CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Data Prefetcher Based on a Temporal Convolutional Network

A machine learning approach to accelerate memory intensive programs

Master's thesis in Computer science and engineering

MATTIAS LARSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Data Prefetcher Based on a Temporal Convolutional Network

A machine learning approach to accelerate memory intensive programs

MATTIAS LARSSON

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Data Prefetcher Based on a Temporal Convolutional Network
A machine learning approach to accelerate memory intensive programs
MATTIAS LARSSON

Supervisor: Pedro Petersen Moura Trancoso, CSE
Examiner: Miquel Pericas, CSE

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2022

Data Prefetcher Based on a Temporal Convolutional Network
A machine learning approach to accelerate memory intensive programs
MATTIAS LARSSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Cache memory serves a crucial role in alleviating the difference in speed between the computer's processor and main memory, which has become a growing problem over the years. However, the cache can only hide the whole memory access latency if the requested data is present in it, and only parts of it if the data is already on its way. For this reason, the technique called data prefetching has proven to be an effective way of increasing performance. This technique entails predicting which memory addresses will be accessed in the future and bringing the corresponding data to the cache ahead of time. This thesis explores the design of a data prefetcher based on a Temporal Convolutional Network (TCN), focusing on low storage overhead to make its corresponding implementation size realistic for hardware implementation. In performance simulation tests performed on 15 memory-intensive benchmarks, the TCN prefetcher achieved an average speedup of 30.5 % over a no prefetching baseline, while adding only 14.4 KB of storage overhead. The result shows that the TCN architecture can be a contender for future ML-based prefetchers and that it might work as a good substitute for larger multilayer perceptron (MLP) models. However, the results also suggest that the trade-offs necessary for practical implementation size of a neural network prefetcher make it challenging to advance the average performance beyond rule-based offset prefetchers.

Keywords: data prefetching, TCN, cache memory, machine learning, computer architecture.

Acknowledgements

First and foremost, I would like to thank my supervisor Pedro Petersen Moura Trancoso, for suggesting this thesis work and all the helpful guidance and encouragement. I would also like to thank my family, who have been a tremendous support over the years of my studies.

Mattias Larsson, Gothenburg, June 2022

Contents

List of Figures

List of Tables

1
Introduction

Processors have seen a rapid increase in performance over the years, creating the need for ever-faster main memory. Unfortunately, the latency and bandwidth of high-capacity storage have not been able to keep up in pace [1]. To aid bridge this gap, small and fast caches have come to serve a crucial role in computer architecture. However, if the requested data is not present in the caches, the processor still has to wait while it is being fetched from the main memory.

An effective technique to hide long-latency memory requests, called data prefetching, entails speculation on which data to bring into the caches before the processor has requested it. This speculation often implies finding patterns in previous memory accesses, and using them to bring in data according to those patterns [1]. Traditionally, prefetchers have generally used quite simple yet effective techniques to find patterns in memory accesses, such as finding the offset(s) that are the most common between their memory addresses [2]. Though, with time more workloads have become harder to predict due to their irregular access patterns, making these types of prefetchers less effective [2].

Recently, active research in data prefetching involves incorporating machine learning (ML) into it, following the significant advances achieved by applying it in many other areas, such as image recognition and natural language processing. In a prefetching competition hosted by ISCA in 2021, the applicants were also encouraged to use ML-based approaches for their prefetchers, with the reasoning that it could lead to advances in the state-of-the-art of data prefetching [3]. However, the competition did solely focus on high predictive performance. Hence, it did not enforce any restriction on inference latency, model size, and the ability for the prefetcher to learn online (i.e., learn from the program while it is running). Due to this, the organizers also stated that the participants' prefetchers likely would not be practical for hardware implementation [3].

1.1 Problem Statement

There are not many high-performance data prefetchers that utilize ML and still fulfill the tight requirements placed upon them to be practical as hardware prefetchers, both in terms of inference latency and implementation size [4]. This thesis explores using an ML algorithm within the context of hardware data prefetching that has not previously been tested in that context (or not been well documented), focusing on keeping the model size small. To this end, the following questions will be answered:

1. Which ML algorithms have been tested and well documented before in the context of hardware data prefetching?
2. Which ML algorithm that is not a part of (1) seems to have good potential to be used in hardware data prefetching?
3. Can the chosen algorithm be implemented as a working data prefetcher in such a way that, within the scope of this project, it gets a storage overhead comparable to published *spatial* hardware data prefetchers?
4. Can the chosen algorithm be implemented as a data prefetcher in such a way that, within the scope of this project, (3) holds true while the prefetcher's performance also is competitive with published hardware data prefetchers?

1.2 Limitations

This thesis uses the training and testing environment provided by ISCA for their 2021 ML competition [3], and the limitations it poses will hold for this thesis work as well. The three major limitations are as follows. 1) The prefetcher is trained *offline* on a section of each program, unlike real hardware prefetchers that continuously learn *online* while the program is running. 2) The prefetcher is only tested at the last level cache (LLC). 3) The performance is only evaluated in single-core processor simulations. Other than the limitations set by the chosen environment, the prefetcher's size will also only be measured in terms of storage overhead. This is because accurate physical size estimation would require implementing the final design in a hardware description language, such as the authors of [4] did. Such implementation would add an extra level of complexity that would likely not fit within the project's time frame.

2

Background

This chapter starts by covering theoretical background relevant to data prefetching. Thereafter, it covers basic machine learning concepts and the algorithm used in this thesis data prefetcher.

2.1 Main Memory and Caches

The main memory of a computer is divided into blocks and pages, where one block consists of multiple bytes, and a page consists of multiple blocks. A typical block consists of 64 bytes, and a typical page consists of 64 blocks. Each byte of storage in the memory has an individual address, which the processor uses to fetch data from that memory location. The address space for these locations can be divided into different sections as shown in 2.1.

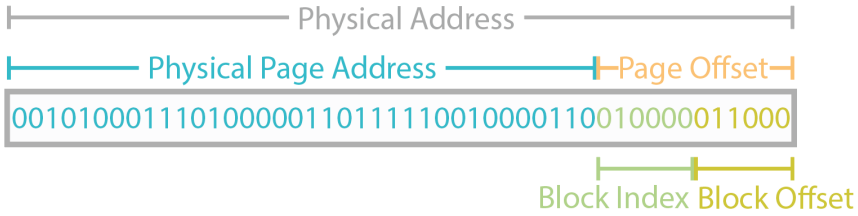

Figure 2.1: Example of a physical address and its sections. The address shown has the following hexadecimal representation: 28E837C86418.

The caches of a processor act as small temporary storage for blocks, where each block is stored in what is referred to as a *cache line*. The caches are divided into different levels based on their relative distance in their chain to the processor cores, as can be seen in Figure 2.2.

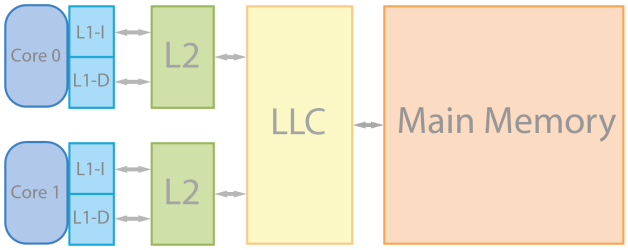

Figure 2.2: Example of a memory hierarchy. The L1 cache is commonly split into an instruction cache (L1-I) and a data cache (L1-D).

The cache closest to a core is referred to as *L1* since it is the first level of cache from the core, and the remaining caches follow the same pattern until the last cache, which usually is referred to as *LLC* (last level cache). L1 is the smallest and has the lowest latency of the caches, and for each higher level of cache, latency is traded off for storage capacity.

2.2 Program Counter

The program counter (PC) is a processor register that depending on implementation, holds the address of the next to be executed instruction or currently executed instruction. Its purpose is to keep track of where a program is in its execution sequence. However, the term PC will be used in this text to refer to the value of the register, rather than the physical register itself.

2.3 Data Prefetching

Data prefetching is a prediction technique with the goal of predicting memory addresses of future memory accesses, and issuing requests for their corresponding blocks before the processor has made explicit requests for them [1]. If the unit which is to perform the prefetching is designed to be implemented in a physical chip, it is known as hardware data prefetcher, and almost all modern high-performance processors have some built into them [5]. A prefetcher can be situated at any cache level, and there can be multiple prefetchers at the same or various levels.

For hardware data prefetchers to work, they often rely on observing one or more features connected to each observed memory access. These features can, for example, be the PC of the instruction which issued the access and the targeted memory address. From the observations of these features, the prefetchers can find patterns between accesses, which they can use to issue prefetches according to those patterns.

2.3.1 Data Prefetcher Types

Data prefetchers are generally divided into different categories based on which type of memory access pattern they exploit while performing predictions. In [1] they are categorized into four broad types, namely *stride* and *stream*, *address-correlating*, *spatial*, and *execution-based* prefetchers.

Stride and *stream* data prefetchers are seemingly the most common type of hardware data prefetcher used in today's commercial processors [1, 5]. They work by observing the distance between following memory addresses to find sequences where the stride is constant, from which they can issue prefetches of blocks on memory addresses with the same stride. This type of pattern where the stride remains constant is commonly occurring during operations on arrays and dense matrices [5], which makes this type of prefetchers effective for these tasks. On the other hand, they are generally ineffective while working on data structures that use pointers, such as

linked lists [1].

Address-correlating data prefetchers, also known as temporal prefetchers [6], store the addresses of memory accesses and makes correlations between them. They work on the principle that addresses accessed in a specific order will likely be accessed in the same order again. Hence, it can capture the access pattern of a given data structure that is repeatedly accessed with a constant stride, but is especially useful for data structures that use pointers, like linked lists [1]. Though, due to the requirement of storing long access histories, these types of prefetchers often use large amounts of memory, which has to be stored in main memory [4]. They are also only able to prefetch data from memory addresses they have seen before. As such, they are unable to prevent cold cache misses [1].

Spatial data prefetches work by observing memory accesses that have been performed close in time to another, and find patterns based on their relative offsets [1]. Therefore, they can capture simple stride patterns but can also find more complex patterns related to data structure layouts, due to these often being fixed and regular. As a result, spatial prefetchers can learn access layouts in one part of memory and use that knowledge to prefetch objects with the same layout in other parts of memory, thereby helping eliminate cold cache misses. This generalization of patterns also leads to high storage efficiency [1], which, together with not needing to store more than offsets/deltas, leads to low area overhead for these types of prefetchers [5]. One downside of spatial prefetchers is that they in general do not work well on data structures that use pointers (e.g. linked lists) [1, 5].

Execution-based data prefetchers use some type of mechanism which allows them to get ahead of the processor's instruction execution and compute which addresses to prefetch from [1]. The mechanism can for example be so-called *helper threads*, which utilize unused computational resources during times the main thread stalls waiting for a long latency memory request [1]. Another example is mechanisms that look at data structure traversals and find patterns that summarize only the instructions necessary to make prefetches (leaving the rest of the instructions out), making the pattern faster to execute than the main program [1]. This type of prefetchers can achieve very high accuracy but generally at the expense of being hard to design [4].

2.3.2 Metrics

There are many factors at play while designing high-performance data prefetchers, especially ones which can feasibly be implemented in hardware. Hence, there are multiple metrics used to evaluate their usefulness. Two basic metrics for prefetching are *coverage* and *accuracy*, where coverage refers to the fraction of cache misses that the prefetcher eliminates (2.1), and accuracy refers to the fraction of prefetches that end up being used by the processor (2.2) [7].

$$\text{Coverage} = \frac{\text{Number of cache misses eliminated due to prefetching}}{\text{Number of cache misses without prefetching}} \tag{2.1}$$

$$\text{Accuracy} = \frac{\text{Useful prefetches}}{\text{Total prefetches}} \tag{2.2}$$

Although having a high coverage and accuracy is desirable, a third metric called *timeliness* is also important. For example, suppose data is prefetched way ahead of the time at which the program requests it. In that case, it can lead to eviction of other data in the cache, which the program might have requested before the prefetched data needed to be in the cache. It could also be the contrary, that the prefetch requests happen too late to hide the whole memory access latency.

Two general performance metrics that are commonly used in regards to prefetching are *speedup* (2.3), which measures the change in throughput, and *misses per kilo instructions* (MPKI) (2.4), which measures how well the cache is able to serve the current workload. These metrics can further be reformulated as *instructions per cycle* (IPC) *improvement* (2.5), and MPKI *improvement* (2.6).

$$\text{Speedup} = \frac{\text{New IPC}}{\text{Baseline IPC}} \tag{2.3}$$

$$\text{MPKI} = \frac{\text{Total cache misses}}{\text{Total number of instructions / 1000}} \tag{2.4}$$

$$\text{IPC improvement} = (\text{Speedup - 1}) \times 100 \tag{2.5}$$

$$\text{MPKI improvement} = \frac{(\text{Baseline MPKI - New MPKI})}{\text{Baseline MPKI}} \times 100 \tag{2.6}$$

For hardware data prefetchers, it is also of high importance to have a small size in terms of square millimeters required to implement them on a chip. Since estimating the physical size can be quite a complex process, it is common to state the amount of storage overhead (i.e., the amount of memory) required by the prefetcher. The memory is chosen since it is likely to contribute to a large part of the design's total area usage.

2.3.3 Prefetch Degree

A data prefetcher can issue one or more prefetch requests for each triggering memory access, where the number of requests is referred to as the prefetch degree, or more commonly, just degree. Issuing requests at a higher degree can often improve coverage but may simultaneously decrease accuracy, resulting in more useless prefetches and increased memory bandwidth usage. Therefore, it is essential to find a balance to maximize performance, which can be done through performance testing.

2.4 Testing of Prefetchers Performance

To properly measure the effectiveness of data prefetchers, their issued prefetches need to be monitored and evaluated on a range of applications that exhibit different memory access characteristics. Standardizing these applications is vital to allow for meaningful comparison of the results that different prefetchers achieve. As such, multiple benchmark suites have been developed that include programs with various workloads. Two of the most prevalent and general ones are SPEC CPU2006 (SPEC06) [8] and SPEC CPU2017 (SPEC17) [9], which consist of a range of workloads derived from real-world applications. There are also other benchmark suites that focus on specific types of workloads, such as the GAP benchmark suite [10] that focuses on graphs.

To be able to perform performance tests without the need to first implement prefetchers in hardware, it is common to implement them in software and perform tests using a trace-driven simulator. This type of simulator models a processor with its caches and accompanying main memory, and takes traces as input consisting of long lists of consecutive instructions with associated meta-data. The traces have previously been collected at the run-time of an application by some form of instrumentation tool, and stored in files. The instruction meta-data stored in these files could for example be the PC of the instruction that generated a memory request, and the memory address of the requested data.

2.5 Machine Learning

Machine Learning (ML) is a subset of artificial intelligence (AI), with the aim of making computers improve automatically from experience [11]. There are three main types of machine learning: supervised, unsupervised, and reinforcement learning. In supervised learning, the algorithms are presented with input data and an expected output given that input data, called a label. By showing the algorithms many examples of these input-output pairs, the idea is that the algorithms shall learn a general mapping between which input shall result in which output. If the mapping has generalized well, it will also work to a high degree for unseen inputs. In contrast, unsupervised learning does not use any labels; instead, the idea is that the algorithms on their own should find patterns in the input data and, for example, group data points into clusters. Finally, reinforcement learning concerns algorithms that also do not use labels, but still receive another form of guidance. Here an algorithm (agent) interacts with an environment by receiving information regarding the environment's state and taking actions that change it. The agent, in turn, receives a numerical reward from the environment, which depends on the action taken, and the agent's goal is to learn how to maximize the rewards it receives over time.

2.5.1 Classification and Regression

Supervised learning can be divided into the two categories *classification* and *regression*. The differentiating factor is that in classification, the output is a label, while in regression, the output is a quantity. A use case for classification could, for example, be to make a computer learn how to recognize handwritten digits. Here the input would be an image of a handwritten digit, and the output would be a guess of which digit is seen in the image. A use case for regression could, for example, be to predict future temperatures based on current conditions, such as temperature and humidity.

2.5.2 Artificial Neural Networks

Artificial neural networks (ANNs) are a subset of machine learning that is inspired by how brains work [12]. They function by connecting together rows (or *layers*) of nodes referred to as *artificial neurons*, or simply *neurons*, forming a network structure. Every connection in the network carries a *weight* (w) and each neuron has a *bias* (b) as well as an *activation function* (φ), as shown in Figure 2.3. Each input x_i to a neuron is multiplied by the weight w_{ij} associated with the corresponding connection, and all inputs to each neuron are summed up and added together with the bias b_j. The result is finally fed into the activation function that determines the final output y_j, as shown in equation 2.7.

$$y_j = \varphi\left(\sum_{i=1}^{n} w_{ij}x_i + b_j\right) \tag{2.7}$$

The weights and biases are called trainable parameters since they can be learned (tuned) through processes such as back-propagation [13] to fit to data. For these

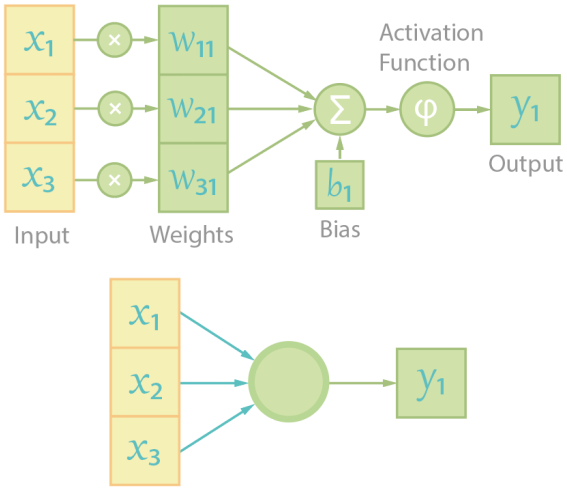

Figure 2.3: Above is a detailed example of an artificial neuron with three inputs. Below is a simplification of the neuron for visualization, where each cyan arrow corresponds to a unique weight.

networks to learn complex relationships between the input and output of the network, non-linear activation functions are used for each neuron that is not in the input layer.

2.5.3 Fully Connected Layer

A layer in an ANN is said to be fully connected if each neuron in the layer is connected to every neuron in the previous layer, as shown in figure 2.4. A multi-level perceptron (MLP) consists of these fully connected layers. It has an input and output layer, as well as at least one intermediate layer referred to as a *hidden* layer.

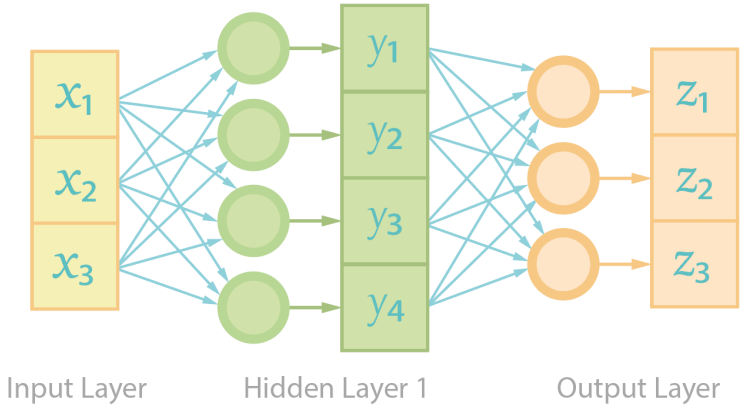

Figure 2.4: A MLP with an input layer of 3 neurons, 1 hidden layer with 4 neurons and an output layer with 3 neurons.

2.5.4 Convolutional Layer

Convolutional layers specialize in finding spatial patterns in data [14], and are most commonly known for their use in computer vision, but have other use cases as well. In contrast to fully connected layers, the neurons in convolutional layers are only connected to a limited set of outputs from the previous layer, referred to as their *receptive field*. Furthermore, the neurons are organized in planes where all neurons within each plane share the same set of weights. The set of unique weights that are used within each plane is called a *filter*, which can consist of multiple *kernels*, where each kernel corresponds to one input *channel*. The set of outputs of a plane is referred to as a *feature map* [14], and is equivalent to an output channel. The number of planes, or equivalently the number of filters in each layer, depends on how many different features are to be extracted from the input data. Figure 2.5 shows a simple example of how a convolutional layer operates on a one-dimensional input array with one channel.

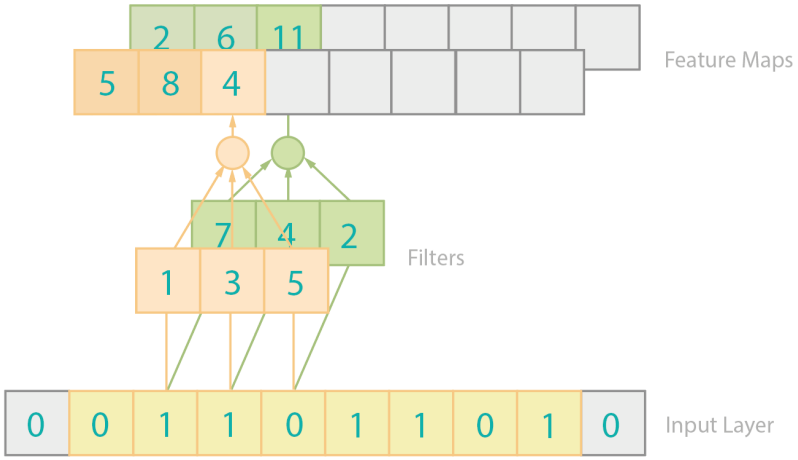

Figure 2.5: In this example, there are two filters with a width of 3 and a stride equal to 1. In a sequential network implementation, each filter incrementally moves one step to the right from the far left while computing the corresponding value in the feature map. The grayed-out zeros in the input layer are padding, which keeps the feature maps the same size as the input (in yellow).

2.5.5 Temporal Convolutional Networks

Temporal Convolutional Networks (TCNs) is a term for a class of networks that was first introduced by Lea et al. [15], along with two TCN models designed for action segmentation. The term TCN was later adopted by Bai et al. [16], which through their evaluation, showed that their TCN model could outperform recurrent neural network models in a range of sequence modeling tasks. The first core principle of a TCN is that the convolutional operations are causal, meaning that given a input sequence of T inputs (x_1, ..., x_T), the output y_t at time t can only depend on the inputs x_1, ..., x_t (i.e. no dependency on any later inputs x_{t+1}, ..., x_T) [16]. The second core principle is that the output sequence (y_1, ..., y_T) shall be of the same

length as the input sequence (x_1 ..., x_T) [16].

The specific TCN architecture presented by Bai et al. uses dilated causal convolutional layers with exponentially increasing dilations (i.e. 1, 2, 4, ..., 2^{n-1}). Such dilations allow effective expansion of the output neurons' receptive field without needing to increase filter size, as shown in figure 2.6. This method was inspired by WaveNet [17] which DeepMind developed for the synthesis of raw audio, and which today is used to generate the US English and Japanese voices of Google Assistant [18]. The dilated causal convolutional layers used in the TCN are accompanied by weight normalization, dropout, as well as a residual connection, which all have been shown to help the training of deep neural networks [19] [20] [21]. As activation function it uses ReLU [22]. Inside of each *residual block*, which is shown in figure 2.7, there are two dilated causal convolutions, which results in an almost doubling of the TCNs receptive field. The total receptive field of the TCN can be computed as follows:

$$Receptive\ field = 1 + 2(k_{size} - 1) \cdot N_{stacks} \cdot \sum_i d_i, \tag{2.8}$$

where the 2 comes from the two dilated convolutions in each residual block, k_{size} is the kernel size, N_{stacks} is how many residual blocks are stacked on top of each other, and d is the dilations in each dilated convolution.

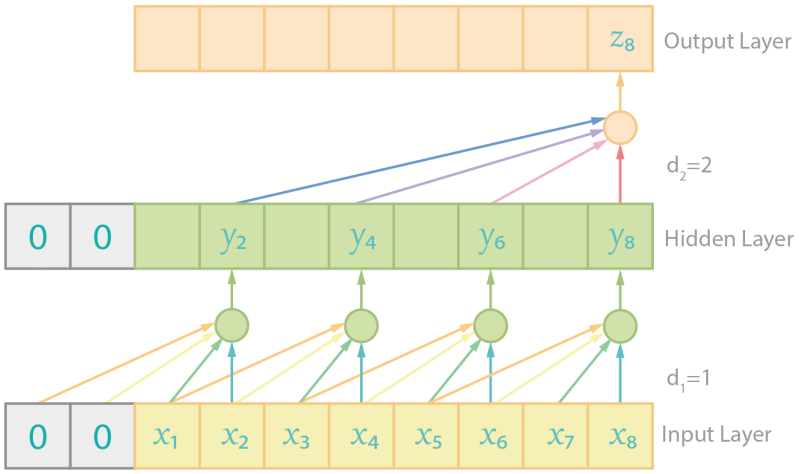

Figure 2.6: The operation of a dilated causal convolution for the final output of a sequence, using a filter size k = 4 and two dilations $d_1 = 1$ and $d_2 = 2$. The figure shows only the operation of one filter per layer (in contrast to Figure 2.5), and each unique weight in both filters is represented with a different color (instead of a square with a number, as in Figure 2.5).

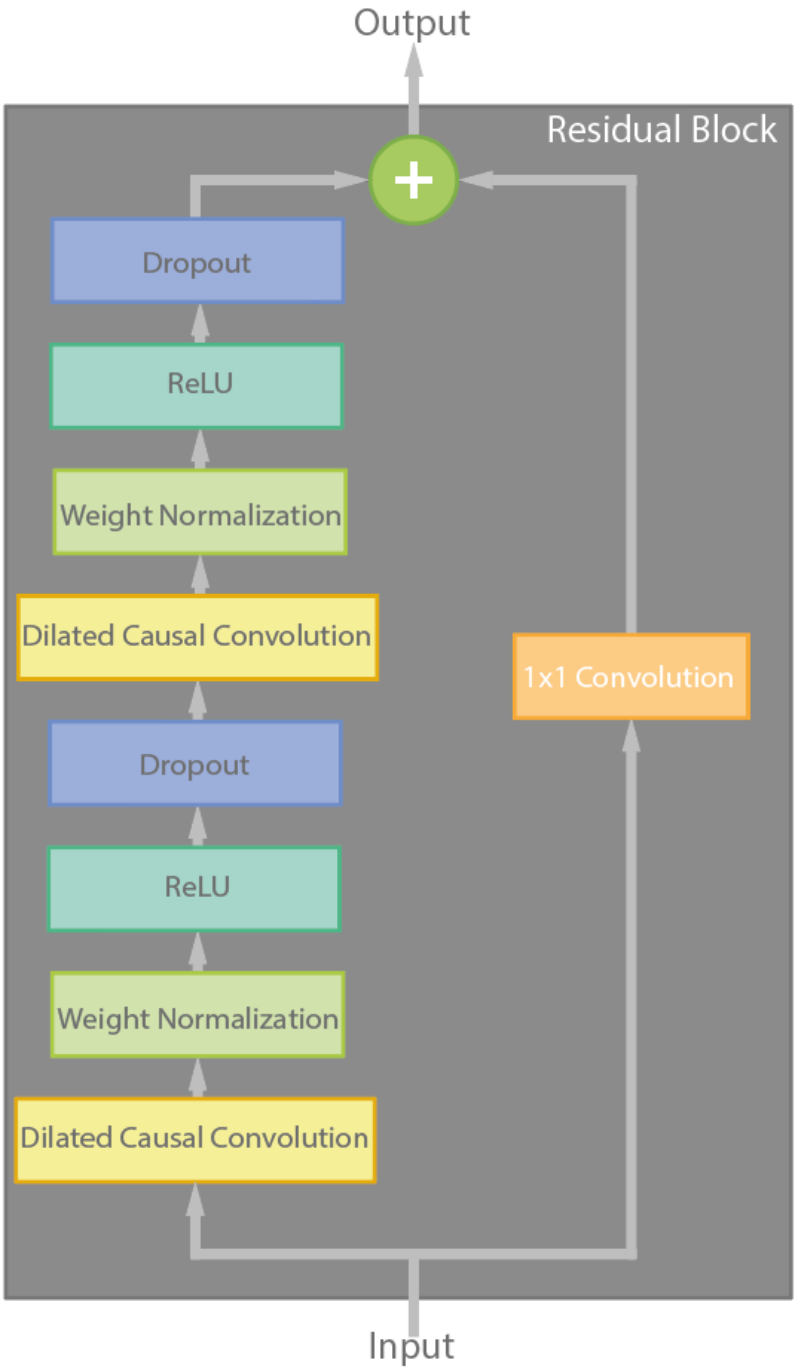

Figure 2.7: Residual block used in the TCN presented by Bai et al. The input can either be a direct input sequence, or it can be a sequence from a previous residual block if $N_{stacks} > 1$.

3
Related Work

This section will cover hardware data prefetchers of the rule-based type and ones that utilize different ML algorithms. As data prefetching has a long history with many different proposals presented in the literature, only a small portion of them will be presented here.

3.1 Rules-Based Prefetchers

Rule-based data prefetchers, often referred to as traditional data prefetchers, work on a set of predefined rules. This section will cover two which are often used as points of comparison for other prefetching works.

3.1.1 Best-Offset

The Best-Offset prefetcher [23] was proposed by Pierre Michaud and is an offset prefetcher designed for L2 caches. It works by prefetching data at a variable distance from the current memory request, such that if the prefetching distance is set to two, the data placed two addresses ahead is prefetched. This variable distance is determined through a process where the addresses that triggered completed prefetches are stored in a table. From there, the addresses of new requests are subtracted by an offset from a rolling list of predefined offsets. Every time this subtraction results in an address that is present in the stored table, the particular offset that was used receives a point. Once an offset has reached the maximum possible points (which is based on the size of the point counter), or alternatively, the lists of predefined offsets have been looped around a fixed number of times (which is set as a fixed parameter), the winning offset will be set as the current offset for prefetching. The process of finding a new prefetching offset is restarted every time an offset has been chosen, and the scores collected by each offset are reset.

3.1.2 Irregular Stream Buffer

Irregular Stream Buffer (ISB) [24] is an address-correlating prefetcher that was proposed by Akanksha Jain and Calvin Lin. The key novelty introduced with this prefetcher lies within how it manages the storage of correlated addresses. Instead of directly storing them as physical addresses in a purely temporal access order, it introduces a bidirectional mapping mechanism that maps between physical addresses and ones of a separate address space. In this address space, the physical

addresses are given sequential addresses according to their temporal access order. This introduces spatial correlation between the addresses, which the ISB utilizes to make on-chip caching of the most relevant addresses efficient, since it simplifies the eviction of unwanted data. The remaining address correlations that are not cached are stored off-chip in the main memory. In terms of performance, the prefetcher achieved an average speedup of 23.1% on eight memory-intensive benchmarks from SPEC06, while the storage overhead for the tested configuration was 32 KB on-chip cache and 8 MB off-chip in main memory.

3.2 ML Based Prefetchers

ML-based data prefetchers use some ML algorithm trained in either of two ways. If it is trained *offline*, it is pre-trained on memory traces before being used to make prefetches. If it is trained *online*, it learns what to prefetch during the execution of a program. This section will cover four hardware data prefetchers that utilize different ML algorithms, two of which are trained *offline* and two *online*.

3.2.1 Using Multilayer Perceptrons

Asgari et al. proposed a hybrid prefetcher for the LLC named MPMLP [2], consisting of a rule-based prefetcher (Best-Offset) and an original ML prefetcher based on a multilayer perceptron (MLP). The reasoning behind using two prefetchers in tandem was based on two observations made by the authors. The first was that rule-based prefetchers could efficiently prefetch data blocks if the workload exhibits regular access patterns, but they work poorly on workloads with irregular access patterns. The second was that an MLP prefetcher could learn irregular access patterns, but it may be too complex to train effectively for simpler access patterns. Based on these observations, they argued that the two prefetcher types would complement each other.

In the MPMPL prefetcher, both sub-prefetchers issue one prefetch request each for every memory access that reaches the LLC. For each memory access seen by the MLP sub-prefetcher, it makes predictions for which blocks will be accessed in the future by the current PC within two pages. The first is the current page. The second is a potential next page, determined through a table that stores the most recent page transition made from the current page.

MPMLP was tested on 40 traces generated from randomly selected benchmarks from the SPEC06, SPEC17 and GAP benchmark suits. The results showed that this hybrid prefetcher achieved an average 32% IPC increase over a no prefetcher baseline, whereas their MLP prefetcher alone achieved an average of 25%. As MPMLP was designed to enter the ISCA 2021 ML prefetching competition [3], the authors did not need to provide any implementation size calculations/estimations, and all training of the prefetcher was performed *offline*. However, the MLP model used 250 632 trainable parameters with 32-bit precision, which amounts to about 1 MB of storage overhead.

3.2.2 Using LSTM

There have been numerous proposed data prefetchers that use long short-term memory (LSTM) [6, 25–29]. Voyager [6] presented by Shi et al. is one example, which is a hybrid between address correlating and spatial prefetcher for the LLC. Voyager learns *online*, and takes sequences of PCs, memory addresses, and deltas as input to make predictions on future memory access addresses. One of the key challenges that Voyager addresses is the enormous output space caused by making predictions on 64-bit addresses. By splitting the address prediction into two separate predictions using two LSTMs, one which predicts which page will be accessed, and one which predicts which block within that page will be accessed, the LSTMs individual output space was considerably reduced. However, there was also the need to introduce a novel layer that gave the block prediction context clues from the page prediction to avoid poor predictive performance.

In a test of Voyager based on nine irregular benchmarks taken from the SPEC06 and GAP benchmark suits, the results showed an average increase of 41,6% in IPC over a baseline with no prefetching. For comparison, the average increase in IPC was 28.2% for ISB and 13.3% for Best-Offset on the same benchmarks. In terms of model size and inference latency, it was compared to a previously presented prefetcher based on LSTM [25]. The authors stated that Voyager was about 110 to 200 times smaller while reducing training and inference costs by 15 to 20 times. Though, the model was also said to be between five to ten times smaller than ISB, which means that it is in the 0.8 - 1.6 MB range. Hence it is large compared to spatial prefetchers with storage requirements in the order of tens of KB. The authors also stated that Voyager's inference latency is too high to be practical for hardware implementation.

3.2.3 Using SARSA

Pythia [4] presented by Bera et al. is a prefetcher designed for L1/L2 caches formulated as a reinforcement learning (RL) agent, based on the SARSA algorithm [30]. It takes various program context information as input and learns to correlate them to timely and accurate prefetch actions by receiving numerical rewards. Moreover, its reward structure also considers the negative influence prefetching can have on memory bandwidth usage. Hence, it learns not to issue prefetches if the memory resources are constrained. For Pythia, an action is to choose the prefetching offset (from a limited set of offsets) that it predicts it can use to prefetch the following access in the current page. An action can also be not to make a prefetch request.

The authors of Pythia argued that RL is highly suitable for hardware prefetching due to three major reasons, 1) It can adapt its learning in complex state spaces, such as prefetching while considering both the positive and the negative effects that prefetching can have on overall system performance. 2) Practical hardware prefetchers are required to learn *online*, which RL agents do by using the rewards they receive from the environment to improve their decision-making over time. 3) The implementation of it incurs low processing and area overhead.

Testing on a set of traces from five benchmark suites (SPEC06, SPEC17, PARSEC, Ligra, Cloudsuite) showed that Pythia resulted in an average IPC improvement of 22.4% over a baseline with no prefetcher. This was higher than all five spatial prefetchers that the authors used as a comparison, although with higher storage overhead (25.5 KB) than three of them. The corresponding physical size of Pythia was also evaluated, which showed that it would result in a 0.33 mm^2/core area overhead on a 14 nm node. This was stated to be only 1,03% of a desktop quad-core processor. Of the total area overhead, over 90% was consumed by its storage components.

3.2.4 Using Transformers

Zhang et al. proposed a prefetcher [31] for the LLC based on the transformer model [32]. They argued that this model, which excels in NLP problems, also might be a good fit for dealing with the prefetching problem since natural language and memory addresses have a similar grammar. The authors also stated that since this model avoids using loops (unlike for example LSTMs) it is more feasible to implement for parallel execution in hardware, effectively reducing its inference latency.

To handle the input vocabulary of 64-bit addresses, which is much larger than usual for NPL problems, the authors decided to encode the addresses in bit arrays before sending them to the transformer. This reduced the vocabulary to a size of 2, at the expense of a 64-fold increase in input sequence length. The transformer's output is in the form of block indices predicted to be accessed in the future. Thus, the offsets for these blocks are concatenated with their corresponding page address to obtain the physical address used for prefetching.

TransformMAPs performance was tested on a set of 14 traces generated from benchmarks in the SPEC06 and SPEC17 benchmark suites, which showed an average IPC improvement of 20.55%. As this prefetcher was designed to enter the ISCA 2021 ML prefetching competition [3], the authors did not need to provide any implementation size calculations/estimations, and all training of the prefetcher was performed *offline*.

4

Environment

This project used the training and testing environment that was provided for the ML prefetching competition hosted by ISCA in 2021 [3]. The environment is set up such that the ML prefetchers are trained *offline* on the first section of each benchmark, and make predictions in the following section. The length of the training section is determined by the specified number of *warm-up* institutions, while the testing section is determined by the specified number of *simulation* institutions. The core of the testing part of the environment consists of a modified version [33] of a trace-based simulator named ChampSim [34], which models an out-of-order (OoO) processor. The modified version is configured as a single-core processor with no prefetchers at the L1 and L2 caches. More of the accompanying configurations are shown in Table 4.1.

Table 4.1: Configuration of the LLC, main memory, and branch predictor used by the ChampSim fork provided for the ISCA 2021 prefetching competition.

LLC	
Ways	16
Sets	2048
Size	2 MB
Replacement Policy	LRU
Main Memory	
Size	4096 MB
Channels	1
Width	64 bits
Data rate	3200 MT/s
Branch predictor	Hashed Perceptron [35]

The environment also includes two sets of instruction traces, each consisting of 99 traces generated from 25 benchmarks from the SPEC06 [8], SPEC17 [9], and GAP [10] benchmark suites. The first set of traces is referred to as *execution traces*, and contains all instructions generated during each trace generation period. Each of these traces can be passed to a compiled ChampSim binary to simulate an execution sequence of the corresponding benchmark, which will result in an output file with performance statistics. The second set of traces is referred to as *load traces*, and contains only information regarding the instructions that result in memory accesses

seen by the L3 cache. These are used to train ML prefetchers and generate the prefetch files that will be discussed later. Figure 4.1 shows a short section of the first instructions of a *load trace*. Note that the numbers to the far left, the Unique Instruction IDs, do not start at 0 (or 1) and are not sequential. This means that if for example 100 million *warm-up* instructions are used for training, the number of memory accesses that the prefetcher trains on are substantially smaller.

```
14, 152, 28e837c89f00, 14f7c945d230, 0
17, 169, 28e837c8af40, 14f7c945d240, 0
18, 502, 28e837c8c840, 14f7c945d247, 0
19, 726, 28e837c8c740, 14f7c945d24b, 0
22, 727, 28e837c8c7c0, 14f7c945d254, 0
65, 728, 28e837c8c800, 14f7c93863cc, 0
24, 967, 28e837c8d8c0, 14f7c945d264, 0
27, 1200, 28e837c8e8c0, 14f7c945d26f, 0
62, 1329, 28e837c8ae40, 14f7c93863bc, 0
63, 1459, 28e837c89e40, 14f7c93863c3, 0
85, 1746, 28e837c91040, 14f7c934c040, 0
94, 2305, 28e837c92940, 14f7c93dc143, 0
```

Figure 4.1: First 12 rows of load trace "SSSP-5". The numbers from left to right represents: Unique Instruction ID, Cycle Count, Load Address, Program Counter of the instruction which issued the load, LLC hit (1)/miss (0).

Along with the simulator and traces, the competition also provided a framework for how the ML prefetchers were to be implemented, and a Python script that takes five types of console commands as input and performs various operations based on them.

The *build* command compiles five ChampSim binaries with different LLC prefetchers. The first has no LLC prefetcher, and is used as a baseline for the performance metrics computed with the *eval* command. The second uses the Best-Offset prefetcher [23], which has been altered to work for the LLC. The third uses the Irregular Stream Buffer (ISB) prefetcher [24]. The fourth uses a combination of Best-Offset and ISB prefetchers. The final binary uses a prefetch file as an LLC prefetcher, where the contents of the file are generated based on a *load trace* by a prefetcher that follows the competition framework. The contents of the file are in the form of two numbers per row. The first number is the *unique instruction id* (row number in the corresponding *execution trace*) of the instruction from which a prefetch shall be issued. The second number is the physical address of the block to be prefetched. During simulation with this ChampSim binary on an *execution trace*, the simulator checks in the prefetch file if there are any row(s) which *Unique Instruction ID* corresponds with the currently simulated instruction. If it finds any, it will issue prefetch requests for the data at the physical address(es) found on the same row(s).

The *train* command loads a *load trace* that is given to it as an argument and puts all of its rows with a *Unique Instruction ID* $<$ the specified number of *warm-up* instructions into a train set, and sends the set to the prefetcher for training.

The *generate* command loads a trained prefetcher and a *load trace* that is given to it as an argument and puts all of its rows with a *Unique Instruction ID* $\geq$ the specified number of *warm-up* instructions into a test set. The test set is sent to the prefetcher to generate prefetches that are then stored in a prefetch file. This file is later used with the ChampSim binary that can use it as LLC prefetcher.

The *run* command starts the performance simulation using the *execution trace* that is provided as an argument. It can run the simulation for all five ChampSim binaries, or just the four that do not use a file as a prefetcher, or just the one that uses a file as a prefetcher.

The *eval* command processes the outputs from the simulator binaries into a CSV file with the following relevant statistics: Accuracy, Coverage, Misses Per Kilo Instructions (MPKI), MPKI Improvement (%), Instructions Per Cycle (IPC), and IPC Improvement (%).

5
Design

This chapter covers the design of this project's hardware data prefetcher. It starts by covering high-level design decisions in Section 5.1. It then continues with more specific decisions based on performance testing in Section 5.2. Finally, it gives an overview of the final design in Section 5.3.

5.1 Design Considerations

This section will review the design options that have been considered for this project's ML hardware data prefetcher, and motivate design decisions that have been made.

5.1.1 ML Algorithm Selection

In order to select an ML algorithm for prefetching, the first thing to consider is the nature of the data prefetching task. The goal is to based on a triggering event, fetch one or multiple memory blocks that the processor will request in the near future. As there is no information in the raw data available to a prefetcher regarding which trigger should be connected to which block in order to ensure a timely prefetch, some form of external guidance is needed. The guidance can be either labels or rewards, which leaves algorithms within supervised learning and reinforcement learning as possible candidates.

Since part of the project is to test an algorithm that has not been tested in the context of hardware data prefetching (or at least not well documented in a published work), a literature review was performed to find out which algorithms have already been tested. The result of the review was that the following algorithm has already been tested: MLP [2, 36], LSTM [6, 25–29], SARSA [4], Contextual Bandit [37] and the Transformer model [31]. Peled et al. also mentioned that they tried CNNs, but that they were not able to get any stable results with their implementation of them [36]. Thus, they decided not to share those results, nor did they go into detail about their network setup. This left an opening for this project to further explore the use of convolutional networks.

The use of convolutional layers in the context of hardware prefetching is particularly interesting since their sparse connectivity between each layer, along with weight sharing within each layer, leads to a substantially lower number of parameters as

compared to the fully connected layers used in an MLP. This is of high importance since it results in lower storage overhead and thereby a smaller implementation size, which is a critical consideration for hardware implementation. An interesting development in the use of convolutional layers for applications outside of imagery, is the introduction of TCNs. These algorithms have been shown to be excellent performers on sequence modeling tasks, even outperforming recurrent neural networks such as LSTMs [16]. Furthermore, in contrast to LSTMs, which need to process each input sequence sequentially due to their recursive use of their output as part of their input at each time step, TCNs can process each input sequence as a whole in parallel [16]. This should intuitively allow for low latency predictions, which also is a highly important factor in hardware data prefetching. For the reasons previously discussed, a TCN was chosen for this project's prefetcher, more specifically, the TCN algorithm presented by Bai et al. [16].

5.1.2 Classification vs Regression

With the chosen algorithm being a TCN, a supervised learning algorithm, the prefetching problem could be viewed as either a classification or regression problem. Both ways to view the problem of data prefetching present their own set of challenges.

Viewing prefetching as a classification problem, where some input data is classified as a unique absolute address, results in the *class explosion* problem [6]. The output layer of a TCN used for classification consists of a fully connected layer, which grows with the number of classes (labels) it can classify. If, for example, a TCN was trained to classify a set of images based on which of the two animals *cat* or *dog* appears in each image, it would have two output neurons, one corresponding to the probability of each class "Cat" and "Dog". However, in the case of predicting memory addresses, modern high-performance processors use 48 bits to address memory locations [38] [39]. The number of classes needed for block address predictions would therefore be 2^{42} (the six block offset bits are not predicted), which would require equally many neurons in the output layer. Thus, the number of neurons would vastly exceed the available memory to store them and the computational resources available for reasonable training and inference latencies.

A possible solution to the class explosion problem would be not to make predictions on absolute addresses. Instead, each prediction could be made in the form a delta between the currently seen memory address and some future memory access address, such as Hashemi et al. did [25]. This delta would then be added to the current memory address to receive the final predicted absolute address. One would, however, need to put a restriction on the number of the deltas to be considered, as they are in the order of millions in many applications [25]. Another alternative would be just predicting a part of future addresses in the form of an offset, such as the block index of future memory accesses. Each prediction could then be inserted into the block index portion of the currently seen memory address to receive the final predicted absolute address. This approach was taken in by both Asgari et

al. [2] and Zhang et al. [31].

Viewing prefetching as a regression problem, where the addresses are seen as a quantity, would completely avoid the problem of class explosion. However, regression models aim at minimizing the difference between the actual outcome and the model's prediction, which introduces an accuracy problem for use in prefetching. In contrast to problems such as temperature forecasting, where close estimations are sufficient, there are not necessarily any "close enough" when it comes to predicting addresses. Suppose the memory access pattern does not exhibit a constant stride. In that case, a predicted address that is off by just one block numerically will likely result in an equally useless prefetch as if the prediction was off by 10000 blocks. In both cases, the prefetched block will likely either just remain unused, or in the worst case, might evict another cache block that might have been useful, resulting in degraded performance.

As the approach of treating prefetching as a classification problem and predicting block indices seemed to require the least amount of resources and give good results [2, 31], this approach was chosen for this project.

5.2 Design Space Exploration

With the ML algorithm selected to be a form of TCN, and the predictions being in the form of labels for future block indices, the rest of the prefetcher's design was derived from performance testing. This section will first present the two main types of input data tested with the TCN, and show the performance results used to decide which of them to use in the final design. The TCN configuration values presented in Table 5.1 were used for these tests, which were chosen to be in the ranges of recommended values in [40]. This section will further cover tests with tweaked TCN configuration values and show their effect on prefetching performance and the number of used parameters.

Table 5.1: The base configuration of the TCN model is used in this section.

Setting	Value
Residual blocks	1
Kernel size	6
Filters	20

All tests in this section were conducted with the prefetch degree set to 1. Furthermore, the TCNs receptive field was set using formula 2.8 to be as small as possible while still covering each sequence of input data. The number of warm-up instructions was set to 100 million to give ample room for learning. To keep the number of simulation instructions consistent across all tests, they were always set to 200 million. The reason for this number of simulation instructions was that the GAP traces provided by the ISCA competition were 300 million instructions long.

5.2.1 Two Prefetcher Versions

Of the five types of metadata available in the ISCA competitions load traces, two types could be used as a basis for predictions. The first was the physical address of each memory access, and the second was the PC of the instruction which issued it. Based on this metadata, two versions of TCN prefetchers were tested. The first version, hereafter referred to as the Page version, used a concatenation between the physical page address and block index of each observed memory access as input. This version was aimed at finding patterns in the order of block indices accessed within a given physical page. The second version, hereafter referred to as the PC version, used a concatenation between the PC and block index of each observed memory access. This version was aimed at finding patterns in the order of block indices accessed by each PC.

5.2.1.1 Correlation Distance

With a supervised machine learning model such as a TCN, the training consists of presenting the algorithm with an input and an expected output (label) that corresponds to the desired classification. However, there are no "true" labels within data prefetching to be used for training. The prefetcher can be set to learn correlations between accesses spaced apart by an arbitrary distance in the access stream. An example of this is shown in Figure 5.1.

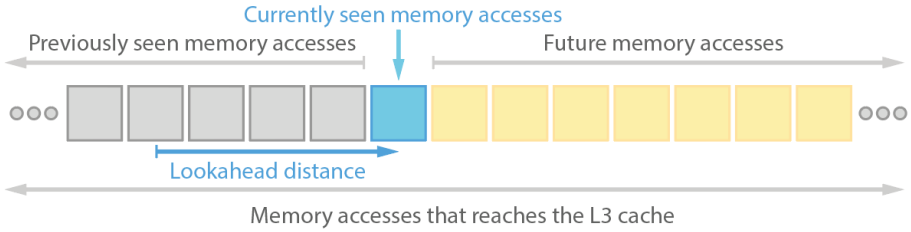

Figure 5.1: Each block represents a memory access. Here a lookahead distance of 4 is used to learn correlations between an access observed four steps in the past and the currently seen access.

Though, the chosen relative distance (lookahead distance) is important since it affects the timeliness of each prefetch. The reason is that once the same information about a memory access reappear in the future, the TCN will trigger a prefetch with the same lookahead distance, as shown in 5.2.

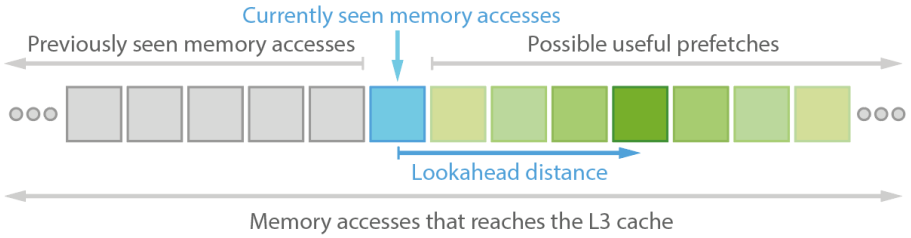

Figure 5.2: The future memory accesses are colored in a deeper green color the more timely a prefetch for that block would be.

Depending on whether the prefetcher makes correlations between memory accesses in the global L3 memory access stream, or correlations between parts of the stream that share some type of feature (for example, issued by the same PC), the optimal lookahead distance might be different. For instance, if the lookahead distance is set to 4 for a prefetcher that learns correlations in the global access stream shown in Figure 5.3, then an accurate prefetch issued from instruction 526563 targets the data that will be requested by instruction 526710 (the data located at address 28e837cf4980). There are only 146 instructions between these two instructions, and thus the lookahead distance of 4 is likely too short to cover the whole memory access latency.

```
526563, 583669, 28e837cf0600, 8bd695, 0
526686, 584143, 28e837cee400, 8bd938, 0
526690, 584146, fdfd3a8c7080, 8bd94f, 0
526720, 584147, 28e837cf2080, 8bd9ef, 0
526710, 584150, 28e837cf4980, 8bd9ba, 0
```

Figure 5.3: Five consecutive accesses found in the global L3 memory access stream of trace 607.cactuBSSN-s0.

However, this is not the case for a prefetcher that observes the corresponding physical page address stream using a lookahead distance of 4, shown in Figure 5.4. An accurate prefetch issued from instruction 526563 targets the data (located at address 28e837cf0680) that will be requested by instruction 614595. There are 88031 instructions between these instructions. Thus, the lookahead distance of 4 is likely enough to cover the whole memory access latency (or too long, and the fetched data might evict a useful block in cache).

```
526563, 583669, 28e837cf0600, 8bd695, 0
548547, 605718, 28e837cf00c0, 8bd613, 0
570579, 627891, 28e837cf0640, 8bd695, 0
592563, 649809, 28e837cf0100, 8bd613, 0
614595, 671821, 28e837cf0680, 8bd695, 0
```

Figure 5.4: Five consecutive accesses seen in the L3 physical page address memory access stream of trace 607.cactuBSSN-s0. The physical page address is highlighted in yellow.

The same argument holds for a prefetcher that looks at the corresponding PC memory access stream shown in Figure 5.5. An accurate prefetch issued from instruction 526563 with a lookahead distance of 4, targets the data (located at address 28e837cf0700) that will be requested by instruction 702627. There are 176063 instructions between these instructions, and thus the lookahead distance may or may not be suitable.

```
526563, 583669, 28e837cf0600, 8bd695, 0
570579, 627891, 28e837cf0640, 8bd695, 0
614595, 671821, 28e837cf0680, 8bd695, 0
658611, 715891, 28e837cf06c0, 8bd695, 0
702627, 760658, 28e837cf0700, 8bd695, 0
```

Figure 5.5: Five consecutive accesses seen in the L3 PC memory access stream of trace 607.cactuBSSN-s0. The PC is highlighted in yellow.

The number of instructions between each access in a memory access stream can also vary significantly between programs, and even between parts of a single program. An optimal lookahead distance can therefore be obtained by making it so that it changes based on some type of performance metric during execution. However, the prefetcher in this project is trained *offline*, so there is no performance feedback available to change the lookahead distance automatically on. Instead, the lookahead distances that on average resulted in the highest IPC improvement for each version were found through testing. Figure 5.6 and Figure 5.7 show the results from the tests. The results show that the lookahead distance of 3 was best on average for both versions, although it was only a difference of 0.16 % between lookahead distances 2 and 3 for the PC version.

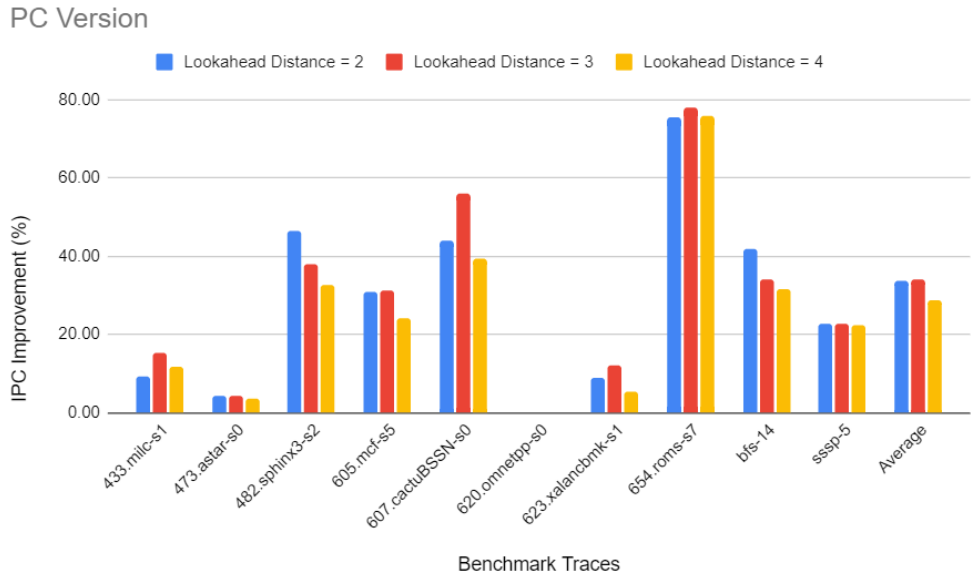

Figure 5.6: IPC improvement over a no prefetcher baseline for the PC version. Each of the ten tested traces is from the benchmark whose name is presented before the dash sign, and the number after the dash sign indicates which specific trace. The results for *620.omnetpp-s0* are not visible since the improvements were almost non-existent.

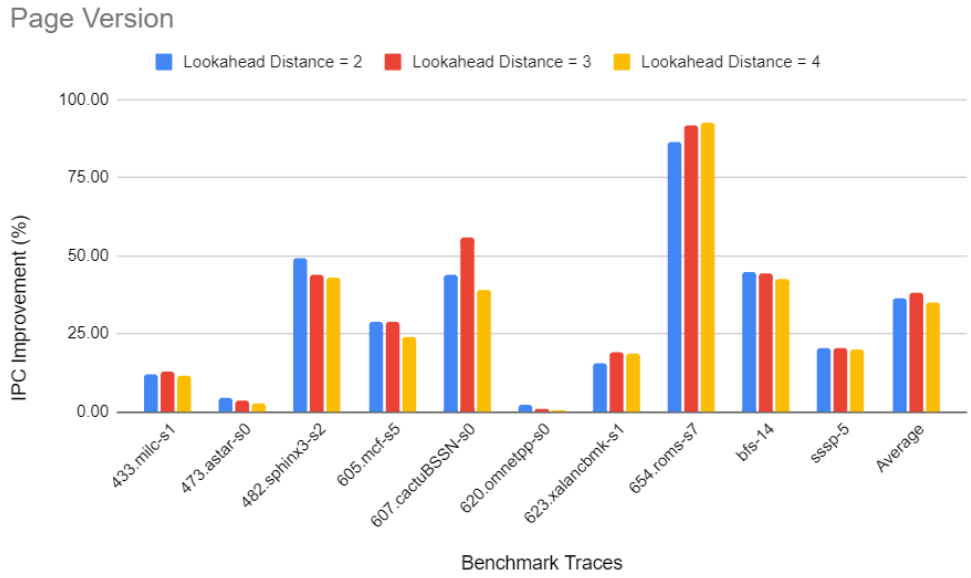

Figure 5.7: IPC improvement over a no prefetcher baseline for the Page version.

5.2.1.2 Choice of version

The results in Figure 5.8 suggest that the Page version is better on average than the PC version in terms of performance. Though, in the benchmark traces where the page version performs better than the PC version, a rule-based and low overhead prefetcher such as the Best-Offset beats both versions (aside from in *654.roms-s7*). In contrast, the PC version is slightly better than the Page version in *473.astar-s0*, *605.mcf-s5* and *sssp-5*, where the Best-Offset prefetcher brings little or no performance improvement.

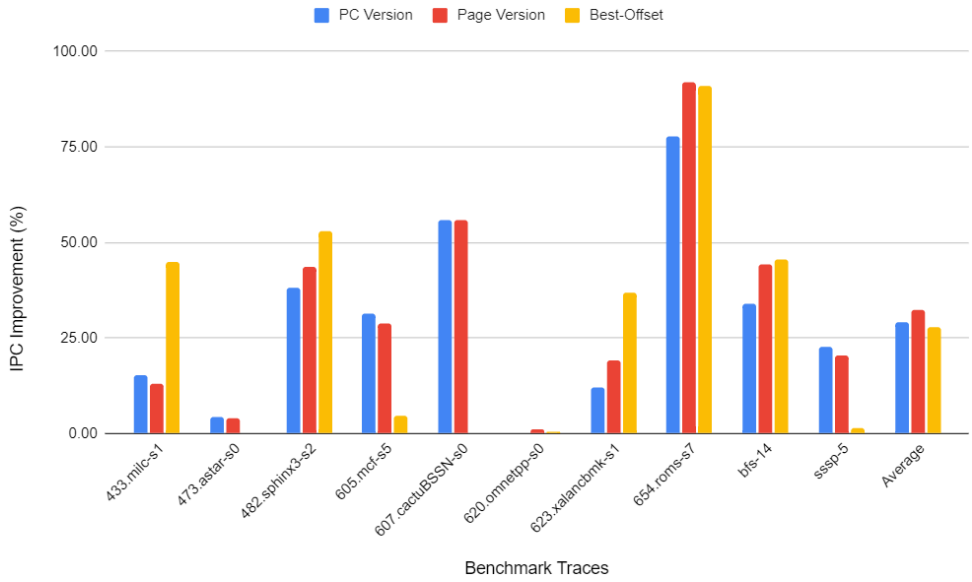

Figure 5.8: IPC improvement over a no prefetcher baseline for both PC and Page versions, using a lookahead distance of 3. Results for the Best-Offset prefetcher are shown as a point of comparison.

As the PC version would work better as a complement to an offset prefetcher such as Best-Offset, this version was chosen for the final design of the TCN prefetcher.

5.2.2 Minimizing Parameter Overhead

Since a primary goal of this project was to keep the size overhead of the proposed prefetcher low, the remaining tweaks to the TCNs configuration were targeted to minimize its use of parameters, with potential trade-offs to the prefetcher's performance.

5.2.2.1 Reducing the Number of Filters

The filters in a TCN are the units that learn patterns in the input data. A reduction in the number of filters reduces the number of unique patterns the TCNs can learn, but also has the most significant impact on reducing the number of parameters it uses. For this reason, multiple filter values were tested to find a sweet spot. Figure 5.9 shows the performance results while Table 5.2 shows the number of parameters. The reduction from 20 to 10 filters brought slight reductions in some benchmark traces, but overall the results stayed mostly the same. The only outlier is *654.roms-s7* which saw an increase in performance. This is probably due to the PCs in this trace mostly moving with a constant stride of one, and by using fewer filters, only the most dominant access patterns were captured. At 8 filters, most of the performance was still intact, while multiple benchmarks performed significantly worse at 5 filters. For this reason, it was decided to reduce the number of filters from 20 to 8.

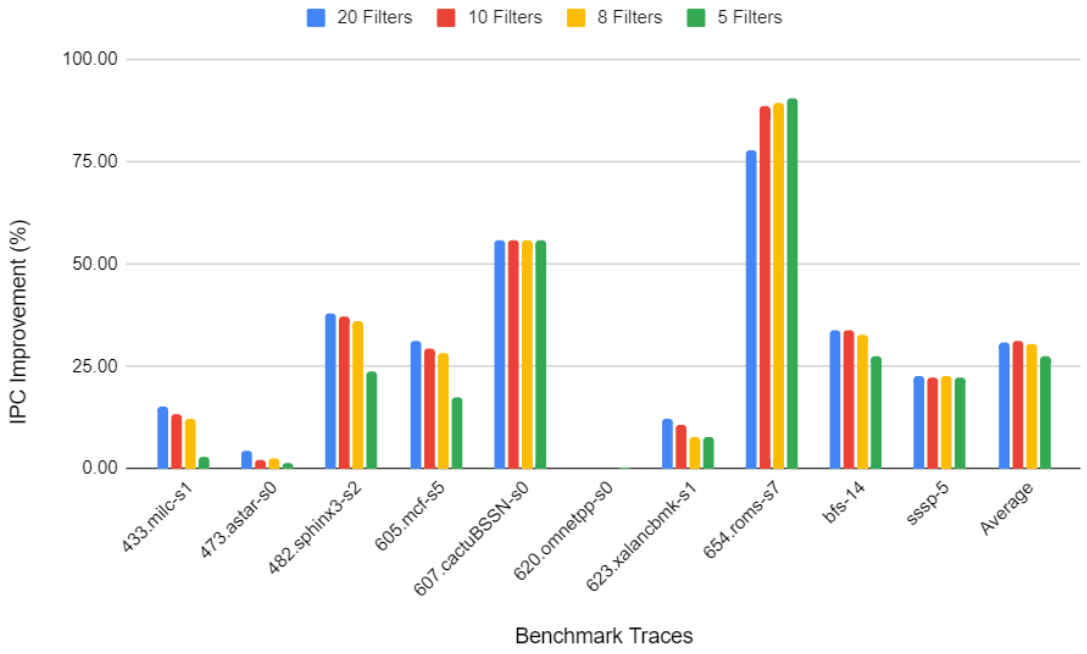

Figure 5.9: Performance comparison of the TCN prefetcher using four different numbers of filters.

Table 5.2: The number of parameters used by the TCN prefetcher for four different numbers of filters.

Filters	Parameters
20	25990
10	7030
8	4678
5	2050

5.2.2.2 Reducing the Receptive Field

Another way to reduce the number of parameters used by the TCN is to reduce the required receptive field needed to cover each input sequence. Since each input sequence consists of a PC (48 bit) and block index (6 bit), the length of each input sequence is 54 bits. It is not possible to reduce the number of block index bits used as part of the input, since they are all needed for the TCN to find patterns between the block index values. On the other hand, from looking at Figure 5.10, it is clear that the number of used PCs is low, and that it might be fine to reduce the number of most significant bits used by the TCN to distinguish them. This is certainly the case for all traces provided from SPEC benchmark suits, since they only use the low-order 24 bits of the PC. It should also be the case for the traces provided from the GAP benchmark suite, which use all 48 bits. In the worst case, there could be multiple PCs that share the same low-order PC bits that also have different block access patterns, making the TCN unsure which block to predict. However, leaving sufficiently many PC bits should make this risk minimal. It was therefore decided to reduce the number of PC bits as input to the TCN by half (keeping only the 24 low-order bits) since it still gives room for around 17 million possible unique PCs.

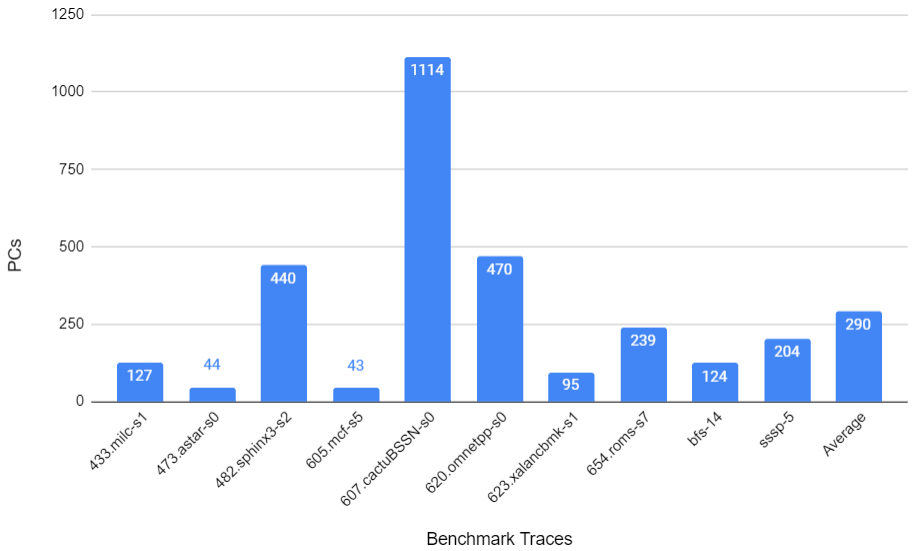

Figure 5.10: The number of unique PCs found in each of the ten tested benchmark traces.

In order to cover the new length of the input sequence, the TCNs receptive field needs to be at least 30 (24-bit PC + 6-bit block index). The cheapest way to achieve this in terms of parameters is to use one residual block with a kernel size of 2 and four dilations (1,2,4,8), resulting in a receptive field of 31. The next cheapest way, which also results in a receptive field of 31, is to use a kernel size of 6 with two dilations (1,2). Both of these options were tested, and the results are shown in Figure 5.11. As can be seen, the version with kernel size 2 slightly outperformed the version with kernel size 6 on average. However, the kernel size 6 was better in the benchmarks where the Best-Offset prefetcher brought little or no performance improvement. For this reason, and the fact that the difference in parameters was small, as shown in Table 5.3, the version with kernel size 6 was selected.

Table 5.3: The number of parameters used by the TCN for two different kernel sizes.

Kernel size	Parameters
2	2616
6	3092

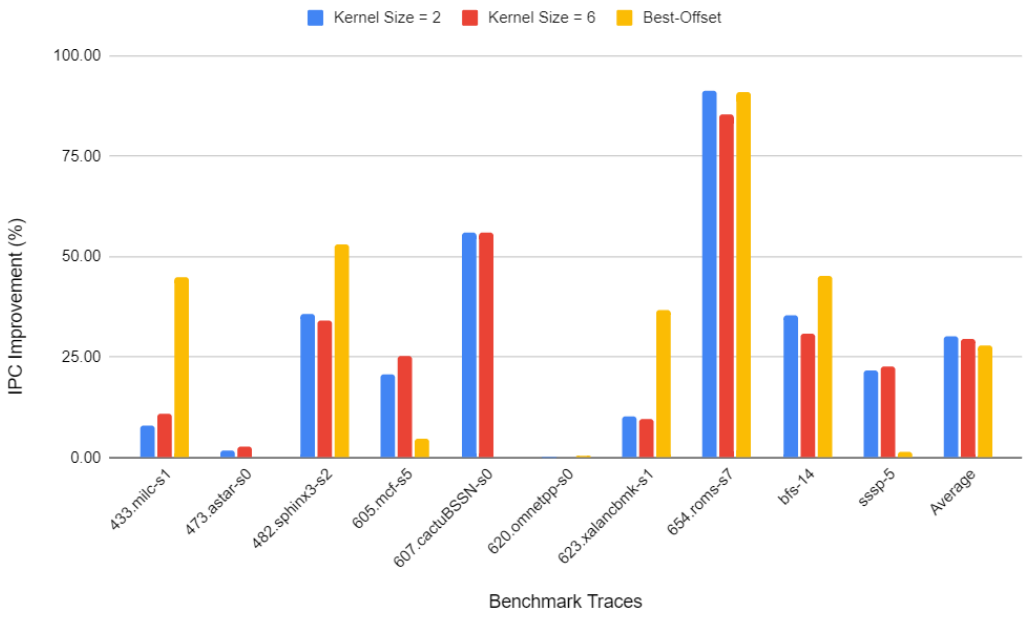

Figure 5.11: IPC improvement for the TCN prefetcher using two kernel sizes. The Best-Offset prefetcher is shown as a comparison.

5.2.2.3 Restricting Queues

Although this project's prefetcher was trained *offline*, it would need to be able to learn *online* if it were to be implemented into hardware. This would imply a restriction on the number of queues the prefetcher can use during training to keep track of each PC's separate access stream. Therefore, to obtain a more correct storage overhead estimation of the prefetcher in Section 6.3, the number of queues was restricted. To re-allocate queues for new PCs if all of them were already occupied, the Least Recently Used replacement policy was used.

The result shown in Figure 5.12 indicates that around 180 queues were needed to achieve full performance in the trace *607.cactuBSSN-s0*, which has the largest number of PCs of the ten benchmark traces tested in this section. In order to accommodate benchmarks with potentially larger amounts of actively used PCs, it was decided that 500 queues would be used in the final design.

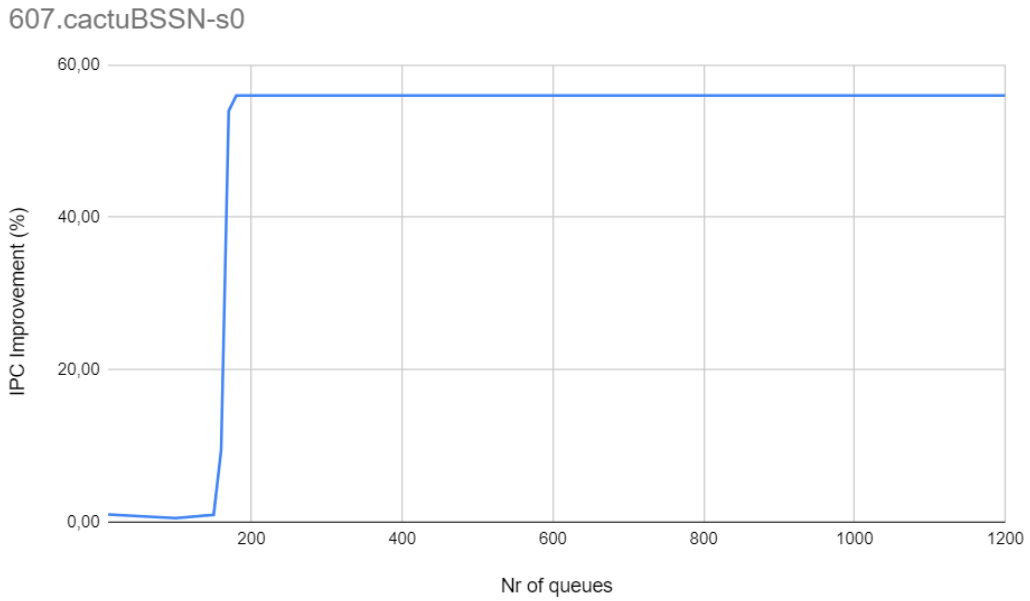

Figure 5.12: IPC improvement over a no prefetcher baseline, using multiple queue quantities.

5.3 Prefetcher Design

The prefetcher proposed in this project consists of two major components. The first component is a TCN used for predictions on which block indices will be accessed in the future by a given program counter. The TCN is based on the one presented by Bai et al. [16], with a receptive field of 31, using one residual block with 8 filters of size 6 x 1, and dilations $d_1 = 1$ and $d_1 = 2$. The TCN is connected to a fully connected output layer with 64 neurons used for the classification of numbers between 0-63, which corresponds to every possible block index for a 4096-byte page with 64-byte blocks. The second component is a queue system that stores the block indices from the last four memory accesses made by the 500 most recently seen PCs. Figure 5.13 shows an overview of the prefetcher's operation. The raw data that the prefetcher takes as an input is the PC and physical address associated with each memory access seen by the LLC (the memory accesses seen are filtered through the L1 and L2 caches).

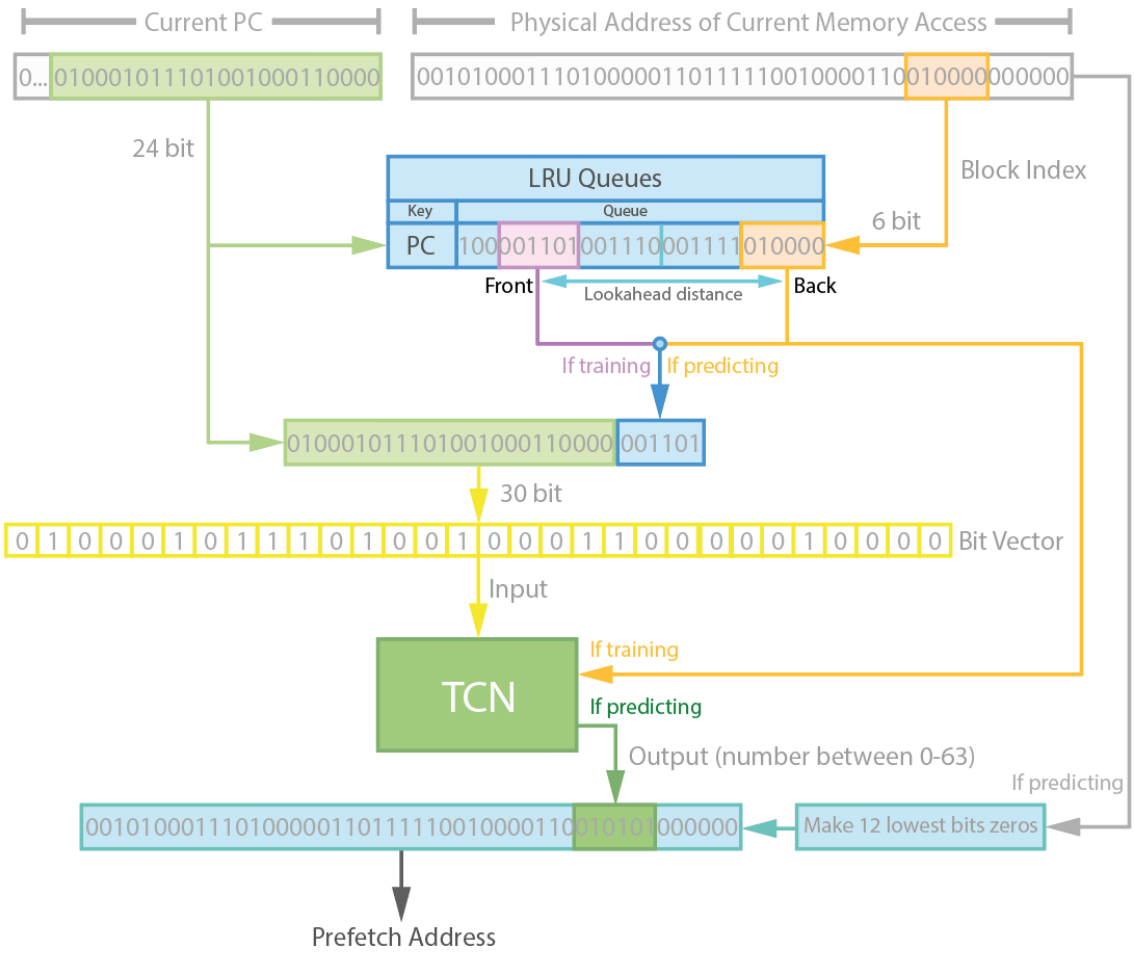

Figure 5.13: Overview of the TCN prefetcher, with example input and output.

For the memory access currently seen by the prefetcher, the low-order 24 bits of the PC are used as part of the input data to the TCN. Furthermore, these bits are also

used as the *key* for the queue system. If the *key* is not already connected to a queue, it will be connected to one. If all queues already have a *key* connected to them, the queue with the least recently used (LRU) *key* will be cleared, and the new *key* will be connected to that queue. Each queue consists of a 27-bit number divided into five sections. The first section is 3 bits long and is used as a counter from 0 to 4, which keeps track of whether the queue is full or not. The remaining 24 bits are split into four equally sized 6-bit sections (queue slots) used to store the four last block indices accessed (including the current one) by the corresponding *key* (current PC).

The block index bits of the current physical address are entered into the back of the currently active queue, and the contents of the queue are shifted one step towards the front before each new entry to it. This results in the queue being filled after four accesses by its corresponding PC, and for each following access, the block index in the front of the queue gets discarded. The remaining operation of the prefetcher depends on which *mode* it is in.

In *training* mode, which requires a filled queue associated with the current PC, the 6 bits of the block index stored in the front of the queue are concatenated with the low-order 24 bits of the current PC. The resulting 30-bit number is transformed into a bit vector of length 30 (each bit receives its own index in an array), which is the final input to the TCN. In this mode, the back of the queue (current block index) is used as a label for the input. Through this process, the TCN learns correlations between the block index that was accessed three accesses in the past by the current PC and the block index of the currently seen access by the same PC.

In *prediction* mode, the 6 bits of the current block index are concatenated with the low-order 24 bits of the current PC. The resulting 30-bit number is transformed into a bit vector of length 30, which is the final input to the TCN. In this mode, the TCN performs predictions on which block index will be accessed three steps in the future by the current PC. The predictions are in the form of scores (probabilities) from each neuron in the output layer of the network, each indicating how confident the network is that the corresponding block index will follow in the PC memory access stream. Depending on the prefetching degree (d) (see 2.3.3), the blocks with the d highest scores are selected to be used for prefetching requests. The predicted block indices are each inserted into the block index portion of a copy of the currently seen physical address, which has had all of its page offset bits (low-order 12 bits) set to zero. The resulting address(es) is/are finally used to issue prefetch requests for the data on that address/those addresses.

6
Results

This chapter covers performance results and storage overhead of the final version of this project's prefetcher. Other prefetchers' performance and storage overhead are also presented for the sake of comparison.

6.1 Methodology

All performance testing in this chapter was performed using 100 million warm-up instructions and 200 million simulation instructions, and all metrics are relative to a no prefetcher baseline. The traces used for testing were provided by the ICSA 2021 prefetching competition [3]. Those that start with four are from the SPEC06 benchmark suite, while those that start with six are from the SPEC17 benchmark suit. Finally, those that do not begin with a number are from the GAP benchmark suit. The presented results are based on the average of three traces from each benchmark, besides for *620.omnetpp* which is based on the average of the only two traces which were provided for that benchmark. The results for each individual trace can be viewed in Appendix A.

From here on, this project's prefetcher is referred to as the TCN prefetcher, or simply TCN, which was implemented using the *compiled_tcn* from the Keras TCN library [40]. The performance of the TCN prefetcher is compared to four other prefetchers. The first is a form of offset prefetcher, which issues prefetch requests for the data that lies on addresses at a fixed distance from each observed access. This prefetcher will be referred to as Fixed-Offset, which at degree 1 (see Section 2.3.3) uses a distance of 3, while at degree 2, it uses distances 2 and 3. The second prefetcher is the Best-Offset prefetcher [23], which won the 2nd Data Prefetching Championship [41]. The third is the Irregular Stream Buffer (ISB) prefetcher [24]. The final prefetcher is the MLP sub-prefetcher from the MPMLP prefetcher [2], which placed second in the ISCA 2021 prefetching competition (the entry that won did not use ML). For a fair comparison, only the MLP sub-prefetcher of the MPMLP prefetcher is used, since otherwise it would mix its results with its second sub-prefetcher, the Best-Offset prefetcher. The two ML prefetchers tested in this section, TCN and MLP, were trained *offline* with a batch size of 256 on the warm-up instructions during a single epoch of each benchmark. The Fixed-Offset prefetcher does not perform any kind of learning, and the Best-Offset prefetcher and ISB prefetcher learn *online*.

6.2 Performance Testing

This section will first cover results where each prefetcher issued prefetch requests at degree 1, followed by results where each prefetcher issued prefetch requests at degree 2. For information about what prefetching degree is, see Section 2.3.3.

6.2.1 Prefetch Degree 1

Figure 6.1 shows a performance comparison between the five tested prefetchers, across 15 memory-intensive benchmarks. The benchmarks were chosen to represent both regular and irregular accesses of varying degrees. The results show that the TCN prefetcher brought the largest improvement of the five prefetchers in the *605.mcf* and *sssp* benchmarks, however, only by slight margins. The TCN prefetcher also brought the largest improvement in *607.cactuBSSN*, together with the Fixed-Offset prefetcher, and the second-largest performance gain in *654.roms* and *cc*, behind the Fixed-Offset prefetcher. The benchmarks that proved to be most challenging for all but one of the prefetchers were *473.astar* and *620.omnetpp*, resulting in no noticeable change in performance. The only prefetcher that significantly improved these highly irregular benchmarks was the address-correlating prefetcher, ISB.

Overall, the results show that the two offset prefetchers performed best on average in these tests, and that the only benchmark where the two ML-based prefetchers (TCN, MLP) simultaneously outperformed the rule-based prefetchers was in *605.mcf*.

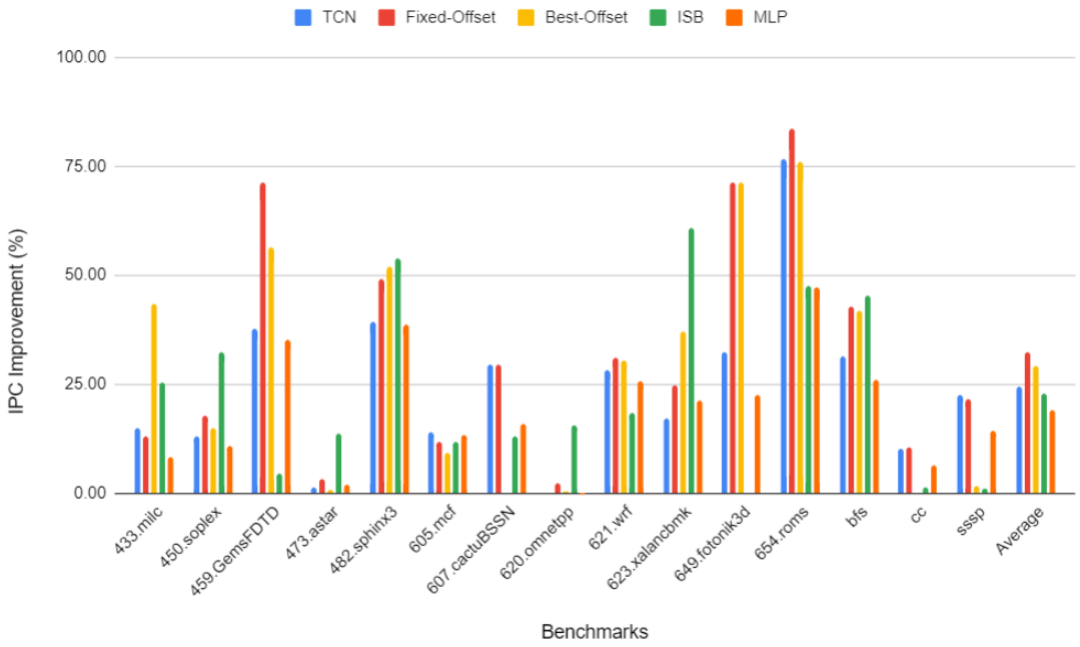

Figure 6.1: IPC improvement, as defined by equation 2.5. Invisible bars means no improvement.

Figure 6.2 shows the improvement in the number of cache misses due to each prefetcher. As can be seen, the TCN prefetcher significantly reduced the number of

cache misses in most benchmarks. An interesting observation is that while both the TCN prefetcher and Fixed-Offset prefetcher brought a similar IPC improvement in the cc benchmark, the TCN prefetcher provided the largest reduction to the number of cache misses, while the Fixed-Offset prefetcher brought almost no reduction. However, as shown in Figure 6.3 and Figure 6.4, the Fixed-Offset prefetcher had both higher accuracy and coverage than the TCN prefetcher in this benchmark. The results also show that the simple Fixed-Offset prefetcher brought the largest average reduction to misses in these tests, followed by the TCN prefetcher.

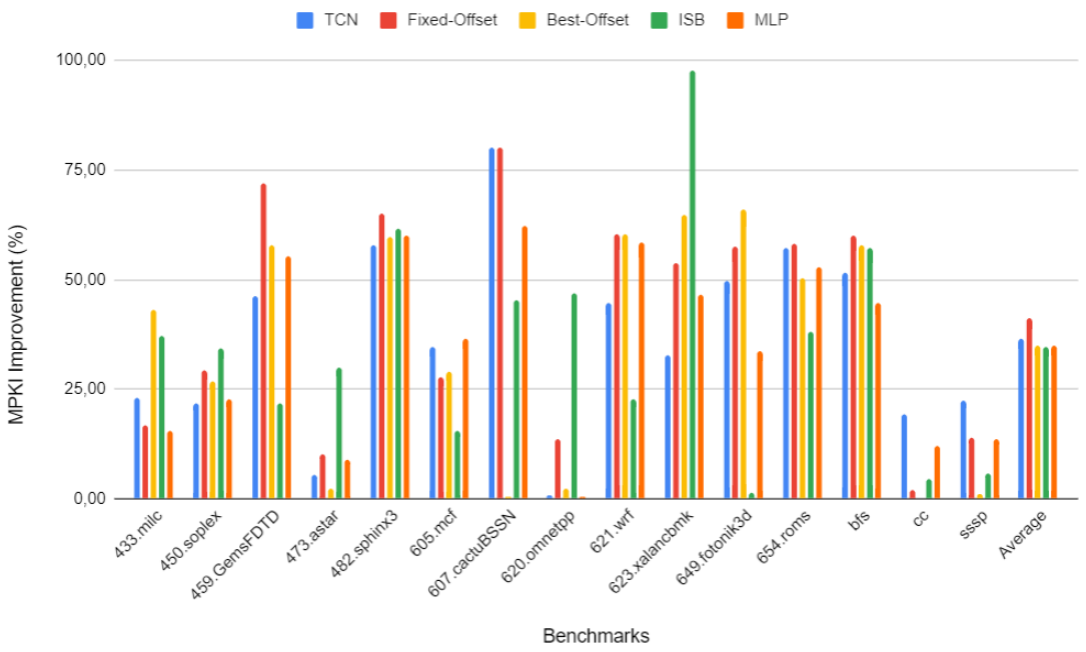

Figure 6.2: MPKI improvement, as defined by equation 2.6. Invisible bars means no improvement.

Figure 6.3 shows the five prefetchers accuracy on the tested benchmarks. The TCN prefetcher achieved its highest accuracy in 607.cactuBSSN, 654.roms, and 621.wrf. In contrast, it achieved its lowest accuracy in 620.omnetpp and cc. The results also show that the TCN prefetcher achieved the highest accuracy of the five prefetchers in 605.mcf. Overall all, the ISB prefetcher proved superior to the rest of the prefetchers in this metric, surpassing 78 % accuracy on average.

Figure 6.4 shows the coverage obtained by the tested prefetchers in each benchmark. The TCN prefetcher achieved its highest coverage in 607.cactuBSSN, 649.fotonik3d and 654.roms, and also achieved comparatively good coverage in 605.mcf and sssp. On the other end of the spectrum, it reached its lowest coverage in 620.omnetpp. In terms of average coverage, the Fixed-Offset prefetcher performed best in these tests.

A summary of the average metrics on the 15 tested benchmarks is presented in Table 6.1. The TCN prefetcher placed third in IPC improvement, second in MPKI improvement, fourth in accuracy, and third in coverage.

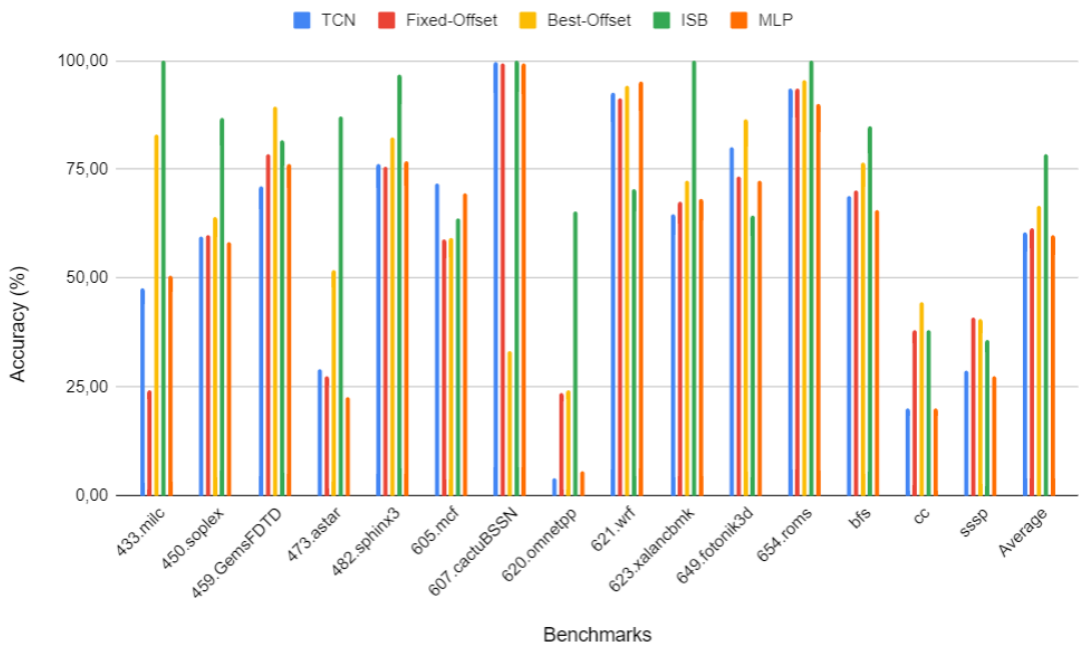

Figure 6.3: Accuracy, as defined by equation 2.2. Invisible bars means no accuracy.

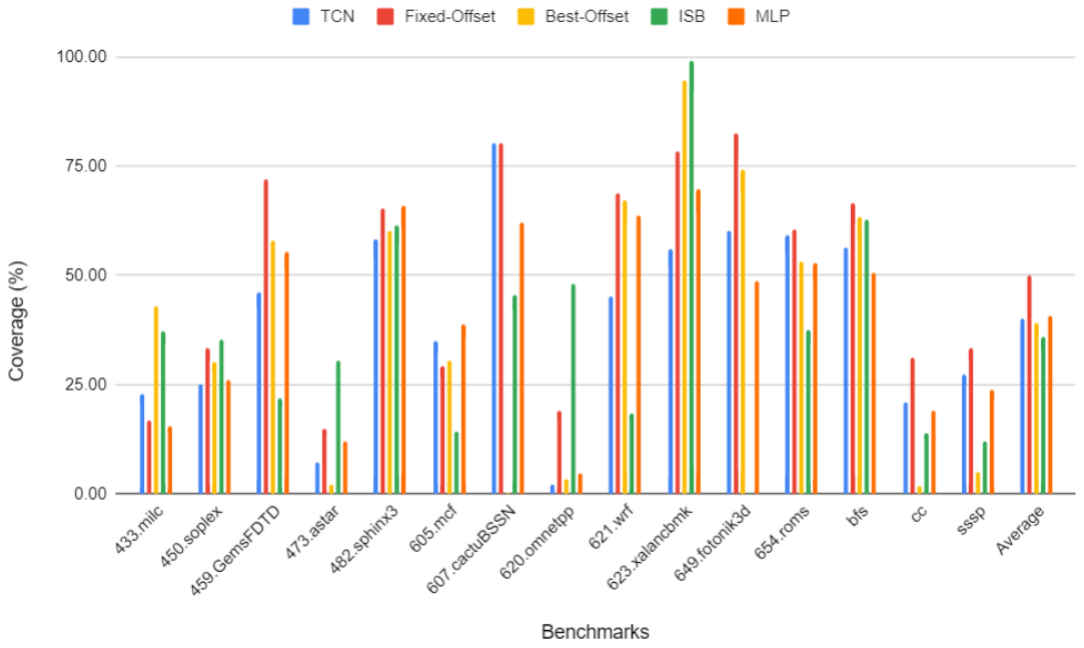

Figure 6.4: Coverage, as defined by equation 2.1. Invisible bars means no coverage.

Table 6.1: Overview of the average IPC improvement, MPKI improvement, Accuracy and coverage, for five tested prefetchers.

Prefetcher	IPC Imp (%)	MPKI Imp (%)	Accuracy	Coverage
TCN	24.53	36.48	60.52	40.16
Fixed-Offset	32.24	41.38	61.55	50.11
Best-Offset [23]	29.04	34.80	66.52	39.15
ISB [24]	22.94	34.67	78.36	35.85
MLP [2]	19.16	34.91	59.89	40.62

6.2.2 Prefetch Degree 2

Up until this point, all tested prefetchers have issued prefetches at degree 1, which showcases the prefetches at their least aggressive prefetch setting. The used testing environment did however allow for prefetching up to degree 2, which was also the default setting on the provided Best-Offset and ISB prefetchers from the ISCA competition. In order to see how all prefetchers' performance would change by increasing the prefetch degree to 2, this was also tested, and the results are presented in this subsection.

Figure 6.5 shows a performance comparison on the same 15 benchmarks used in Section 6.2.1. Overall, every prefetcher increased their average IPC improvement due to the increased prefetch degree. The MLP prefetcher saw the largest increase with 10,9 %, followed by Best-Offset: 7.15 %, TCN: 5.97 %, ISB: 4.37% and Fixed-Offset: 2.85 %.

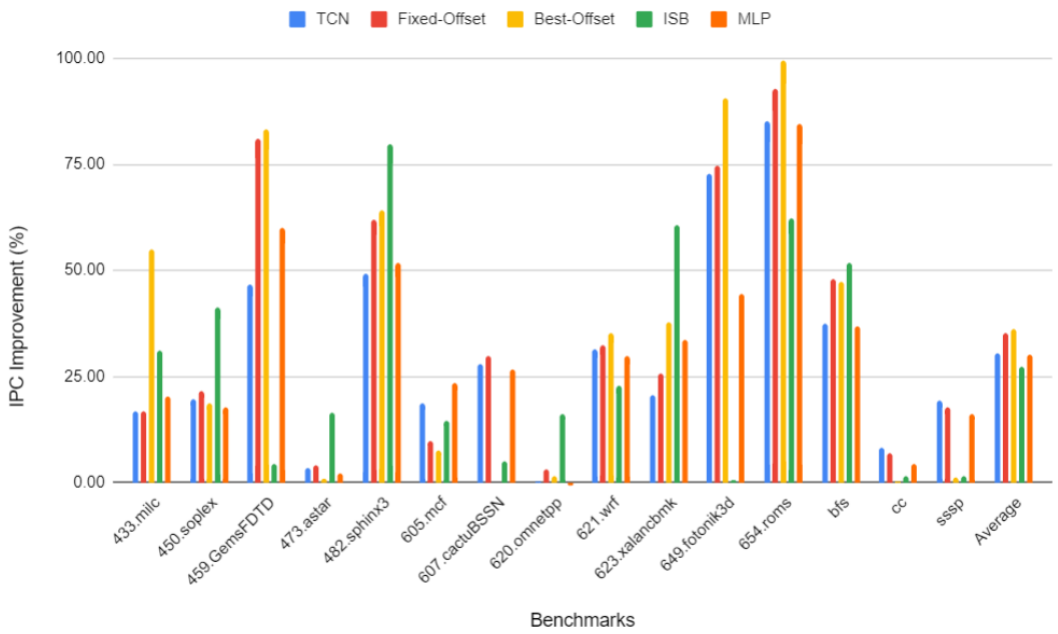

Figure 6.5: IPC improvement, as defined by equation 2.5. Invisible bars means no improvement.

The benchmark where the TCN prefetcher made the most significant gains was *649.fotonik3d*, with a more than doubling from 32.22 % to 72.75 %. The only two benchmarks which did not benefit from the increased prefetch degree of the TCN prefetcher were *607.cactuBSSN* and *cc*, which decreased by 1.49 % and 1.94 % respectively. The degree increase also changed which benchmarks the TCN prefetcher performed best on of all the tested prefetchers. Due to the Fixed-Offset prefetcher losing more performance than the TCN prefetcher in *cc*, the TCN prefetcher ended up on top of this benchmark. In contrast, the MLP prefetcher made an even larger gain than the TCN prefetcher in the *605.mcf* benchmark, which made the TCN prefetcher end up in second place in this benchmark.

Figure 6.6 shows the improvement in the number of cache misses due to each prefetcher. The results show that the switch from degree 1 to 2 was most beneficial for the MLP prefetcher in this metric, with an increased average MPKI improvement by 9.49 %, followed by TCN: 6.65%, ISB: 5.76% Best-Offset: 5.63, % and Fixed-Offset: 1.37 %. These changes also place the TCN and MLP prefetcher slightly above the Fixed-Offset prefetcher in average MPKI improvement.

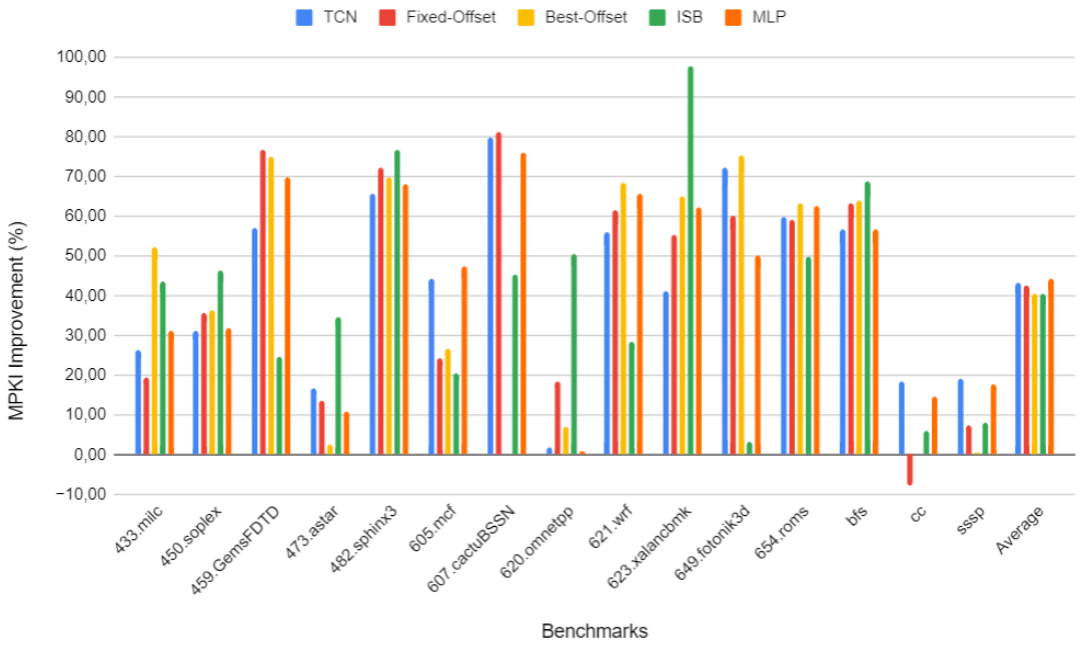

Figure 6.6: MPKI improvement, as defined by equation 2.6. Invisible bars means no improvement.

Figure 6.7 shows the five prefetchers' accuracy on the tested benchmarks. The increase in prefetch degree impacted the Best-Offset prefetcher the most, with an average reduction of 5.44 %, followed by TCN: 4.78 %, Fixed-Offset: 4.42 %, MLP: 3.1 % and ISB: 2.31 %. The worst affected benchmark for the TCN prefetcher was *432.milc* and *bfs*. Interestingly, the increased prefetch degree had the opposite effect on the 473.astar benchmark for the TCN prefetcher. Here it saw a very modest increase of 0.6%, while all other prefetchers saw a decrease. Overall, due to the TCN prefetcher dropping more accuracy than the MLP prefetcher with the increase in degree, the TCN prefetcher fell slightly behind the MLP prefetcher and ended up

in last place.

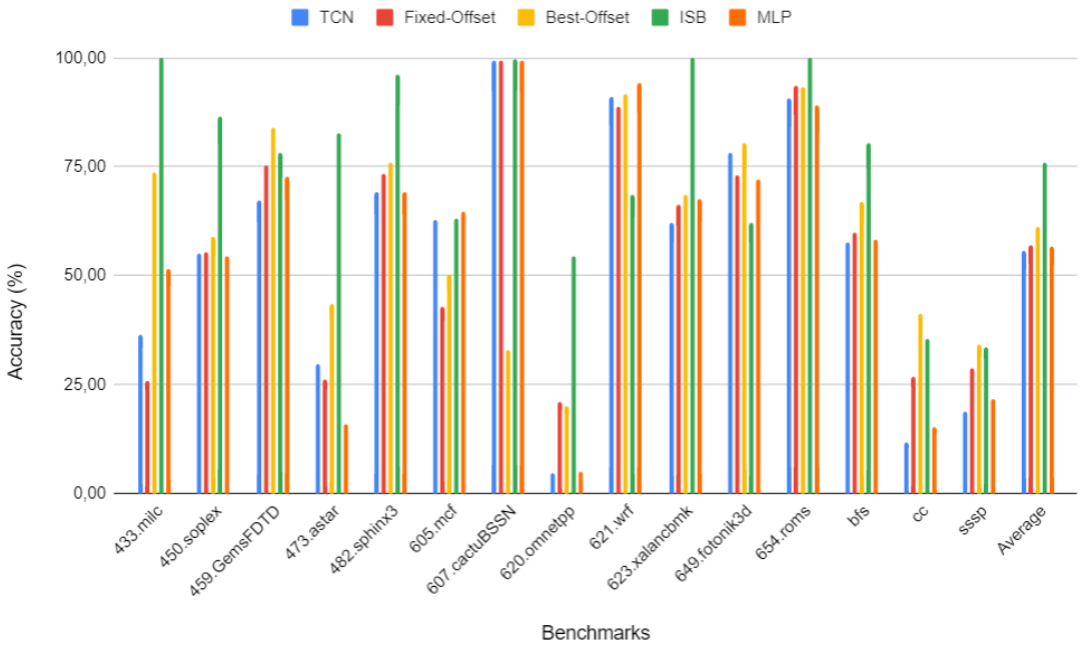

Figure 6.7: Accuracy, as defined by equation 2.2. Invisible bars means no accuracy.

Figure 6.8 show the prefetchers coverage in each of the tested benchmarks. The MLP prefetcher benefited the most from the increased prefetch degree, increasing its average coverage by 10.98 %, followed by TCN: 8.34 %, Best-Offset 6.72 %, ISB: 5.74 %, Fixed-Offset: 3.36 %. The benchmarks where the TCN prefetcher saw the most significant gains in coverage was the *649.fotonik3d*, *621.wrf* and *473.astar*. Both of the TCN and MLP prefetchers also saw a major increase in the *605.mcf*, with 10.04% and 12.3 % respectively, further increasing their lead in that benchmark. On the whole, the Fixed-Offset prefetcher retained its lead over the other prefetchers in coverage, though it is smaller at degree 2 than it was at degree 1.

A summary of the average metrics on the 15 tested benchmarks is presented in Table 6.2. The TCN prefetcher placed third in IPC improvement, second in MPKI improvement, fifth in accuracy, and third in coverage.

Table 6.2: Overview of the average IPC improvement, MPKI improvement, Accuracy and coverage, for five tested prefetchers.

Prefetcher	IPC Imp (%)	MPKI Imp (%)	Accuracy	Coverage
TCN	30.50	43.13	55.72	48.50
Fixed-Offset	35.09	42.75	57.13	53.47
Best-Offset [23]	36.19	40.43	61.08	45.87
ISB [24]	27.31	40.35	76.05	41.59
MLP [2]	30.06	44.40	56.79	51.60

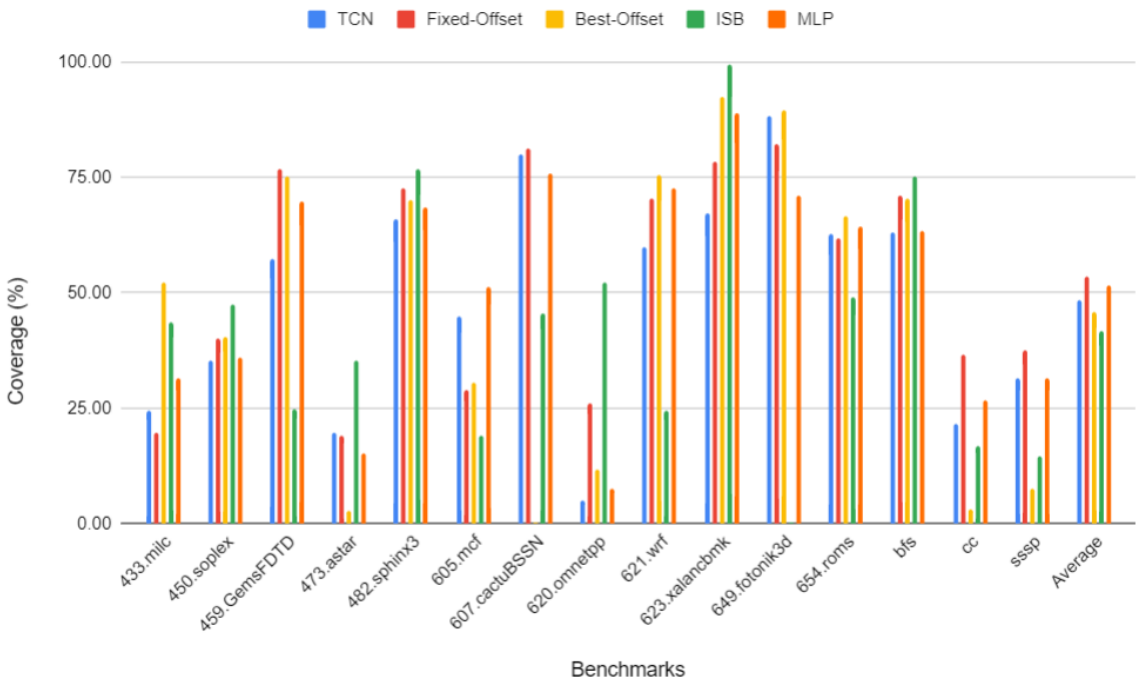

Figure 6.8: Coverage, as defined by equation 2.1. Invisible bars means no coverage.

6.3 Storage Overhead

According to the output from the Keras overview tool, presented in Table 6.3, the complete ML model used in the TCN prefetcher uses a total of 3092 parameters. The trainable parameters are 1856 in total, which are updated through gradient descent during training. The remaining 1236 parameters are updated with mean and standard deviation values of the TCNs neuron activations, for use in the TCNs two weight normalization layers during training. Each parameter is a 32-bit float, meaning that the model's storage overhead is around 12.4 KB. The storage overhead for the prefetchers LRU queue component is 4 * 500 = 2 KB, assuming each of the 500 27-bit queues is placed in their own 4 bytes of storage. The total storage overhead of the prefetcher is therefore around 14.4 KB.

Table 6.3: Setup of the TCN prefetcher's ML model, along with the number of parameters used.

Layer	Output Shape	Parameters
Input	(None, 30, 1)	0
TCN	(None, 8)	2516
Dense	(None, 64)	576
Activation	(None, 64)	0
Total parameters:	3092	
Trainable parameters:	1856	
Non-trainable parameters:	1236	

For the sake of comparison, the TCNs storage overhead is listed together with the tested MLP and ISB prefetchers in Table 6.4. Since the storage overhead of the Fixed-Offset and Best-Offset are negligible/unspecified, these are not shown. The TCN prefetcher uses about 70 times less storage than the MLP prefetcher. Important to note is also that the storage overhead for the MLP prefetcher presented in Table 6.4 only includes the storage needed for its parameters. Hence, it does not include the storage required for its table used to keep track of page transitions, nor the storage it uses to keep a history of previously accessed block indices. This is because there is no set upper limit for how large they can grow. Comparing the TCNs storage overhead to the cache size used by the ISB, the TCN uses around 2.2 times less memory. In terms of total storage overhead, the TCN prefetcher uses about 558 times less memory than ISB.

Table 6.4: Storage overhead of three of the tested prefetchers.

Prefetcher	Storage
TCN	14.4 KB
MLP [2]	1 MB
ISB [24]	32 KB cache + 8 MB main memory

Since the storage overhead for the two tested offset prefetchers is negligible, the MLP prefetcher is impractically large, and the ISB uses a hybrid cache/memory approach, these are not optimal points of comparison for practical storage overhead. The TCN prefetcher will therefore now be compared against the storage overhead of published *spatial* prefetchers. Spatial prefetchers are chosen since they need a non-negligible amount of storage but are still small enough to be practical for hardware implementation. Table 6.5 shows the TCNs storage overhead along four other spatial prefetchers. As can be seen, the TCN prefetcher's storage overhead is within the range of these other proposed prefetchers.

Table 6.5: Storage overhead of five spatial prefetchers.

Prefetcher	Storage
TCN	14.4 KB
Pythia [4]	25.5 KB
DSPatch [42]	3.6 KB
SPP [43]	5.37 KB
PPF [44]	39.34 KB

7
Discussion

This chapter discusses some of the results presented in 6.2.2, possible improvements to the TCN prefetcher, and the challenges presented for achieving major performance improvements. It then continues with remaining challenges for the practicality of the TCN prefetcher, followed by answers to this thesis research questions. Finally, it offers a reflection on the final results.

7.1 Performance Analysis

The test results presented in Figure 6.5 show that the TCN prefetcher performed best or next best on *605.mcf*, *607.cactuBSSN*, *cc* and *sssp*, while bringing little to no benefit in 473.astar and 620.omnetpp. In order to give insight into the reason for these results, this section will go over the characteristics of the access found in these benchmarks, and how they relate to the TCN prefetcher's performance. References to the performance of the other tested prefetchers will also be made.

In *605.mcf*, some program counters access pages with a constant stride, while others access the blocks in an irregular but consistent order multiple times through execution. What sets the TCN and MLP prefetchers apart from the offset prefetchers in this benchmark is that they capture not only the constant stride accesses, but also the irregular but consistent accesses. However, for these irregular accesses, the TCN prefetcher would have brought a more significant improvement if it had used a history of accessed block indices as input instead of only the current one. The reason is that the access patterns are not only PC-specific, but also page-specific, and a short history would have helped distinguish different patterns found on different pages. The MLP's use of a short history is one of the reasons that it performs better in this benchmark. The other reason is that the MLP prefetcher maintains a table that keeps track of page transitions. Since the page transitions remain consistent for these irregular parts, this helps achieve higher accuracy and coverage.

The reason why the TCN prefetcher works well on *607.cactuBSSN* is that most PCs exhibit a constant stride pattern while accessing pages. Since the TCN prefetcher views the PC access stream, this is an easy pattern to recognize, resulting in almost perfect accuracy and high coverage. In contrast, the Best-Offset prefetcher views the global access stream and can thus not capture this pattern. Consequently, it decides to stop issuing prefetch requests, resulting in it bringing no performance benefit. Though, for this benchmark, the TCN prefetcher essentially works as the

Fixed-Offset prefetcher with a distance (offset) of 3.

In the *cc* benchmark, the TCN prefetcher does not find a clear pattern between accesses since they are highly irregular, and there are only around 60 PCs to learn from in the tested traces. Instead, it finds which blocks each PC accesses the most and issues prefetch requests for them. As shown by the results, this does not give high accuracy or coverage, but it seems to ensures that some often requested blocks are put into the cache.

In the *sssp* benchmark, the memory accesses do not have a constant stride, but the same pages are accessed multiple times throughout execution with a periodically similar access pattern. Although the Fixed-Offset prefetcher achieves higher accuracy and coverage than the TCN prefetcher in this benchmark, the TCN prefetcher seems to bring in blocks that are used more times.

The TCN prefetcher offers almost no speedup in *473.astar* and no speedup in *620.omnetpp*. These benchmarks mostly contain highly irregular accesses, and the accesses to the same pages are sparsely separated over their execution. The ISB prefetcher is the only of the tested prefetchers that brings a significant improvement in these benchmarks. This is because ISB is an address-correlating prefetcher, which stores the full addresses of previously seen accesses and then recalls their relative order later on in the execution.

7.2 Possible Improvements

As mentioned in Section 7.1, the TCN prefetcher would have benefited from receiving a short history of previous accesses as input to help distinguish more access patterns. This was confirmed later in the project when a history-based version of the TCN prefetcher was tested. This version worked mostly like the current version, but it replaced the bits of the current PC as part of the input, with a history of three block indices that the PC has accessed. Performance tests of this version on a small selection of traces showed that it outperformed the current TCN prefetcher by a few percent on average. However, the improvement was not significant enough to justify any further time investment into it.

The history-based version also came with two downsides. The first was that it needed to maintain three more block indices per LRU queue, resulting slightly larger storage overhead. The second was that if the content in one of the LRU queues were to be evicted because of infrequent access during parts of the execution, it would need to refill a queue associated with that PC before predictions could be made again for that PC. This was not found to be a problem on the benchmark traces tested. Nevertheless, it could become a problem if the prefetcher issued prefetches for larger programs with more active PCs, or if multiple programs are executed interchangeably. This is in contrast to the current version of the TCN prefetcher, which only needs to maintain a short history of accesses for its learning process, while it is not required for its predictions.

The TCN prefetcher would probably have benefited from a multi-label approach such as Shi et al. [6] presented. This is because patterns can be more or less apparent depending on which access stream is viewed. For example, a simple constant stride pattern in the global access stream can become complex if viewed from the PC access stream, and vice versa. Multi-labeling would, however, increase the design complexity of the prefetcher. It also requires storing information from multiple access streams, which increases the storage overhead, but the potential benefits are likely worth it.

7.3 Challenges for Major Improvements

The TCN prefetcher presented in this project has shown that a small 1-D convolutional network can achieve similar prefetching performance as a large MLP model. However, as shown by the results in Figure 6.5, the two offset prefetchers performed better on average than the two ML based prefetchers. This is largely because of the three following reasons.

The first reason is that there is no significant advantage to using neural networks to capture accesses with a constant stride. In contrast, the Best-Offset prefetcher, for example, offers the advantage of dynamically adjusting its prefetching distance to optimize timeliness. If prefetchers based on neural networks would do the same, they could at best even out the performance gap for these types of patterns. However, it is unclear how well this would work in a neural network framework since every time the lookahead distance changes, the network would need to be completely re-trained to learn the new classifications.

The second reason is that it is seemingly rare that programs access blocks within the same page in such a way that there are clear patterns to the accesses, without the patterns also exhibiting a constant stride. In the instances these patterns do appear, for example in 623.xalancbmk, it is beneficial to have a per-page knowledge of accesses since it presents clear patterns, but this is difficult to obtain. The Page version of the TCN prefetcher evaluated in Section 5.2.1 was an attempt to achieve this. However, in most cases, it learned to ignore the page bits that were part of the input, and only learned patterns in the order of block indices. Thus, this version ended up being close to a fixed offset prefetcher with an offset of 3 in most benchmarks. Though, the benchmark *605.mcf* proved to bring a special case of irregular access patterns. Here, not only per-page knowledge of accesses results in clear patterns. In this benchmark, most PCs also enter the same pages in the same way within longer bursts, many times throughout each trace. This allowed the TCN and MLP prefetchers that view the PC access stream to learn the patterns and issue prefetches according to them.

The third reason is that to make a substantial improvement in highly irregular benchmarks, such as 473.astar and 620.omnetpp, it is not sufficient to only predict which block will be accessed; the page also needs to be predicted. This presents

a major challenge since predictions on the vast address space of a computer are resource-intensive, and for the prefetcher to be considered practical, it can not use more than a few tens of kilobytes of memory. The ISB prefetcher solves this by storing address correlations in the main memory and using a clever caching and replacement policy to maintain low latency predictions. However, it is uncertain how well this works in practice, and this approach is not possible for neural networks.

7.4 Challenges for Practicality

Other than predictive performance improvements to the TCN prefetcher, there are also a number of other challenges that would need to be addressed to make it practical for hardware implementation. The first is that it needs to be able to learn *online*. This is crucial since each program's behavior changes throughout execution, and the operating system performs context switches between different programs which have entirely different program counters and access patterns.

The second challenge is inference latency. To be able to issue timely prefetches and keep up with the flow of LLC accesses, the inference latency needs to be low enough for this. This has not been tested in this project, but there are many more operations needed to make a prediction through a multi-layer neural network than a rule-based prefetcher. However, there are many opportunities for parallel execution of the TCN model, and there are no recurrences like there are in LSTM models.

The third challenge is energy consumption. As mentioned in the second challenge, neural networks require more operations to perform predictions than rule-based prefetchers. As such, their energy consumption is likely higher. Evaluating the TCN prefetcher's energy consumption is outside of this project's scope. However, its energy consumption would need to be evaluated before it could be implemented in real hardware.

The final challenge is that each page is assumed to be conventionally sized at 4 KB. If the standard page size increased to, for example, 1 MB, while the block size remains the same, the number of classes to predict between will increase from 64 to 16384. This would increase the number of parameters required by the fully connected output layer from 576 to 147456. If the precision remained the same (32-bit), the storage overhead would increase from 14.4 KB to about 602 KB, making the prefetcher impractical from storage overhead alone. Reducing the precision to 16 or 8 bit would alleviate some overhead, but not enough to make up for the difference. Also, in a few tests conducted toward the end of the project to see if the TCN prefetcher could be switched to 16-bit precision, the training process became unstable. Thus the performance results were significantly degraded.

7.5 Answers to Research Questions

Here the research questions of the project will be repeated, and a brief answer to them will be given.

1) *Which ML algorithms have been tested and well documented before in the context of hardware data prefetching?*

The literature review that was conducted in connection to Section 5.1.1 showed that the following ML algorithms had already been tested in this context: MLP [2, 36], LSTM [6, 25–29], SARSA [4], Contextual Bandit [37] and the Transformer model [31].

2) *Which ML algorithm that is not a part of (1) seems to have good potential to be used in hardware data prefetching?*

As described in Section 5.1.1, TCN was found to have good potential for the task of prefetching, since it offers low parameter usage and has good potential for parallel execution of each input sequence.

3) *Can the chosen algorithm be implemented as a working data prefetcher in such a way that, within the scope of this project, it gets a storage overhead comparable to published spatial hardware data prefetchers?*

The storage overhead for the final version of the TCN prefetcher is 14.4 KB, which is within the range of previously presented spatial data prefetchers.

4) *Can the chosen algorithm be implemented as a data prefetcher in such a way that, within the scope of this project, (3) holds true while the prefetcher's performance also is competitive with published hardware data prefetchers?*

In the performance tests conducted in Section 6.2.2, the TCN prefetcher was tested along with four other prefetchers, of which the Best-Offset prefetcher and ISB prefetcher have been published. On average, the Best-Offset prefetcher improved the performance more than the TCN prefetcher, while the TCN prefetcher made a larger improvement than the ISB prefetcher. But unlike both of these published prefetchers, which have their own areas in which they excel, the TCN prefetcher does not. However, it gives a pretty stable performance improvement across the tested benchmarks.

7.6 Reflection

Regarding the final performance results, it was an oversight to not include an offset prefetcher with a fixed offset as a point of comparison early on in the design phase of the prefetcher. Since the Best-Offset prefetcher is an optimization of such a prefetcher with the ability to change offset to optimize timeliness dynamically, my perception during the design phase was that it would work as a better point of comparison. This turned out to only be partially true. Due to how the Best-Offset prefetcher selects offset, it can only capture constant stride patterns that are visible in the global access stream, not on a per PC / page basis. If it does not find a pattern, it will stop issuing prefetch requests until it has found one in an attempt to reduce the number of useless prefetches. This has the side effect that it in for example *607.cactuBSSN* offers no benefit while a fixed offset prefetcher performs very well. Using the Best-Offset as the point of comparison therefore resulted in skewed conclusions regarding the benefits that the TCN prefetcher brought. If I had been aware of this early on, a different prefetcher design could have been explored. It is however uncertain at this point what exact design that would have been and what result it would have yielded.

8

Conclusion

This thesis explored using a Temporal Convolutional Network (TCN) as the basis for a hardware data prefetcher designed for the LLC cache. A primary goal was to maintain a storage overhead low enough to be practical for hardware implementation, while striving for good prefetching performance. To this end, two prefetcher versions with different inputs were designed and evaluated, where the most promising version was selected to be optimized for low storage overhead and further evaluation.

The performance of the final TCN prefetcher was evaluated on a set of 15 memory-intensive benchmarks from the SPEC06, SPEC17, and GAP benchmark suits. The result showed an average IPC improvement of 30.5 % over a baseline with no prefetcher, while incurring only 14.4 KB storage overhead. The result further showed that it is possible to achieve reliable performance gains using a TCN for prefetching, which can greatly reduce the number of parameters needed to make predictions compared to MLP based prefetchers, and without the use of recursion such as in LSTM based prefetchers. The result further suggests that rule-based offset prefetchers can capture the majority of prefetching patterns that are presented within the page-limit of real-world programs, and that to achieve a major advancement in prefetching for irregular accesses, page predictions are necessary.

Future work on the TCN prefetcher would need to adapt it to train *online* and estimate its inference latency. Furthermore, exploration of using labels from multiple memory access streams is suggested to find otherwise missed access patterns, and some form of page prediction that only entails a small storage overhead.

Bibliography

[1] Babak Falsafi and Thomas Wenisch. A primer on hardware prefetching. *Synthesis Lectures on Computer Architecture*, 9:1–67, 05 2014.

[2] Alitaghi Asgari, Andrew Gunter, Mehdi Saeidi, Mieszko Lis, and Prashant J. Nair. MPMLP: A case for multi-page multi-layer perceptron prefetcher. *The 2021 Workshop on ML for Computer Architecture and Systems (MLArchSys 2021), held in conjunction with ISCA*, 2021.

[3] International Symposium on Computer Architecture (ISCA). Mlarchsys. [Online]. Available: `https://sites.google.com/view/mlarchsys/isca-2021/ml-prefetching-competition?authuser=0`, 2021. (accessed on: 2021-11-18).

[4] Rahul Bera, Konstantinos Kanellopoulos, Anant Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. Pythia: A customizable hardware prefetching framework using online reinforcement learning. *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, Oct 2021.

[5] Mohammad Bakhshalipour, Mehran Shakerinava, Fatemeh Golshan, Ali Ansari, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. A survey on recent hardware data prefetching approaches with an emphasis on servers. *CoRR*, abs/2009.00715, 2020.

[6] Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin. A hierarchical neural model of data prefetching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 861–873, New York, NY, USA, 2021. Association for Computing Machinery.

[7] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. *SIGARCH Comput. Archit. News*, 25(2):252–263, may 1997.

[8] Standard Performance Evaluation Corporation. Spec cpu 2006. [Online]. Available: `https://www.spec.org/cpu2006/`, 2006. (accessed on: 2021-11-24).

[9] Standard Performance Evaluation Corporation. Spec cpu 2017. [Online]. Available: `https://www.spec.org/cpu2017/`, 2017. (accessed on: 2021-11-24).

[10] Scott Beamer, David Patterson, and Krste Asanović. Gap benchmark suite. [Online]. Available: `http://gap.cs.berkeley.edu/benchmark.html`, 2015. (accessed on: 2021-11-24).

[11] T.M. Mitchell. *Machine Learning*. McGraw-Hill International Editions. McGraw-Hill, 1997.

[12] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958.

[13] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323:533–536, 1986.

[14] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[15] Colin Lea, Michael D. Flynn, René Vidal, Austin Reiter, and Gregory D. Hager. Temporal convolutional networks for action segmentation and detection. *CoRR*, abs/1611.05267, 2016.

[16] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *CoRR*, abs/1803.01271, 2018.

[17] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W. Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *CoRR*, abs/1609.03499, 2016.

[18] Aäron van den Oord and Tom Walters. Wavenet launches in the google assistant. [Online]. Available: `https://deepmind.com/blog/article/wavenet-launches-google-assistant`, 2017. (accessed on: 2022-03-22).

[19] Tim Salimans and Diederik P. Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. *CoRR*, abs/1602.07868, 2016.

[20] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, jan 2014.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[22] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, page 807–814, Madison, WI, USA, 2010. Omnipress.

[23] Pierre Michaud. Best-offset hardware prefetching. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 469–480, 2016.

[24] Akanksha Jain and Calvin Lin. Linearizing irregular memory accesses for improved correlated prefetching. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, page 247–259, New York, NY, USA, 2013. Association for Computing Machinery.

[25] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. Learning memory access patterns. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1919–1928. PMLR, 10–15 Jul 2018.

[26] Yuan Zeng and Xiaochen Guo. Long short term memory based hardware prefetcher: A case study. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '17, page 305–311, New York, NY, USA, 2017. Association for Computing Machinery.

[27] Ajitesh Srivastava, Angelos Lazaris, Benjamin Brooks, Rajgopal Kannan, and Viktor K. Prasanna. Predicting memory accesses: The road to compact ml-driven prefetcher. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, page 461–470, New York, NY, USA, 2019. Association for Computing Machinery.

[28] Peter Braun and Heiner Litz. Understanding memory access patterns for prefetching. *International Workshop on AI-assisted Design for Architecture (AIDArc), held in conjunction with ISCA*, 2019.

[29] Pengmiao Zhang, Ajitesh Srivastava, Benjamin Brooks, Rajgopal Kannan, and Viktor K. Prasanna. Raop: Recurrent neural network augmented offset prefetcher. In *The International Symposium on Memory Systems*, MEMSYS 2020, page 352–362, New York, NY, USA, 2020. Association for Computing Machinery.

[30] G. Rummery and Mahesan Niranjan. On-line q-learning using connectionist systems. *Technical Report CUED/F-INFENG/TR 166*, 11 1994.

[31] Pengmiao Zhang, Ajitesh Srivastava, Rajgopal Kannan, Anant V. Nori, and Viktor K. Prasanna. TransforMAP: Transformer for memory access prediction. *The 2021 Workshop on ML for Computer Architecture and Systems (MLArchSys 2021), held in conjunction with ISCA*, 2021.

[32] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[33] International Symposium on Computer Architecture (ISCA). Quangmire/champsim. [Online]. Available: `https://github.com/Quangmire/ChampSim`, 2021. (accessed on: 2022-01-20).

[34] ChampSim. Quangmire/champsim. [Online]. Available: `https://github.com/ChampSim/ChampSim`, 2020. (accessed on: 2022-03-30).

[35] Anthony S. Fong and C.Y. Ho. Global/local hashed perceptron branch prediction. In *Fifth International Conference on Information Technology: New Generations (itng 2008)*, pages 247–252, 2008.

[36] Leeor Peled, Uri Weiser, and Yoav Etsion. A neural network prefetcher for arbitrary memory access patterns. *ACM Transactions on Architecture and Code Optimization*, 16(4):1–27, dec 2019.

[37] Leeor Peled, Shie Mannor, Uri Weiser, and Yoav Etsion. Semantic locality and context-based prefetching using reinforcement learning. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 285–297, New York, NY, USA, 2015. Association for Computing Machinery.

[38] Advanced Micro Devices. *Preliminary Processor Programming Reference (PPR) for AMD Family 19h Model 01h, Revision B1 Processors Volume 1 of 2*. AMD, 2485 Augustine Drive, Santa Clara, CA 95054, 55898 rev 0.50 edition, may 2021.

[39] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*. Intel, 2200 Mission College Blvd, Santa Clara, CA, 253665-076us edition, December 2021.

[40] Philippe Rémy. Keras tcn. [Online]. Available: `https://github.com/philipperemy/keras-tcn`, 2022. (accessed on: 2022-04-13).

[41] DPC2. The 2nd data prefetching championship (dpc2). [Online]. Available: `https://comparch-conf.gatech.edu/dpc2/`, 2020. (accessed on: 2022-05-08).

[42] Rahul Bera, Anant V. Nori, Onur Mutlu, and Sreenivas Subramoney. Dspatch: Dual spatial pattern prefetcher. *CoRR*, abs/1910.03075, 2019.

[43] Jinchun Kim, Seth H. Pugsley, Paul V. Gratz, A.L. Narasimha Reddy, Chris Wilkerson, and Zeshan Chishti. Path confidence based lookahead prefetching. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[44] Eshan Bhatia, Gino Chacon, Seth Pugsley, Elvira Teran, Paul V. Gratz, and Daniel A. Jiménez. Perceptron-based prefetch filtering. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–13, 2019.

A

Appendix 1

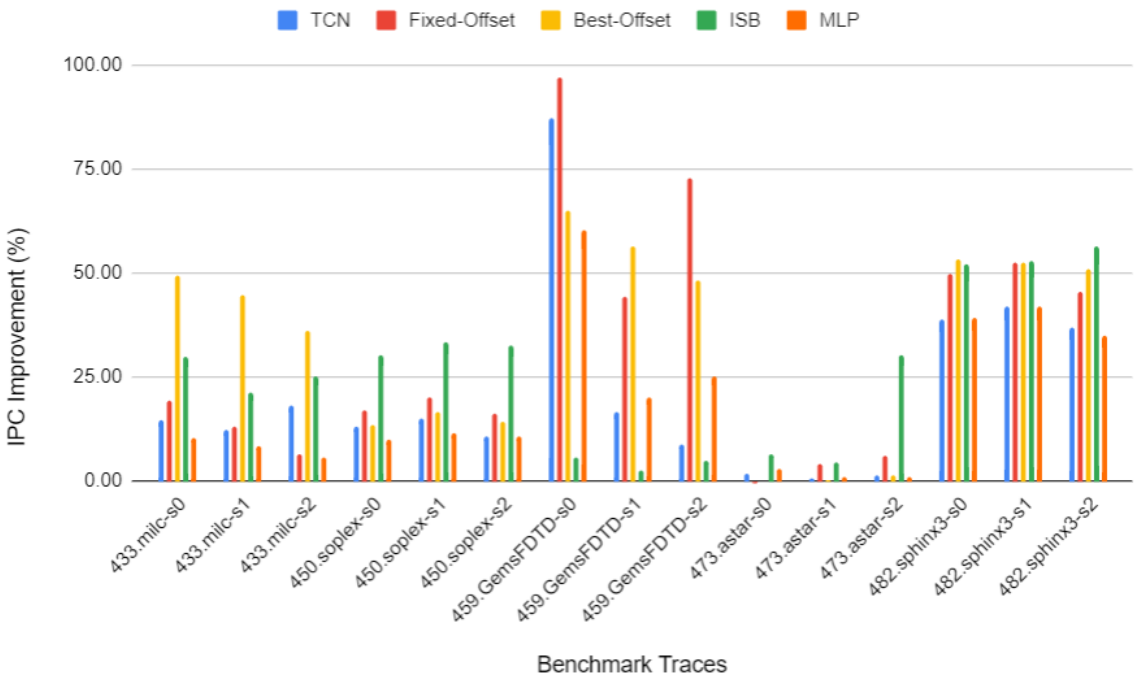

Figure A.1: IPC improvement at degree 1, part 1.

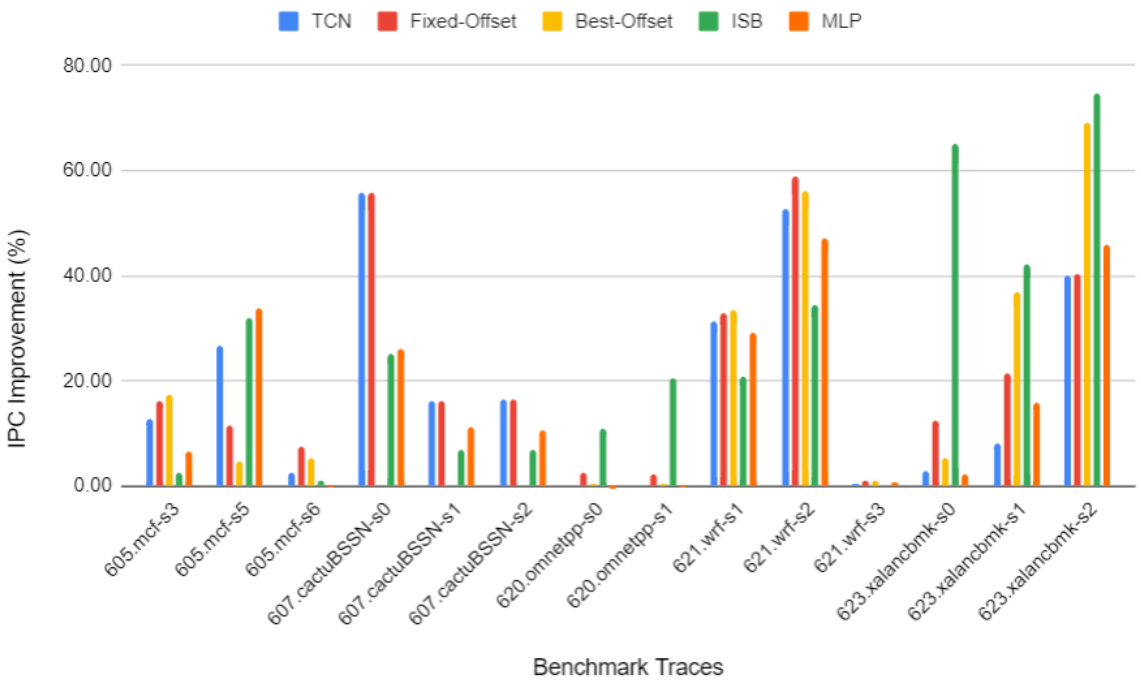

Figure A.2: IPC improvement at degree 1, part 2.

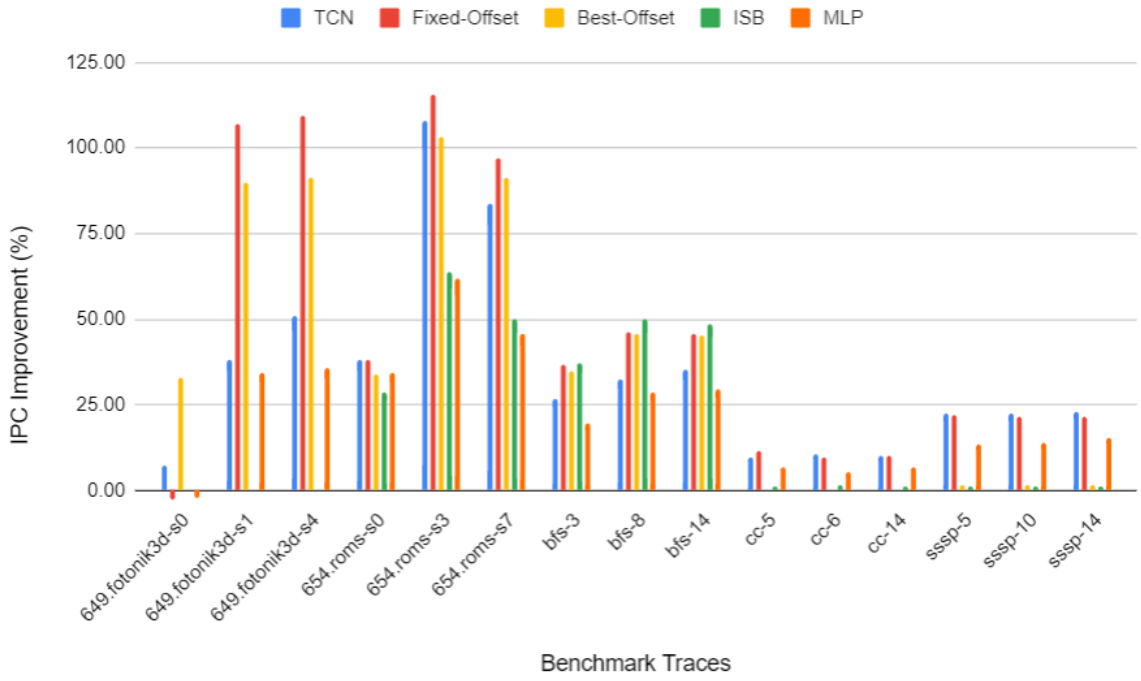

Figure A.3: IPC improvement at degree 1, part 3.

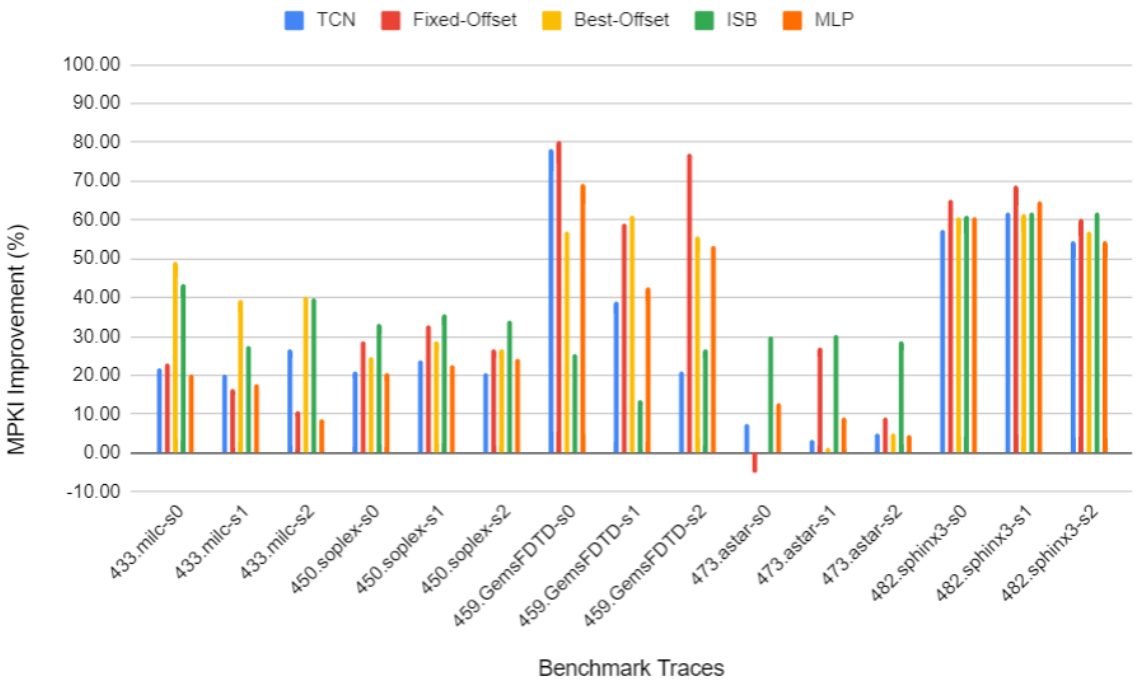

Figure A.4: MPKI improvement at degree 1, part 1.

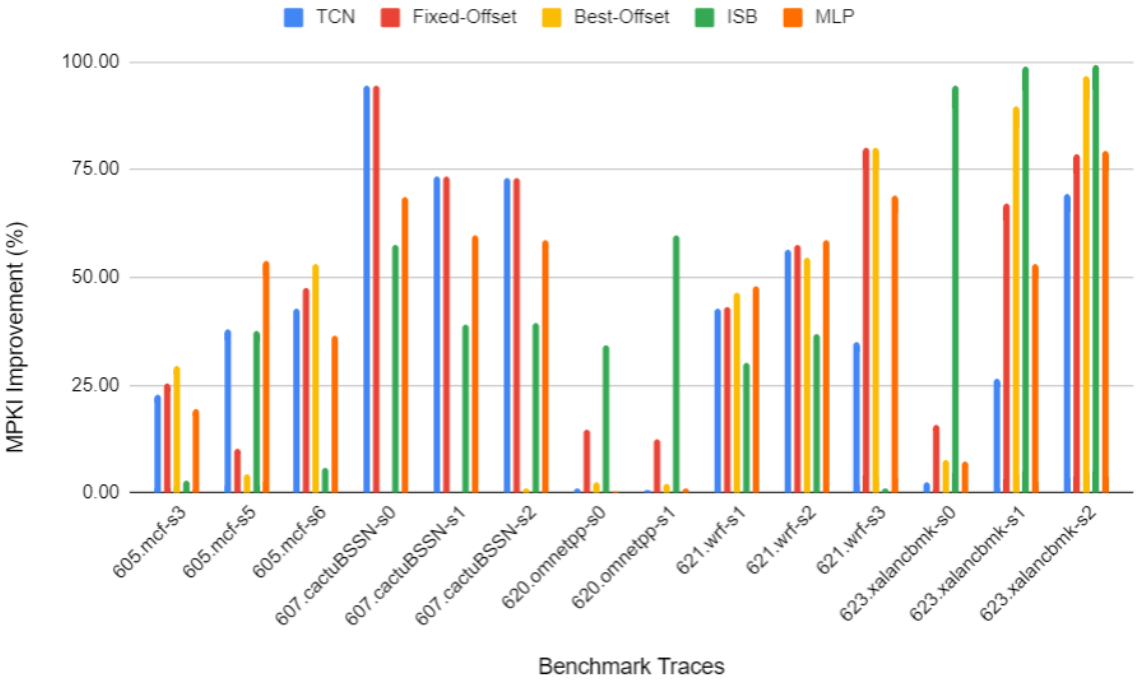

Figure A.5: MPKI improvement at degree 1, part 2.

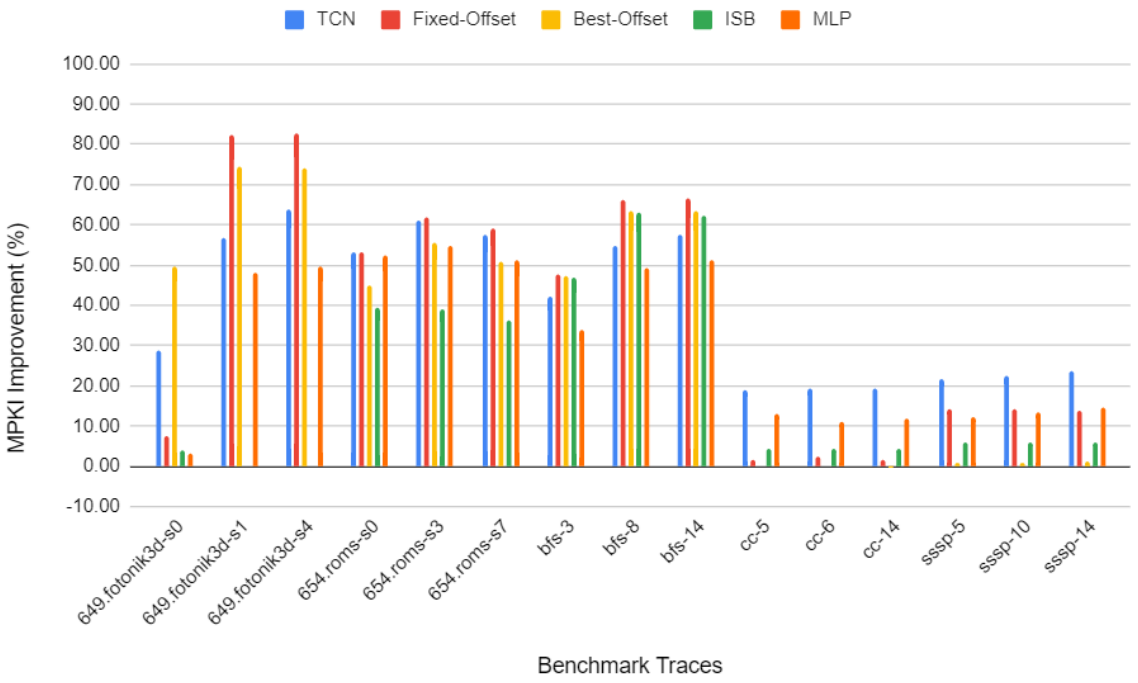

Figure A.6: MPKI improvement at degree 1, part 3.

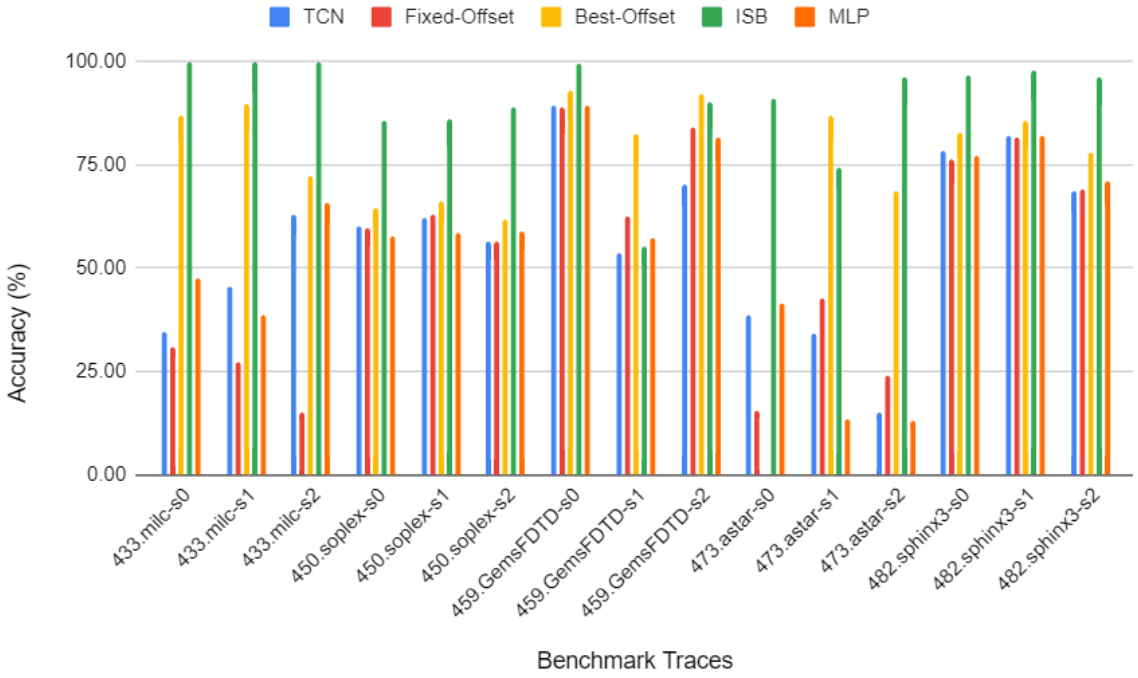

Figure A.7: Accuracy at degree 1, part 1.

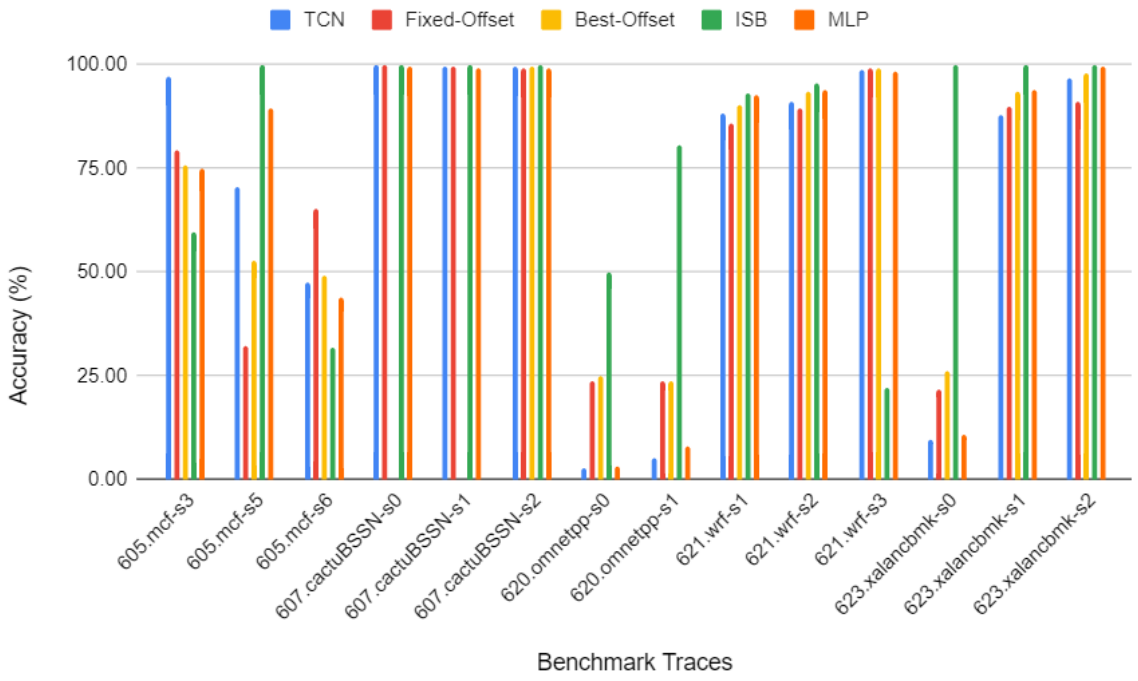

Figure A.8: Accuracy at degree 1, part 2.

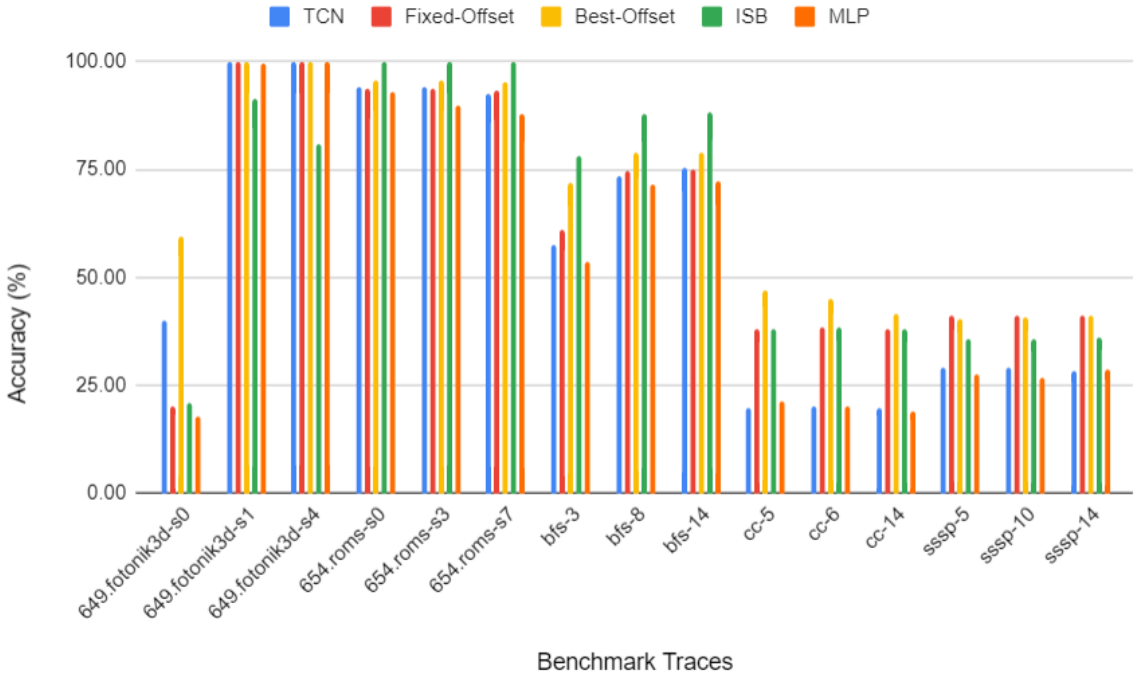

Figure A.9: Accuracy at degree 1, part 3.

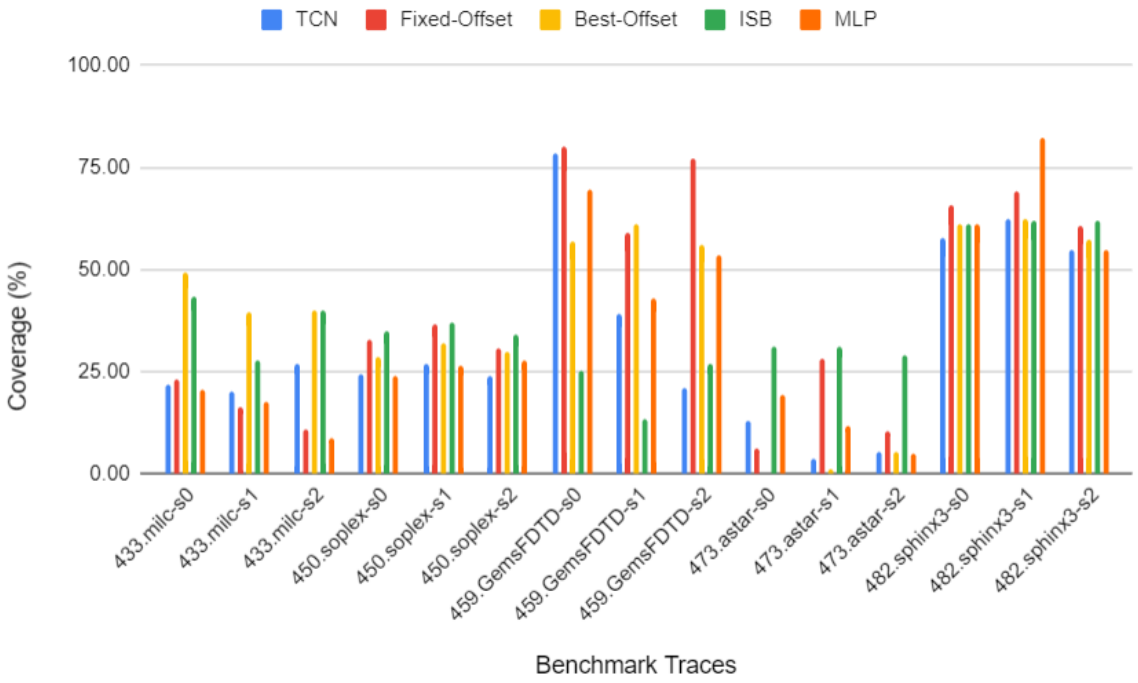

Figure A.10: Coverage at degree 1, part 1.

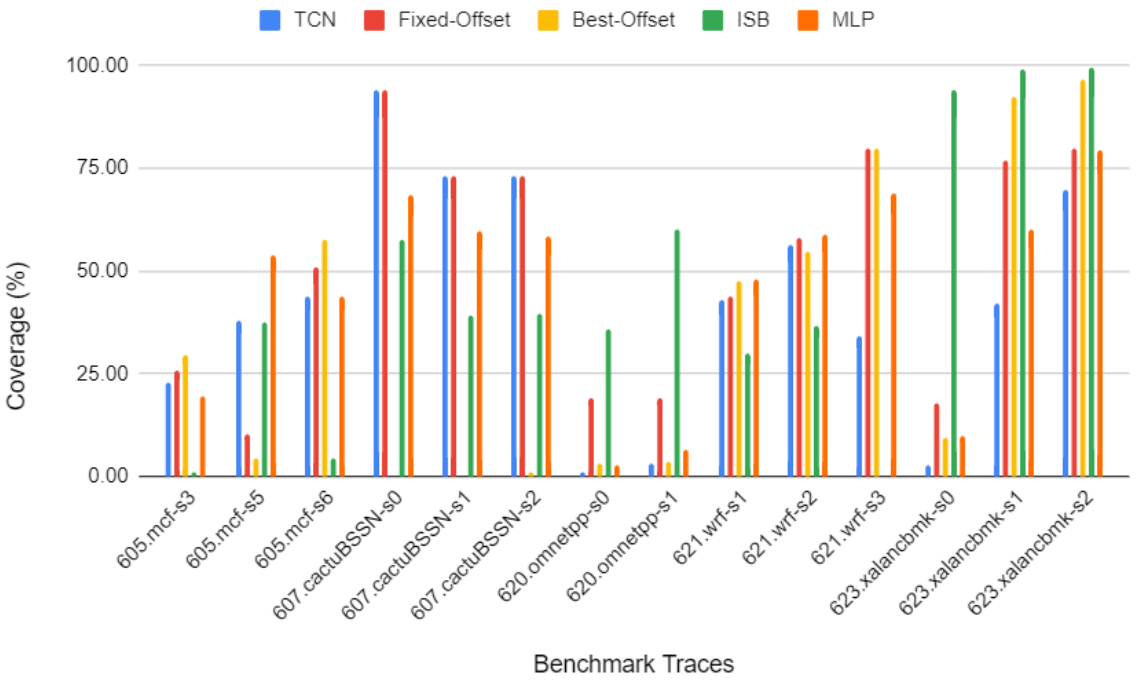

Figure A.11: Coverage at degree 1, part 2.

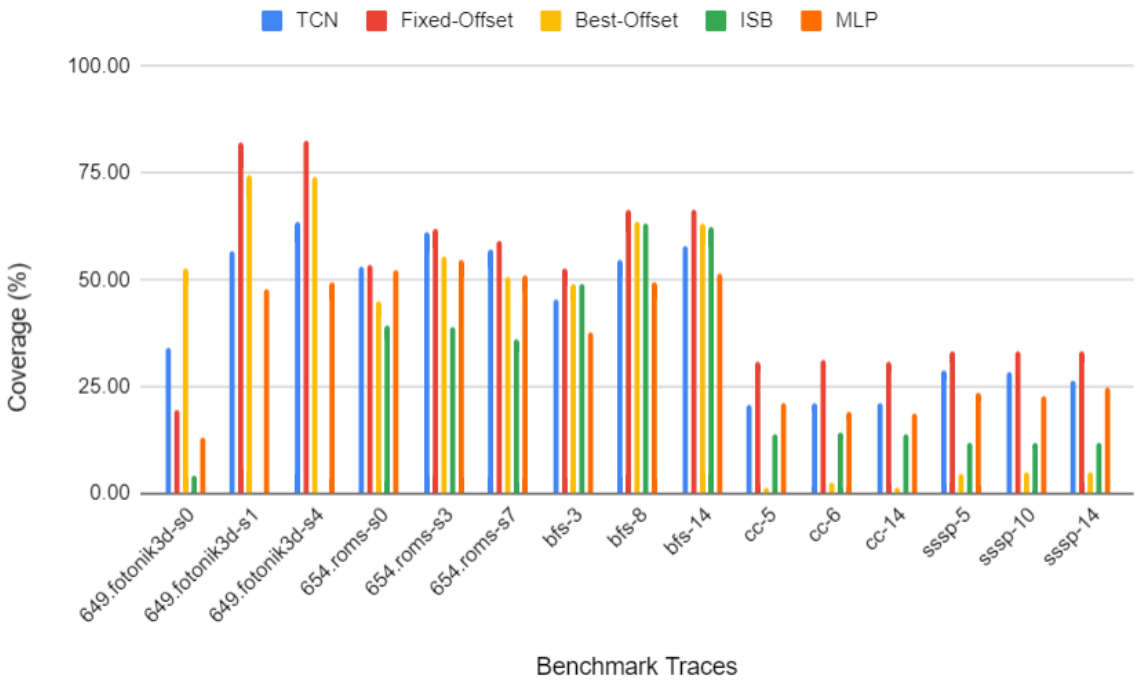

Figure A.12: Coverage at degree 1, part 3.

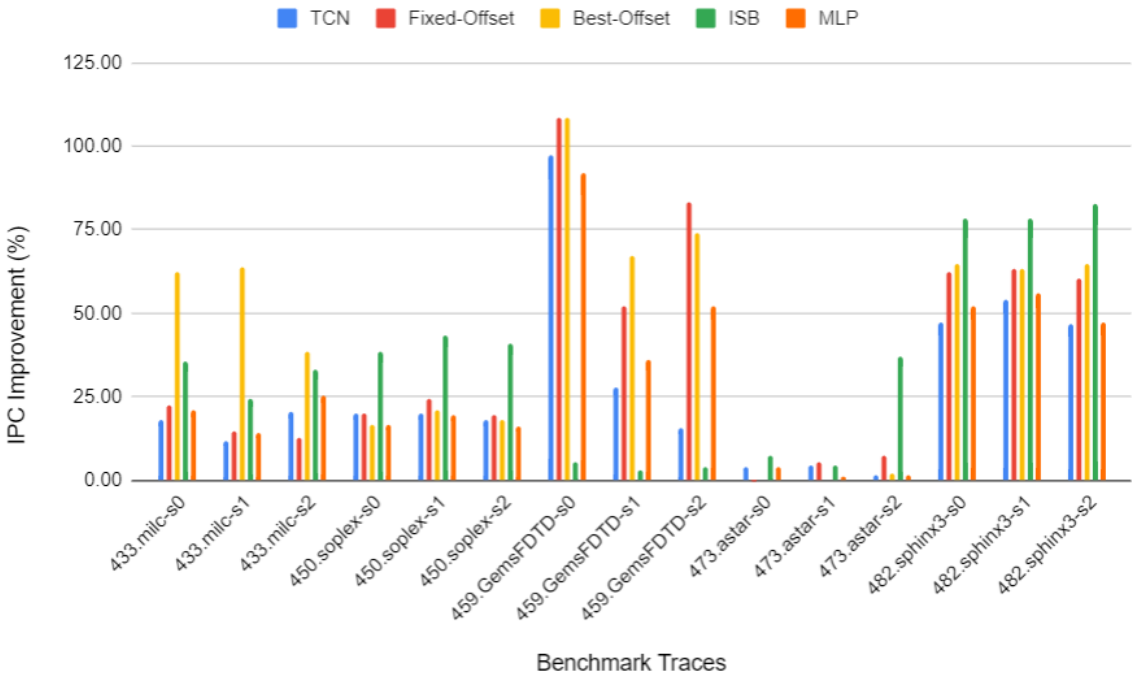

Figure A.13: IPC improvement at degree 2, part 1.

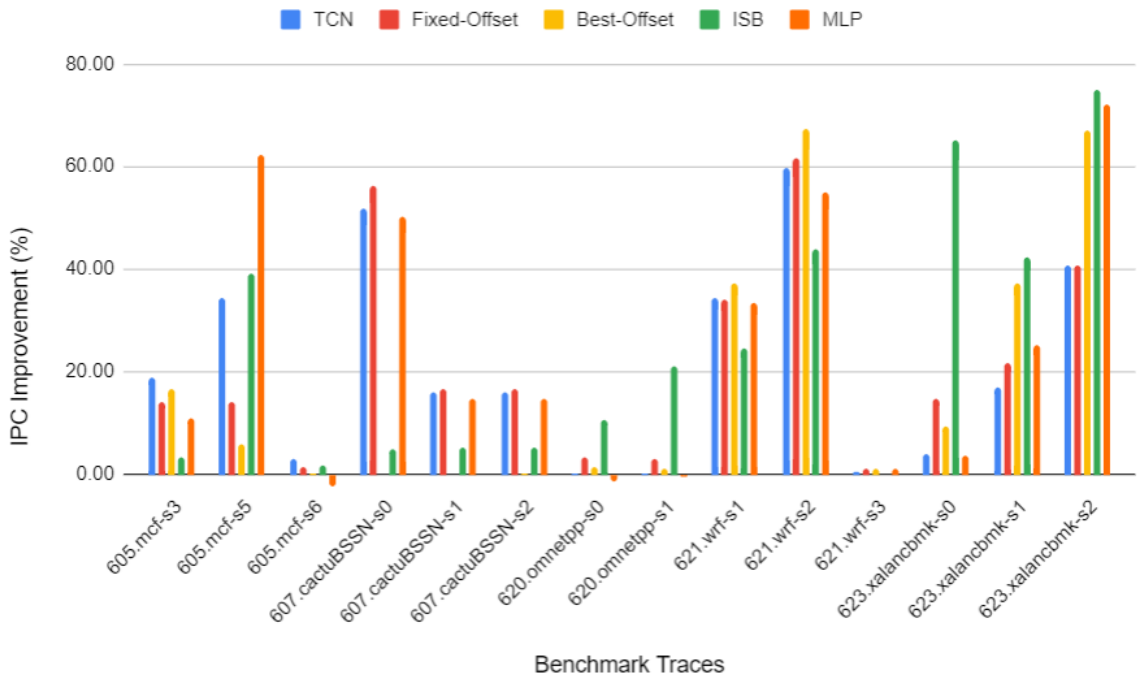

Figure A.14: IPC improvement at degree 2, part 2.

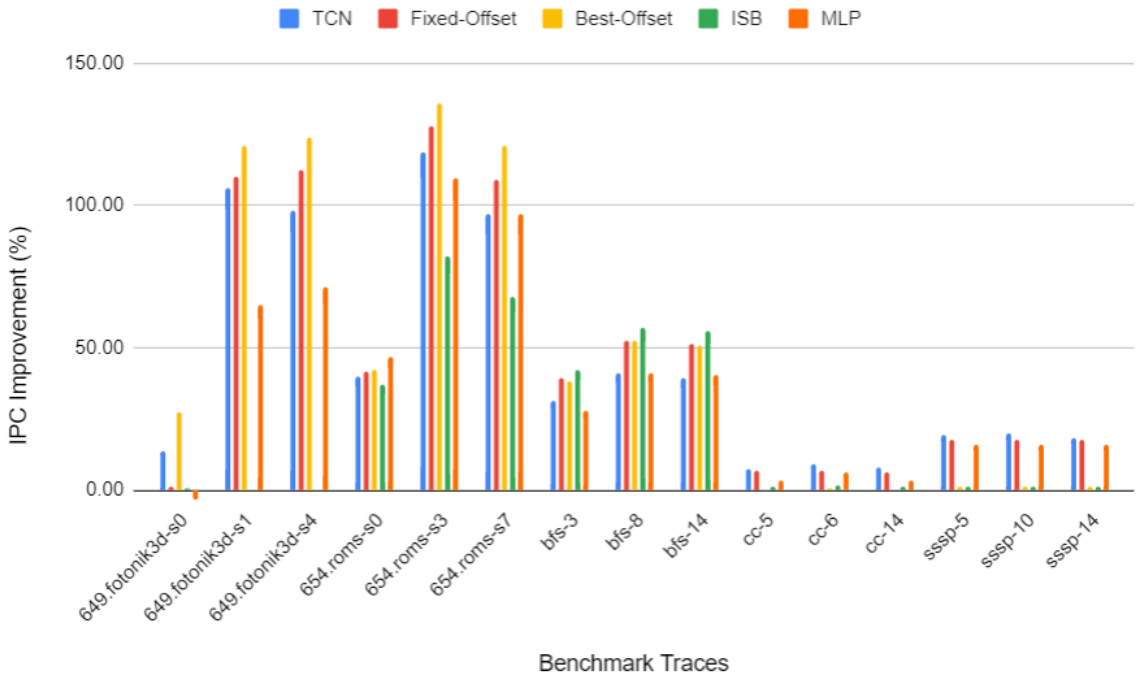

Figure A.15: IPC improvement at degree 2, part 3.

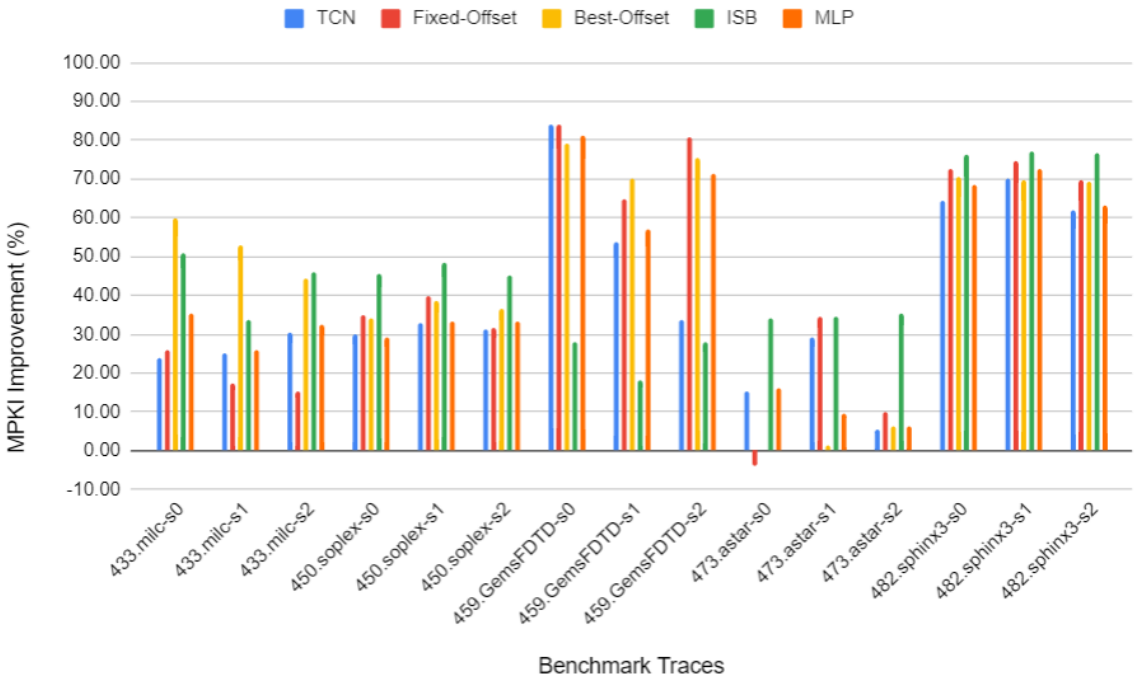

Figure A.16: MPKI improvement at degree 2, part 1.

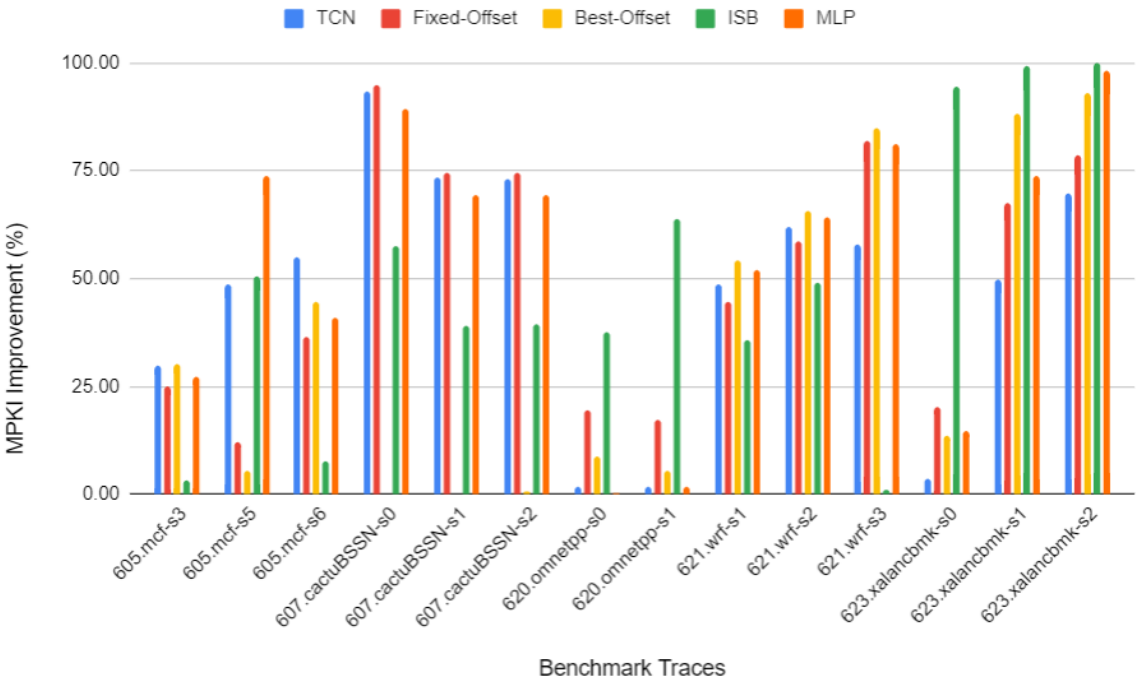

Figure A.17: MPKI improvement at degree 2, part 2.

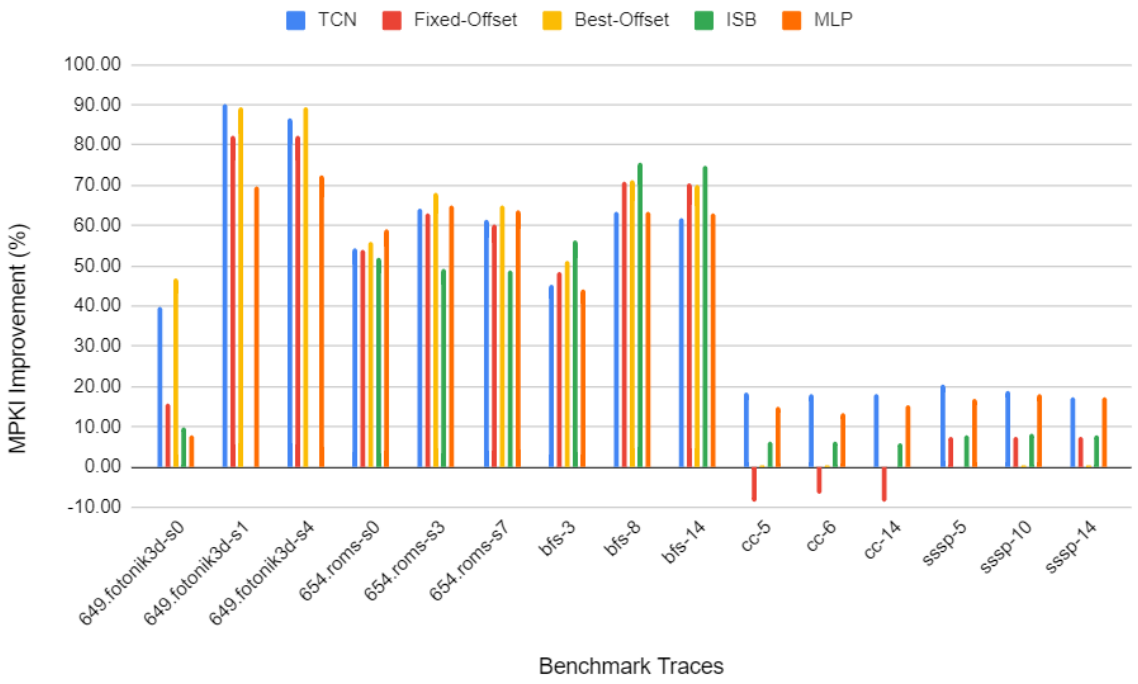

Figure A.18: MPKI improvement at degree 2, part 3.

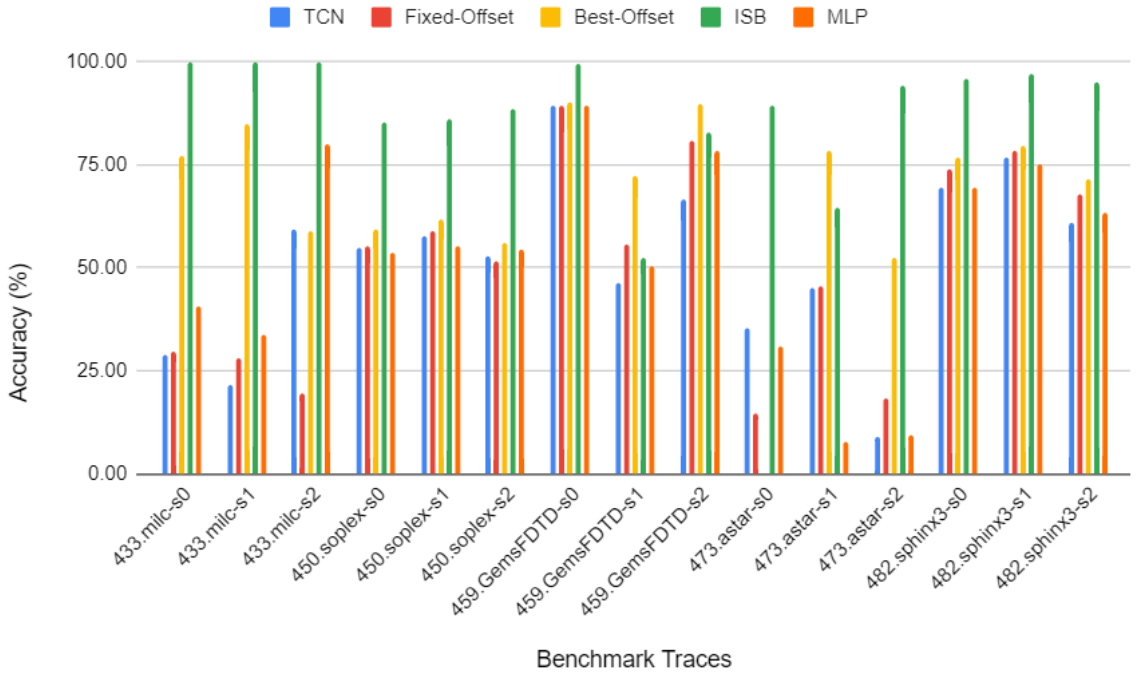

Figure A.19: Accuracy at degree 2, part 1.

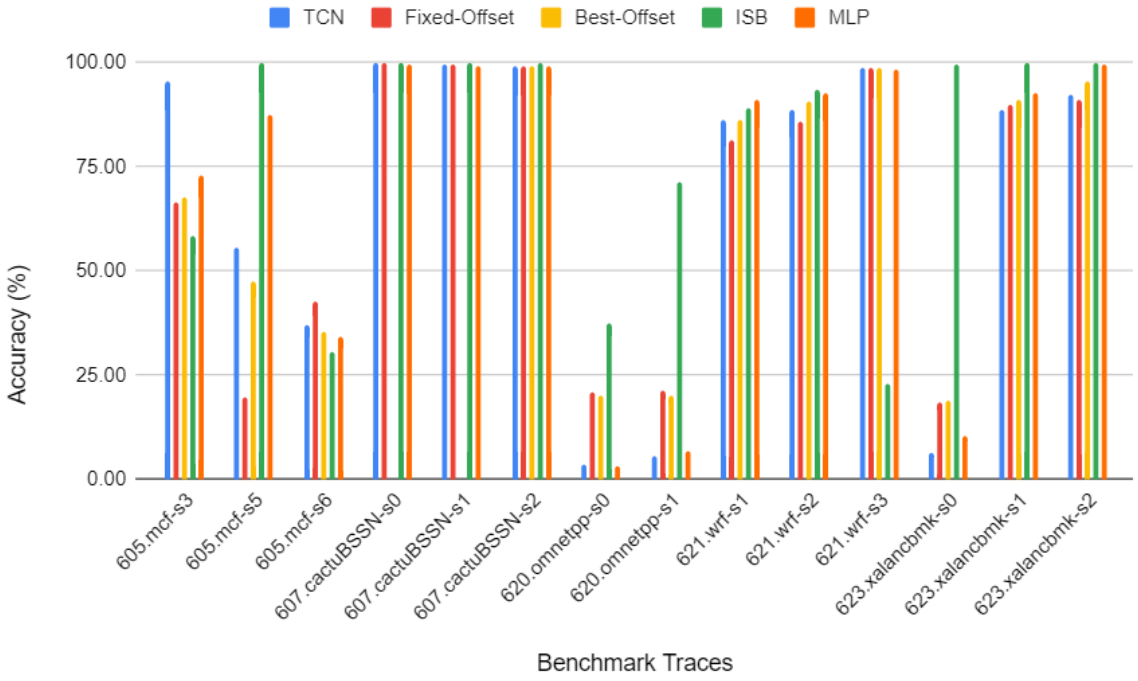

Figure A.20: Accuracy at degree 2, part 2.

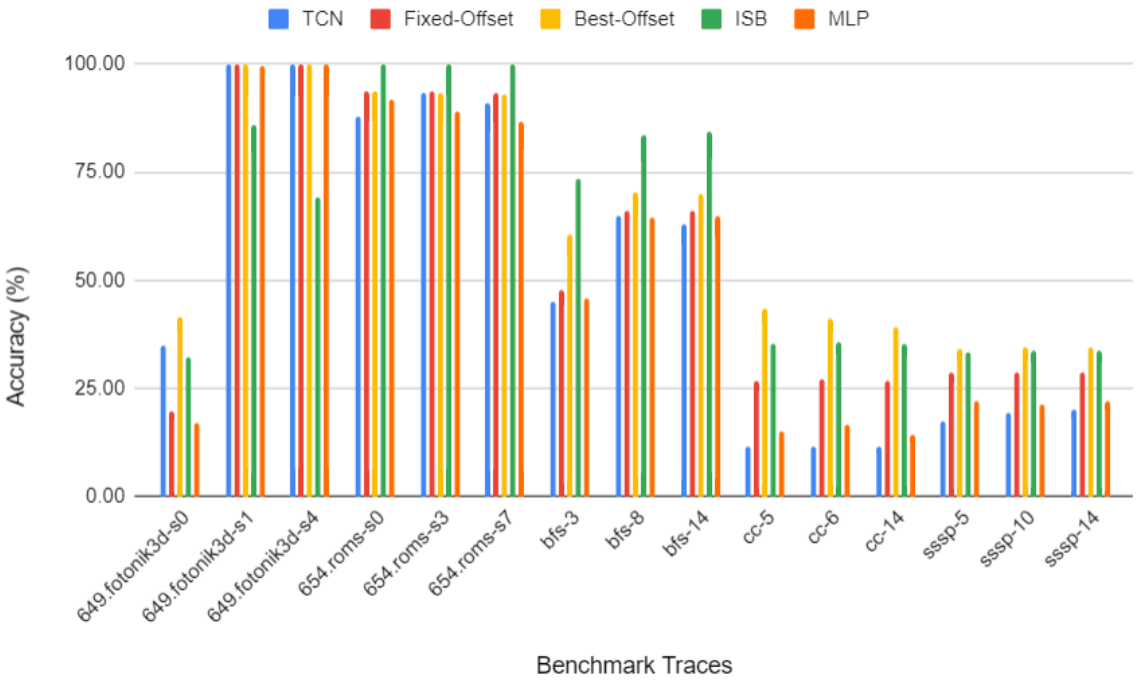

Figure A.21: Accuracy at degree 2, part 3.

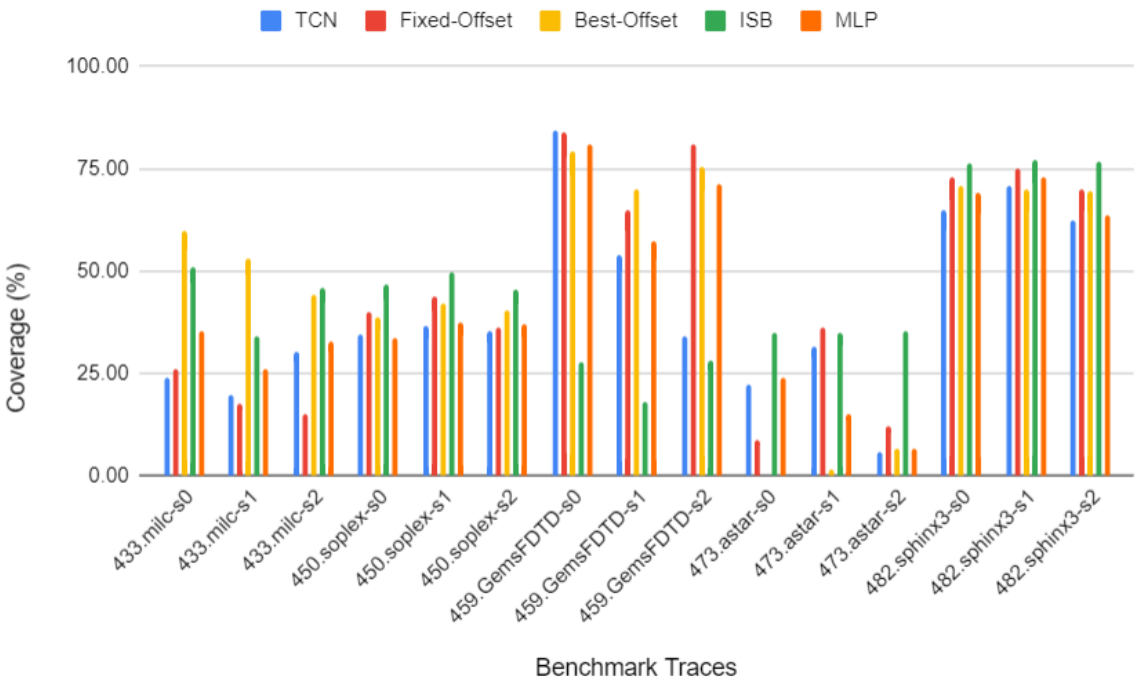

Figure A.22: Coverage at degree 2, part 1.

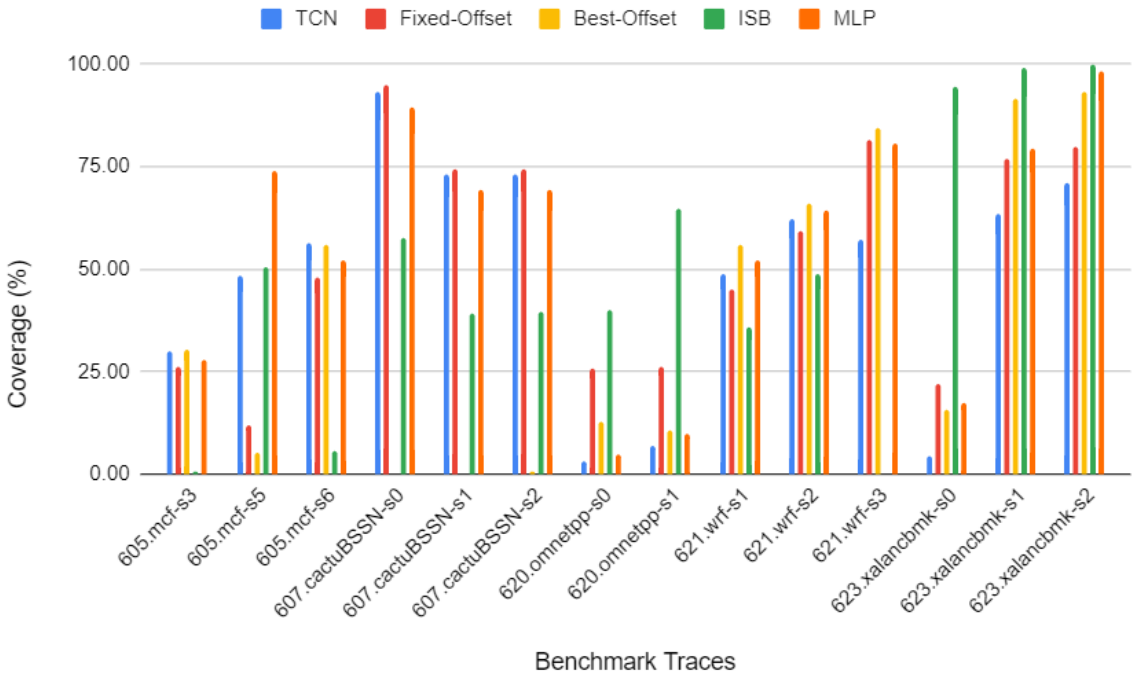

Figure A.23: Coverage at degree 2, part 2.

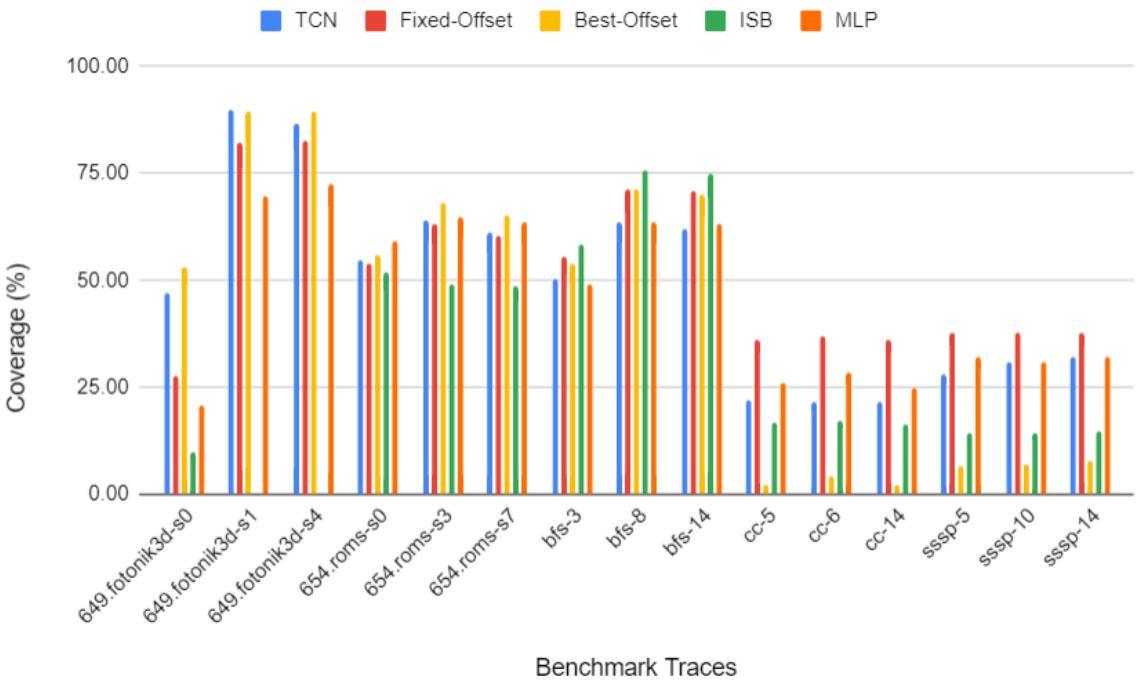

Figure A.24: Coverage at degree 2, part 3.