ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal
Regular Issue, Vol. 14 (2025), e31782
eISSN: 2255-2863
DOI: https://doi.org/10.14201/adcaij.31782

ADDP: The Data Prefetching Protocol for Monitoring Capacity Misses

Swapnita Srivastava, and P. K. Singh

Department of Computer Science and Engineering, Madan Mohan Malaviya University. Deoria Road, Singhariya, Kunraghat, Gorakhpur, Uttar Pradesh 273016, India
✉ swapnitasrivastava@gmail.com, topksingh@gmail.com

KEYWORDS

data prefetcher; instruction prefetcher; multi-core; instruction per cycle; coverage

ABSTRACT

Prefetching is essential to minimizing the number of misses in cache and improving processor performance. Many prefetchers have been proposed, including simple but highly effective stream-based prefetchers and prefetchers that predict complex access patterns based on structures such as history buffers and bit vectors. However, many cache misses still occur in many applications. After analyzing the various techniques in Instruction and Data Prefetcher, several key features were extracted which impact system performance. Data prefetching is an essential technique used in all commercial processors. Data prefetchers aim at hiding the long data access latency. In this paper, we present the design of an Adaptive Delta-based Data Prefetching (ADDP) that employs four different tables organized in a hierarchical manner to address the diversity of access patterns. Firstly, the Entry Table is queue, which tracks recent cache fill. Secondly, the Predict Table which has trigger (Program Counter) PCs as tags. Thirdly, the (Address Difference Table) ADT which has target PCs as tags. Lastly, the Prefetch Table is divided into two parts, i.e., Prefetch Filter and the actual Prefetch Table. The Prefetch Filter table filters unnecessary prefetch accesses and the Prefetch Table is used to track other additional information for each prefetch. The ADDP has been implemented in a multicache-level prefetching system under the 3rd Data Prefetching Championship (DPC-3) framework.

ADDP is an effective solution for data-intensive applications since it shows notable gains in cache hit rates and latency reduction. The simulation results show that ADDP outperforms the top three data prefetchers MLOP, SPP and BINGO by 5.312 %, 13.213 % and 10.549 %, respectively.

1. Introduction

Computer architects know that there is a gap between the speed of the processor and the memory. The gap continues to persist because it is very difficult to create memory that is large enough to store the whole working set of the majority of programs. This is because the size of the memory is inversely proportional to the memory access time as shown in Figure 1. According to Patterson (2006), by implementing a memory hierarchy with register files, caches, main memory, and disks, it is possible to take advantage of the fact that smaller memories are faster and that most applications exhibit spatial and temporal locality. Temporal locality is important, since if memory access patterns were randomly distributed, the majority of references would go to the slower memories, leaving the fast memories with little benefit. Temporal locality enables us to arrange data in such a manner that the overwhelming majority of accesses in many applications occur in the faster memory, albeit smaller, memory. Data arrangement is critical to accomplishing the aim. It is more critical to effectively shift data to faster memories than to move data to slower memories. When the processor needs data that is not in fast memory, then it waits for the data to be transferred from slow memory. If this occurs on a regular basis, functional units may find themselves underused as a result of their inability to access data.

In addition, data that has to be transferred to slower memory may normally be placed in a buffer and scheduled at a later time for write-back. Without cache prefetching, data is transferred explicitly from the lowest memory level to a register Reduced Instruction Set Computer (RISC) or functional unit Complex Instruction Set Computer (CISC) through an instruction. A Cache Prefetching technique speculatively moves data to higher levels in the cache hierarchy in anticipation of instructions that need the data. Prefetching handled by the compiler is referred to as software prefetching. The alternate scenario is hardware prefetching, in which a hardware controller makes prefetching requests based on information it may collect at run-time from various sources (e.g., cache miss addresses and memory reference). Software prefetchers, on the other hand, make use of profiling information and compile-time, whereas hardware prefetchers utilize run-time information. Both prefetchers have significant benefits, and the potential to be quite successful. Cache prefetching lowers the cache miss rate because it removes the need for demand fetching of cache lines in the cache hierarchy (Wu and Martonosi, 2011). Also known as the latency hiding method, it is used to conceal long-latency transfers from lower to higher levels of a memory hierarchy by concealing them behind periods of time while the CPU is executing instructions.

A program experiences capacity misses when its working set grows larger than the cache's storage capacity, which results in frequent data replacements and decreased performance. These mistakes can result in major performance bottlenecks in contemporary systems with large data requirements, such

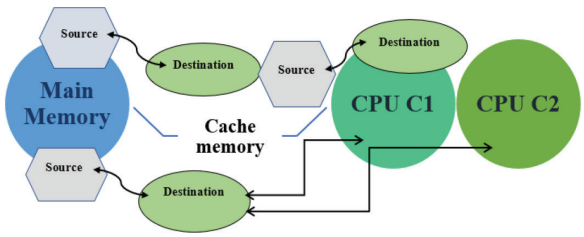

Figure 1. The latency gap

as big data apps and AI-driven workloads. Improving memory access efficiency, lowering latency, and increasing system throughput all depend on monitoring and addressing capacity misses.

This paper is organized as follows. Section 2 provides background on Instruction and Data Prefetching technique. Section 3 explains the taxonomy of prefetching. Section 4 proposes an Adaptive Delta-Based Data Prefetching (ADDP). Section 5 contains the results and analysis of ADDP. Section 6 concludes the research paper with future directions.

2. Background

Instruction prefetching in the instruction cache is an important approach for creating high-performance computers. In the Entangling Instruction Prefetcher (EIP) three main aspects were considered when designing an efficient prefetcher with maximum performance: timeliness, coverage, and accuracy. A prefetcher's effectiveness depends on its timing. Fetching instructions too early may lead to their eviction before use, while fetching them too late may cause them to arrive after their scheduled execution. Prefetching is vital for reducing instruction cache misses, but it must not contaminate the cache or interfere with other hardware operations. The prefetcher decides which instruction should start a prefetch for the next, taking into consideration the prefetch delay. The coverage and connectivity both are adjusted by the prefetcher carefully.

The authors said that prefetch requests are triggered when the I-Shadow cache is not found. The I-Shadow cache solely monitors demand misses. The Footprint Next Line prefetcher and Multiple Miss Ahead Prefetcher (FNL+MMA) is a combination of two prefetchers that make use of two aspects of I-cache utilization (Seznec, 2020). The next line is frequently used by the application in the near future. However, systematic next-line prefetching results in cache pollution and over fetching.

In modern applications, instruction cache misses have become a performance constraint, and numerous prefetchers have been developed to disguise memory access delay. Nakamura et al. (2020) proposed the Distant Jolt Prefetcher (D-JOLT); a prefetcher that uses the call and return history function. It has features linked to history and miss addresses, history duration, and distance to the prefetch target. The D-JOLT prefetcher comprises many prefetchers with varying features, including a long-range prefetcher, a short-range prefetcher, and a fallback prefetcher. Thus, the D-JOLT prefetcher forecasts near future with higher accuracy. Table 1 shows a comparison between cache with top three instruction prefetchers and cache with no instruction prefetcher.

Pakalapati and Panda (2019) proposed BOUQUET; an instruction pointer solution for the DPC-3. The authors employed numerous instruction pointer based prefetchers to cover a wide range of access patterns. Instruction pointers were classified by the classifier at the L1 cache level and communicated to the L2 prefetcher by the classifier. The prefetching system, IPCP, improved single-core performance by 43.75 % and 25 multi-core mixes by 22 %. IPCP requires 16.7KB hardware overhead per core.

BINGO, for spatial data prefetching, uses memory page access patterns to predict future memory references (Bakhshalipour et al., 2019). Existing spatial data prefetchers store correlation records using fragments of information from trigger accesses (i.e., the first access to memory pages). The link observed access patterns to either a brief, high-recurrence event, or a long, low-recurrence event. Prefetchers, on the other hand, either have low accuracy or lose substantial prediction opportunities.

Shakerinava et al. (2019) demonstrated in their study that past ideas for offset prefetching either ignore or sacrifice coverage for timeliness when selecting the prefetch offset. To address the shortcomings of previous offset prefetchers, the authors proposed the Multi-Lookahead Offset Prefetcher (MLOP), the use of a new offset prefetching method that considers both missed coverage and response

Table 1. Comparison of instruction prefetcher

Parameters		No Prefetcher	EIP	FNL+MMA	D-JOLT
L1D	TA	3477670	3477704	3477717	3477651
	H	3476375	3476409	3476422	3476356
	HR	99.96276242	99.96276279	99.96276293	99.96276222
	M	1295	1295	1295	1295
	AML	141.788	141.786	141.788	141.632
L1I	TA	1636806	4819814	1651135	2206856
	H	1636355	4818757	1650190	2205793
	HR	99.97244634	99.97806969	99.94276664	99.95183193
	M	451	1057	945	1063
	AML	81.4412	41.4011	45.072	41.2681
L2C	TA	1903	2509	2397	2515
	H	776	1382	1270	1394
	HR	40.77771939	55.08170586	52.98289529	55.42743539
	M	1127	1127	1127	1121
	AML	171.306	171.306	171.306	170.975
LLC	TA	1127	1127	1127	1121
	H	0	0	0	0
	HR	0	0	0	0
	M	1127	1127	1127	1121
	AML	140.866	140.866	140.866	140.533

TA: Total Access; H: Hit; HR: Hit Rate; M: Miss; AML: Average Miss Latency; L1D: L1 Data Cache; L1I: L1 Instruction Cache; L2C: L2 Cache Memory; LLC: Last Level Cache.

Table 2. Comparison of data prefetcher

Parameters		No Prefetcher	BOUQUET	BINGO	MLOP
L1D	TA	3477670	3792715	3479750	3481890
	H	3476375	3784113	3477781	3476270
	HR	99.96276242	99.77319677	99.94341548	99.83859341
	M	1295	8602	1969	5620
	AML	141.788	60.1403	95.9726	58.0899
L1I	TA	1636806	1636725	1636754	1636717
	H	1636355	1636274	1636303	1636266
	HR	99.97244634	99.97244497	99.97244546	99.97244484
	M	451	451	451	451
	AML	81.4412	59.9956	57.4545	58.643

Table 2. Comparison of data prefetcher (continued)

Parameters		No Prefetcher	BOUQUET	BINGO	MLOP
L2C	TA	1903	18810	4294	9168
	H	776	15738	2515	6821
	HR	40.77771939	83.66826156	58.57009781	74.40008726
	M	1127	3072	1779	2347
	AML	171.306	183.264	168.883	183.909
LLC	TA	1127	3076	1967	2486
	H	0	7	191	143
	HR	0	0.22756827	9.710218607	5.752212389
	M	1127	3069	1776	2343
	AML	140.866	151.58	139.68	151.698

TA: Total Access; H: Hit; HR: Hit Rate; M: Miss; AML: Average Miss Latency; L1D: L1 Data Cache; L1I: L1 Instruction Cache; L2C: L2 Cache Memory; LLC: Last Level Cache.

time when sending out prefetch request messages. Table 2 shows a comparison between the top three data prefetchers and cache with no data prefetcher (Hosseinzadeh et al., 2024).

3. Cache Prefetching Techniques

T-SKID: Prefetching is fundamental to decreasing the number of reserve misses and further developing processor execution. Numerous prefetchers have been proposed, including basic prefetchers, as well as highly viable stream-based prefetchers and prefetchers that foresee complex access designs dependent on constructions, for example, history cradles and touch vectors. Despite these efforts, reserve misses still occur frequently in many applications. We identified numerous applications with simple access patterns that existing prefetchers often fail to predict due to the extended time intervals between their accesses. Furthermore, in these access patterns, even when reserve lines are successfully prefetched, they are often evicted before being accessed due to the long-time intervals between requests. In this paper, we propose a planning slide prefetcher, which freely learns the locations and the entrance timing. We assessed T-SKID with SPEC CPU 2017 benchmarks as per the standard of DPC3 and the assessment results show an over 40 % improvement in execution contrasted with a processor without prefetching (Souza and Freitas, 2024) (Wang et al., 2024).

Memory access inactivity is a significant bottleneck in program execution, and store misses cause huge execution degradation in PC frameworks. Prefetching is one of the most fundamental strategies aimed at decreasing the quantity of reserve misses and further developing processor execution. Thus, numerous prefetchers have been proposed, including straightforward stream/step-based prefetchers and prefetchers that foresee complex access designs utilizing different delta histories or utilizing bit-vectors recording access designs. Notwithstanding, many reserve misses happen in numerous applications (Abella et al., 2005). To resolve this issue, we initially examined store access designs that are hard to anticipate utilizing existing prefetchers. We focus is on L1D store access designs since 1) L1D reserve hit rates altogether affect execution and 2) an L1D prefetcher can make an exact prediction of all the

hits that cannot be retrieved in an L2 reserve or LLC. We found out that numerous applications have basic location sequences that current prefetchers frequently cannot anticipate because of the long-time gap between their access. In this section, the upward hub addresses access time, the flat hub addresses address space, and each plotted point addresses memory access. On the other hand, accessing data within the same cache block in a structured pattern involves step accesses, which can be easily predicted by simple prefetchers. Nonetheless, regardless of whether a store line is effectively prefetched by basic step forecast, whipping access will remove the prefetched line before interest access is given because the limit of the L1D reserve is tiny.

Pangaloos: With a restricted data degree and space/rationale intricacy, pangaloos can recreate an assortment of both basic and complex access designs. This is accomplished by a profoundly proficient portrayal of the Markov chain to give accurate estimates to progress probabilities. What is more, we have added a component to remake delta changes initially jumbled by the mixed-up execution or page advances, such as when streaming information from numerous sources. When joined with an identical counterpart for the L1 reserve, the speedups ascend to 6. In the multi-center assessment, there is a significant exhibition improvement as well. Markov models have been utilized widely in earlier requests for prefetching purposes, by assessing and using address progress probabilities for resulting gets to. Distance prefetching is a speculation of the normal Markov model prefetchers, that utilizes deltas rather than addresses to fabricate broader models. In such cases, the information gained is applied to different addresses, including those already concealed. A devoted execution of a Markov-chain for delta advances would be a coordinated chart, with deltas as states/hubs and probabilities as weighted transitions/curves. A delta is a contrast between two back-to-back advertisement addresses. As we can see from the worked-on model beneath, given an underlying location and a surge of deltas, the location stream can be recreated: 1 4 2 7 8 9 Delta: 3 - 2 5 1. In genuine frameworks, we have page limits, which oblige the compass of deltas. Both the virtual and actual memory spaces are separated into pages. For security and integrity reasons, the page designation is normally not viewed as consecutive. The page substance is recorded by the leftover least significant address pieces and stays unaltered between interpretations. While prefetching, any anticipated addresses that fall outside are discarded. One challenge in distance prefetching is that many pages may be accessed in interleaving patterns, in this manner disturbing the generated delta stream. The delta stream, which would otherwise be fully utilized to update the Markov model, contains invalidated deltas caused by comparing addresses from different pages, such as when accessing data from multiple sources iteratively. Our main idea is to track deltas per page rather than universally, while still building an accurate Markov model for global decisions. The principal commitment of this paper is the presentation of an effective, more-steadfast portrayal of a Markov chain which provides a measure of delta-change likelihood (Li et al., 2024).

We outline the real-world complexity of delta-change Markov chains to gain insights into the associated challenges and advancements. Utilizing a straightforward analysis, we screen all the delta changes utilizing an evaluation system from the state of the art (Srinath et al., 2007). We execute a spurious store prefetcher, where all events of legitimate delta advances (from addresses falling on a similar page) are counted inside a nearness grid. The prefetching shows a representation of the frequencies for the delta advances in a run of 607.cactuBSSN_s 3477B. On the right, we can see the created Markov chain, with the width of the bolts addressing the probability of progress. The amount of the width of all circular segments leaving a hub aggregate to 1 (a few advances with low likelihood are rejected). Another one shows the separate perception of the nearness matrix for all benchmark follows. There are some intriguing perceptions: 1) The lattices are sparse, yet 2) not as sparse to justify supporting only customary steps, i.e., the next-line/consecutive prefetcher). 3) Restrictions on the inclusion of deltas

are not beneficial, it is beneficial to also include negative deltas. 4) Matrices that are excessively inadequate or void show basic examples or negated deltas. Some extra perceptions: 1) The slanting lines are undoubtedly from instances of changes from apparently arbitrary accesses within a page, while an ordinary step is performed. For instance, in a streaming activity with a delta change, any optional get to would yield advances of the structure, where the step is represented and 0 is a new temporary delta. 2) This also explains the vertical and horizontal lines near the axes, as these changes happen before and after certain points. 3) The explanation that there appear to be 'internal' limits that cause the general shape to appear as though a hexagon is because such anomalies would infer two continuous deltas pointing outside the page edges (Butt et al., 2007; Srinath et al., 2007).

One in carrying out a Markov chain in equipment is that a straightforward precise execution would require N*N positions, where N is the number of states, for keeping up with the change probabilities in a contiguous framework. Those affiliated designs (like a completely cooperative or set-acquainted reserve) as a rule utilize a Least Recently Used, or a First-In-First-Out substitution strategy. However, both methods are prone to losing important data due to frequent evictions (thrashing). Additionally, with the data kept by LRU and FIFO, there is no genuine measurement of recurrence/probabilities, which is the thing that Markov-anchors are initially expected to give. It is a set-affiliated reserve, giving delta advances dependent on the current delta It is indexed by the current delta, and the squares in each set represent the most frequently occurring next deltas. This sums in a delta size of 7 pieces, addressing values from - 64 (barring, since it focuses to an alternate page) to +63. Consequently, a substitution technique akin to the Least Frequently Used (LFU) strategy is employed. This rapidly eliminates less significant data while maintaining precise change probabilities. Every block in a set has a counter and a delta value. All counters are cut in half when there is no more space, but the relative disparities between them are retained. Thus, a value is divided by the sum of the values in the set to determine the likelihood of a change. This method guarantees effective prefetching and balanced substitution.

Berti: The Berti prefetcher improves computer memory speed by retrieving data at the optimal moment. Performance deteriorates if data is fetched too late, while cache misses might rise if it is fetched too early. Berti operates in two ways. In regular mode, it uses historical trends to anticipate which block will be fetched next. To increase efficiency while visiting a new (cold) page, it retrieves more data in burst mode. Berti determines the optimal step (delta) for every page rather than following a set pattern like conventional prefetchers. As a result, fetching is more precise and effective. In order to anticipate what data will be required next, it additionally uses instruction pointers (IP) to group comparable memory accesses. Berti examines stored data in several tables to determine when to prefetch. It only retrieves data when it is certain it is required and gains confidence based on previous accesses. It prevents needless fetching and utilizes saved deltas if no match is found. By decreasing cache misses and retrieving data at the appropriate moment, Berti enhances performance by facilitating quicker and more effective memory access.

BARCA: By analyzing the program's control flow and predicting which data blocks will be needed next, BARÇA preloads data in computer memory to speed up processing. Rather than using traditional branch predictors, BARÇA creates a control-flow graph, in which each node represents a fixed-size memory region and edges indicate how frequently control moves between these regions. BARÇA searches this graph up to a certain depth and stops searching when the probability of reaching a block drop below a threshold. The algorithm does not focus on specific branch instructions, but rather tracks overall control flow, and return instructions are given special handling because they have multiple potential targets. By analyzing the program's control flow and predicting which data blocks will be needed next, BARÇA preloads data in computer memory to speed up processing. Rather than using traditional branch predictors,

BARÇA creates a control-flow graph, in which each node represents a fixed-size memory region and edges indicate how frequently control moves between these regions. BARÇA searches this graph up to a certain depth and stops searching when the probability of reaching a block drop below a threshold. The algorithm does not focus on specific branch instructions, but rather tracks overall control flow, and return instructions are given special handling because they have multiple potential targets.

TAP: TAP is a method that helps computer programs run faster by anticipating and retrieving the instructions that will be needed next. It accomplishes this by monitoring trends in the flow of instructions and examining previous misses or failures to locate data in the cache. The idea of transitive closure aids in figuring out whether a program's flow has a path connecting two points. To determine which instructions are likely to follow which in weighted programs, TAP gives these routes probabilities (Luo et al., 2017). Although TAP operates in tandem with next-line prefetching, it does not retrieve the subsequent line as that duty is already taken care of independently. To increase accuracy, it tracks missed commands and their relationships using a table instead. Managing metadata is a significant difficulty because TAP necessitates the storage of a substantial amount of history. In contrast to other prefetching techniques, it effectively compresses this data, lowering memory usage. TAP performs very well in server workloads and increases performance by 23 % when compared to no prefetching. It makes use of a "shadow cache" to track the usefulness of prefetches and makes use of the lengthy lifespan of instructions in the cache. TAP adapts to enhance future predictions if a prefetch proves ineffective (Duong et al., 2012).

EIP: EIP is a method that helps minimize instruction cache misses while preventing space wastage from needless prefetching. It ensures that necessary instructions arrive on time by monitoring the time it takes to acquire missing data and connecting those misses to instructions that should initiate prefetching. EIP keeps track of memory line connections. It anticipates and retrieves another memory line (target) that will probably be required shortly after one (source) is accessed (Liu et al., 2020; Qureshi and Patt, 2006). This increases efficiency by guaranteeing the objective is available precisely when needed. EIP determines the delay for each cache miss by comparing the request time and the actual data arrival time. This data is utilized to improve prefetching choices in the future and is kept in a table. A history buffer built into the system keeps track of the initial instruction that caused a memory block to load. Additionally, it keeps a timing table that records cache misses and prefetch events. EIP uses these records to increase the precision of its future forecasts (Srivastava and Sharma, 2019). EIP modifies its prediction confidence when a new cache entry is filled and determines if it was previously predicted. Confidence rises when the prefetch was beneficial and falls when it was not. Over time, this aids in improving the prefetching procedure. Additionally, EIP keeps track of source-target memory pairs and their confidence ratings in a trapped table. Prefetching efficiency is maintained by the trapped table, which also optimizes cache performance by minimizing needless memory accesses (Johnson and Shasha, 1994)

MLOP: By anticipating and loading memory blocks that are a predetermined distance (k blocks) from the one being accessed, offset prefetching operates. This lessens the delays brought on by cache misses. By taking into account several potential offsets rather than depending just on one, MLOP enhances conventional offset prefetching (Akram and Sawalha, 2019) (Aleem et al., 2016). After evaluating various offsets and giving them scores, it chooses the best one to prefetch. This improves efficiency and precision. In contrast to earlier approaches that simply employed one optimal offset, MLOP selects the most promising offset at each stage by looking ahead several steps into the future. More cache misses are covered as a result. MLOP enhances system performance by 4 % over the best prefetcher currently in use and by 30 % over no prefetching. It is made to function well with contemporary systems, cutting down on cache miss delays while keeping overhead minimal (Bao et al., 2017; Borgström et al., 2017).

4. Prefetching

Data prefetching is now acknowledged as an effective method to conceal the data access latency incurred through cache misses and to cover up the space of performance between the memory and the processor (Long et al., 2023; Byna et al., 2008). The data prefetching technique, as illustrated in Figure 2, focuses on moving data closer to the CPU in advance of its actual use, with initial support provided by hardware or software. To leverage the advantages of multi-core architectures, various prefetching techniques have been developed over time to reduce data access latency. A taxonomy of fundamental five concerns (what, when, where, who and how to prefetch), which is required to design the prefetching method, is provided in this work. The key operation of prefetching is to anticipate the probable accesses accurately and to shift the anticipated data from its origin to the final destination timely. There are various types of highly recommended forecasting techniques

- Pursuing updated history of data accesses for pattern recognition,
- Operating compiler and user-suggested indications,
- Anticipating elements of past operation performed by applications and loops,
- Operating a helper-thread strategy to assist real-time application execution in anticipating cache misses.

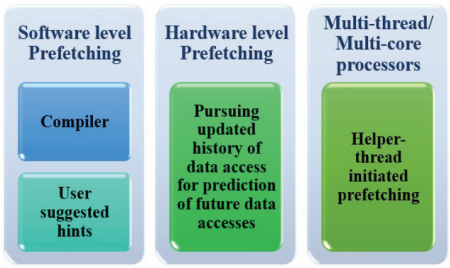

Figure 2. Prefetching Levels

The data prefetching technique is a challenging task to execute with precise timing. The system anticipates future data requests, begins fetching data, and transfers data closer to the processor before it is required, reducing the amount of time the CPU is interrupted when a cache miss event occurs.

4.1. Taxonomy of Prefetching

This taxonomy of prefetching consists of five fundamental concerns as shown in Figure 3.

What to prefetch: The initial stage of the prefetching technique involves identifying what data to prefetch. In a hardware-controlled prefetching technique, the most popular and recommended methods are- offline analysis, run-ahead execution based and history based. In a software-controlled prefetching technique, the most used methods are: compiler-based, application function calls based, post-execution analysis, etc. (Bera et al., 2021). On the other hand, if a hybrid technique (hardware/software controlled), history-based and pre-execution-based methods are more recommended. As less exactness can cause cache pollution, predicting future data accesses precisely may be difficult.

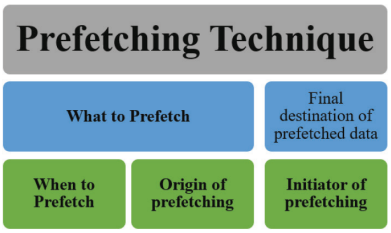

Figure 3. Taxonomy of prefetching

When to prefetch: The exact time to execute prefetching data must occur and reach its final place before a new cache miss occurs. This whole process can be divided into several modules. Firstly, event-based prefetching includes methods encompassed in each memory access on a cache miss, on the branch, and on accessing prefetched data. Secondly, lookahead program counter (LA-PC) modifies interval by operating pseudo counter, including some cycles to the pathway of a real PC. Thirdly, in the software-controlled method, a compiler is required to decide when to prefetch data productively for applications. The last one, the prediction-based method, follows a server-based push notification prefetching strategy to examine when to prefetch.

Origin of prefetching: The origin/source consists of three levels, namely, cache memory, main memory, and storage (Pugsley et al., 2014). The prerequisite to understanding the design of a prefetching technique is to identify where the current prefetch copy of data is located. Single-core processors usually show that cache or main memory is the initial source of prefetching. Contrarily, multi-core processors have local cache memories assigned to each private core within the memory hierarchy, and multiple cores share cache memories. Since the entire process focuses on the cache and memory prefetching level, following various copies of prefetched data at a local level may lead to consistency issues (Kim et al., 2016).

The final destination of prefetched data: This stage is one of the most imperative layers of the prefetching technique. The destination of prefetched data must be closer to the CPU than the origin of prefetching. It boosts the gains of performance. Prefetching can be done within a local cache memory, or a shared cache memory of multiple processors, or a separated cache which also includes a distinct private prefetch and a shared prefetch accordingly.

Initiator of prefetching: This stage is represented by "who." The instructions for prefetching can be commenced by two types of processors that either require data or render such services. Initiators are categorized into two kinds: pull-based (client-initiated) and push-based (Young et al., 2017). The pull-based technique is widely used in 1-core CPU. In multi-threaded processors, prefetching makes sure to decline the entrance of data from the calculation. In helper thread, prefetching typically brings data closer to the CPU rather than retrieving it from the main memory. Push-based prefetching enables data to be pushed into a cache shared for computational purposes on the processor side. In comparison to others, memory-side prefetching is a fairly new method that involves pushing anticipated data from the main memory closer to the processor. On the other hand, without any delay for the requests from the processor, the server-based method pushes data to its destination from its origin. While pull-based prefetching leads to an issue of complexity, push-based prefetching transports complexity to the outer side of the processor. The push-based is faster than the pull-based method. Push-based also follows the usage strategy of dedicated servers. Thus, the push-based strategy is more efficient and effective than the pull-based strategy (Yovita et al., 2022).

5. Proposed Adaptive Delta-Based Data Prefetching (ADDP)

Current prefetching methods, such BINGO, MLOP, and SPP, are good at reducing gate access latency, but since they depend on predefined off-set prediction models, they frequently cannot handle capacity misses. As an example, MLOP concentrates on offset prefetching but has trouble with dynamically changing access patterns, which in certain cases results in significant overprediction rates. In order to dramatically reduce capacity misses, ADDP sets itself apart by using a hierarchical structure of four tables to adaptively forecast and filter prefetches based on dynamic access patterns. In contrast to other protocols, ADDP uses temporal correlation of trigger PCs and delta-based address prediction to efficiently handle capacity-driven evictions.

The proposed ADDP is explained in Figure 4. The ADDP comprises the following four modules.

1. Entry table is a table that records tags, recent access addresses and their PCs. The entry table comprises features such as recency, frequency, offset and valid bit.
2. When a new page is accessed (i.e., trigger access), it creates an entry in its Predict Table. The Predict Table keeps track of the trigger program counters (PCs) and the target PCs that correlate to them. The table anticipates the memory locations that will probably be visited next, after a trigger PC is accessed, allowing pertinent data to be prefetched in a timely manner.
3. The Address Difference Table (ADT) determines the delta, or difference, between consecutive memory addresses that the same computer may access. Accurate prefetching is ensured even for non-sequential data by using this delta to forecast future addresses based on current patterns.
4. The Prefetch Table is divided into two parts, i.e., Prefetch Filter and the actual Prefetch Table. Prefetch Filter duplicates prefetch bits and cache tags in the cache and simulates cache behavior to filter unnecessary prefetch accesses. It contains features such as tags and prefetch bits. Prefetch addresses and their associated PCs are kept in the Prefetch Table until a prefetch is filled. When a prefetch is filled, the memory system provides a fill address but does not notify the trigger PC or

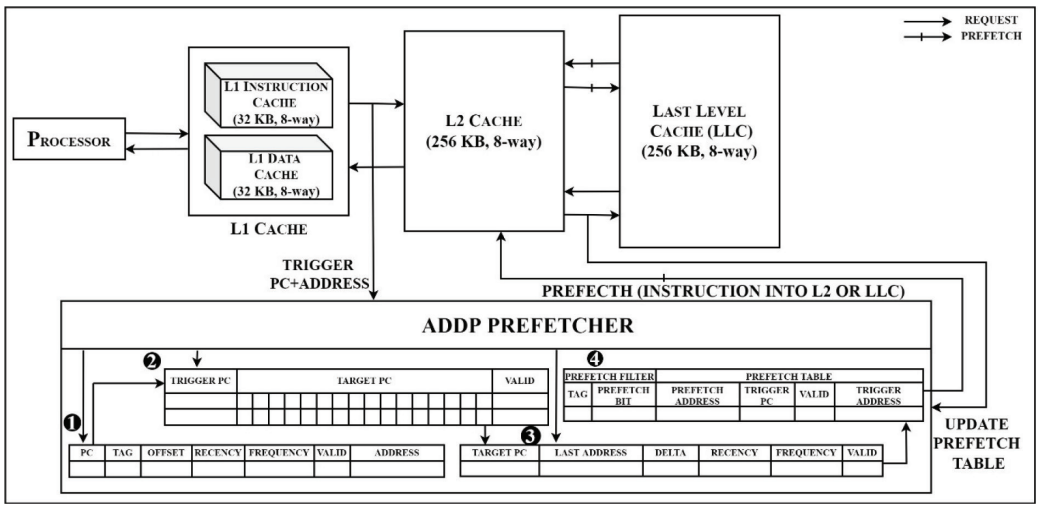

Figure 4. Block diagram of ADDP

the memory address of the triggering access. The Prefetch Table tracks such additional information for each prefetch.

5.1. Methodology

Prefetching is fundamental to decreasing the number of reserve misses and further developing processor execution. ADDP decouples address prediction and trigger timing prediction. ADDP exploits time correlation between PCs. ADDP can timely issue prefetch with temporally distant accesses. ADDP can prefetch lines evicted due to capacity miss. ADDP is divided into two phases: Training and Testing. Figure 5 shows the working of ADDP.

ADDP consists of several tables. The Entry Table is a queue which tracks recent cache fill. Its entries are PCs, tags, offset, recency, frequency, valid bits, and addresses. The Predict Table uses trigger PCs as tags. The entries are target PCs. In this paper, the number of target PCs is dependent on the n-way set associative. Such as the number of target PCs is n for each trigger PC if cache is of the n-way set associative. The Address Difference Table (ADT) has PCs as tags. The entries are delta and a last address to calculate the delta. The following two phases are discussed below:

Testing Phase: The ADT and Predict Table are used for prediction. First, an entry from the Entry Table is searched from the Predict Table with a trigger PC as the access PC. In case of HIT, ADDP gets the target PC of the entered trigger PC and there can be an n number of target PCs depending on the cache associativity. Then, each entry is searched from the ADT with each target PC. In the case of HIT, the prefetch address is calculated using delta and last access address of the trigger PC. Then, the Prefetch Table removes the unnecessary prefetches whose addresses are already in cache.

Training Phase: At cache fill, the PC of the access is stored in the entry table. At a certain moment, the Predict Table records a current access PC as target PC and all the entries of the recent request table

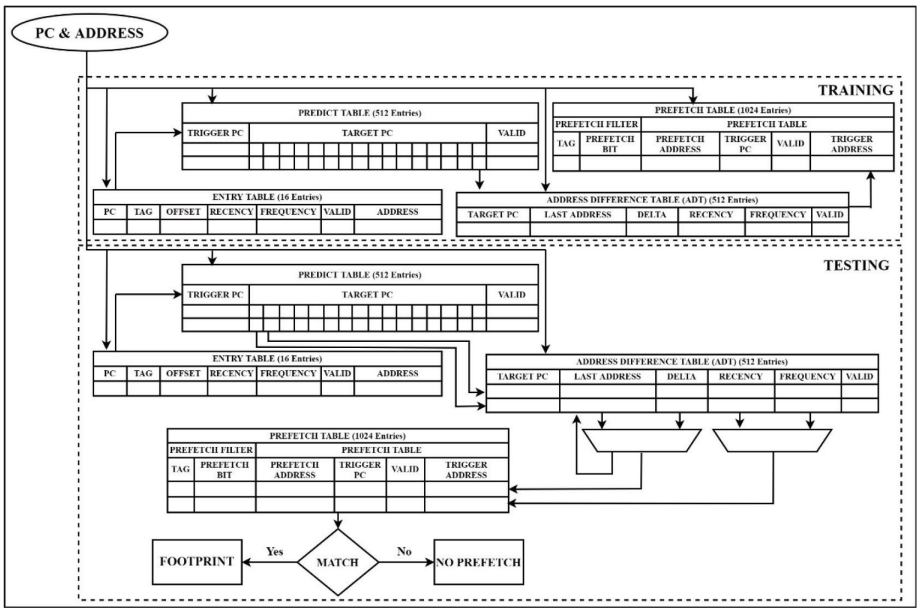

Figure 5. The operation of ADDP

as the trigger PC. ADDP repeats this operation for each access to learn the set of trigger PC and target PC. ADT learns the PCs not only for misses but also for hits as target PC. An entry is searched from the ADT with a current PC as target PC. The ADT calculates a delta using the last recorded address of an entry and the current access address. It then updates the entry with the calculated delta and access address. The ADT repeats this process to learn address prediction.

5.2. Prefetching in the framework of ChampSim

ChampSim (ChampSim, n.d.), a trace-based microarchitecture simulator is used to test and estimate speedup on a single core and multi core processor. ChampSim simulates a processor with a branch predictor, L1 Instruction Cache, L1 Data Cache, L2 Cache, Last Level Cache (LLC) prefetcher, LLC replacement strategy, and the number of CPU cores. When a simulation is finished the output file is checked for results. The 1st Instruction Prefetching Championship and 3rd Data Prefetching Championship models were studied in this paper. This competition compared several instruction prefetching techniques with a 128 KB storage allowance. The paper also includes a proposed Adaptive Delta-based Data Prefetching (ADDP) technique to increase the L1 cache Instruction Per Cycle (IPC) performance. The ADDP is then compared with the policies used in the competition. The competition's policies were updated to operate with the latest ChampSim version and the L1 Instruction cache.

The Standard Performance Evaluation Corporation (SPEC) CPU2017 (SPEC CPU, n.d.) suite are widely used in both industry and academia. This suite covers various aspects of system design, including CPU, memory systems and compiler optimizations. SPEC CPU2017 is made of benchmarks representing real life applications rather than synthetic kernels or benchmarks. CPU2017 has 43 benchmarks, classified in 4 suites. In which first two suites have 20 integer benchmarks, and the last two suites have 23 floating point benchmarks. In this paper authors have randomly selected 20 benchmarks from both integer and floating benchmark to compare the performance of ADDP. All these benchmarks are single-threaded written in C, C++, and Fortran. In a simulation infrastructure with multicores, the multi-programmed workloads are formed by running one individual instance of one CPU2017 benchmark on one core.

6. Results and Analysis

The ADDP is implemented in the ChampSim simulator, an improved model of the 3rd Data Prefetching Championship simulation architecture (DPC-3) (Third Data Prefetching Championship, n.d.). The cache parameters can be found in Table 3.

The results obtained from ADDP are compared with the three top performing data prefetchers: Signature Path Prefetcher (SPP), BINGO and MLOP. The ADDP is evaluated on cache with 1 core and 4 core configurations with all SPEC CPU2017 traces with an LLC MPKI of at least 1.0. All simulation results are warmed up with 50 million instructions and simulated for an additional 100 million instructions.

The storage computation of ADDP is shown in Table 4. The entry table contains: Tag of 6 bits, Offset of 6 bits, Recency of 8 bits, Frequency of 8 bits, Valid of 1 bit, PC of 11 bits and Address of 48 bits. The predict table contains: Trigger PC of 11 bits, Target PC of 16 bits and Valid of 1 bit. The ADT table contains: Trigger PC of 11 bits, Last address of 48 bits, Delta of 7 bits, Valid of 1 bit, Recency of 8 bits, Frequency of 8 bits. The Prefetch table is categorized into two parts. The first part is the Prefetch filter, and the second one is the octal prefetch table. The Prefetch table contains: Tag of 6 bits and 1 Prefetch bit. The Prefetch table consists of 48 bits of Prefetch address, 6 bits of trigger address, 1 valid bit and 48 bits of Valid Trigger address.

Table 3. Configuration of cache

Cache Structure	**Configuration**
L1 Instruction Cache (L1I)	32 KB, 4-cycle Latency, 8-way
L1 Data Cache (L1D)	32 KB, 4-cycle Latency, 8-way
L2 Cache (L2C)	256 KB, 8-cycle Latency, 8-way
Last Level Cache (LLC)	2 MB, 12-cycle Latency, 16-way

Table 4. Computation of storage overhead

Table	**Σ (Entry Size × Entry)**	**Total**
Entry Table	16 × 6 + 16 × 6 + 16 × 8 + 16 × 8 + 16 × 1 + 16 × 11 + 16 × 48	1408 bits
Predict Table	512 × 11 + 512 × 16 + 512 × 1	14336 bits
ADT	512 × 11 + 512 × 48 + 512 × 7 +512 × 1 + 512 × 8 + 512 × 8	42496 bits
Prefetch Filter	1024 × 6 + 1024 × 1	7168 bits
Prefetch Table	1024 × 48 + 1024 × 6 + 1024 × 1 +1024 × 48	105472 bits
1408 + 14336 + 42496 + 7168 + 105472 = 170880 bits (**20.858 KB**)		

ADDP was evaluated using instruction per cycle (IPC), average miss latency (AML), and hit/miss rates. These measurements offer a comprehensive view of how well the protocol mitigates capacity misses. For example, improved IPC shows faster throughput for workloads, while decreased AML suggests fewer cache evictions.

Figure 6 shows the Instruction Per Cycle of ADDP over three other prefetchers for single core. ADDP improves the IPC speedup of benchmarks such as 605.mcf_s-994B, 623.xalancbmk_s-10B, 654.roms_s-1007B, 627.cam4_s-573B, 621.wrf_s-8065B, 621.wrf_s-6673B, 628.pop2_s-17B, 607.cactuBSSN_s-2421B and 603.bwaves_s-2931B.

Figure 7 shows the Instruction Per Cycle of ADDP over three other prefetchers for multi core. ADDP improves the IPC speedup of benchmarks such as 619.lbm_s-4268B_4T, 623.xalancbmk_s-10B_4T and 628.pop2_s-17B_4T.

ADDP improves the coverage of benchmark 623.xalancbmk_s-10B, 620.omnetpp_s-141B, 605.mcf_s-994B, 620.omnetpp_s-874B, 623.xalancbmk_s-592B, 627.cam4_s-573B, 649.fotonik3d_s-10881B, 619.lbm_s-4268B, 602.gcc_s-734B, 619.lbm_s-2677B, 621.wrf_s-8065B, 621.wrf_s-6673B, 619.lbm_s-3766B, 628.pop2_s-17B, 654.roms_s-1007B, 607.cactuBSSN_s-2421B, 602.gcc_s-1850B, 603.bwaves_s-2931B by 6.83 %, 23.78 %, 23.92 %, 24.47 %, 27.02 %, 58.96 %, 59.22 %, 90.35 %, 91.64 %, 91.69 %, 92.52 %, 92.92 %, 92.94 %, 94.59 %, 96.79 %, 97.39 %, 97.64 % and 98.26 %, respectively. Figure 8 shows the coverage of different benchmarks for single core. Coverage is defined as below.

$$C = \frac{M_P}{M_T} \tag{1}$$

Where C is Coverage M_P is Misses evicted by prefetching and M_T is Total Cache Misses.

In order to overcome the shortcomings of current prefetching methods, ADDP introduces a revolutionary adaptive delta-based data prefetching protocol that successfully mitigates capacity misses. On comparison to high-performing prefetchers like MLOP and BINGO, ADDP showed notable gains in IPC, miss rate reduction, and AML through simulation using SPEC CPU2017 benchmarks. These findings highlight the potential of ADDP to improve system performance in situations with one or more cores.

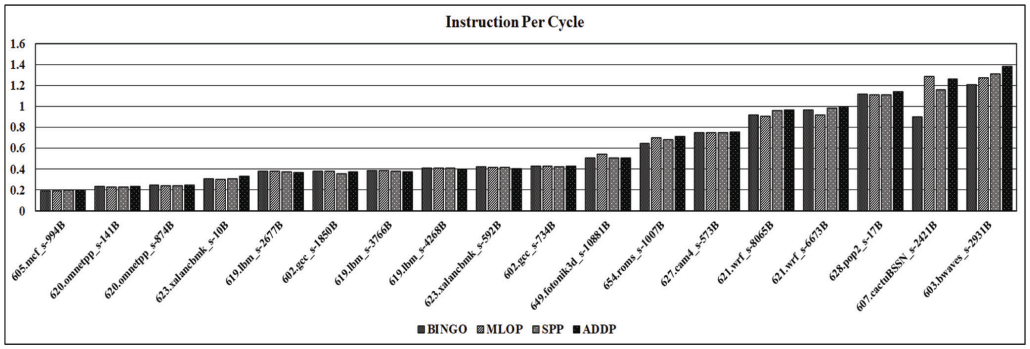

Figure 6. Instruction Per Cycle for Single Core

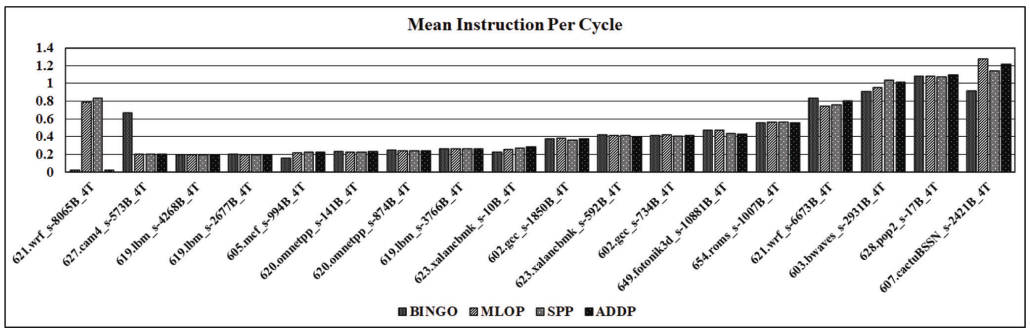

Figure 7. Instruction Per Cycle for Four Core

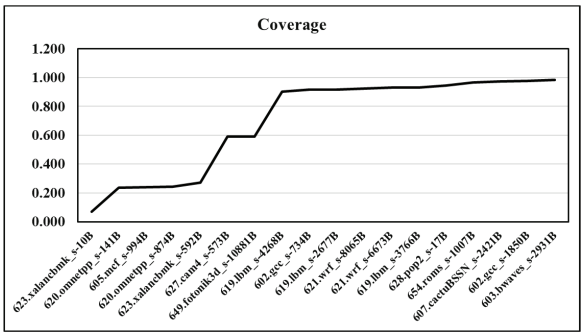

Figure 8. Coverage

7. Conclusions and Future Work

Prefetching strategy performance is determined by a variety of factors. With the introduction of multi-threaded and multi-core processors, new problems and concerns in prefetching data are addressed. A taxonomy of the 5 key challenges that must be considered when constructing the prefetching technique is presented in the paper. A prefetching technique for multicore processor must be adaptable in order to

be successful; it must be able to choose from among many processes, anticipating future data accesses. Prefetching techniques should use history-based prediction algorithms to anticipate future accesses when the access pattern is simple to develop. The paper demonstrates the recent instruction and data prefetching techniques used in caches. The trade-offs used in prefetching and several important features of the prefetching were also identified. The classification helped to design an Adaptive Delta-Based Data Prefetching (ADDP) that employs four different tables organized in a hierarchical manner to address the diversity of access patterns. ADDP learns address patterns and timing to trigger prefetch accesses independently. The ADDP is implemented in the ChampSim simulator for single and multi-core processors using SPEC CPU 2017 benchmarks. In simulations, ADDP outperforms MLOP by 5.312 %, SPP by 13.213 % and BINGO by 10.549 %, respectively. The different features such as confidence value can be used in future versions of the ADDP since there are still significant numbers of unnecessary prefetches in the Prefetch Table.

Future research will concentrate on expanding ADDP's functionality to accommodate new memory technologies including non-volatile memory (NVM) and high-bandwidth memory (HBM). Furthermore, using AI-based workload predictors might boost ADDP's capacity to adjust to various and intricate access patterns, guaranteeing long-term performance gains in the systems of the future.

Data Availability Statement

Datasets derived from public resources and made available with the article https://www.spec.org/cpu2017/.

References

Abella, J., González, A., Vera, X., & O'Boyle, M. F. (2005). IATAC: A smart predictor to turn off L2 cache lines. *ACM Transactions on Architecture and Code* Optimization *(TACO)*, *2*(1), 55-77. https://doi.org/10.1145/1061277.1061282

Akram, A., & Sawalha, L. (2019). A survey of computer architecture simulation techniques and tools. *IEEE Access*, *7*, 78120-78145. https://doi.org/10.1109/ACCESS.2019.2921799

Aleem, M., Islam, M. A., & Iqbal, M. A. (2016). A comparative study of heterogeneous processor simulators. *International Journal of Computer Applications*, *148*(12). https://doi.org/10.5120/ijca2016911332

Bakhshalipour, M., Shakerinava, M., Lotfi-Kamran, P., & Sarbazi-Azad, H. (2019). Accurately and maximally prefetching spatial data access patterns with bingo. *The Third Data Prefetching Championship*. https://doi.org/10.1145/3309200.3309204

Bao, W., Krishnamoorthy, S., Pouchet, L. N., & Sadayappan, P. (2017). Analytical modeling of cache behavior for affine programs. *Proceedings of the ACM on Programming Languages*, 2(POPL), 1-26. https://doi.org/10.1145/3158103

Bera, R., Kanellopoulos, K., Nori, A., Shahroodi, T., Subramoney, S., & Mutlu, O. (2021, October). Pythia: A customizable hardware prefetching framework using online reinforcement learning. En *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (pp. 1121-1137). https://doi.org/10.1145/3466752.3480071

Borgström, G., Sembrant, A., & Black-Schaffer, D. (2017, January). Adaptive cache warming for faster simulations. En *Proceedings of the 9th Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools* (pp. 1-7). https://doi.org/10.1145/3156403.3156404

Butt, A. R., Gniady, C., & Hu, Y. C. (2007). The performance impact of kernel prefetching on buffer cache replacement algorithms. *IEEE Transactions on Computers*, *56*(7), 889-908. https://doi.org/10.1109/TC.2007.1048

Byna, S., Chen, Y., & Sun, X. H. (2008, May). A taxonomy of data prefetching mechanisms. En *2008 International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN 2008)* (pp. 19-24). IEEE. https://doi.org/10.1109/I-SPAN.2008.4549991

Duong, N., Zhao, D., Kim, T., Cammarota, R., Valero, M., & Veidenbaum, A. V. (2012, December). Improving cache management policies using dynamic reuse distances. En *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (pp. 389-400). IEEE. https://doi.org/10.1109/MICRO.2012.42

GitHub - ChampSim/ChampSim: ChampSim repository. (s.f.). Recuperado el 14 de abril de 2022, de https://github.com/ChampSim/ChampSim

Hosseinzadeh, M., Moghim, N., Taheri, S., & Gholami, N. (2024). A new cache replacement policy in named data network based on FIB table information. *Telecommunication Systems*, 1-12. https://doi.org/10.1007/s11235-023-01012-5

Johnson, T., & Shasha, D. (1994, September). 2Q: A low overhead high-performance buffer management replacement algorithm. En *Proceedings of the 20th International Conference on Very Large Data Bases* (pp. 439-450). https://doi.org/10.5555/645920.672996

Kim, J., Pugsley, S. H., Gratz, P. V., Reddy, A. N., Wilkerson, C., & Chishti, Z. (2016, October). Path confidence-based lookahead prefetching. En *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (pp. 1-12). IEEE. https://doi.org/10.1109/MICRO.2016.7783725

Li, X., Sun, H., & Huang, Y. (2024). Efficient flow table caching architecture and replacement policy for SDN switches. *Journal of Network and Systems Management*, 32(3), 60. https://doi.org/10.1007/s10922-024-09685-6

Liu, E., Hashemi, M., Swersky, K., Ranganathan, P., & Ahn, J. (2020, November). An imitation learning approach for cache replacement. En *International Conference on Machine Learning* (pp. 6237-6247). PMLR. https://proceedings.mlr.press/v119/liu20g.html

Long, X., Gong, X., Zhang, B., & Zhou, H. (2023). Deep learning-based data prefetching in CPU-GPU unified virtual memory. *Journal of Parallel and Distributed Computing*, *174*, 19-31. https://doi.org/10.1016/j.jpdc.2023.01.005

Luo, H., Li, P., & Ding, C. (2017). Thread data sharing in cache: Theory and measurement. *ACM SIGPLAN Notices*, *52*(8), 103-115. https://doi.org/10.1145/3155284.2790007

Nakamura, T., Koizumi, T., Degawa, Y., Irie, H., Sakai, S., & Shioya, R. (2020). D-jolt: Distant jolt prefetcher. *The 1st Instruction Prefetching Championship (IPC1)*. https://doi.org/10.1145/3400302.3415760

Pakalapati, S., & Panda, B. (2019). Bouquet of instruction pointers: Instruction pointer classifier-based hardware prefetching. *The 3rd Data Prefetching Championship*. https://doi.org/10.1145/3309200.3309205

Patterson, D. A. (2006). Future of computer architecture. *Berkeley EECS Annual Research Symposium (BEARS2006)*. University of California at Berkeley, CA. https://doi.org/10.1109/BEARS.2006.1704817

Pugsley, S. H., Chishti, Z., Wilkerson, C., Chuang, P. F., Scott, R. L., Jaleel, A., & Balasubramonian, R. (2014, February). Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers.

En *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)* (pp. 626-637). IEEE. https://doi.org/10.1109/HPCA.2014.6835962

Qureshi, M. K., & Patt, Y. N. (2006, December). Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. En *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)* (pp. 423-432). IEEE. https://doi.org/10.1109/MICRO.2006.49

Seznec, A. (2020, May). The FNL+ MMA instruction cache prefetcher. En *IPC-1-First Instruction Prefetching Championship* (pp. 1-5). https://doi.org/10.1145/3400302.3415761

Shakerinava, M., Bakhshalipour, M., Lotfi-Kamran, P., & Sarbazi-Azad, H. (2019). Multi-lookahead offset prefetching. *The Third Data Prefetching Championship*. https://doi.org/10.1145/3309200.3309206

Souza, M. A., & Freitas, H. C. (2024). Reinforcement learning-based cache replacement policies for multicore processors. *IEEE Access*. https://doi.org/10.1109/ACCESS.2024.3314805

SPEC CPU® 2017. (s.f.). Recuperado el 14 de abril de 2022, de https://www.spec.org/cpu2017/

Srinath, S., Mutlu, O., Kim, H., & Patt, Y. N. (2007, February). Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. En *2007 IEEE 13th International Symposium on High Performance Computer Architecture* (pp. 63-74). IEEE. https://doi.org/10.1109/HPCA.2007.346203

Srivastava, S., & Sharma, S. (2019, January). Analysis of cyber-related issues by implementing data mining algorithm. En *2019 9th International Conference on Cloud Computing, Data Science & Engineering (Confluence)* (pp. 606-610). IEEE. https://doi.org/10.1109/CONFLUENCE.2019.8776900

Third Data Prefetching Championship | SIGARCH. (s.f.). Recuperado el 14 de abril de 2022, de https://www.sigarch.org/call-contributions/third-data-prefetching-championship/

Wang, Z., Hu, J., Min, G., Zhao, Z., & Wang, Z. (2024). Agile cache replacement in edge computing via offline-online deep reinforcement learning. *IEEE Transactions on Parallel and Distributed Systems*, *35*(4), 663-674. https://doi.org/10.1109/TPDS.2024.3320449

Wu, C. J., & Martonosi, M. (2011, April). Characterization and dynamic mitigation of intra-application cache interference. En *IEEE ISPASS: IEEE International Symposium on Performance Analysis of Systems and Software* (pp. 2-11). IEEE. https://doi.org/10.1109/ISPASS.2011.5762690

Young, V., Chou, C. C., Jaleel, A., & Qureshi, M. (2017, June). Ship++: Enhancing signature-based hit predictor for improved cache performance. En *Proceedings of the Cache Replacement Championship (CRC'17) held in conjunction with the International Symposium on Computer Architecture (ISCA'17)*. https://doi.org/10.1145/3109904.3109913

Yovita, L. V., Wibowo, T. A., Ramadha, A. A., Satriawan, G. P., & Raniprima, S. (2022). Performance analysis of cache replacement algorithm using virtual named data network nodes. *Jurnal Online Informatika*, *7*(2), 203-210. https://doi.org/10.15575/join.v7i2.1689