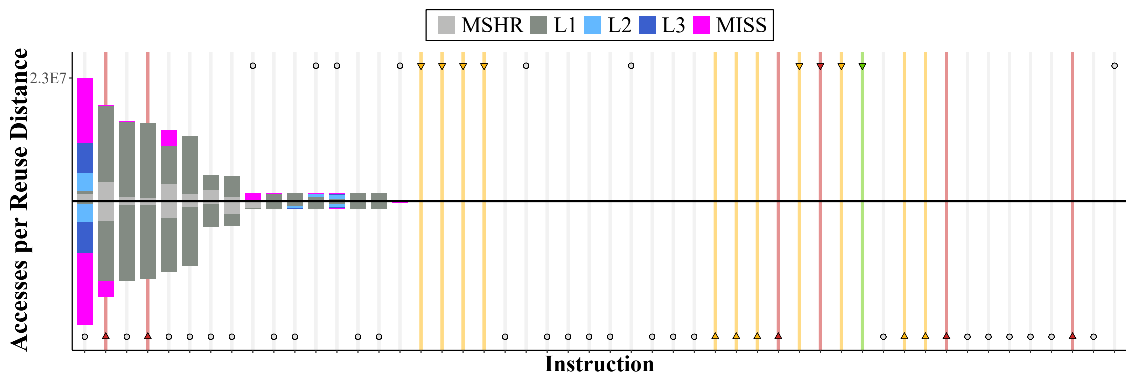

Figure B.10 – Frequency of reuse distances for the 50 most frequently occurring instructions for *sssp*.

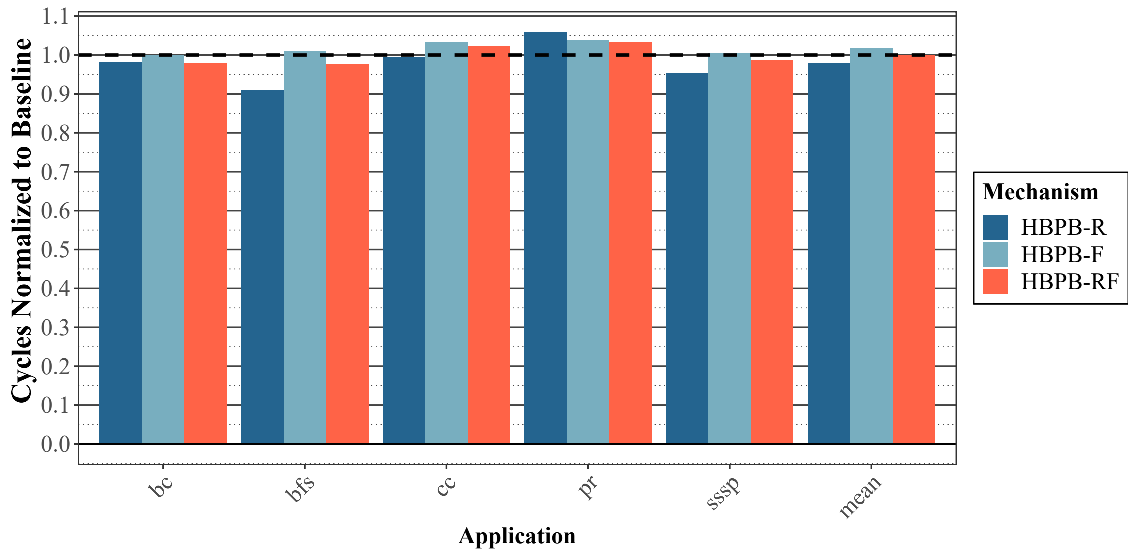

Figure B.11 – Effect of HBPB on the cycles for the GAP benchmarks.

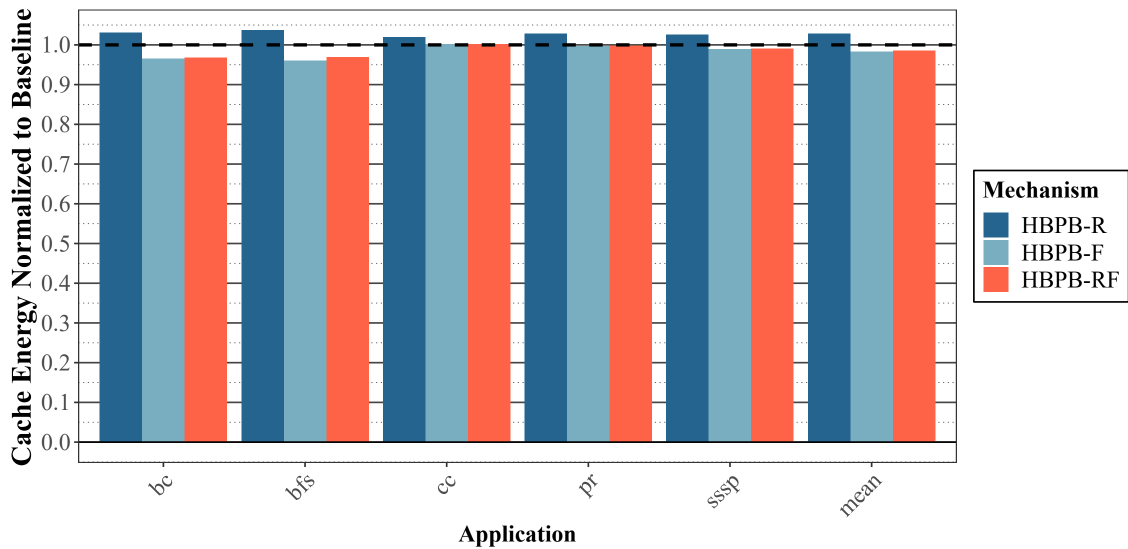

Figure B.12 – Effect of HBPB on the cache energy for the GAP benchmarks.

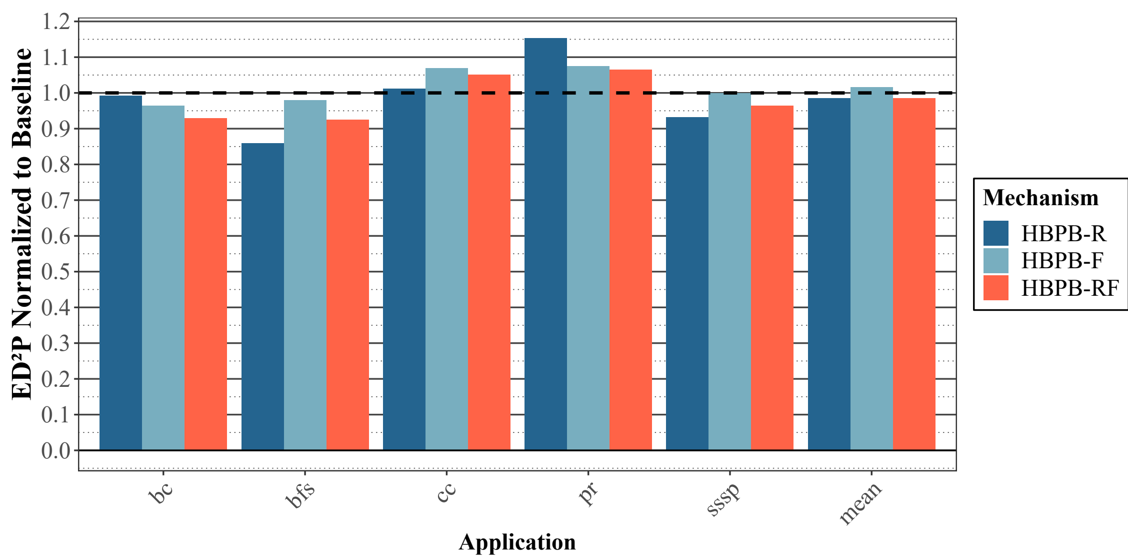

Figure B.13 – Effect of HBPB on the ED^2P for the GAP benchmarks.

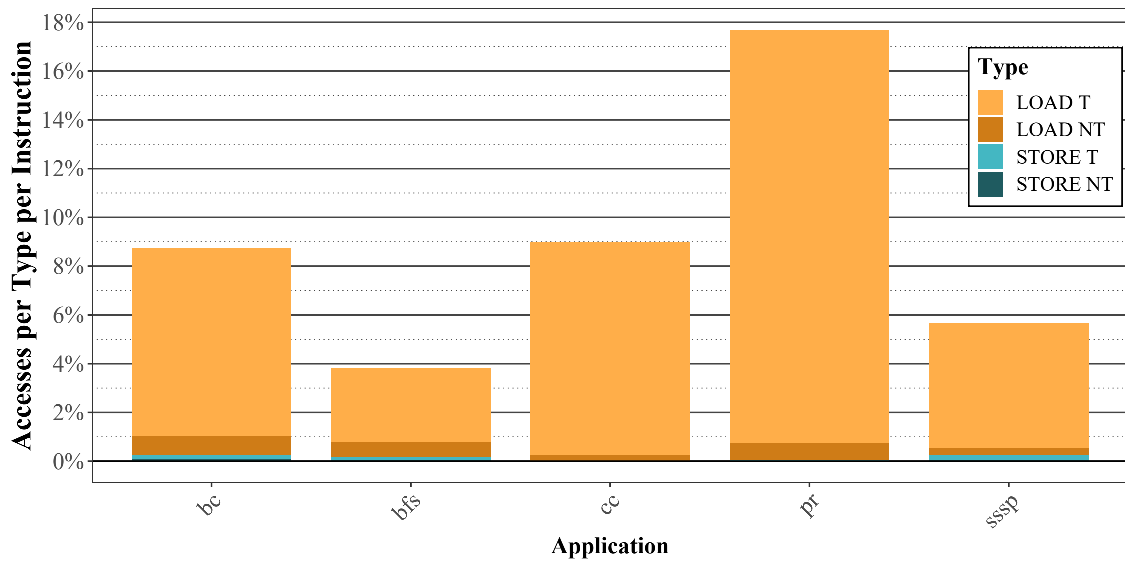

Figure B.14 – Frequency of each type of access for the GAP benchmarks with HBPB.