Implementing and Evaluating a Best Offset (BO) Hardware Prefetcher using Gem5

***Abstract*—This paper explores prefetching algorithms for optimized memory access performance in caches. We have conducted a literature review, implemented a Best-Offset (BO) prefetcher in gem5, and evaluated its efficiency using benchmark workloads compared to a Next-line prefetcher.**

The goal was to improve the Instructions per Cycle (IPC) of the binary benchmarks, but fell short as a result of the BO prefetcher (mainly) underperforming compared to the Next-line prefetcher (for those specific workloads). Our findings highlight the benefits and shortcomings of implementing prefetching algorithms.

I. Introduction

Memory latency is a critical bottleneck in modern computer systems. A more detailed description of the memory latency problem can be found in Section II. Prefetching is a widely used technique to mitigate the issues presented by the memory latency problem by predicting and loading data before it is needed.

A. Motivation

The motivation behind this paper is to research existing prefetching technologies and understand their effect on computer systems. Furthermore, this paper will aim to implement a "Best-Offset (BO) prefetcher" in gem5 (an open-source computer architecture simulator [1]) and evaluate its performance against various benchmarks.

B. Contributions and Results

This paper explores the use of the BO prefetching algorithm to improve Instructions per Cycle (IPC) in binary execution. Through testing and evaluation, we will determine the effectiveness of this prefetcher. Our results determined that the BO prefetcher improves overall IPC on a general basis. However, given the selected binaries we tested our prefetcher on, there seemed to be little effect on IPC for a majority of them. We later determined that this was a result of the binaries not relying heavily on the L2 cache for execution, and not our prefetcher preforming poorly.

C. Paper Organization

The remainder of this paper is structured as follows:

- **Section II** provides an overview of memory latency issues and prefetching strategies, including classifications and how they are typically implemented.
- **Section III** reviews existing prefetching algorithms and highlights similarities and differences between existing prefetching algorithms and Best-Offset prefetching.
- **Section IV** details the design and integration of our "Best-Offset prefetcher" within the gem5 simulator.
- **Section V** outlines our experimental setup, including hardware configurations, selected workloads, evaluation metrics, and validation procedures to ensure accurate and reproducible results.
- **Section VI** presents the performance evaluation of our prefetcher with metrics such as cache hit rate improvements. Furthermore, execution time reductions and memory bandwidth utilization will be compared to the baseline configuration before any prefetcher implementation.
- **Section VII** discusses the implications of our findings, addresses the limitations of our current approach, and suggests potential optimizations to improve our prefetcher performance.
- **Section VIII** summarizes our contributions, reflects on the impact of our work, and propose future directions in the field of memory prefetching techniques.

II. Background

As mentioned in Section I, memory latency is a critical bottleneck in modern computer systems as a result of variable performance with memory modules. In order to mediate this problem, computer systems use deep memory hierarchies composed of sequential blocks of memory with increasing size and increased latency. When using a memory hierarchy as outlined, memory accessed from the top of the memory hierarchy is very quick (but limited in size) while a memory access further down would have much higher latency. As a result of low-latency memory having small capacity, prefetchers attempt to reduce the average access times by speculatively moving data up the memory hierarchy before they are required [2]. The upper layers of the memory hierarchy will hereafter be reference to as $L1$, $L2$, and $L3$ caches, where

$$L1 < L2 < L3$$

with reference to memory latency and

$$L3 > L2 > L1$$

with reference to memory size.

A. Overview of Prefetching

Prefetching strategies can be classified into hardware-based, software-based, and hybrid approaches. Previous research has demonstrated various algorithms, including stream prefetchers, stride prefetchers, correlation-based prefetchers, and excecution-based prefetchers [2]. In general, prefetcher

implementations vary in the way they attempt to "predict" what the next memory access will be. Computer architectures and the developers of the systems often have a limited understanding of the memory access patterns for a given application, making prefetchers difficult to implement on a general basis. A prefetcher which is not suited for the workload a program is executing can, in many cases, result in a performance reduction for a system [2].

B. Best-Offset Prefetching

The Best-Offset (BO) prefetcher is a next line prefetcher that further develops the Sandbox prefetcher [3] with a special focus on timeliness while still maintaining simple hardware simple. The central idea behind BO is to continuously evaluate the offset size that fits the workload the best [4]. It was first proposed at the 2nd Data Prefetching Championship in 2015 by Pierre Michaud [4] and optimized for the SPEC CPU2006 [5]. The BO prefetcher is intended to prefetch into the $L2$ cache, and is a degree- one prefetcher, which means it only issues one prefetch with one offset per access. Furthermore, the BO prefetcher is considered more aggressive, meaning it speculates more about which data to prefetch. As a result, it may fetch larger amounts of data earlier in the execution process [6].

Offset prefetchers work well on a variety of streams, which are given sequences of bits. For sequential streams, the whole stream will be prefetched if the offset is chosen correctly. Scrambling refers to a non-sequential access pattern, where the sequence of bits varies slightly or drastically from a sequential stream. If the offset is long enough, the BO prefetcher may still prefetch all the relevant data. However, the effectiveness in scrambled streams may vary [4]. In periodic accesses with a fixed distance between consecutive memory accesses, if the offset is set to the sum of stride distances, the BO prefetcher will successfuly predict all future accesses. Interleaved streams S1 and S2 are two access patterns which happen in parallel, either alternatively or mixed together. If the streams respectively have step sizes of 2 and 3, the least common multiple (i.e. 6) would be an ideal offset. This is a result of the fact that a best offset of 6 would cover memory accesses for both streams [4]. In summary, BO prefetchers work well in a variety of different memory access cases, making this implementation quite effective relative to its overhead as a result of its implementation. This is because prefetching inaccurate data to the cache causes contention in the bandwidth and fills up the cache with useless data [3].

III. Related Work

This section will give an overview over common prefetching algorithms which are already widely used. Furthermore, these will be contrasted to and compared with the BO prefetcher to determine differences and commonalities between the different algorithms.

A. Main Types of Prefetchers

- **Next-line prefetching [7]:** Prefetches the next cache line on a cache miss. Similar to the (BO) prefetcher, but with a fixed offset, whereas BO dynamically adjusts the offset.
- **Stream prefetching [8]:** Designed to detect memory access streams and prefetch accordingly. The BO prefetcher performs well on streams but does not explicitly detect them.
- **Stride prefetching [4], [9]:** Identifies memory accesses with a constant stride. Unlike Next-line prefetchers with a fixed offset, stride prefetchers adapt to varying access patterns.
- **Delta-correlation prefetching [4], [8]:** Predicts the next memory access based on observed deltas in access patterns. In contrast, BO prefetchers estimate which line will be accessed soon rather than predicting the exact next line.

B. State-of-the-Art Prefetcher Algorithms

1) Sandbox Prefetching: The Sandbox prefetcher is an offset prefetcher with simple hardware [3]. Its main goal is to mediate the aggressiveness of the prefetcher by evaluating them first. This is done by using a Bloom filter, which is probabilistic data structure that checks whether an element is part of a set [10], to evaluate the prefetching. In Sandbox prefetching, the prefetch address is added to a Bloom filter and the subsequent cache accesses are tested against the contents of the Bloom filter. By doing that, it is checked whether the prefetcher could have accurately prefetched the data. At the same time, it checks if there are prefetchable streams [3]. When the accuracy exceeds a threshold, it performs the actual prefetching. That way it combines some attributes of the confirmation-based prefetcher, which means it only prefetch after having some confidence the prefetch will be useful, and the immediate prefetcher, which is a prefetcher that as soon as given an input address will generate and perform a prefetch. It is made with $L2$ and SPEC2006 in mind [3].

In comparison, the BO prefetcher, which is based on the Sandbox prefetcher, was designed for the same use case and uses the same principle of evaluating the prefetchs but is using the Best-Offset learning algorithm and Recent Requests table instead (see Section IV). Another important difference is that the BO prefetcher does the prefetching and learning algorithm at the same time if assuming the best offset is above the predefined accuracy threshold.

2) Right on Time (ROT) Prefetching: The ROT prefetcher is another hardware based implementation which detects access patterns in memory accesses and brings the data closer to the processor. This prefetcher also adjusts the prefetch distance of the pattern so that the prefetched data arrives just in time for it to be used, hence the name. ROT prefetchers monitor the traffic between the first- and second-level caches ($L1$ and $L2$). It tracks operations made, accessed addresses,

and cache hit/miss information to dynamically initiate or halt the prefetching behaviour [11]. Furthermore, the prefetcher uses this information to adjust the prefetch distance to match application behavior.

This algorithm consists of three main components: detecting new access patterns, maintaining active prefetching, and adapting to new strides. The prefetched lines are directly sent to the second-level cache $L2$ to reduce latency in the system. To do this, it has a list of popular strides and another list of recent miss addresses [4]. This algorithm may struggle with irregular access patterns as a result of its reliance on previous accesses, resulting in a potentially inefficient prefetcher implementation in such cases [11]. This is a lot like how the offset in the BO prefetcher is chosen by trying to adapt looking at previous accesses. The list of popular strides and recent miss addresses corresponds to the offset list and RR table in BO prefetching [4].

IV. Implementation

The Best-Offset prefetcher works, in broad strokes, by detecting access patterns in memory requests and fetching data before the CPU requires it. Figure 1 illustrates the design of the BO prefetcher. Implementing the prefetcher involves modifying cache structures and updating the prefetching policies to implement a dynamic offset learning.

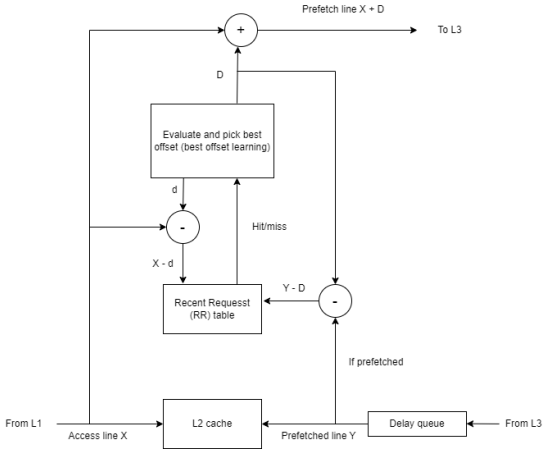

Fig. 1. Illustration of a BO Prefetcher [4] [8]

In Figure 1, X denotes an access request from the CPU which was not found in the $L1$ cache. $X + D$ denotes the line which is prefetched from the $L3$ cache, with D being the current best offset. Y is the prefetched line from $L3$, which is then placed in the $L2$ cache.

This implementation assumes that $X + D$ is in the same memory page as X. The best offset, D, varies dynamically based on application needs and is determined by the use of an iterative learning algorithm [9].

During this learning process, the BO prefetcher determines D by testing and evaluating a series of possible offsets - denoted as d. This is done through the use of three hardware structures (as seen in Figure 1):

1) **Recent Requests (RR) Table:** This table stores the addresses of recently requested memory accesses. It is implemented through hardware as a two-way skewed-associative cache consisting of two banks (a left and a right) which are both accessed through hashing functions [9]. It has also been suggested to implement the RR Table as a directly-mapped cache instead [4].
2) **Offset List:** The offset list contains a set of predefined offsets that the prefetcher tests iteratively. Each offset, d, has an associated score that reflects how often the offset successfully predicts future accesses. As we assume that X and $X + D$ are in the same memory page, the offset list does not consider offsets larger than the page size.
3) **Delay Queue:** The BO prefetcher employs a delay queue to improve coverage and accuracy, enhancing overall performance [9]. The base address X enters the queue after issuing a prefetch request for $X + D$ and is written to the right bank of the RR table. The address X remains in the queue for a fixed number of cycles. This address is then removed and written to the left bank of the RR table [9]. This mechanism helps filter unnecessary prefetches and refines prefetch accuracy.

As seen in Figure 1, when the prefetched line Y is loaded into the $L2$ cache, $Y - D$ is also loaded into the RR table in order to track how successful the current best offset (D) is in predicting future accesses.

A round refers to a full round of testing each offset in the offset list once. A learning phase consists of multiple rounds. At the start of each learning phase, the scores for all offsets in the offset list are set to zero. For every $L2$ cache read access, the prefetcher checks if the line $X - d_n$ exists in the RR table for the nth value of d in the offset list. If the line is found (hit), it means that $X - d_n$ was likely prefetched correctly, resulting in an increment to the score for the offset d_n. If there is a miss, the prefetcher moves to the next offset (d_{n+1}) in the offset list [9].

After completing a learning phase through either an offset reaching a maximum value, or the number or rounds reaches a maximum value, the prefetcher selects which offset has the highest score, which is then selected as the best offset, D. In the case where no offset has achieved a score above a predefined minimum threshold, the prefetcher is temporarily turned off to reduce unnecessary memory traffic. In this case, the learning process is still actively running and will turn the prefetcher back on again once prefetcher accuracy improves, but no prefetches are issued [9].

As mentioned, there are several ways to implement the RR table. In our reference documentation, a direct mapped or 2-way associative is suggested [9]. The size of the RR table varies based on the expected amount of concurrent memory accesses of the system. The offsets defined in the offset list are chosen using the formula $2^i * 3^j * 5^k$, with i, j, k being larger than, or equal to, 0 so long as $X + d_n$ is in the same page as X. This pattern ensures that both small and larger strides are accommodated. We will denote the maximum score for an offset as $SCOREMAX$ and the maximum number of rounds as $ROUNDMAX$. Our minimum threshold will be denoted as $BADSCORE$.

V. Methodology

Gem5 is a leading hardware simulator which we have used to simulate our BO prefetcher implementation [1]. The implemention required defining key elements of the prefetcher using various structures in C++, and ensuring that these worked together to achieve the desired result.

A. Key Components

1) **RR table:** As mentioned in the reference documentation [9], we implemented our RR table as a "2-way skewed-associative cache", which involve two banks denoted "left" and "right". The table was implemented in gem5 as two vectors of addresses (one for each bank), with a hash function to access each of the arrays comprising of the base address and which way the address is in.

2) **Offset list:** The offset list was created as a vector of tuples, where the first element in the tuple is the offset and the second element is the score of that offset in the current round of learning. This was calculated using the equation $2^i * 3^j * 5^k$, with i, j, k being larger than 0. All offsets smaller than, or equal to, $maxOffset$ were added to the offset list. See I for information about this variable.

3) **Offset scoring:** The scoring of the offsets is done in a function which is called for every call to "calculatePrefetch" in gem5. Here, the function takes in the currently accessed address, and checks the RR table using the current testing offset d_n to see if it was "likely prefetched" using that offset. If this is the case, that offset is incremented in the score. After the score is incremented or not, the prefetcher moves to the next offset d_{n+1} in the offset list to test. In the same function, we increment the round the learning phase is in and update if the prefetcher is turned on or not is based on whether one of the scores in the offset list exceeds $SCOREMAX$ or $ROUNDMAX$ is reached. This learning phase is described more in depth in Section IV.

4) **Delay Queue:** The delay queue was implemented as a list of entries that contain both the base address of the request and the process tick for when the request should be executed ($curTick() + delayTicks$). If there exists an entry in the delay queue where the process tick has passed, then it is inserted into the RR table and removed from the queue.

B. Experimental Setup

In gem5, we implemented our BO prefetcher and tuned our configuration iteratively in order to get best results. The initial values for these parameters were based in the reference documentation, but modified for our specific case [9] to gain a best IPC. The optimal setup for a future implementation may vary further based on simulator used and other factors. Table I summaries the variables defined and used in our BO prefetcher implementation.

Variable Name	Description	Value
SCOREMAX	Maximum score for an offset	30
ROUNDMAX	Maximum rounds in learning phase	100
BADSCORE	Minimum score for an offset	1
rrSize	Size of one bank in the RR table	64
tagBits	Number of bits from the address used to access the RR table	12
dqSize	Number of entries in the Delay Queue	15
dqCycles	The number of cycles an entry stays in the Delay Queue	60
maxOffset	Maximum size of offset	64

TABLE I
Overview of experimental variables used in the simulation

The BO prefetcher was implemented in the $L2$ cache per the documentation. The prefetchers in the $L1$ and $L3$ caches were retained as the default "TDTPrefetcher". No other changes were made, other than swapping out the "TDTPrefetcher" in the $L2$ cache with our BO prefetcher implementation. The same binaries were executed on both situations to maintain an identical comparison.

C. The Benchmark

To test our prefetcher implementation and compare it with some SPECspeed®2017 Integer benchmarks [12], we ensured strict testing procedures to maintain fairness and accuracy. We utilised a supporting Python-script - "run_prefetcher.py", which ensured fair testing. In this file, we defined which binaries were to be run and produced the correct statistics which were used further for evaluation of our prefetcher. To test our prefetcher, we first ran this function using our baseline configuration for the cache prefetchers as seen in Table II. This benchmark employed a pre-defined prefetcher - "TDTPrefetcher", which was implemented as a Next-line prefetcher (see Section III). The binaries executed for the benchmark configuration are summarised in Table III.

Parameter	Value
CPU Type	O3CPU
Memory Type	DDR4_2400_8x8
L1 Data Cache Size	48 KiB
L1 Data Cache Associativity	12
L1 Data Cache Prefetcher	TDTPrefetcher
L1 Instruction Cache Size	32 KiB
L1 Instruction Cache Prefetcher	TDTPrefetcher
L2 Cache Size	1280 KiB (1.25 MiB)
L2 Cache Associativity	20
L2 Cache Prefetcher	TDTPrefetcher
L3 Cache Size	3 MiB
L3 Cache Associativity	12
L3 Cache Prefetcher	TDTPrefetcher

TABLE II
SYSTEM CONFIGURATION FOR PREFETCHER TESTING

Binary	Description	Source
`gcc`	GNU Compiler Collection, a widely used C/C++ compiler.	[13]
`exchange2`	An AI benchmark that uses a recursive solution generator to play sudoku written in Fortran.	[14]
`mcf`	A combinatorial optimization benchmark that solves vehicle scheduling problems.	[15]
`deepsjeng`	A chess AI benchmark that evaluates search algorithms.	[16]
`x264`	A video encoding benchmark based on the H.264 codec.	[17]

TABLE III
BENCHMARKS USED FOR PREFETCHER EVALUATION

By using the same support script and the same binaries for execution, but swapping out the "L2 Cache Prefetcher" (from Table II) with our "BestOffsetPrefetcher", and extracting the same statistics, we were able to create a comprehensive and accurate comparative test between the benchmark and our prefetcher implementation. We also created a third benchmark with no "L2 Cache Prefetcher" but the same configuration otherwise. This allowed us to compare the BO prefetcher with Strided, and with no prefetcher at all.

VI. RESULTS AND EVALUATION

The statistics collected from both the benchmark and the BO prefetcher were extensive and detailed, automatically produced by the gem5 simulator. This section will display and evaluate the results for several key metrics, incuding accuracy, coverage, general prefetching behaviour, speedup, and hardware utilization.

A. Performance Analysis

1) Accuracy and Coverage: Figure 2 illustrates the L2 prefetcher accuracy of the BO prefetcher compared to the Next-line prefetcher (TDTPrefetcher), representing the proportion of total prefetches which were correctly predicted and referenced by the application before being replaced.

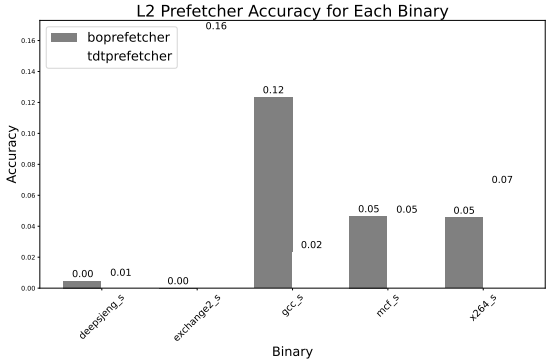

Fig. 2. Accuracy of the BO Prefetcher Implementation

As seen in the figure, the BO prefetcher demonstrates significantly lower accuracy (approximately 0%) on the binary `exchange_s`. On `gcc_s`, there was a 14 times gain from 2% with TDTPrefetcher to 28% using our BO prefetcher. On the binaries `deepsjeng_s` and `mcf_s` there was little change, returning 0% and 0.05% respectively (compared to 0.01% and 0.05% using Next-line). Finally, there was a slight reduction of 3% on the binary `x264_s` when using our prefetcher implementation.

Figure 3, similarly to Figure 2, depicts the coverage of the L2 prefetcher between our BO prefetcher and the Next-line prefetcher. Coverage refers to the ratio of correctly prefetches which are references by the application before being replaced over the number of cache misses without any prefetching. In other words, coverage measures how many of the potential candidates for prefetching were actually retrieved.

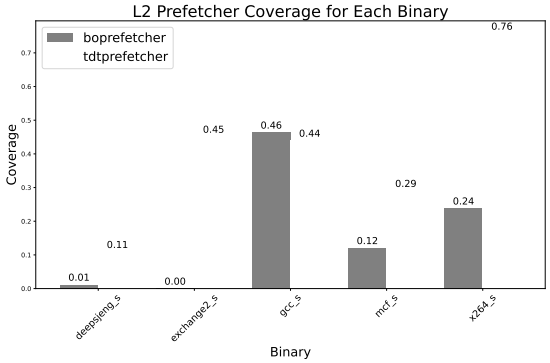

Fig. 3. Coverage of the BO Prefetcher Implementation

Here, the results are more dramatic than the accuracy comparison. For the binaries `deepsjeng_s` and `exchange_s`, the coverage was essentially at 0%. This is likely a result of the prefetcher turning itself off after not being able to determine a best offset. For `gcc_s`, the coverage increased significantly, from 44% to 71%. For `mcf_s` and `x264_s`, there were both falls in the coverage. `mcf_s` experienced a fall from 29% to 10% and `x264_s` a fall from 76% to 5% (a 15 times reduction).

2) Prefetching Behaviour: Figures 4 and 5 illustrate the behavior of the Next-line prefetcher (TDTPrefetcher) and the BO prefetcher during binary execution. The behavior is analyzed using three key metrics:

1) **Covered** (system.l2.prefetcher.pfUseful): The number of prefetches which were successfully used to satisfy memory requests by the application.
2) **Uncovered** (system.l2.prefetcher.demandMshrMisses): The number of L2 cache misses which were not covered by the prefetcher.
3) **Overpredicted** (system.l2.prefetcher.pfUnused): The number of prefetched cache lines which were never used by the application.

The graph is scaled so that sum of "Covered" and "Uncovered" sum to 100% to indicate the full scope of the applications memory accesses. These metrics together help evaluate the effectiveness of each prefetcher by seeing how they behave when executing the binaries. Figure 4 shows this behaviour for the Next-line prefetcher (TDTPrefetcher), and Figure 5 shows the same behaviour but for our BO prefetcher implementation.

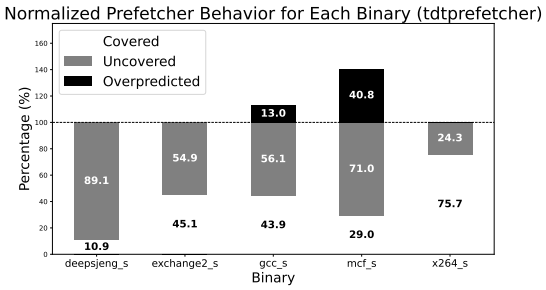

Fig. 4. Prefetching Behaviour of the Next-line

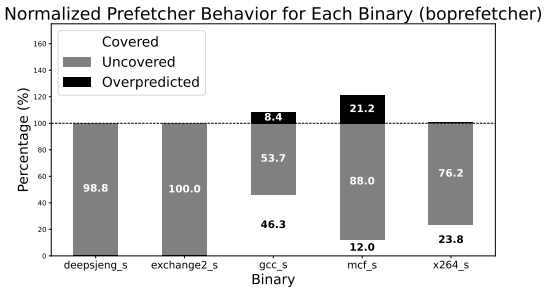

Fig. 5. Prefetching Behaviour of the BO prefetcher

The general trend for the executed binaries is a massive increase in "uncovered" prefetches, in turn a reduction in "covered" prefetches. For the binaries which had an overprediction of prefetcher (`gcc_s` and `mcf_s`) both had a reduction in "overpredicted" prefetches (respectively a reduction of 65% and 27.6%). The only real deviation to these trends are from the binary `gcc_s`. Here, the "covered" prefetches have increased from 43.9% to 71.1%, with uncovered likewise falling from 56.1% to 28.9%. This indicates that more "correct" prefetches were obtained for that binary when using the BO prefetcher compared to the Next-line prefetcher.

3) Instructions per Cycle: Instructions per Cycle (IPC) is a performance metric which measures how efficiently the CPU is executing instructions. A higher IPC indicates a better utilization of the processors resources, which would in turn improve overall performance. An increase in IPC indicates that the prefetcher more effectively reduces memory latency by fetching data before it is needed, thereby improving execution speeds. Figure 6 shows a comparison of the IPCs for each binary, for each configuration of the gem5 simulator.

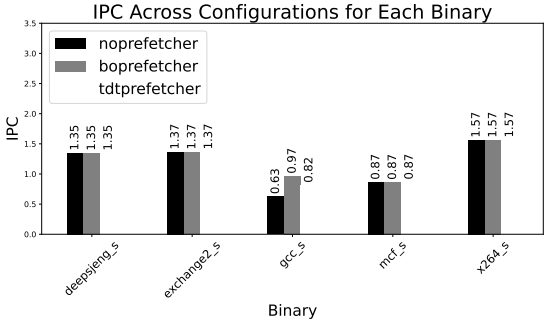

Fig. 6. IPC Comparison

As seen in the figure, there is a disappointing lack of change between the different prefetcher configurations. For the binaries `deepsjeng_s`, `exchange2_s`, `mcf_s`, and `x264_s`, there is no noteworthy change in IPC between the various configurations. This is extremely suprising as this indicates that the processor is equally capable of executing these binaries regardless of if there even is a prefetcher in L2 or not. This indicates that these workloads are not significantly memory-bound, and therefore rely on the L2 cache minimally. In other words, these binaries are effectively handled in within the CPUs core execution, relying more on computation rather than simultaneous accesses to memory.

An interesting exception, and a clear outlier with reference to all metrics measured, is the `gcc_s` binary. With this workload, there is a steady increase in IPC from the no prefetcher configuration, to the Next-line prefetcher, and finally to the BO prefetcher. From no prefetcher to Next-line there was an increase of 0.19IPCs, and a further increase of 0.25IPCs when going to the BO prefetcher. In total, there is an increase in 0.44IPCs, a roughly 1.7 times increase, between having no L2 prefetcher and using our implementation. This indicates that this specific binary is more reliant on memory accesses than the others.

4) Speedup and Harmonic Mean: Performance of prefetchers is typically measured in terms of speedup, which is a metric which measures how much faster a system runs with an optimization (a better prefetcher) compared to a baseline. It is determined using the equation:

$$\text{Speedup} = \frac{\text{E}_{\text{no prefetcher}}}{\text{E}_{\text{with prefetcher}}} = \frac{\text{IPC}_{\text{with prefetcher}}}{\text{IPC}_{\text{no prefetcher}}} \tag{1}$$

where E represents the execution time, and IPC denotes the number of instructions per cycle.

Here, $\text{IPC}_{\text{no prefetcher}}$ is the measured IPC for the configuration with no L2 cache. Therefore, we can compare the speedup of the Next-line prefetcher (TDTPrefetcher) to that of the BO prefetcher for each binary. This can be seen in Figure 7

The harmonic means is used to summarize IPC improvements across multiple workloads, and functions as an average to aggregate each benchmark speedup into one score for that prefetcher. The harmonic mean is determined with the following equation:

$$H_{\text{avg}} = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}} \tag{2}$$

where H_{avg} represents the harmonic mean, n is the total number of values, and x_i are the individual data points. In our case x_i are the individual speedups for that prefetcher and n is the number of binaries. The two harmonic means can be seen in Figure 8.

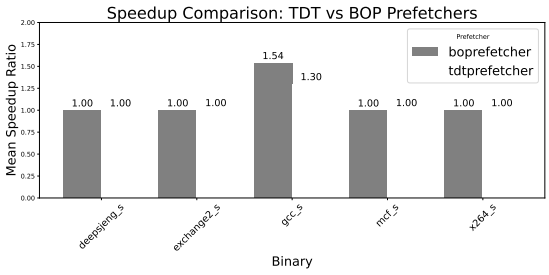

Fig. 7. IPC Comparison

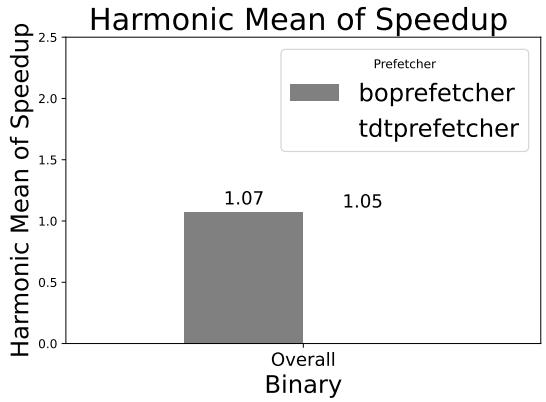

Fig. 8. Harmonic Mean Comparison between Next-line and BO Prefetcher

The results in Figure 7 indicate that, for a majority of the binaries, the prefetcher configuration provides no siginifcant speedup when compared to using no prefetcher. For these uneffected benchmarks, the speedup remains at 1.0, which suggests that these workloads do not benefit from L2 cache prefetching. This supports the conlusions made in our discussion surrounding IPC variations in Section VI-A3.

However, for the binary `gcc_s`, both prefetcher show noticeable improvement in speedup. With the Next-line prefetcher, a speedup of 1.3 is achieved. Furthermore, a speedup of 1.7 is reached when using our BO prefetcher. This further supports the idea that `gcc_s` exhibits memory access patterns which benefit more from L2 prefetching than the other binaries. When aggregating our speedups across all workloads, the harmonic mean speedups are 1.09 for the BO prefetcher and 1.05 for the Next-line prefetcher. This reinforces the observation that the configuration of prefetcher has little to no impact across most workloads. Despite this, the BO prefetcher does deliver slightly better performance overall.

B. Hardware Utilization

The BO prefetcher implementation introduces a couple additional hardware structures which are required. These include the RR table, the Offset List, and the Delay Queue. Each of these requires a certain number of bits of storage in order to function, leading to larger hardware overhead. In this section we will calculate the hardware utilization of our BO prefetcher and determine if it is justified given the performance increases it provides. There are additional structures as well which require bits, such as the current best offset, round, and best scores. However, these require only a couple bits each, so will not be included in the calculation.

1) RR Table: The RR Table was, as mentioned in Section V, represented as two vectors of addresses, each of length 64, with the size of an entry being 12 bits for the tag.

$$\begin{aligned} RR_{\text{bits}} &= 2 \times (64 \times 12) \\ &= 2 \times 768 \text{ bits} \\ &= 1536 \text{ bits} \end{aligned} \tag{3}$$

2) Offset List: From our equation for determining offsets in the offset list - $2^i * 3^j * 5^k$, with i, j, k being larger than (or equal to) 0, and a maximum offset of 64, we have a total of 26 possible unique offsets. As we also have negative offsets, this increases to 52 unique offsets (26 negative and 26 positive). As our maximum offset is 64, we only require 8 bits to store each offset value. For the offset score, as $SCOREMAX$ is 30, we only require 5 bits to determine score. Therefore, one entry in the Offset List is 8 + 5 bits.

$$\begin{aligned} OffsetList_{\text{bits}} &= 52 \times (8 + 5) \\ &= 52 \times 13 \text{ bits} \\ &= 676 \text{ bits} \end{aligned} \tag{4}$$

3) Delay Queue: The Delay Queue contains 15 entries. Each entry was implemented as an object containing both the 12 bit tag, and the current tick when the tag was inserted. The current tick is represented as a 64 bit unsigned integer in gem5. However, this can be implemented as 32 bit signed int

in hardware. When choosing the worst case (upper bound) of 64, we get the following:

$$\begin{aligned} DelayQueue_{\text{bits}} &= 15 \times (12 + 64) \\ &= 15 \times 76 \text{ bits} \\ &= 1140 \text{ bits} \end{aligned} \tag{5}$$

4) Combined: The BO prefetcher requires additional logic for offset tracking and prediction. When combining the total hardware utilization of the main structures, we get the following total hardware utilization:

$$\begin{aligned} BO_{\text{size}} &= RR_{\text{bits}} + OffsetList_{\text{bits}} + DelayQueue_{\text{bits}} \\ &= 1536 \text{ bits} + 676 \text{ bits} + 1140 \text{ bits} \\ &= 3352\text{bits} \\ &= 419\text{B} \\ &\approx 0.409\text{KB} \end{aligned} \tag{6}$$

From this, we gather that our BO prefetcher is not very hardware intensive. However, given the performance characteristics discussed in the results section of the report, it remains ambiguous whether or not this prefetcher is worth implementing. For a majority of the binaries executed there was little variation in IPC regardless of which L2 configuration was used. The only real binary executed where we have comparative results is `gcc_s`, where IPC increased significantly. Considering the quite modest overhead of 0.409 KB, there would be cases where the BO prefetcher would be worth implementing, especially workloads with a larger reliance on the L2 cache (such as `gcc_s`). A more detailed discussion surrounding the worthiness of our prefetcher can be read in Section VII-A.

VII. Discussion

A. Is our BO Prefetcher Worth Having?

With background on all the data collected, it may appear that our BO prefetcher is not very impressive. However, testing on only five of the 43 benchmarks in SPEC2017 may not paint the full picture. All the benchmarks tested were also made for integer operations. This is an interesting observation given that the inventor of the Best-Offset prefetcher expresses how the prefetcher experiences the largest gains in speedup when tested on benchmarks containing floating point operations [4]. In addition to this, the prefetcher also works well on streamed data (sequential, scrambled, and interleaved) as mentioned in Section II. The workload presented for the BO prefetcher has the potential to change the results massively, and may show that it improves IPC dramatically like it did for the `gcc_s` binary. In order to further analyze this, it would be beneficial to test more of the benchmarks in SPEC2017 to better understand how this prefetcher behaves.

Whether or not the BO prefetcher is worth implementing depends, as most things do, on what the intended end goal is. From the statistics collected the prefetcher would, in general, only provide marginal speedup. In some cases, such as `mcf_s`, the IPC would in fact fall slightly, but not noticeably. For a system which compiles GNU C code regularily, implementing this prefetcher would be incredibly beneficial. One would experience a 30% gain in processing when compared to a Next-line prefetcher if the processor was only compiling GNU C code.

B. Challenges and Performance Trade-offs

One of the most complicated (and provocative) challenges faced during this implementation was the tuning of the parameters outlined in Table I. As a result of gem5 requiring meticulous recompiling, and each simulation of the benchmarks taking between 5 and 15 minutes, this process was tedious and time consuming. We iterated through all the parameters in order to find the best combination which increased IPC most for most benchmarks. In addition to this, several parameters had contradictory effects on the results, making it difficult to pick values which increased the performance "overall". It is still unclear what correlation exists between the parameter values and the different performance metrics for the L2 cache and prefetcher. Finding the combination of parameter values that most consistently improved performance took time and ultimately gave results which were nearly identical to those suggested in one of our reference papers [9], which was unsatisfying. This was certainly a weakness in the evaluation of the prefetcher, as we have no way of knowing if our combination of values was truly the most "correct" combination for our implementation. In order to better conclude the best parameters, we suggest that future researchers create a supporting software-based solution which iteratively changes the variables and notes down all the statistics provided. Using the statistics provided, it would be beneficial to mathematically determine the impact of each variable in order to better tweak the prefetcher to perform optimally.

VIII. Conclusion

The goal of this paper was to implement a Best-Offset prefetcher and analyze the impact it has on IPC when compared to Next-line prefetching as a baseline. Our BO prefetcher implementation did not preform as well as expected from reading our reference literature. The prefetcher showed little improvements in accuracy, and massive reductions in coverage on all benchmarks except for `gcc_s`. In this binary, our prefetcher significantly outpreformed the Next-line prefetcher, with an IPC hike of rougly 0.25 IPCs. Across all binaries, our BO prefetcher had no real effect on IPC. As a result of the IPC hike in the `gcc_s` binary, the speedup when using the BO prefetcher was higher than Next-line.

However, these results were gathered from only a small subset of SPEC2017 benchmarks. In order to more accurately determine if the BO prefetcher is worth implementing, more benchmarks should be tested - preferably on all available binaries. There is also a question of which workloads the

processor would be executing as this clearly has a large impact on how the BO prefetcher performs (see Section VI).

IX. AI Declaration

ChatGPT was used to get suggestions on how to divide the text into subsections and paragraphs. Furthermore, it was also used to get an initial outline of how paragraphs should be structured and ideas which could be relevant in relation to a well-structured academic paper. Co-pilot was used to make the coding process of the BO prefetcher in C++ more effective.

References

[1] gem5 Community, "The gem5 simulator system," https://www.gem5.org/, 2025, accessed: 2025-02-04.

[2] G. Ayers, H. Litz, C. Kozyrakis, and P. Ranganathan, "Classifying memory access patterns for prefetching," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 513–526.

[3] S. H. Pugsley, Z. Chishti, C. Wilkerson, P. fei Chuang, R. L. Scott, A. Jaleel, S.-L. Lu, K. Chow, and R. Balasubramonian, "Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Orlando, FL, United States, 2014.

[4] P. Michaud, "Best-offset hardware prefetching," in *International Symposium on High-Performance Computer Architecture*, Barcelona, Spain, 2016.

[5] John L. Henning, SPEC CPU Subcommittee , "Spec cpu2006 benchmark descriptions," in *SPEC CPU2006 Benchmark Descriptions*, 2006, pp. 1–17.

[6] A. E. Papathanasiou and M. L. Scott, "Aggressive prefetching: an idea whose time has come," in *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, California, USA, 2005, p. 6.

[7] J. D. Collins and D. M. Tullsen, "Runtime identification of cache conflict misses: The adaptive miss buffer," in *ACM Transactions on Computer Systems*, San Diego, USA, 2001, p. 422.

[8] P. Michaud. Best-offset hardware prefetching. [Online]. Available: https://inria.hal.science/INRIA/hal-01254863/file/BOP_HPCA_2016.pdf

[9] ——, "A best-offset prefetcher," in *2nd Data Prefetching Championship*, Portland, United States, 2015.

[10] Geeks for geeks. Bloom filters – introduction and implementation. [Online]. Available: https://www.geeksforgeeks.org/bloom-filters-introduction-and-python-implementation/

[11] E. Hagersten, "Toward scalable cache only memory architectures," in *PhD thesis, Royal Institute of Technology (KTH)*, Stockholm, Sweden, 1992, p. 5.

[12] Standard Performance Evaluation Corporation (SPEC), "Spec cpu®2017 overview / what's new?" accessed: 2024-04-03. [Online]. Available: https://www.spec.org/cpu2017/Docs/overview.html#benchmarks

[13] GNU Project, "Gcc, the gnu compiler collection," 2024, accessed: 2024-03-30. [Online]. Available: https://gcc.gnu.org/

[14] Standard Performance Evaluation Corporation (SPEC), "648.exchange2_s spec cpu®2017 benchmark description," 2020, accessed: 2024-04-02. [Online]. Available: https://www.spec.org/cpu2017/Docs/benchmarks/648.exchange2_s.html

[15] ——, "605.mcf_s spec cpu®2017 benchmark description," 2020, accessed: 2024-03-30. [Online]. Available: https://www.spec.org/cpu2017/Docs/benchmarks/605.mcf_s.html

[16] ——, "631.deepsjeng_s spec cpu®2017 benchmark description," accessed: 2024-03-30. [Online]. Available: https://www.spec.org/cpu2017/Docs/benchmarks/631.deepsjeng_s.html

[17] VideoLAN Organization, "Videolan, a project and a non-profit organization." accessed: 2024-03-30. [Online]. Available: https://www.videolan.org/developers/x264.html