National Technical University of Athens
School of Electrical and Computer Engineering
Division of

Cache miss estimator

Predicting cache misses of different cache architectures

Diploma Thesis

of

MARKOS-GEORGIOS RAMOS

Supervisor: Nectarios Koziris
Professor

Athens, April 2024

National Technical University of Athens
School of Electrical and Computer Engineering
Division of

Cache miss estimator

Predicting cache misses of different cache architectures

Diploma Thesis

of

MARKOS-GEORGIOS RAMOS

Supervisor: Nectarios Koziris
Professor

Approved by the examination committee on 5 April 2024.

(Signature)	*(Signature)*	*(Signature)*
.....	.....	.....
Nectarios Koziris	Dionisios N. Pnevmatikatos	Georgios I. Goumas
Professor	Professor	Associate Professor

Athens, April 2024

National Technical University of Athens
School of Electrical and Computer Engineering
Division of

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism.

(Signature)

.....
Markos-Georgios Ramos

5 April 2024

Abstract

Given the increasing difference in processing speed between the main memory and the CPU, the role of CPU caches in achieving the maximum possible processing speed is as big as ever. Every cache access that doesn't find the data has to invoke the main memory, using tens, if not hundreds, of machine cycles. Predicting the behavior of a process prior to execution on different cache architectures is an important task. It will help determine what processor will work best for a program or how to distribute computing power optimally. The immediate goal of this thesis is to propose a machine learning model that will predict as accurately as possible the cache misses of a program depending solely on the cache architecture, the program code, and the reuse-distance histograms. To achieve this goal, we propose a number of transformer models that combine the transformed input code with information within the reuse-distance histograms and lead to an output cache miss ratio. We trained and tested these models with simulated data and produced high-accuracy predictions for a large number of replacement policies and LLC cache sizes. These networks can act as a useful tool in cache prediction and may be used on real machines in future research.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem introduction

Let's delve into the dynamic realm where the architecture of a computer isn't just a blueprint but a catalyst, wielding immense influence over performance outcomes. From the intricate dance of processors to the labyrinthine pathways of memory, every architectural facet orchestrates a symphony that determines the speed, efficiency, and transformative power of computing. Understanding this intricate interplay opens doors to unlocking unparalleled performance potentials, reshaping the digital landscape one architectural innovation at a time.

When establishing the pivotal role of computer architecture in shaping the performance landscape, it's imperative to recognize the intricate dynamics behind this technological symphony. Computer architecture goes beyond being a mere blueprint; it serves as the cornerstone dictating the efficiency and capabilities of modern computing. To comprehend its profound impact, it's essential to examine the intricate orchestration of processors and memory pathways, exploring how each architectural facet harmonizes to define the speed, efficacy, and transformative potential within the digital realm.

The relentless pursuit of faster and more efficient CPUs has been a tale of continual innovation and technological advancement. Since the inception of computing, the quest for speed has driven engineers and scientists to push the boundaries of what's possible. Initially, CPUs were constructed using basic electronic components. However, as technology advanced, the late 20th century witnessed a monumental shift with the advent of integrated circuits and microprocessors. Moore's Law, formulated by Intel co-founder Gordon Moore in 1965, predicted the doubling of transistor counts on a chip approximately every two years. This principle became a guiding force, driving the industry's aspirations for rapid advancements. Over time, the manufacturing process underwent significant enhancements, transitioning from larger transistors to smaller, more densely packed ones, leading to increased processing speeds and improved efficiency. Innovations in semiconductor technology, such as the development of silicon-based chips, the introduction of multicore processors, and the refinement of architectures through pipe lining and parallel processing, have been pivotal in the relentless march towards faster CPUs. Additionally, advances in materials science, nanotechnology, and chip design methodologies have collectively contributed to the ongoing evolution of CPUs, fueling an

era of computational power that continues to redefine the boundaries of technological possibility.

CPU caches serve as a crucial mechanism to alleviate bottlenecks in performance by minimizing the time it takes for the processor to access frequently used data. When a CPU performs operations, it constantly fetches data and instructions from memory. However, accessing data directly from the main memory can be relatively slow due to the speed difference between the CPU and the memory. Caches act as a high-speed intermediary between the CPU and the main memory, storing frequently accessed data and instructions. By doing so, they reduce the latency in retrieving information required by the CPU. When the processor needs data, it first checks the cache. If the required data is found in the cache (cache hit), it can be accessed much faster than fetching it from the slower main memory. This avoids the need to wait for data from the main memory, thus alleviating the performance bottleneck caused by memory access latency. Effectively utilized caches optimize the CPU's efficiency by reducing idle time, allowing for quicker access to data and enhancing overall system performance across various computational tasks.

Cache misses induce substantial time penalties in computational workflows. When the CPU seeks data that isn't stored in the cache, it triggers a cache miss, prompting the processor to pause its execution while retrieving the required information from the slower main memory. This transition between the cache and main memory incurs a notable delay due to the significant speed gap between these memory tiers. This delay, termed the cache miss penalty, results in tens if not hundreds of idle processor cycles, hindering the smooth execution of instructions and impeding overall system performance. Therefore, identifying and elucidating performance bottlenecks that result from cache misses is crucial.

1.2 Suggested solution

The diversity in cache designs, sizes, mapping strategies, replacement policies, and technological advancements among different CPUs results in varying cache miss behaviors, making it essential to consider the specific characteristics of each CPU architecture when analyzing cache performance.

Cache miss behavior can fluctuate significantly across different CPUs due to variations in cache architecture, size, organization, and access policies. Different CPUs may employ diverse cache designs, such as different levels of cache (L1, L2, L3), cache sizes, associativity, replacement policies, and prefetching strategies. These architectural discrepancies can impact how cache misses occur and their subsequent penalties.

Moreover, architectural advancements and innovations in newer CPU generations often introduce optimizations aimed at reducing cache misses. Improved prefetching techniques, smarter prediction algorithms, or changes in cache hierarchy can impact how cache misses manifest on newer CPUs compared to older ones.

Simulation of a CPU cache serves as an indispensable tool in modern computing, offering crucial insights and optimizations. By mimicking the behavior of caching mechanisms, simulations enable in-depth analysis of cache performance, aiding in the design,

evaluation, and optimization of cache architectures. These simulations facilitate the exploration of diverse cache configurations, replacement policies and associativity allowing engineers to pick the most efficient setups for specific applications. Moreover, cache simulations offer a deeper understanding of how different workloads interact with the cache and impact the cache misses.

The process of simulating CPU caches can be time-consuming due to various factors. The complexity involved in emulating cache behavior requires significant computational resources and time. Handling large traces of memory access patterns for accurate simulations adds to the time needed for processing. Furthermore, incorporating intricate cache designs, such as multiple cache levels (L1, L2, L3), diverse cache sizes, associativity schemes and advanced cache algorithms, contributes to the computational complexity and duration of the simulations. Hence, these simulations prove to be extremely time consuming and a better way of computing miss ratio is needed.

The solution to this problem are prediction mechanisms. The importance of a model that can accurately predict the miss ratio of a workload without having the burden of simulating each aspect of it is immense. Almost all of the existing models for predicting miss ratios are created for LRU or random replacement policies [3, 4, 1]. The only existing model [3] that can use other replacement policies than LRU, needs to know, prior to execution, analytically all the configuration details. This leaves a large part of the problem unanswered, since most modern CPU processors don't use the LRU replacement policy in their caches and, in most cases, we don't know what replacement policy is being used.

1.3 Thesis structure

In this thesis we are tackling this problem of predicting cache misses on any given architecture by using machine learning. We propose a number of different networks that have the goal of estimating cache miss ratios as accurately as possible. Inspired by StatCache 2.2, the existing probabilistic approach to this question, we decided to use the same data that it uses for its prediction to train and evaluate a machine learning network. The current rise in popularity of NLP networks motivated us to add a transformer branch to the networks, that uses this new technology of understanding the source code, to obtain better results.

We created a total of three different network architectures that predict the cache miss ratio of any program for set cache sizes with any replacement policy. These networks perform really well for any replacement policy. We also introduced one that predicts the miss ratios for any cache size and any replacement policy, whose function we showcased on LRU. In the case of LRU, all of these networks predict the cache miss ratios significantly better than the existing probabilistic approach to the question.

This thesis is structured into five main chapters, each contributing uniquely to the exploration and analysis of miss ratios. Chapter 1 introduces the fundamental concepts and the theoretical framework that supports the study, presenting an overview of the historical context and the significance of predicting cache misses. Chapter 2 delves into the existing literature, critically examining prior research that will help understand what

this thesis is trying to address. Chapter 3 outlines the research methodology adopted, detailing the chosen approach, data collection methods and network architectures used. It also holds individually the results for each one of the networks that we will propose. Chapter 4 presents the collected results derived from our networks, offering an analysis and comparison of the data collected. Finally, Chapter 5 synthesizes the findings, discusses their implications, and offers conclusions along with recommendations for future research cache miss ratio prediction.

In summary, this thesis aims to use the current advancements in machine learning in order to predict the cache miss ratios of any program on any cache architecture. By exploring possible solutions to this problem such as the probabilistic approach or NLP algorithms, this study intends to shed light on the difficulty and the many aspects of predicting cache misses accurately. The following chapters will examine these concepts in detail. Chapter 2 will focus on related work that has been done, while Chapter 3 will delve into our proposal to solve this issue. This structured approach will provide a comprehensive understanding of predicting cache misses. Now, let's proceed to delve deeper into these areas, starting with the StatCache probabilistic approach.

Chapter 2

Related Work

2.1 Cache hierarchy

Back in the history of computer development, CPU speeds were outpacing memory access speeds. This discrepancy led to idle CPUs waiting for data from main memory. Enter cache memory, a concept first proposed by British computer scientist Maurice Wilkes in 1965 [5]. Early cache models improved data access latency, but making main memory entirely high-speed was prohibitively expensive. Researchers explored better designs, eventually leading to the idea of multi-level caches. These multi-level cache models, such as the three-level caches found in Intel's Core i7 products, strike a balance between cost and performance. Now, CPUs can tap into a hierarchy of caches, each level serving as a buffer between the processor and main memory, ensuring efficient data flow and faster execution.

Each one of these caches has its own size, and inner structure, independently of caches of other levels. There are many ways to configure a cache as to the structure. The important ones that we are going to discuss are the size, associativity and replacement policy. For the size, its only dependency is to be larger than the size of the cache in the lower levels. Generally meaning that the size(L3) > size(L2) > size(L1). This is because the speed of lower level caches is supposed to be higher and therefore a smaller cache is required. Associativity is referring to the inner structure of this cache, specifically, it deals with the methods used to determine where data can be stored within the cache. In our case, we will be using set-associative caches, but there also exist fully-associative and direct mapping caches.

Lastly, it is important to define the replacement policy of the cache. Replacement policies are algorithms or strategies used to decide which cache entries to replace or evict when new data needs to be loaded into a full cache. These policies are crucial for maintaining the efficiency and performance of the cache system, as they directly influence the hit rates of cache accesses. LRU is the most known replacement policy that removes the cache entry that has not been accessed for the longest time. Modern processors use more modern replacement policies that are often not known to the user of a CPU. Therefore, we will be using a range of different replacement policies in this thesis, to show that the predictions consistently exhibit high quality across various machines. SHiP [6], SRRIP [7] and Mockingjay [8] are modern replacement policies that aim at predicting the

reuse of a cache entry and replace the least likely to be reused.

2.2 StatCache

There have been statistical attempts of estimating cache-miss ratios for the LRU replacement policy. Namely, StatCache [1] offers a probabilistic approach towards this goal using the reuse distances of said program.

Firstly, it is necessary to state that the data is stored in cache-line-sized pieces of data within the memory. We denote as a memory access the access to a cache-line-sized block. Let N be the number of memory accesses that occur during the execution of the program. We can enumerate those accesses from 1 to N. Suppose i < j < N where i and j accesses to the same block and no accesses to the same block have occurred between i and j. We can say that the reuse distance of this is i-j-1, or, equally, the accesses that happened between 2 consecutive accesses to this block are i-j-1. We gather these reuse distances into a histogram h, where h(i) denotes the accesses that have a reuse distance of i.

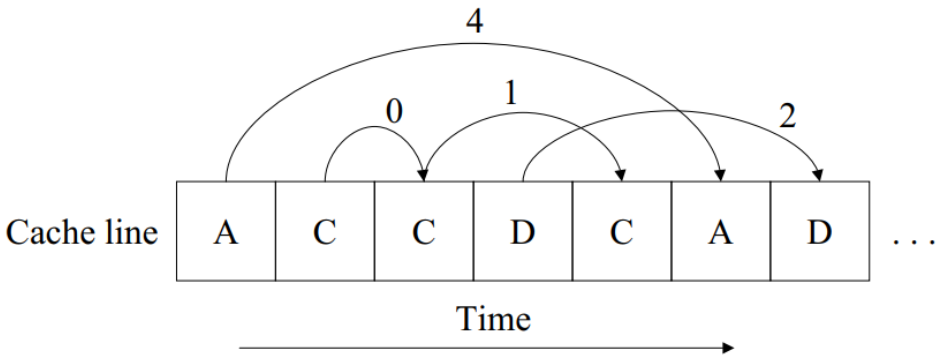


Figure 2.1. *The figure illustrates the reuse distances. The arrows indicate reuse of cache lines, and the numbers next to the arrows are the corresponding reuse distances assigned to the memory references pointed at by the arrows [1]*

Using those reuse distances, a mathematical equation is derived:

$$RN \approx h(1)f(R) + h(2)f(2R) + h(3)f(3R) + \ldots \tag{2.1}$$

where R is the miss ratio, N is the highest possible reuse distance, h is the aforementioned histogram where h(i) tells us that there are h(i) references with reuse distance i.

F(n) denotes the probability of the cache line not staying in cache after n misses. So, assuming the cache is fully associative and has L cache lines, the probability that a line will not remain in the cache after n misses is:

$f(n) = 1 - (1 - 1/L)^n$

This is basically StatCache's probabilistic approach to the question. Each reuse distance together with its probability of staying in the cache computes a number of cache misses that, summed up, give the total misses of the execution. Benefits of this method of

estimation are that it's simple and quick to calculate. The main downside of this method is that it only works for the LRU replacement policy.

2.3 Neural Networks

2.3.1 Other usages

Deep neural networks have shown their capabilities of analyzing and understanding complex patterns in multiple kinds of problems, such as image classification [9, 10]. In our case, helpful are researches that are concerned with computer code [11, 12].

In parallel with these, there is also a rise in popularity of neural networks within the contexts of computer architectures. For example, many compiler optimizations have resulted from the implementation of machine learning techniques. In instruction scheduling, the preference function of one scheduling over another can be computed by the RL algorithm's temporal difference [13]. The LSTM-based model [14], circumvents manual feature engineering, autonomously acquiring compiler heuristics from raw code. This enables the construction of appropriate embeddings for programs while simultaneously mastering the optimization process.

Many different machine learning networks have been used for predicting similar problems to ours [15]. Dong et al. [16] use artificial neural networks to predict higher-level features (like miss of cache read/write and instructions per cycle) from-lower level features (cache associativity, capacity and latency) for non-volatile memory based cache hierarchies. Other machine learning networks have been introduced to help predict efficient resource allocation [17] and task scheduling [18], to always select the path of maximal instructions per cycle.

Numerous introductions of machine learning into the realm of computer architectures, both similar and distinct, have sparked our curiosity. This motivation led us to explore the cache miss ratio prediction problem-solving through the lens of machine learning.

2.3.2 Multi-Layered Perceptron

Via stimuli that happen on receptors all around the body, the human brain can reach several conclusions about the environment and on what actions to take accordingly. Inspired by the structure of the human brain, researchers proposed a structure that would process information similarly to the human brain, by breaking it down into smaller chunks and understanding the dependencies between those. These structures revolutionized the modern era of computing. Due to their similarity to the human brain, they were named neural networks and proved to have significant success in understanding massive data sets.

Just like a physical neural networks consists of neurons, neural networks in artificial intelligence consist out of many smaller parts, that cooperate to create a system. One of those many smaller parts is called a perceptron [19]. Perceptrons are so called neurons that execute one simple function each. They take as input a number of values, say x_i where $i \in [1, n]$. These values get multiplied by a weight, say w_i where $i \in [1, n]$. Then the

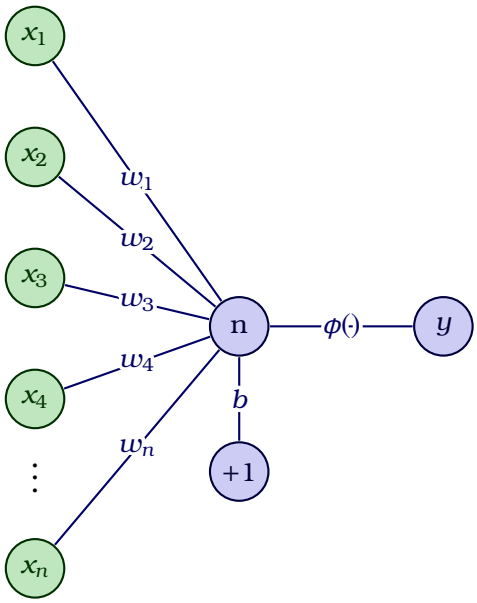


Figure 2.2. *One perceptron neuron. X_i are the inputs to the neuron. Each input is multiplied by the weights W_i on the connecting arrow. Each neuron has one bias b. The output y is derived by applying the activation function $\phi()$ over the sum of weighted inputs and the bias.*

output is computed by adding all those values together, adding an extra weight b that's known as a bias and applying a function φ onto the result, aka the activation function.

Meaning, each perceptron neuron derives it's output from the formula:

$y = \varphi(\sum_{i=0}^{n} x_i * w_i + b)$

This is a linear equation which can also be described as a multiplication of two vectors $x * w^T$, where $x = [1, x_1, x_2, ..., x_n]$, $w = [b, w_1, w_2, ..., w_n]$.

It is clearly visible that the parameters we can control are the weights of the vector w. These are the parameters that are changed according to the data set to compute the best possible output and are usually initialized to random float numbers and fitted in training.

Feeding the same data into a large number of perceptrons creates a group of perceptrons where each neuron will compute a different output according to the value of it's weights. All the outputs of those perceptrons are collected and fed into either another group of neurons or an output neuron that will calculate the final output of the whole structure, according to their outputs. This group of neurons is called a layer and since it's output is not visible to the outside, but fed into the next layer it's named a hidden layer. If more than 2 hidden layers exist in a network it is called a deep neural network [20]. Deep neural networks are particularly interesting in cases of difficult problems, since they detect more similarities and connections between the input data than a shallow perceptron can with only one hidden layer. This is because the modular function is applied more times to each data point, discarding for each perceptron layer more unimportant features of previous layers. Therefore, since the multi-layer perceptron is either way the multiple application of the above function, the understanding of the data will be more accurate and more efficient.

Fitting such a network to data is called training and it is the process of changing

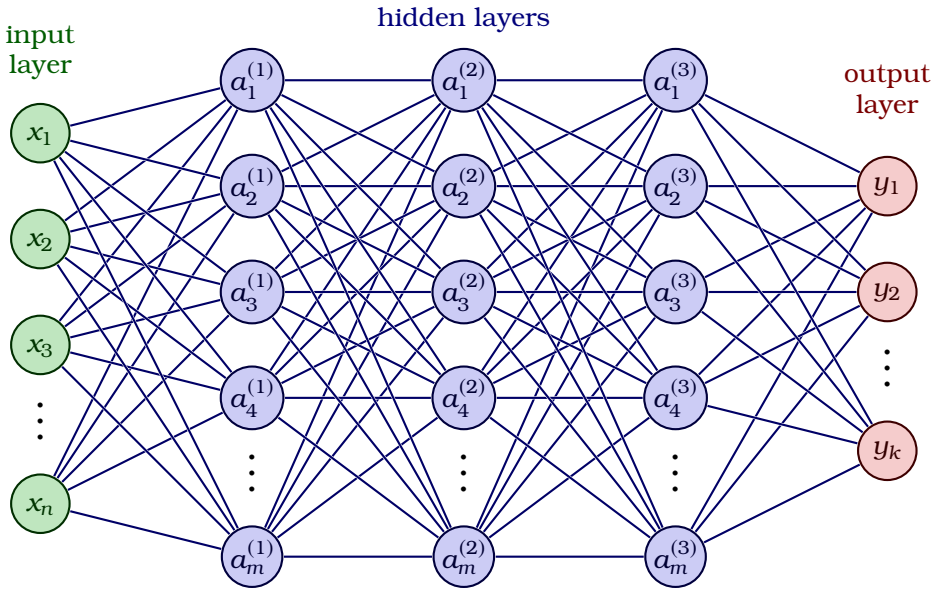


Figure 2.3. *Multi-Layered perceptron with 3 hidden layers*

the parameters w_k^j for each of the perceptrons k in each layer j of our system with the goal of achieving the least amount of possible error. To train a MLP a process called backpropagation is used.

Forward propagation is the calculation of the output. Since we are using a MLP where each neuron applies the function $y_i = \varphi(\sum_{i=-0}^{n} x_i * w_i + b)$, we just have to substitute x_i and w_i with the values of the previous layer. Therefore, if the intermediate values of the layer j are:

$a_k^j = \varphi(\sum_{i=1}^{m} a_i^{j-1} * w_{k,i}^j + b)$, $k \in [1, layer\_size]$, m = neurons of layer k-1.

Or, equivalently:

$a_k^j = \varphi(a^{j-1} w_k^j)$, $w = [b, w_1, .., w_n]$, $a^{j-1} = [1, a_1^{j-1}, .., a_n^{j-1}]$

The first layer a^1 relies solely on the input layer x and the known weights, therefore we can calculate it's values. Then the next layer can be computed and so on, until we reach an output y (aka a^L where L denotes the last hidden layer). The output of the forward propagation, has a deviation from the real value of y - v and a loss of C(y - v), where C is a loss function and v the real value.

In the backpropagation step derivatives are being used to fit the values of the weights.

Firstly, if we declare the weighted input of a layer l as z^l, a derivative of the loss towards the input can be expanded to:

$\frac{\partial C}{\partial x} = \frac{dC}{da^L} \cdot \frac{da^L}{dz^L} \cdot \frac{dz^L}{da^{L-1}} \cdot \ldots \cdot \frac{da^1}{dz^1} \cdot \frac{\partial z^1}{\partial x}$

The derivative of $\frac{dz^l}{da^{l-1}}$ equals to the weights that are used for this layer w^l. Additionally, the derivative $\frac{da^l}{dz^l}$ denotes the derivative of the activation function φ. Hence, the previous equation can be denoted as:

$\nabla_x C = \nabla_{a^L} C \cdot (\varphi^L)' \cdot (w^L)^T \cdot (\varphi^{L-1})' \cdot \ldots \cdot (\varphi^1)' \cdot (w^1)^T$

Here, we declare an additional parameter δ^l which stores the values after the layer l:

$\delta^l = \nabla_{a^L} C \cdot (\varphi^L)' \cdot (w^L)^T \cdot (\varphi^{L-1})' \cdot \ldots \cdot (\varphi^{l+1})' \cdot (w^{l+1})^T \cdot (\varphi^l)'$

Clearly, δ^l has a size of the neurons in layer l. Each one of those neurons gets a value

and that value is interpreted as the contribution of this neuron towards the computed output and therefore loss. Also this helps in the recursive computation, since we can express δ^{l-1} as :

$\delta^{l-1} = (\varphi^{l-1})' \cdot (w^l)^T \cdot \delta^l$, and

$\delta^L = (\varphi)' \cdot \nabla_{a^L} C$

which is computable. Lastly, due to the layer output of layer l-1 being left when we differentiate over the weights of layer l, the gradient of the weights of layer l is:

$\nabla_{w^L} C = \delta^l (a^{l-1})^T$

As previously mentioned, this whole process is called backpropagation and the objective is tweaking the weights of our network after each iteration over an object in the training dataset to give a better result. That is how a neural network learns.

2.3.3 Long Short-Term Memory

Recurrent neural networks are another type of neural network, different from the previously mentioned MLP. RNNs consist of neurons that perform a feedback operation. This means that neurons, together with passing their result to the next layer to create the forward passing, just like the aforementioned MLP networks, they re-feed the output as input to themselves. This concept has been applied to many areas of interest, such as handwriting recognition [21] or acoustic modeling [22]. Similarly, we hypothesize that this method could help our model understand the reuse distances better since the reuse distances are not incoherent data-points that just exist; they are a sequence where consecutive buckets have similar access points.

The most known type of RNNs are the Long Short-Term Memory models. These were introduced in 1977 [23] and has since been used in multiple applications. Goal of these networks is combating the vanishing gradient problem. This is a problem that occurs in RNNs where one gradient is applied multiple times and if it's small it will multiply itself into insignificance. This is a problem which will cause parts of the network not being able to fit [24].

LSTM units take as input 3 parameters one being the input step from the input sequence and the other 2 a cell state and a hidden state. The cell state is the important one, which holds the information passed to our unit from previous time steps. The hidden state is the state that holds the output of the previous timestep and that will hold the output of this timestep once finished.

Each unit of a LSTM network goes through 3 stages in it's execution.

Firstly, the forget state determines how much of the input state C_{t-1} will be forgotten. It executes the formula:

$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$

Basically, this takes into account the h_{t-1} and x_t, then, due to the sigma function, computes a number between in the range $[0, 1]$ and multiplies the C_{t-1} state with it.

Second phase is the input gate. This is when the new information is being added to the cell state $f_t * C_t - 1$ to compute it's new value C_t. For this purpose, we need to calculate a $\tilde{C_t}$ value from the inputs x, h_{t-1} and a multiplier i_t which is going to decide how big the

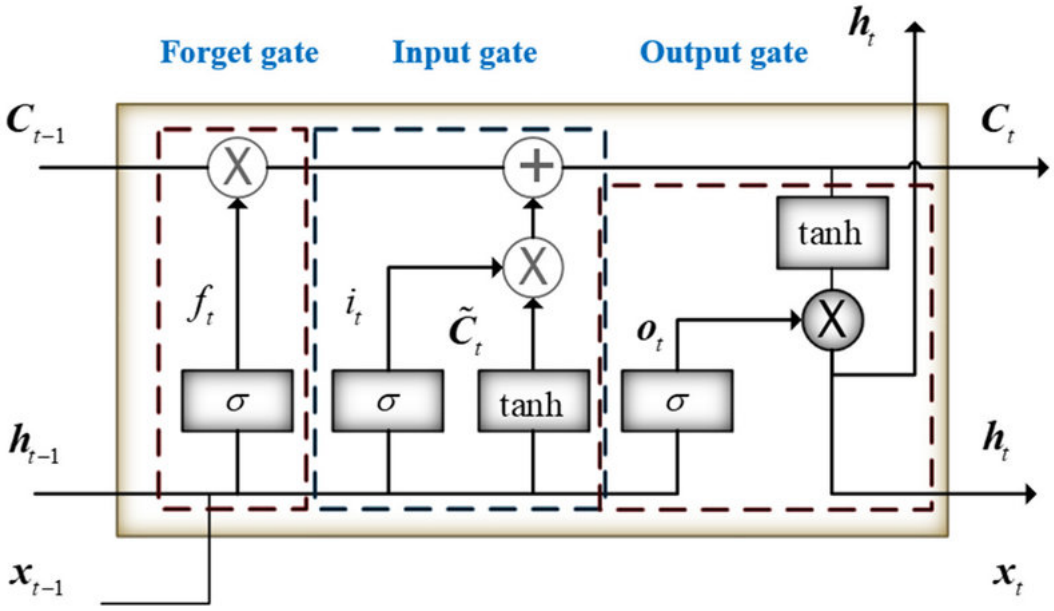


Figure 2.4. *LSTM Unit*

impact of the new state will be. The way to do that is just as before, using the respective functions and weights.

$\tilde{C}_t = tanh(W_t \cdot [h_{t-1}, x_t] + b_t)$

$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$

Consequently, the current state is computed as:

$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$

Lastly, we have the output phase. To compute an output given an input we'll just apply tanh to the cell state and multiply it with a factor that results from the previous hidden state.

$h_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \cdot tanh(C_t)$

This is the output h_t that will be forwarded to next layers.

2.3.4 Convolutional Neural Networks

Convolutional Neural Networks are a fundamental component of image recognition and classification. They appear in all sorts of problems involving image classification and or recognition [25, 26] , but they appear in other sectors as well such as stock market analysis [27] or NLP [28]. They use a simple yet effective method of tackling the problem of having too many parameters and limited memory. This method involves convolution from which the name is derived.

First step, is creating a kernel of a specific size, which holds the weights that we want to create a convolution with. Then, this kernel is moved over every possible position of the input matrix and make a convolution between it's values and the input matrix's values. A convolution is simply a sum of the multiplication of each value that is covered by the kernel times the value of the kernel in it's respective cell.

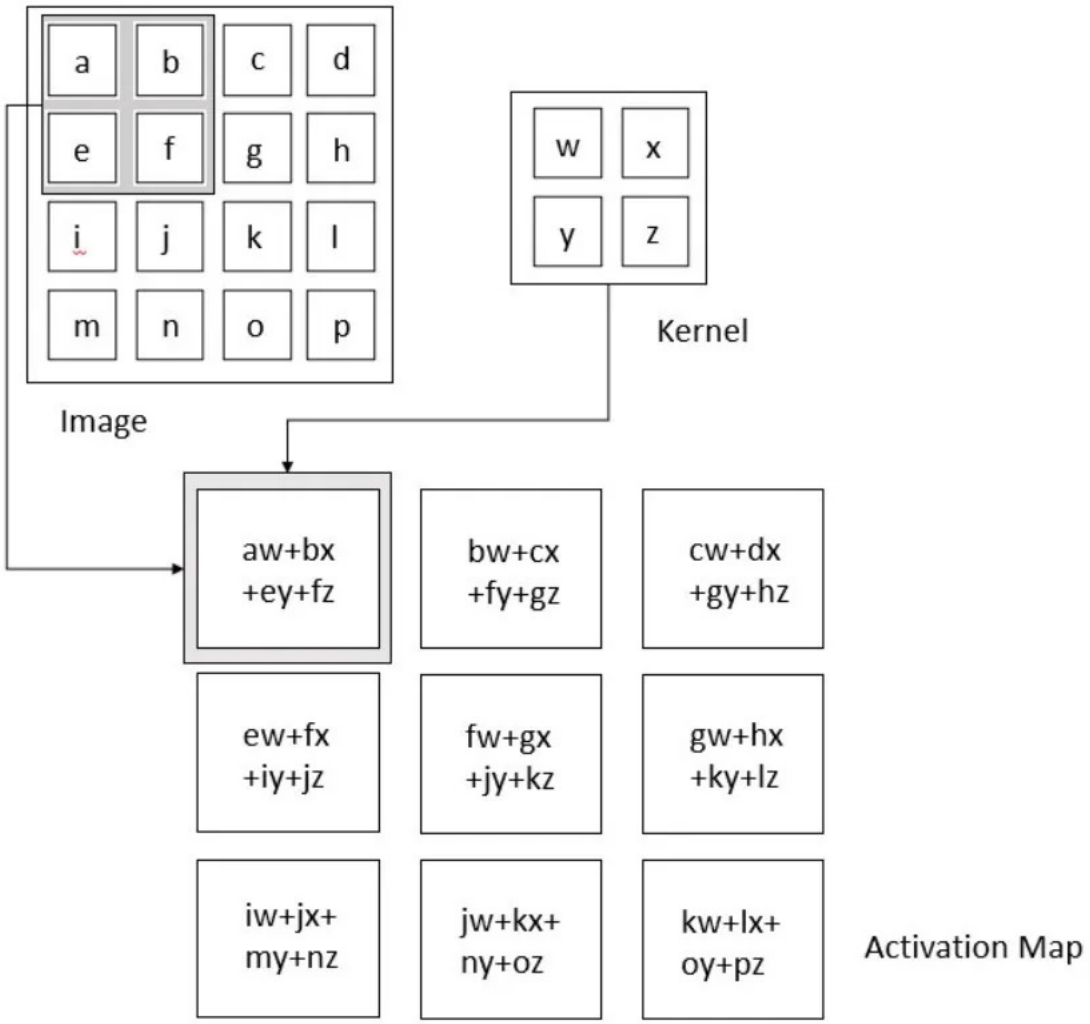


Figure 2.5. *Convolution Operation [2]*

After a CNN layer a pooling layer is often added [29]. Max and average pooling layers are the most frequently used. It has been shown that max-pooling layers accelerate convergence, enhance generalization as well as selection of better invariant features by [30]. This has also been mathematically proven by [31].

These pooling layers, unlike other layers that have been discussed, have no weight parameters and aim at lowering the data dimensions. They construct a Kernel of shape (K_h, K_v) and apply it to the input matrix in contiguous rectangular areas. To each one of those clusters the function from which the layer takes it's name is applied, i.e. a max-pooling layer takes the max from each cluster it covers and an average-pooling layer the average. This reduces the dimensions of the CNN's output by (K_h, K_v) respectively.

2.4 Embeddings

To use the program code as input for our network we will need to convert it into usable data by a neural network. Embeddings are these representation of objects as vectors. A function converts an object which is recognizable to a human into a vector of numbers,

which can be recognized by a program with the least possible loss of data. This method of representation is used frequently in NLP where each word in the dictionary is converted into a number and fed into a neural network.

A similar technique has been developed for creating embeddings of code [32, 33] where the symbols of code are being used to convert a sequence of code symbols into the embedding vector. Each symbol takes a value from 0 to N where N is the arbitrary number of symbols we want to consider (the rest we discard) and this is known as a vocabulary. Then this conversion of symbols will be used by some mechanism to transform those arrays of symbols into a vector with the least possible loss of data.

In our case we will be using the IR2Vec [32] for the creation of our embeddings. IR2Vec is an embedding calculator that has the goal of creating a vector representation as accurate as possible of an IR file. IR file stands for intermediate representation file. It is the file that the input code is transformed into by the compiler before it gets transformed to machine code. In this stage of compilation the code has already been optimized and all the flags have been applied which makes it the ideal starting point to create an accurate vector representation of the program.

Via unsupervised learning the authors of IR2Vec created a network that identifies and represents these IR files accurately. Their network was trained and tested on the SPEC CPU 17 benchmarks and Boost library. Their experimental outcomes ensure the practical viability of this network in our problem, since our experimental data also stems from the SPEC benchmark suites as we'll discuss later on. Thus, we are going to use this network and vocabulary as a black box to transform the source code of our dataset. Output of this procedure is a vector of size 300 which is supposed to give our model vital information about the proceedings of the program and how to understand the reuse distances.

Chapter 3

Approach and Methods

3.1 Overview

Our goal with this thesis is to predict the cache miss ratios given a program and a cache architecture. To use a program as input, we need to convert its features into usable data for our network to predict from. For this purpose, according to StatCache 2.2, we need the reuse distances of each program in our benchmark set. Reuse distances, as mentioned previously, are the distances between consecutive accesses to the same address in memory. These distances depend mainly on the program and its inputs; they vary very little between different architectures and therefore will only be captured once from each trace of a program. They are going to be stored in a histogram, and one branch of the network will try to predict the miss ratios using them.

Next, we need to transform the input code into a vector (embedding) so that the network can retrieve even more information from the input code about the structure of the program, the flow of data, etc. To predict the cache miss ratios accurately, we need to interpret the transformed data appropriately. The embeddings vector can be fed into a dense neural network, so this is what we do for the transformer branch. The reuse distance histograms, on the other hand, are more complex to understand; a simple dense neural network is not the most well-suited network to fully understand the complexity of this problem.

For this purpose, we introduce three different models that understand reuse distances differently. The first one is a shallow multi-layered perceptron that computes the miss ratio solely based on reuse distances. Next, we introduce a double LSTM layer and a CNN network that has the purpose of better understanding the reuse distances given the neighboring reuse distances. These two networks compute the cache miss ratio better, using the reuse distances as well as the transformer branch.

Lastly, we introduce a deep MLP that takes as input the reuse distances and embeddings, introduces the LLC size as a parameter, and computes the miss ratios for other cache architectures with the same replacement policy.

3.2 Experimental setup

3.2.1 Simulator

The initial input data for our system consists of the SPEC 2007 and SPEC 2016 benchmark suites [34, 35]. SPEC are benchmark suites designed to provide a comparative measure of compute-intensive performance across the widest practical range of hardware using workloads developed from real-user applications. These suites are perfect to showcase our goal since they provide a large variety of problems and algorithms covering a wide range of computing tasks.

To simulate these programs, we use ChampSim, a processor simulator aimed primarily at simulating the memory subsystem and branch prediction as accurately as possible [36]. We use this simulation tool to collect the runtime data that we will need in the next section. It executes the trace of a program on a simulated machine based on a given architecture. For this purpose, we use the traces given by the 3rd Data Prefetching Championship. These traces consist of 2 billion instructions each and cover large parts of the execution of a benchmark. Each benchmark has anywhere from one to six traces to its name, each representing a given percentage of its execution. Later on, we'll use this percentage as a weight to compute the cache misses of a benchmark as a weighted average of the miss ratios of its traces.

The cache that suffers the longest miss latency is the LLC. This makes the prediction for it far more valuable than for other-level caches. Therefore, in our approach, we keep the lower-level caches invariant. L1D is of size 48KB, associativity of 12 and LRU replacement policy, and the L2 cache has a size of 512KB, with associativity 16 and LRU replacement policy. We conducted the simulations mentioned in the next chapters with a wide range of cache sizes and four replacement policies for the LLC to cover a large set of real-world architectures. We used the cache sizes mentioned in table 3.1. The LLCs with sizes of 768KB, 1536KB, 3072KB, and 6144KB are 12-way associative, whereas the 1024, 2048, 4096, and 8192KB-sized caches have an associativity of 16.

Replacement policy	LLC size [KB]
LRU	768, 1024, 1536, 2048, 3072, 4096, 6144, 8192
SHiP	1024, 2048, 4096, 8192
SRRIP	1024, 2048, 4096, 8192
Mockingjay	1024, 2048, 4096, 8192

Table 3.1. *LLC cache configurations*

3.2.2 Dataset

Collecting reuse distances

The collection of reuse distances during run time is pretty simple. We implemented a reuse distance profiler to be used within ChampSim. It is given an address and a state, and it calculates the reuse distance. We did this via a simple function that checks the

last time this block was accessed and increments the reuse distance in the histogram. Afterward, the data address gets stored for future reuse distance calculations, and we continue. This implementation is lightweight, since nothing but a few instructions are added on execution, and effective, since efficient data structures are used, such as hash maps.

To use this reuse distance profiler, we added it to the replacement policy of L1D that executes this aforementioned process every n-th time the cache is accessed. This n has to be small enough to get a representative view of the reuse distances in our dataset, but not too small, since we don't want to run the reuse distance profiler for each data access because it increases processing time. In our case, we use $n = 16$ accesses. This is a bit too meticulous for 512k accesses, but we wanted to ensure that the outputs don't get affected by this and thus sacrificed some processing time.

The fluctuation of cache misses throughout the execution of the program can be an issue that negatively influences our estimation. As the program goes through the different execution phases, the cache miss ratio changes over time. According to the StatCache paper [1], dividing the profilers' output into windows with a smaller number of accesses is the solution to this issue. The rationale behind it is that the number of accesses will be small enough so that each window's cache miss ratio is likely to be constant over time.

To accomplish this, we simulated the traces, and for every 512k accesses, we returned the state of the reuse distance histogram. We noticed that this size works best in our case since it gives the simulation time to fill part of the caches and still sample the program's execution frequently enough to not notice the fluctuation. Now, for each benchmark trace, we have a number of windows with 512k accesses each. Each window consists of the reuse distances that occurred within those 512k accesses and the LLC miss ratio that those correspond to. Let it be known that some traces have far fewer than 512k accesses within their execution. We used a window for each one of these traces regardless, so that their data doesn't get lost, but some of them will result in some problems later on due to their different scale.

To calculate the overall miss ratio of a benchmark, one has to average the miss ratio of every window in a trace. This miss ratio is calculated by *cache misses/cache accesses* to the LLC over each window's period of 512k accesses. Then, each trace's miss ratio is multiplied by a weight, which represents the percentage of the total execution of the benchmark that this trace simulates. The sum of these miss ratios multiplied by the trace's weight constructs a weighted average, which will be the miss ratio of the whole benchmark.

As we can see, the average cache miss ratio of all the benchmarks is at 90% for the smallest cache and keeps gradually getting lower, until 55%. This is an immediate consequence of the fact that, on average, larger caches result in lower miss ratios since they can store more information.

To summarize and be more precise, our dataset contains 1 to 373 windows for each one of the 186 traces that result from 47 benchmarks. Each window contains the histogram of reuse distances and the cache miss ratios that were captured during 512k continuous memory accesses during the benchmark's execution. The cache miss ratios are numbers

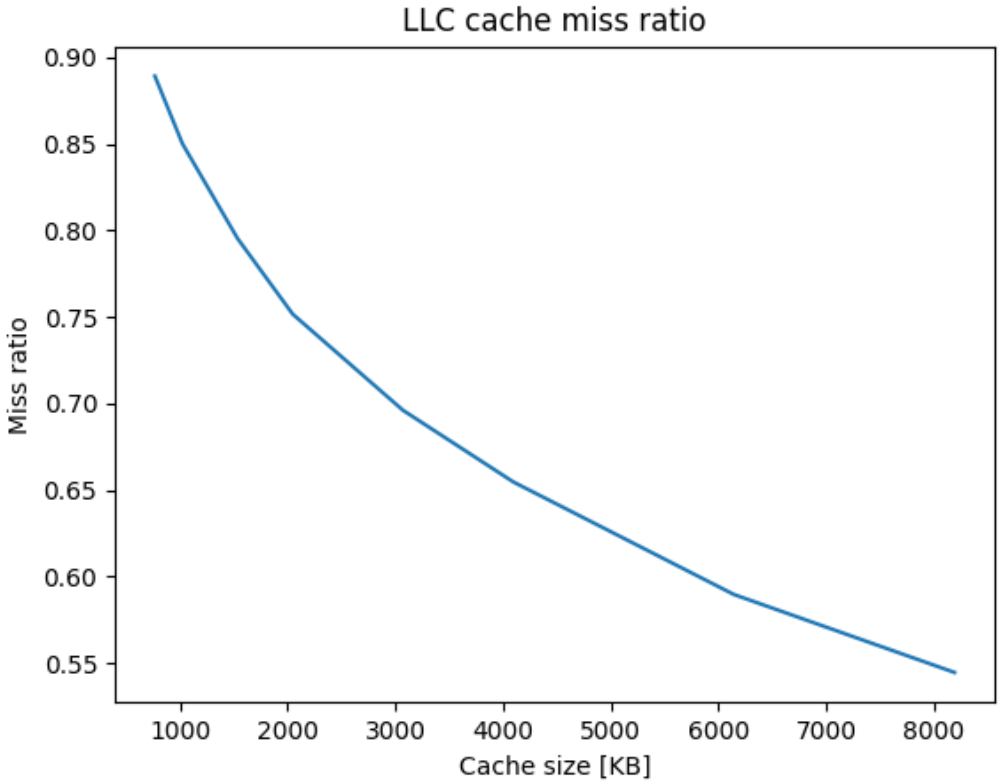


Figure 3.1. *Average miss ratio of all benchmarks with LRU replacement policy*

in the range [0, 1] based on the simulated values of ChampSim for each one of the LLC configurations mentioned in Table 3.1. These reuse distances will be the input for the first branch of our network, and the miss ratios are the real output onto which the output of the network will be fitted in training and compared against in testing.

Embeddings

Each benchmark's code needs to be transformed into a form that can later be converted into an embedding. For this reason, we compiled the SPEC files via clang-16 into IR files. LLVM IR files are essentially intermediate assembly instructions that are used to create the compiled code. The compiler options we used to generate these IR files are the default ones suggested by the SPEC toolchains.

We will be using the pre-trained model of IR2Vec for the transformation of the code into vectors. With it, we create 47 vectors (one representing each benchmark) of 300 values. This is the transformer branch of our networks. This helps the rest of the network, via the program code, understand program properties, which will help compute a more accurate miss ratio.

IR2Vec generates either one embedding for each function in the program or one embedding for the whole program at once. The benefit of having one embedding for each function is the precision as to what part of the execution we want to focus on. The program-wide embedding will convert the whole benchmark into one vector of 300 di-

mensions. In our case, we do not know the function or the part of the program that is being executed in each trace; thus, we use one vector that's the embedding of the whole benchmark and one only with the main function's vector for each benchmark.

Train-test split

Another concern is the split of the dataset, into a train and test set, which is not trivial at all. Our dataset consists of tuples. As described previously, each tuple contains a window's reuse distance histogram, miss ratios, as well as the embeddings of the benchmark that it stems from.

The reuse distances have been selected from one trace file at a time, so all the items that result from the same trace file will have a similar reuse distance structure. The same is true for the embeddings; we have collected them from benchmark-compiled code, and all the trace files that have been produced by the same benchmark with different configurations will have the same embedding file as input. For example, the traces 435.gromacs-111B, 435.gromacs-134B, 435.gromacs-226B, and 435.gromacs-228B have all been produced by the 435.gromacs benchmark from SPEC and thus have the same embedding in each item of the dataset. Due to this similarity, we can't have items of a benchmark in the train set as well as in the test set; it would skew our data since the network will have been trained on this structure.

Our initial thought for the train-test split was to separate 20%-30% of the dataset's benchmarks for the test set. But, since our dataset is pretty small, only consisting of 47 benchmarks, separating 20%-30% of the benchmarks for a test set wouldn't be optimal. It would result in a test set that is very small and, most likely, not indicative of all possible programs that the network could face. For this reason, we decided to use a method called leave-one-out cross-validation [37]. We separate one benchmark at a time and try to predict its miss ratios with the network trained on the other 46 benchmarks. This way, we cycle through the 47 benchmarks and get 47 results, one for each benchmark of the dataset.

3.2.3 Metrics

For training and testing of the networks, we used the TensorFlow libraries in Python. The processing times for the networks are from our training and testing procedures that were conducted using an NVIDIA T4 GPU.

We used a mean squared error metric for the training process. This works because, regardless of the window's weight in the final outcome, we want the least possible deviation from its estimated value. It works better than a weighted average (where the weight of a window will be $trace\_weight * 1/windows\_per\_trace$) or a mean absolute error function because it forces the network to reduce the largest deviations. This, in turn, forces the overall average deviation to be smaller.

A normalization layer can boost the accuracy of our model.There are enormous deviations between the numbers in the reuse distance histogram buckets since they are reflective of real cache accesses. A simple standard scaler that is being fit on the training

set proves very helpful for resolving this issue. This one performs a simple transformation of $z = (x - u)/s$, where x is our row, u is a row of means, and s is the standard deviation of each column. Then, we apply this scaler to the train and the test set and continue. Note that this scaler has to be trained only on the train set to not falsely create better results. Calculating the standard deviation on the whole dataset could introduce a bias that would cause the result to be better than the real one in the test set.

The training was done with an Adam optimizer [38], a stochastic method for tweaking the weights of the networks for optimization. Our learning rate was 0.001, batch size was 16, and we trained the network for 50 epochs. For these variables, we tried lots of different configurations and kept this one since it provided the most accurate results.

To evaluate how good a prediction of a network is, we are going to evaluate its absolute error from the real miss ratio. To compute this, we need to average the output of the network for each window in a trace. Afterwards, a weighted average over a benchmark has to be calculated with the trace weight and subtracted from the real value. We'll refer to the absolute value of this subtraction as the absolute prediction error. For each one of the 47 benchmarks in our dataset, an error will be computed with the training set for the other 46 benchmarks, as explained in 3.2.2. To compare these sets of errors with each other and with StatCache predictions, we will use a geometric mean. It has to be mentioned that this way of calculating the overall result of a benchmark loses some information in the data in the analysis. Some traces' differences between prediction and real value vary significantly in some benchmarks, up to 70%. Also, the windows' difference between prediction and real value can vary within a trace. These differences are supposed to even out over the multiple windows and traces.

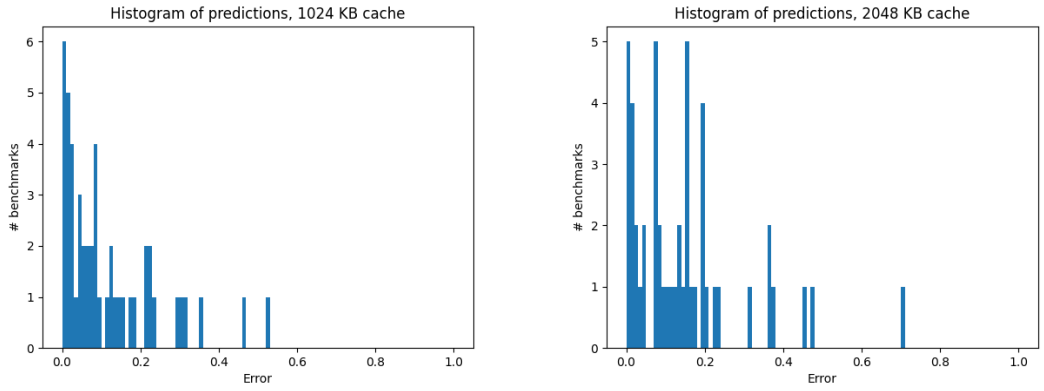
3.3 StatCache calculation

The outputs of StatCache need to be calculated. It is important to have a competitive, state-of-the-art predictive approach that also relies on reuse distance histograms to compare our results with. Comparing the two will allow us to quantify the effect of using neural networks as opposed to an analytic approach, given the same input data.

StatCache is a probabilistic approach to the problem for LRU replacement policy, and its values are obtained by the formula 2.1. When computing this formula, we get a miss ratio for each one of the windows. After averaging them over each trace and computing the weighted average over each benchmark, just like before, we compute the predicted miss ratio for each benchmark. The absolute error deviations of the predictions from the real values, as depicted in figure 3.2 for the 47 benchmarks, seem to be following a folded normal distribution. This is expected since the approach is trying to predict the value of the real miss ratio. Hence, the absolute error of this prediction should be the absolute of a normal distribution around zero. We'll use the geometric mean to summarize the distributions as one number and compare the outputs with each other. The geometric mean of absolute prediction errors from the StatCache model is 5.4%, 8.3%, 8.7%, and 8.5% for the caches of sizes 1MB, 2MB, 4MB, and 8MB, respectively.

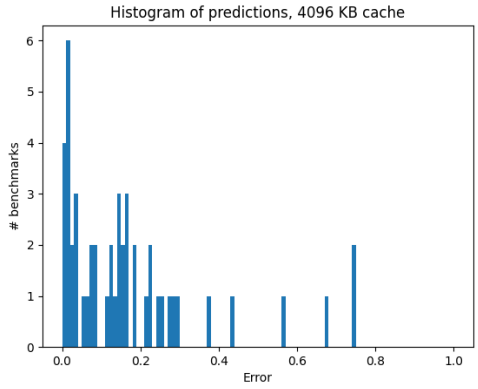
This folded normal distribution results in boxplots such as the ones shown in figure

3.3. The orange line represents the median. The interquartile range (IQR) represents 25% of the values on each side of the median. The whiskers have a maximum length of 1.5 * IQR and all values outside of this range are considered outliers. All the boxplots of absolute prediction errors that we'll discuss in this thesis will have a similar structure for this problem. The mean is very low and thus we'll have little boxes and some outliers in the boxplot.

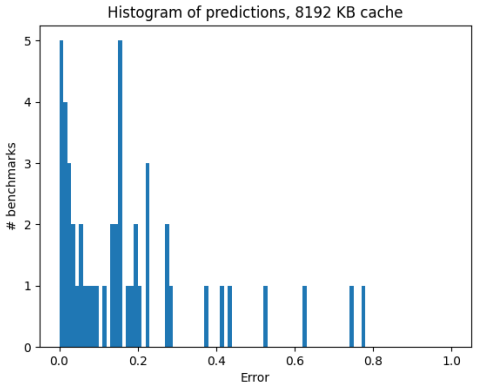


(a) 1024KB

(b) 2048KB



(c) 4096KB



(d) 8192KB

Figure 3.2. *StatCache's absolute prediction errors for LRU*

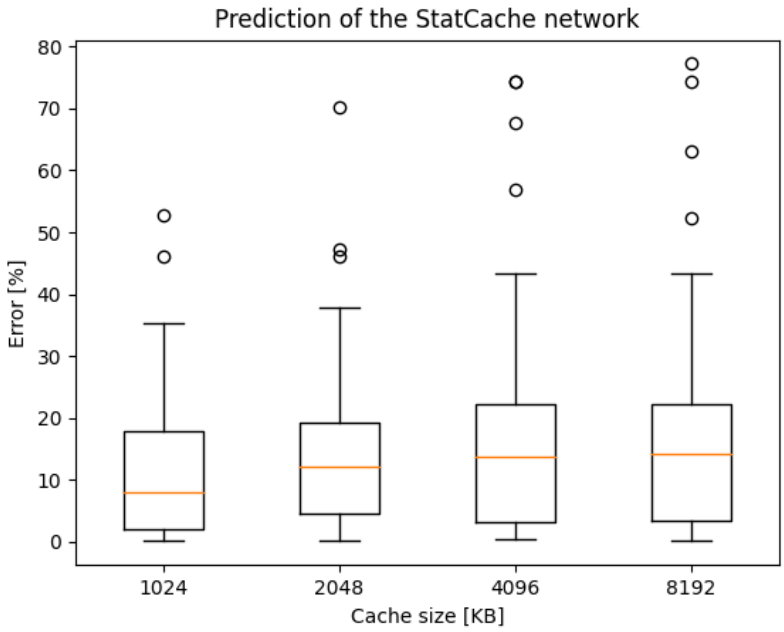


Figure 3.3. *StatCache's absolute absolute prediction errors.*

3.4 Shallow Multi-Layered Perceptron

First, we created a simple MLP to predict the miss ratios solely from the reuse distances. Reuse distances are enough data to predict such values with high accuracy, especially for the LRU replacement policy. StatCache predicts the miss ratios with a low error, only knowing those; therefore, we suppose that there's enough data to make an initial prediction and create a first model for it.

The model we created is an MLP with one hidden layer of 512 neurons. It takes as input the reuse distance histogram and produces four outputs, one for each cache size (1MB, 2MB, 4 MB, and 8MB). These four outputs are the cache miss predictions of the network for how the machine with the specific architecture will respond to the given problem.

Since this is the simplest of the networks that we are going to examine, its training time is also the lowest. It only needs 1ms/step, resulting in 1 second per epoch.

3.5 LSTM Network

Our quest for a superior solution has lead us down this path of maximizing our understanding of the reuse distances. When looking at a reuse distance, it is necessary to take into consideration the reuse distances larger and smaller than it to reach a conclusion. For this purpose we will try using an LSTM network.

An LSTM takes as input a series of consecutive values, iterates over them, and computes for each value an output, keeping in a hidden state parameter the values that have already passed. For our problem, we converted our array of reuse distances into a series

and fed it into an LSTM network. Thus, the network can compute a value for each one of the reuse distance histogram's buckets given the previous values that have passed from smaller reuse distances' buckets.

Similarly, we want the network to be able to compute a value for a certain histogram value given the values that are after it from larger reuse distances' buckets. For this reason, we reversed the reuse distance histogram and fed it into a second layer of LSTM units. For this second layer to act as a continuation of the first layer, the output cell and hidden state have to be initialized as the final ones from the first one. Now, the second layer will act as a continuation of the first layer with separate weights.

Originally, the thought behind this network was to utilize attention. This would benefit us in terms of understanding how important a feature is and where to focus when looking at the data. With this reasoning, we implemented an attention layer right after the two LSTM layers to combine them. Attention layers are attention mechanisms inspired by the human brain's cognitive attention. They detect the importance of each datapoint within the outputs via a softmax function [39].

Even though the intuition of using such a network seems correct, the training and output did not perform as well as expected. Its predictions were slightly worse than StatCache's, and therefore we won't show them. This is probably a consequence of not having enough training data. The attention matrix is computed from every output of every LSTM unit within the LSTM layers. This puts an enormous weight on every one of them and includes too many weights to optimize with training.

As an alternative to that, as depicted in figure 3.4, we opted to use a dense perceptron layer to understand the outputs of the LSTM layers. A concatenate and a flatten layer transform the output shapes from the LSTM layers into shapes acceptable to the dense layer. This dense network understands the problem better and can predict miss ratios with more accuracy.

For this network, in addition to the reuse distances, we implemented the transformer branch. We added the embeddings of the code into a parallel layer of perceptrons, input_1 in figure 3.4. The embeddings resulting from IR2Vec are not normalized, so a normalization with a similar scaler as we used for the reuse distances provides us with better results. Then, we connected all those to the last hidden perceptron layer and, lastly, to the output layer. All of the dense neural networks use a relu activation function except the output layer, which uses the sigmoid function since the miss ratio is a probability in the range [0, 1].

We added some normalization layers in between the small dense layers of the inputs and the big dense layer of the output. These seem to have a positive impact on the outcome of the model. Each one trains on the data of the training set and normalizes the outputs created by the dense neural networks. Layer normalization is beneficial, even though the input data is normalized, because it enables smoother gradients, faster training, and better generalization accuracy. We tested all possible placements of these layers in the network, and the best-performing one was right after the small, dense layers.

As to the specifics of the network, our LSTM layers are of size 4 units each. We experimented with lots of values for the network in the range 1–16 LSTM units and

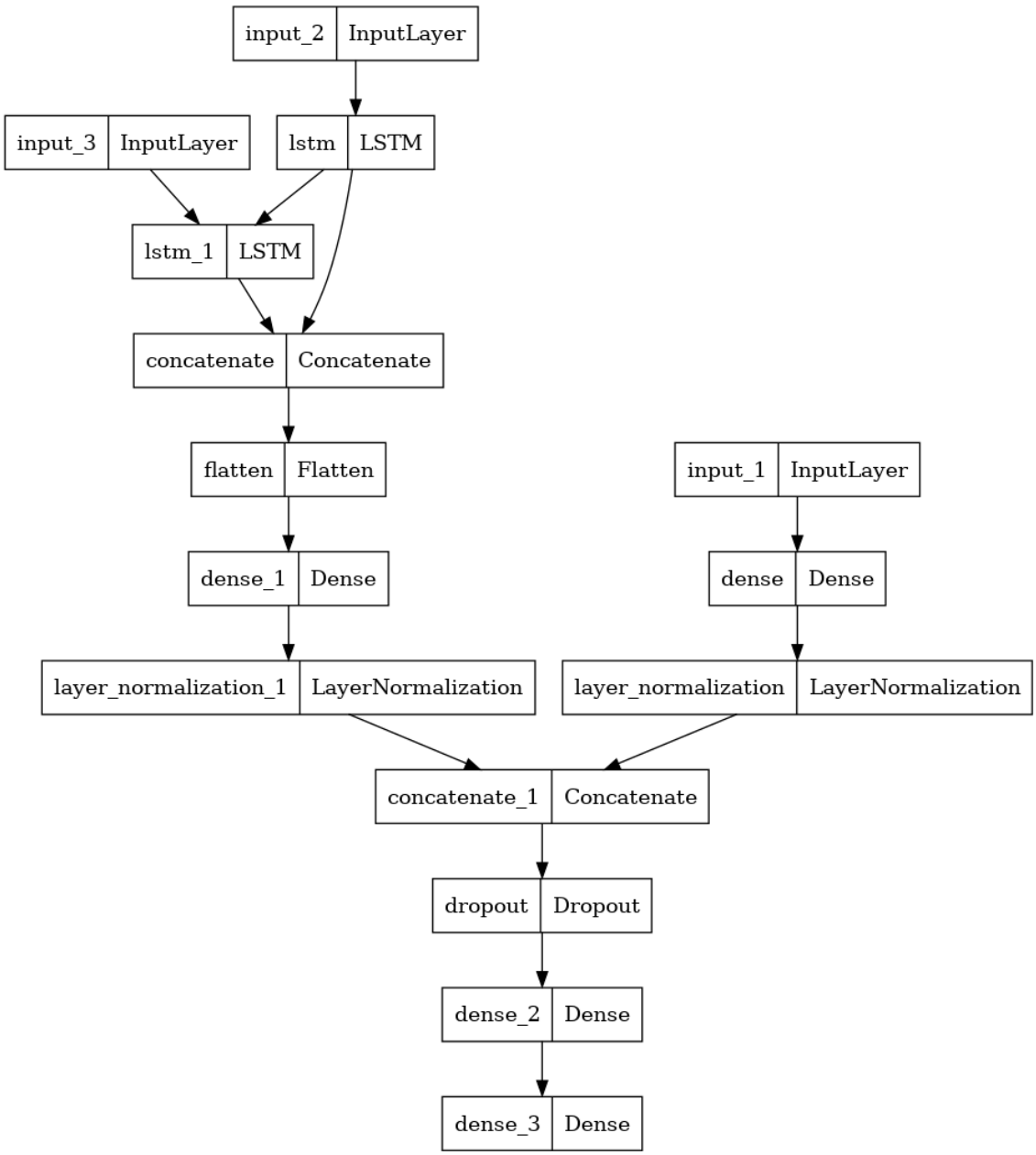


Figure 3.4. *LSTM network's structure. input_1 are the embeddings, input_2 is the reuse distance histogram and input_3 is the reuse distance histogram reversed (from largest to smallest)*

discovered that 4 units per layer seems to be computing best.The dense layers right after (dense and dense_1 in figure 3.4) are of size 128 neurons, whereas the last dense layer (dense_2) is larger with a size of 1024. The size of the smaller layers doesn't influence the precision too much; sizes of 128, 256, and 512 all have similar results. This size of the larger layer, on the other hand, worked better than other values such as 512 and 2048.

In addition, we use a drop-out layer. This layer randomly sets 20% of its inputs to zero to prevent over-fitting.

This network is the slowest and most difficult to train network that we'll introduce. It trains with an average of 50ms/step or 18 seconds per epoch.

3.6 Convolutional Neural Network

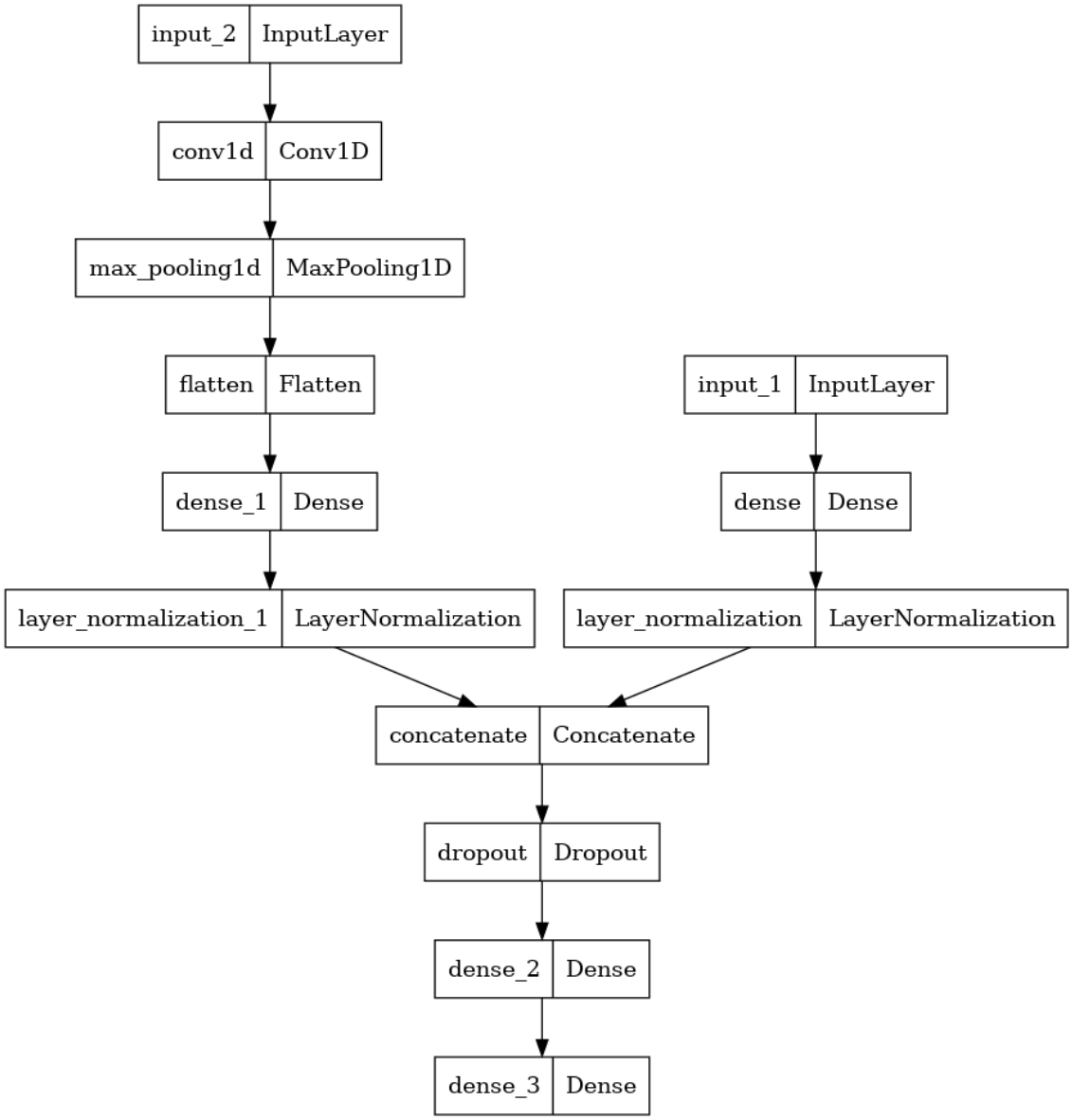


Figure 3.5. *Convolutional neural network's structure. input_1 are the embeddings and input_2 are the reuse distances.*

Convolutional layers exist to help the network identify similarities between features close to each other that might get lost or overlooked if we use a dense network. Convoluting neighboring reuse distances could result in better performance since the input to the dense layer will be more informative.

Then, immediately after, a pooling layer has the purpose of reducing the dimensionality of the previous output. The convolutional layer has n filters, thus producing n results per histogram input. The histogram input is a matrix of shape (896,1); therefore, the output matrix will be $((896 - n)/stride, n)$. We're using our max-pooling layer to reduce those to dimensions horizontally to $((896 - n)/(stride * K), n)$, where K is the kernel size. There are alternatives to the pooling layer for reducing dimensions, some of the more

popular being max and average pooling or using a larger stride. In this problem, a small stride and max-pooling computed the best outputs, so we used those.

Again, just like in the LSTM implementation, the output of this pooling layer is connected to a dense MLP layer with relu and a normalization layer. The rest of the network is the same as before.

It is worth noting that, with this network, unscaled reuse distances work best. The StandardScaler on the unscaled reuse distances that we used previously doesn't improve our prediction at all. Quite the opposite, the prediction gets worse because the scaling influences the output of the convolution; this results in smaller differences between the resulting values of the convolution and consequently worse understanding. The network prefers unscaled reuse distances passed through it, which will then be passed by the first perceptron layer. Then, a normalization layer will normalize the results of this dense layer, rather than having pre-scaled data. All the possible places to insert normalization layers, with all the possible combinations, were tried, and simply one layer after the perceptron produces the best predictions.

As to the specifics of the network, our CNN network consists of filters that iterate over the input reuse distances with a stride and compute the convolution over a kernel. We experimented with lots of combinations for these three numbers (filter number, stride, and kernel size) in the ranges $[1, 20]$ for each number. This testing concluded that the best combination is 6 filters, kernel size 8, and stride 4. The dense layers right after (dense and dense_1 in figure 3.5) are of size 512 neurons, whereas the last dense layer (dense_2) is larger with a size of 1024. The size of the smaller layers influences the precision a little bit; sizes 128 and 256 have a geometric mean of absolute prediction errors of about 0.5% less. This size of the larger layer, on the other hand, worked better than other values such as 512 and 2048.

This network's training needs on average 6 ms/step or 3 seconds/epoch, which is far quicker when compared to the LSTM network's 50 ms/step or 15 seconds/epoch.

3.7 Deep Neural Network

Lastly, we wanted to see if we could create a network that predicts the miss ratio on any LLC cache size. This has the goal of accurately estimating the miss ratio of each cache size and, ultimately, showing which is the best cache architecture for a specific problem. For this purpose, we simulated the traces again, using in-between cache sizes to create a bigger dataset. This contains the data for LLC cache sizes [768, 1024, 1536, 2048, 3152, 4096, 6044, 8192] KB with the LRU replacement policy.

This network can be used in two separate ways. If we have a program for which we have the data on one or more machines, we can add it to the training set and predict how high of a miss ratio it will have with other cache sizes. The second way of using it is for a new program that hasn't been seen by the network. This second way will in general be more useful than the previous three networks since it will be able to predict the miss ratio for any cache size and not just be restricted to the ones that it has been trained on.

Inspired by the MLP network 3.4, we implemented a deep neural network that is a bit

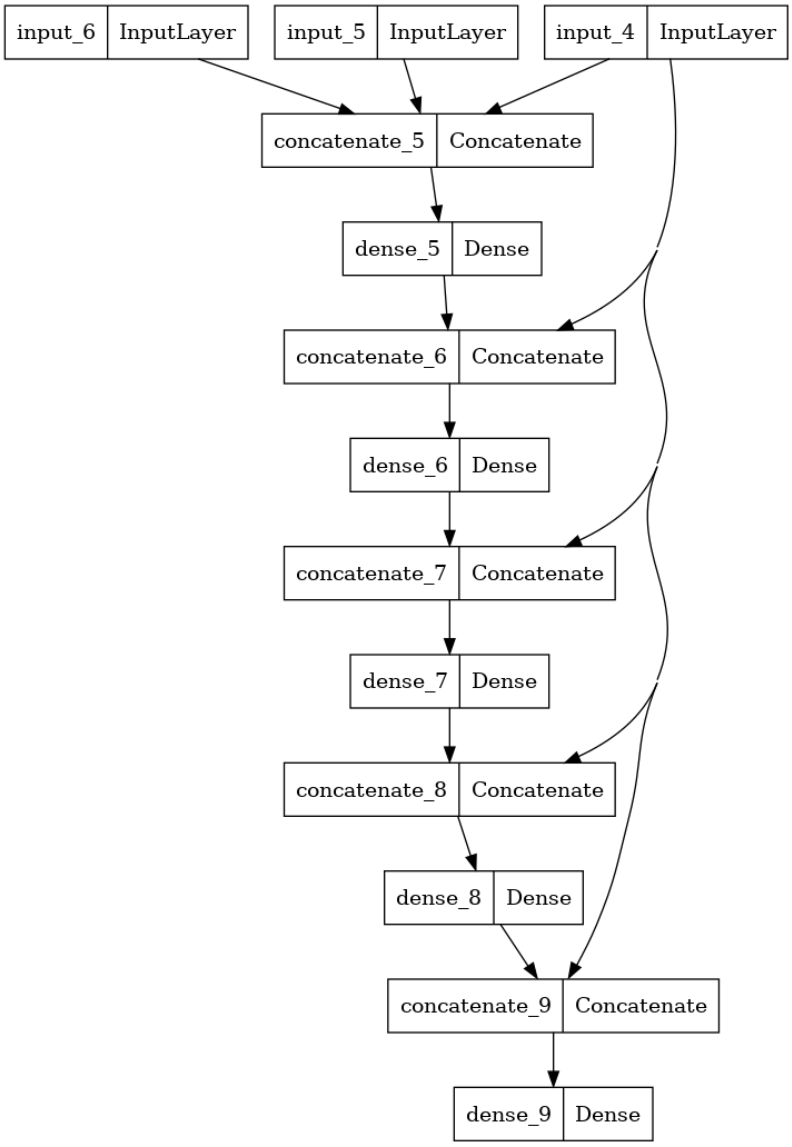


Figure 3.6. *Deep Neural Network for predicting other cache sizes. input_4 is the size of the cache, input_5 are the reuse distances and input_6 are the embeddings.*

more complicated. For this network, as shown in figure 3.6, we use 4 hidden layers. We want each layer to make predictions according to the size of the cache; that's why we feed the input size to each hidden layer by concatenating it with the output of the previous layer. Each one of the hidden layers has 512 neurons; the first four use the relu activation function, and the last one and the output layer use the sigmoid activation function. This configuration was tested and seemed to have the best results. More hidden layers or more neurons didn't compute a better prediction, perhaps due to creating too many variables.

All three of the inputs contain a normalized version of reuse distances, embeddings, and cache sizes. The reuse distances and embeddings are normalized with the same method as in other networks. The input size was divided by 512KB, resulting in values ranging from 1.5 - 16.0 for each one of the caches.

Chapter 4

Experimental evaluation

4.1 Predicting the miss ratio of an unknown application

4.1.1 Predicting for LRU replacement policy

The problem that we are about to discuss is very straight forward to describe. We are trying to predict the miss ratios of any application that our networks are met with. To show that we use the aforementioned method of leave-one-out cross-validation where we separate one benchmark, train the network on the other benchmarks and try to predict it. This way we will have the best possible prediction for every workload without having seen it previously.

The search for the most accurate miss ratio prediction has led us to introduce four models in total: the MLP, CNN, LSTM, and DNN networks. Each of these models represents a distinct approach to predicting cache miss ratios as accurately as possible. Let's compare and discuss the overall predictions of our networks. For this purpose, we will choose a replacement policy and compare the prediction errors that we have computed with each one of our networks.

The LRU is the only replacement policy for which we can compare StatCache and the DNN networks as well. StatCache was invented and works only on the LRU replacement policy. Our DNN network is probably capable of predicting any replacement policy, but we would need more data on other replacement policies to confirm that it actually predicts any cache size. The DNN network that we are going to show results from is the DNN that has been trained on the 4 cache sizes (1MB, 2MB, 4MB, and 8MB) with the same method of leave-one-out cross-validation as the other networks.

It is clear that all of our networks outperform significantly the StatCache predictions. The outliers, the boxes, and the medians of absolute prediction errors in the boxplot of 4.1 are noticeably higher for the StatCache prediction. It is also clearly noticeable that the geometric means of prediction errors in table 4.1 are clearly worse than any of the other networks in the three lower cache sizes and equal to the worst for the 8MB sized cache.

It is also clearly visible that for larger cache sizes, the absolute prediction errors become larger. For larger cache sizes, the average miss ratio drops drastically. This leads us to believe that for larger caches, even though it is easier to predict a range where the

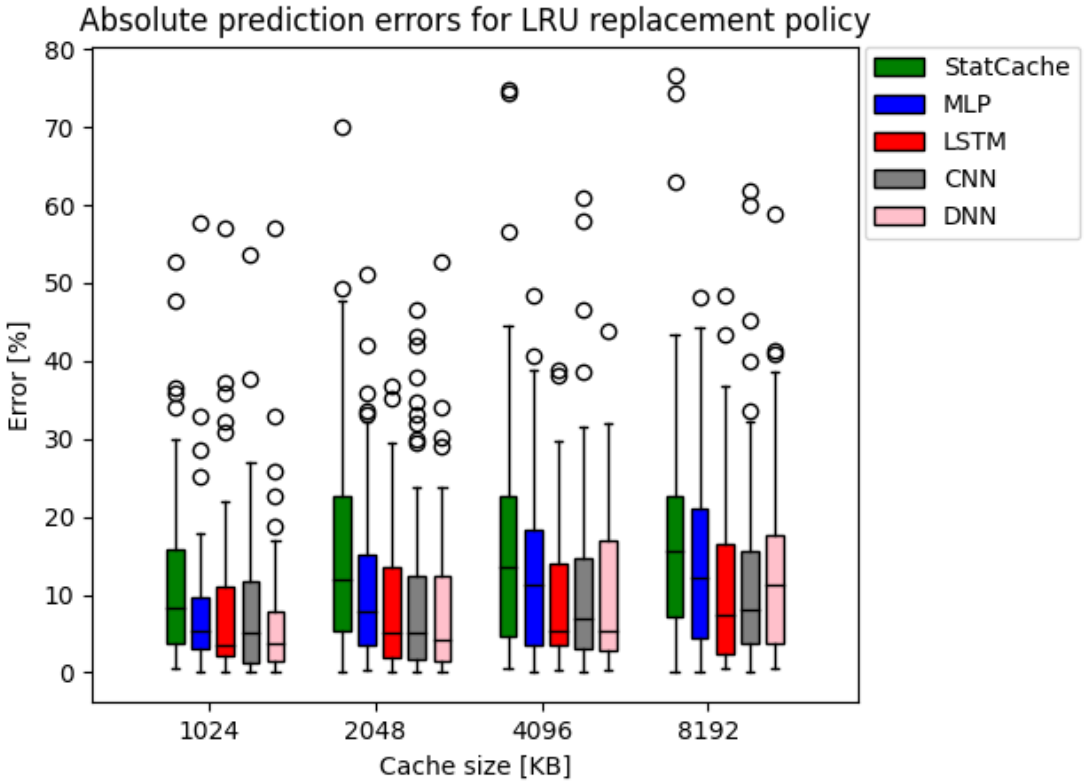


Figure 4.1. *Prediction errors of the MLP, LSTM, CNN, DNN networks and StatCache for LLCs with LRU replacement policy*

miss ratios will be, it is harder to predict the exact miss ratios.

The shallow MLP is overall the weakest of our networks. The fact that it is not getting as much information as other networks, due to it not having the transformer branch, as well as the worse understanding of the reuse distances, result in larger errors than the other networks. Its prediction errors have higher medians, larger boxes, and many outliers when compared to the CNN and LSTM networks' prediction errors. We also see that by the fact that it consistently produces the second largest prediction error in table 4.1. The goal of this network was to show that it is possible with a simple MLP to produce similar, if not better, results than StatCache only using reuse distances. The results indicate that we have successfully reached this goal.

The other three networks have access to the transformed code of the program, so their predictions are undeniably better than the ones from the MLP. To analyze their predictions more precisely, we will inspect the histograms of their absolute prediction errors 4.2. Since the histograms are similar for other cache sizes as to the shape, we will only show the ones from the 4MB cache size that are representative.

It is obvious that the prediction of StatCache is worse than any of the other predictions. Many large outliers, fewer benchmarks close to zero error, and an almost even distribution in the range of its values make it clear why the predictions of Statcache are worse. The highest column for all other histograms is the smallest one. The LSTM and the CNN

Network	Cache size [MB]			
	1	2	4	8
StatCache	5.4%	8.3%	8.7%	8.5%
MLP	4.1%	7.2%	6.3%	8.5%
LSTM	3.6%	3.9%	5.3%	6.4%
CNN	3.9%	4.6%	5.5%	6.2%
DNN	3.6%	4.0%	6.0%	8.1%

Table 4.1. *Geometric mean of absolute prediction errors for the caches with LRU replacement policy.*

networks absolute prediction errors have very similar shapes close to zero, with many benchmarks very close and less further away. What differentiates them is that CNN has many outliers. This is also visible in 4.1, where the CNN network's absolute prediction errors have far more outliers than the LSTM network's.

Assessing the DNN network's prediction in comparison to the others is a bit more difficult. Its absolute prediction errors seem to have a more even distribution within the range of 0 to 0.2 with very few outliers. This is slightly worse than the LSTM network's absolute prediction errors. When looking at the cumulative boxplot 4.1, its predictions seem to be better than LSTM's for smaller values of cache size. The geometric means for both of these are almost equal. This means that the network has the capability of predicting cache miss ratios as well as the LSTM network, if not better, for small cache sizes. This also means that this DNN-structured network has a weakness for larger cache sizes.

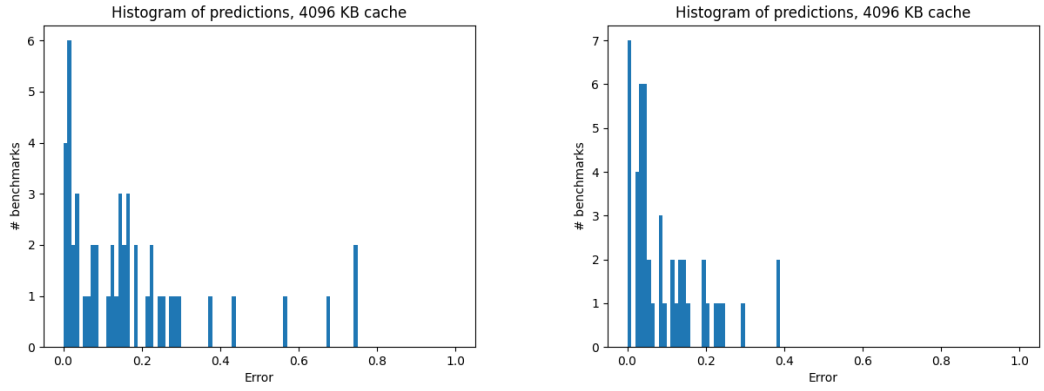
Overall, the best-performing network for all cache sizes with an LRU replacement policy is the LSTM network, closely followed by CNN. The DNN network has a certain weakness for larger values, for which it doesn't predict as well as the other networks.

4.1.2 Predicting any cache size with LRU replacement policy

After analyzing the results of all networks on these four cache sizes, we want to present the versatility of the DNN network. The DNN network takes as input the cache size parameter and can understand any cache size's miss ratio. Therefore, we have the miss ratios of more cache sizes in the LLC to show that this DNN network can actually predict them.

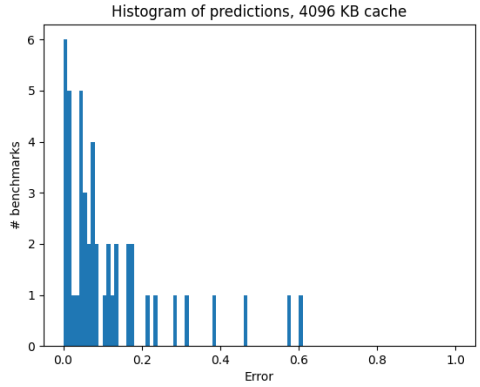
In this section, we try to predict any unknown program for any cache size with the given replacement policy. We have the LLC sizes [768, 1024, 1536, 2048, 3152, 4096, 6044, 8192] KB and replacement policy LRU. We are going to use the method of cross-validation for our test set, just like in chapter 3.4, where we'll iterate through the benchmarks and predict them with a network trained on the other 46 benchmarks. In the end, we will have 47 predictions, one for each benchmark, with the network trained on all other benchmarks each time.

For less than 3 cache sizes in the training set, the prediction holds no value since the network can't understand the size parameter. When the network is trained on one

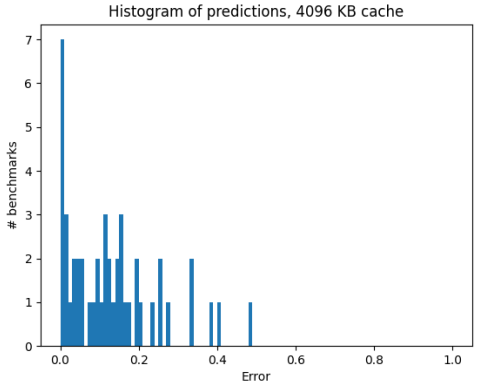


(a) StatCache

(b) LSTM



(c) CNN



(d) DNN

Figure 4.2. *StatCache, LSTM, CNN, and DNN absolute prediction errors for LLC of size 4MB with LRU replacement policy*

cache size, the size parameter is not changing in the training set, so the network will just assign random values to it. When two cache sizes are used in the training set, the network understands it as a binary problem and tries to apply a sigmoid function to it. This makes the predictions unreliable. So we will show the predictions with 3, 4, and 6 cache sizes in the training set.

The network has been trained with the 1MB, 2MB, and 8MB cache sizes in the training set. As we can see in figure 4.3, the absolute prediction errors for the lower caches are very good. The medians are low and steady. There is mostly only one outlier for these caches, and that is the prediction for the 416.gamess benchmark; its miss ratio is always predicted to be higher, and it will be miss-predicted in the future as well, for good reason. This benchmark consists of one trace, which only has 17k accesses to the LLC, which means that the network reads one window of reuse distances with 17k accesses and tries to predict the miss ratio of it. The majority of reused distance windows that our network is trained on consist of 512k accesses. Thus, the network sees the low number of reuse distances and assumes lots of accesses that only happen once and therefore don't have reuse distances. These result in misses (cold misses), and therefore the network computes a higher miss ratio. We can safely ignore this output since, with a 30x longer simulation,

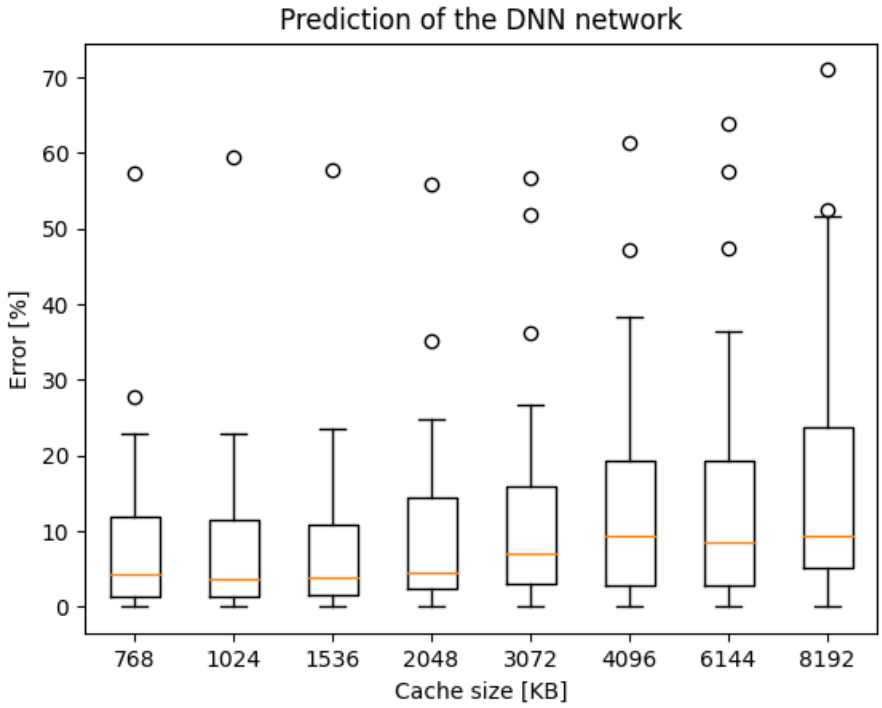


Figure 4.3. *DNN network's absolute prediction errors, trained on the 1MB, 2MB and 8MB cache sizes.*

the output would most likely get closer to the real value.

As for the larger LLCs, the predictions are gradually getting worse. The precision of the network for these caches is significantly worse than the ones from the LSTM network, as suggested by their geometric means 4.2. The geometric means of prediction errors are close to those predicted by the LSTM for smaller cache sizes and get drastically worse for larger cache sizes.

Train-set cache sizes [MB]	**Cache size [KB]**							
	768	**1024**	**1536**	**2048**	**3072**	**4096**	**6144**	**8192**
1, 2, 8	3.0%	3.1%	3.8%	5.0%	5.8%	6.2%	6.5%	8.5%
1, 2, 4, 8	3.8%	3.6%	3.1%	4.0%	5.1%	6.0%	6.9%	8.1%
0.75, 1, 2, 4, 6, 8	2.5%	3.4%	4.1%	3.5%	4.2%	5.3%	7.2%	9.7%
LSTM Network	-	3.6%	-	3.9%	-	5.3%	-	6.4%

Table 4.2. *Geometric mean of DNN network's absolute prediction errors, trained on different cache sizes. Together with the LSTM network's Geometric mean of absolute prediction errors, for comparison.*

Next, we added the 4MB-sized cache's data to the training set. Now we have the data for 1, 2, 4, and 8 MB-sized LLCs in the training set. This means that the network trains on the exact same data as the previous networks that we introduced.

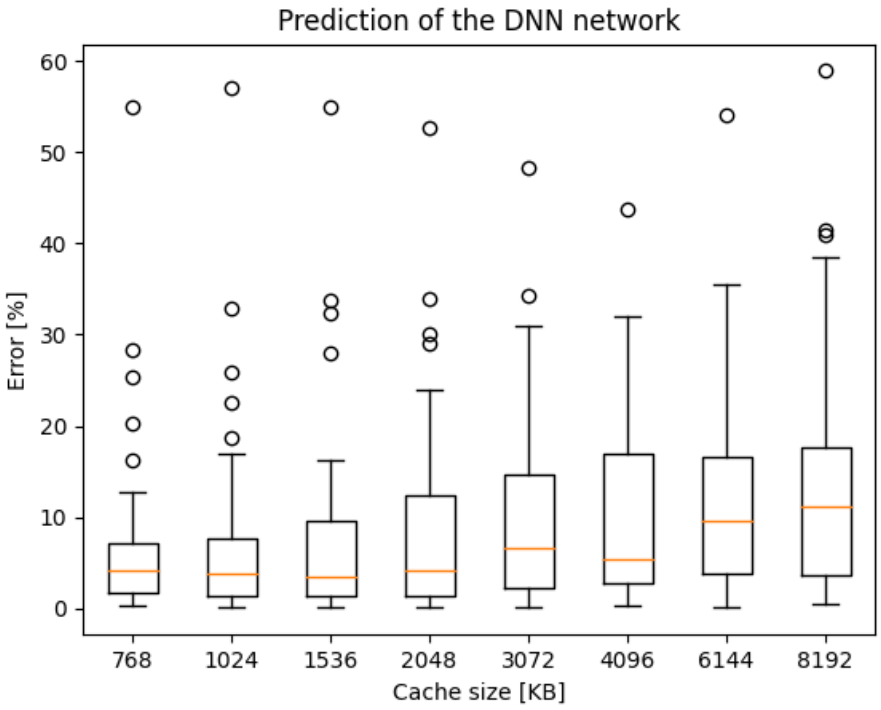


Figure 4.4. *DNN network's absolute prediction errors, trained on the 1MB, 2MB, 4MB and 8MB cache sizes.*

It is interesting how this addition influenced the prediction of the 4MB-sized cache very little, in the figure 4.4 or in the table 4.2. Apparently, the network could already predict the miss ratios for the 4MB cache size well enough when trained on three cache sizes, therefore, the addition of it changed it very little.

This addition helped our network understand some of the smaller caches better. When first training a network, we trained it only on the 2MB cache size. Only knowing the 2 MB cache gave us a prediction with a geometric mean of absolute prediction errors of 4.7% for the 2MB column. It is interesting how the geometric mean of the 2MB-sized cache (row 2 of table 4.2) is lower than the one predicted with only this cache in the training set. This means that the network gained some insight into the miss ratios of the 2MB-sized cache from the addition of the 4MB-sized cache to the training set.

Lastly, we trained the network on six cache sizes: 0.75, 1, 2, 4, 6, and 8 MB-sized LLCs. This is just to see how the network will behave with such an addition of cache sizes.

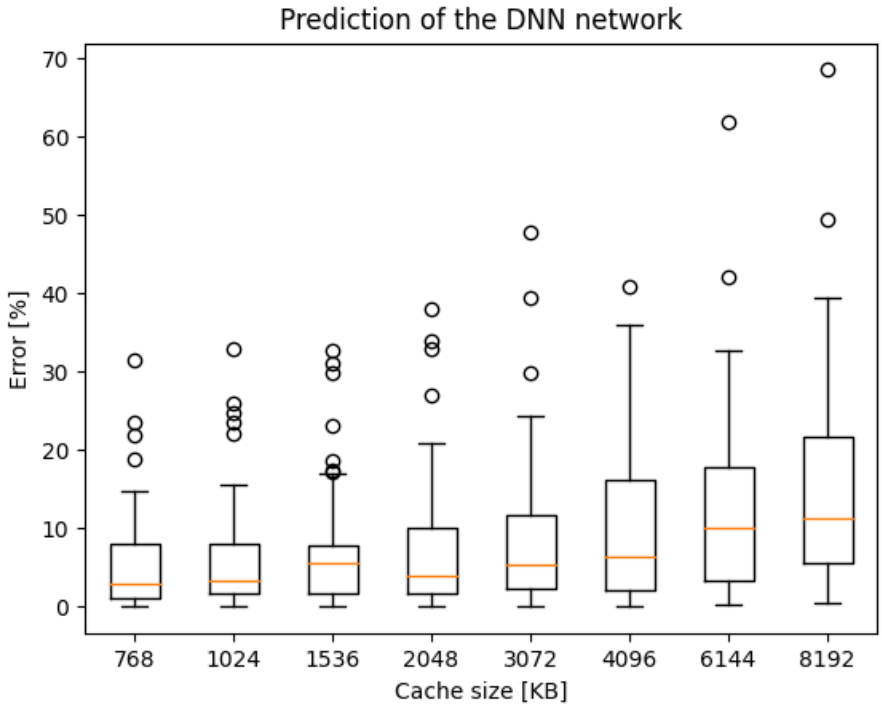


Figure 4.5. *DNN network's absolute prediction errors, trained on the 0.75, 1, 2, 4, 6, 8 MB cache sizes.*

The predictions of this training set in figure 4.5 seem very similar to the previous prediction's errors. The sizes and shapes of the boxes in the boxplot are fairly similar to the one trained with 4 cache sizes. The only difference is that 416.gamess's predictions have gotten a bit better for smaller cache sizes, but not enough.

In the third row of the table 4.2 we can see that the prediction of the 768 KB cache got even better, as well as the prediction for the 2MB-sized cache. Now, the geometric mean of absolute prediction errors for the 2MB-sized cache is lower than the LSTM network's prediction that we mentioned at the start. This means that most likely for any cache size lower than 4MB this DNN network will have similar results to the LSTM network. It is also interesting that the geometric mean of absolute prediction errors for the 6 and 8 MB-sized caches has risen. The network focuses more on predicting the small cache sizes exactly and apparently cannot understand the larger ones as precisely.

4.1.3 Predicting other replacement policies

SHiP replacement policy

For the other three replacement policies, we only have the prediction errors from the MLP, CNN, and LSTM networks. The StatCache's prediction is designed to describe the behavior of LRU caches, and therefore can't be applied to other cache replacement policies. The DNN network is only trained and tested on the LRU replacement policy; for other replacement policies, we don't have enough data for enough cache sizes to accurately evaluate its performance in between cache sizes.

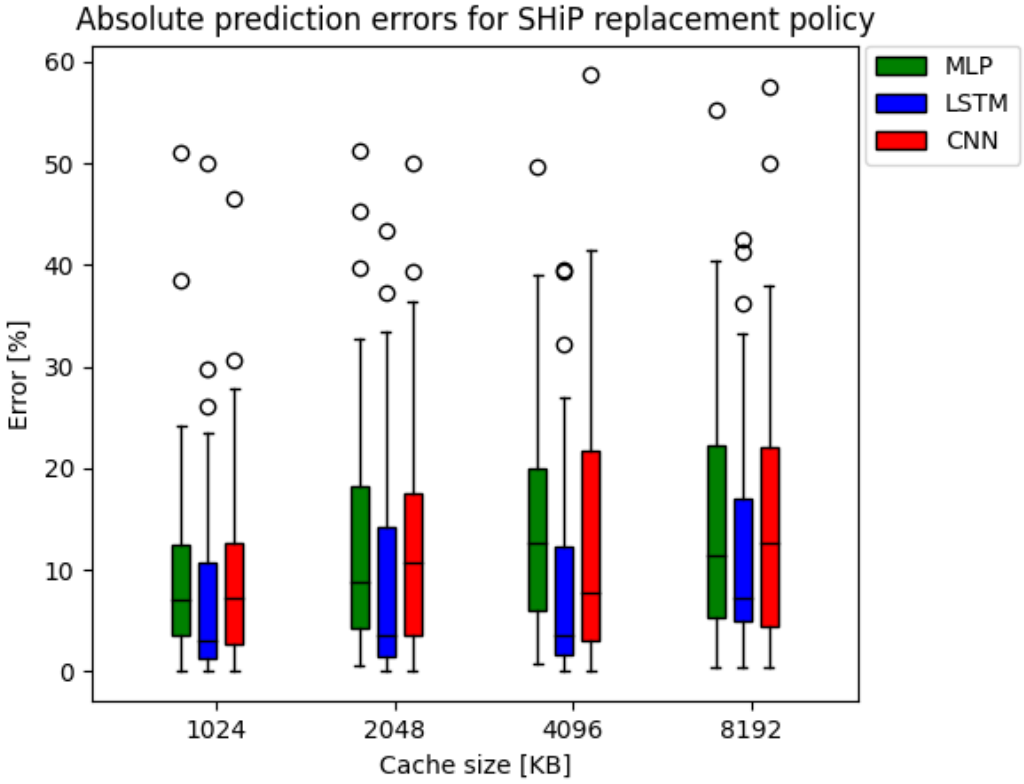


Figure 4.6. *Prediction errors of the MLP, LSTM, CNN networks for LLCs with SHiP replacement policy*

Network	Cache size [MB]			
	1	2	4	8
MLP	5.6%	7.7%	9.8%	8.6%
LSTM	2.7%	4.1%	3.7%	7.6%
CNN	5.3%	7.3%	5.8%	9.4%

Table 4.3. *Geometric mean of prediction errors for the caches with SHiP replacement policy.*

The best prediction when looking at the prediction errors, figure 4.6, or their geometric means, table 4.3, is clearly the LSTM prediction. All the means are closer to zero than any of the others, and the boxplots look better for all cache sizes. The geometric means

of the network are at least 2% better than the geometric means of the smallest of the other two networks. They also don't deviate more than 1% from the geometric means of prediction errors of LSTM for LRU replacement policy. This means that they have the same range as the ones predicted for LRU and show the stability of this network despite the change in replacement policy. The prediction errors from the CNN network for this replacement policy are very similar to the MLP network's prediction errors, which indicates that the CNN network shows some weakness for this replacement policy. This still raises the question, what makes the LSTM predict so much better than the other two in this replacement policy.

SRRIP replacement policy

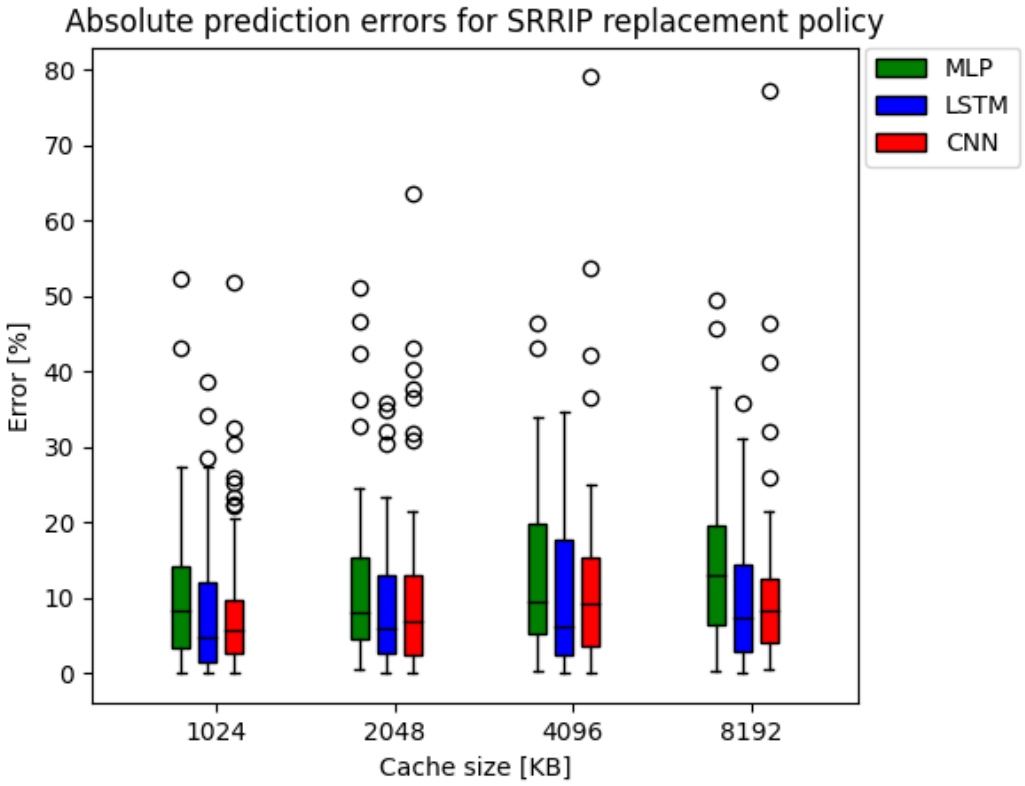


Figure 4.7. *Prediction errors of the MLP, LSTM, CNN networks for LLCs with SRRIP replacement policy*

Again, the predictions of the LSTM network seem to be the strongest ones. The prediction errors from the CNN network are better than the ones that we had for SHiP, looking similar to the LSTM network's prediction errors. They have smaller boxes in the boxplot, meaning that the prediction error is pretty stable. Still, the LSTM network's prediction errors are better regarding means and geometric means. What is also worth mentioning is that this is the lowest geometric mean of prediction errors that we have had for the 8MB-sized cache.

Network	Cache size [MB]			
	1	2	4	8
MLP	5.7%	8.0%	8.6%	9.3%
LSTM	3.1%	4.6%	5.5%	5.7%
CNN	4.7%	5.2%	6.4%	7.4%

Table 4.4. *Geometric mean of prediction errors for the caches with SRRIP replacement policy.*

Mockingjay replacement policy

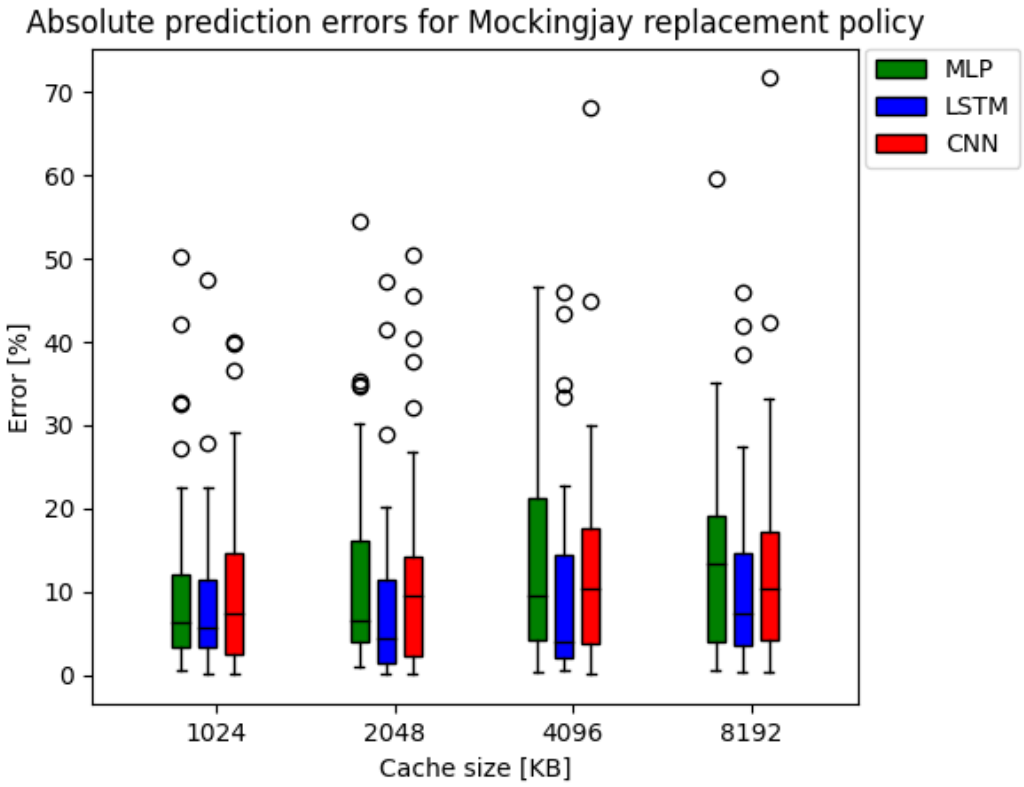


Figure 4.8. *Prediction errors of the MLP, LSTM, CNN networks for LLCs with Mockingjay replacement policy*

Lastly, there is the Mockingjay replacement policy. Just like before, the MLP network is clearly worse than the other two networks in predicting the caches with this replacement policy. The LSTM network outperforms the other networks significantly. Every box in the boxplot of prediction errors from the LSTM network is smaller, lower, and has a smaller mean than the others.

Mockingjay is a cache replacement policy that relies on reuse distances. It tracks them, calculates the likelihood of reuse, and then decides if an object will stay in the cache. This would lead us to the conclusion that a process such as a double LSTM layer helps the network understand reuse distances better and how the cache will behave with

Network	Cache size [MB]			
	1	2	4	8
MLP	6.2%	7.8%	9.0%	8.8%
LSTM	4.9%	3.6%	5.3%	6.2%
CNN	5.3%	6.2%	7.6%	8.3%

Table 4.5. *Geometric mean of prediction errors for the caches with Mockingjay replacement policy.*

this replacement policy. But the LSTM network doesn't predict Mockingjay significantly better than any of the other replacement policies. This leads us to believe that there is still room for improvement when it comes to this network.

4.1.4 Benchmarks

This section of the thesis aims to dissect and compare the outcomes of the aforementioned networks. We are going to compare and discuss the strengths and weaknesses of our networks, on individual benchmarks. Let's begin this result comparison by assessing the predictions of our networks for one specific benchmark.

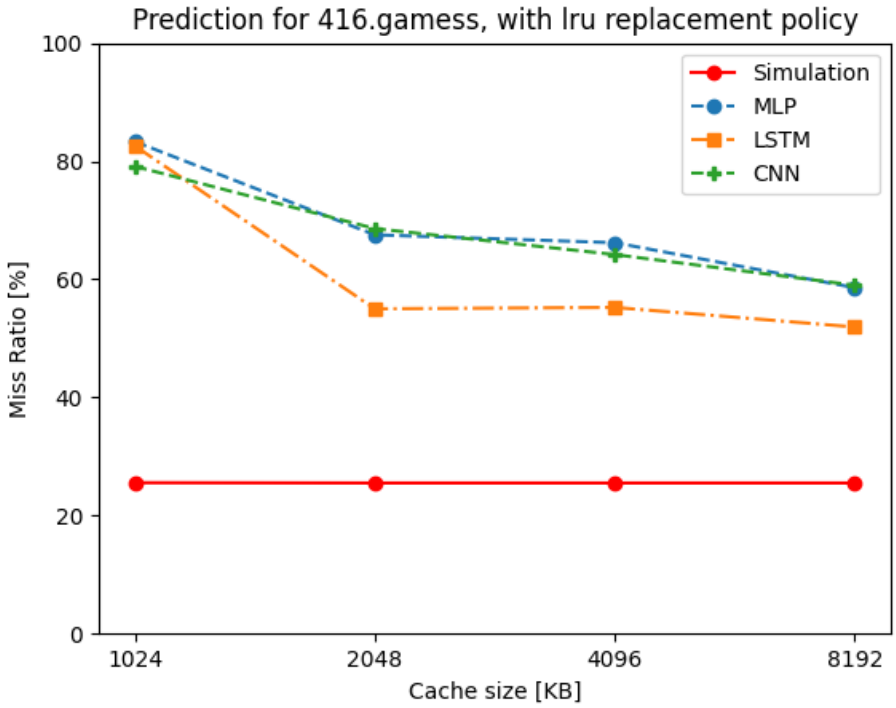


Figure 4.9. *Network predictions for 416.gamess with LRU replacement policy.*

Foremost, let's address the most miss-predicted benchmark in our dataset. The 416.gamess benchmark consists of one trace with 17k accesses within it. The majority of windows that our network is trained on consist of 512k accesses. When confronted with a window of this size, our network assumes that most of the accesses that happen

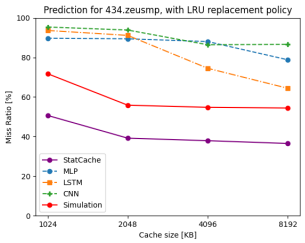
are cold misses, and as a result, they don't have reuse distances. Therefore, the prediction is much higher than the actual value. As we can see above, in figure 4.9, the three first networks predict a much higher miss rate than the actual. This type of benchmark can be safely ignored since, with a longer simulation, the result will be more accurate.

Moving onto some of the worst-predicted benchmarks by all of our networks. In figure 4.10 we are presenting three of the benchmarks for which we have the largest deviations between the predicted and the simulated value. These three benchmarks are 401.bzip2, 434.zeusmp, and 456.hmmer. All of them are highly intensive programs with millions of accesses each. We show the outputs of our networks for the four replacement policies. For LRU, since we depict separately the predictions of StatCache, MLP, CNN, and LSTM from the DNN networks to preserve clarity. The DNN predictions (plots d, e, and f) are labeled as DNN-i, where i is the number of training cache sizes of the network. We also show the predictions from the DNN-1 and DNN-2 networks, to show that their predictions indeed are unreliable. It is clear to see that all the DNN-1 networks are random, and that is because the network doesn't understand the size parameter, since it was trained on only one size parameter.

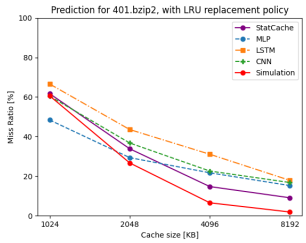
The 434.zeusmp is a program that simulates astrophysical phenomena based on the ZEUS-MP computational fluid dynamics code. It is clear that the prediction from StatCache for this benchmark is the only one that is lower than the actual simulated miss ratios. All the predictions from the networks that we proposed seem to agree that the miss ratio should be significantly higher than the simulated one. Even though this holds true for most networks, LSTM seems to have made a valiant attempt at closely predicting the miss ratio for the Mockingjay replacement policy (plot m).

401.bzip2 is a compression algorithm that compresses a range of files throughout its execution. StatCache's prediction for this benchmark is very accurate, beating our networks by quite a bit. The misprediction of our networks for this benchmark is not as bad as for the other two benchmarks. The prediction of miss ratios is always accurate for the smallest cache size, and then as the cache sizes get larger, it deviates.

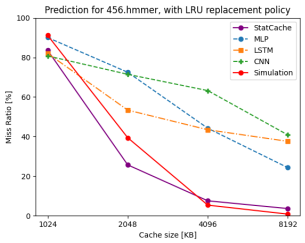
Lastly, the 456.hmmer file is a computation of profile Hidden Markov Models to do sensitive database searching using statistical descriptions of a sequence family's consensus. Again, the prediction of Statcache is far better for this benchmark. The steep drop-off in miss ratio for large cache sizes seems to confuse our networks, since they always predict higher for the larger cache sizes, even though for the smallest ones they get it correct. It is interesting how the DNN network (plot f) creates seemingly the same prediction shape when it is trained with 1, 2, or 3 cache sizes, upwards of the fourth. On the other hand, it seems to understand the shape of the benchmarks better and move to overcome it. The LSTM also seems to have the correct shape when compared to the other two networks, but it doesn't quite seem to understand the steepness of the curve.



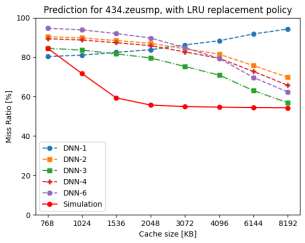
(a) 434.zeusmp, LRU



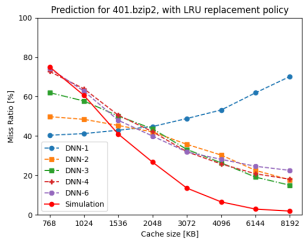
(b) 401.bzip2, LRU



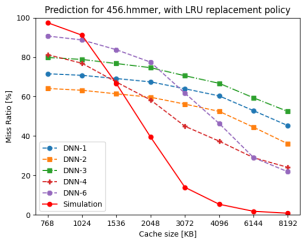
(c) 456.hmmer, LRU



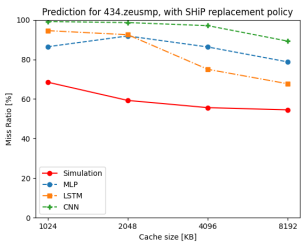
(d) 434.zeusmp, LRU



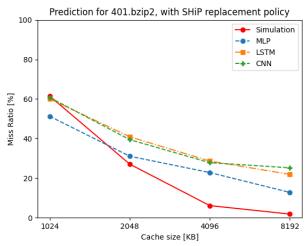
(e) 401.bzip2, LRU



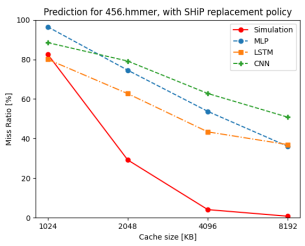
(f) 456.hmmer, LRU



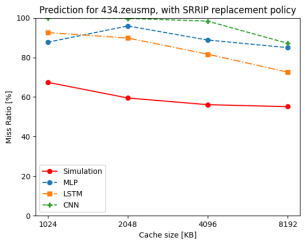
(g) 434.zeusmp, SHiP



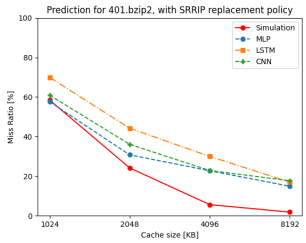
(h) 401.bzip2, SHiP



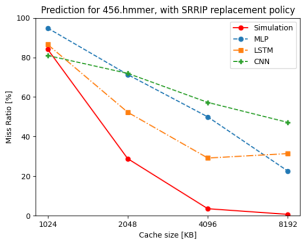
(i) 456.hmmer, SHiP



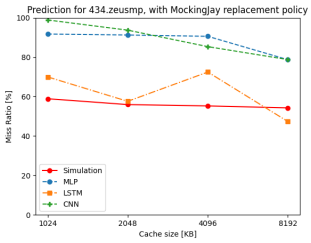
(j) 434.zeusmp, SRRIP



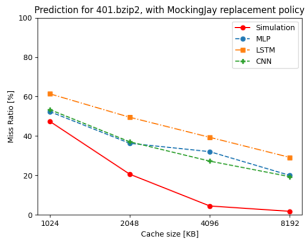
(k) 401.bzip2, SRRIP



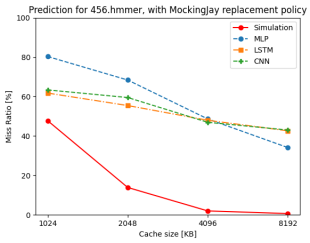
(l) 456.hmmer, SRRIP



(m) 434.zeusmp, Mockingjay

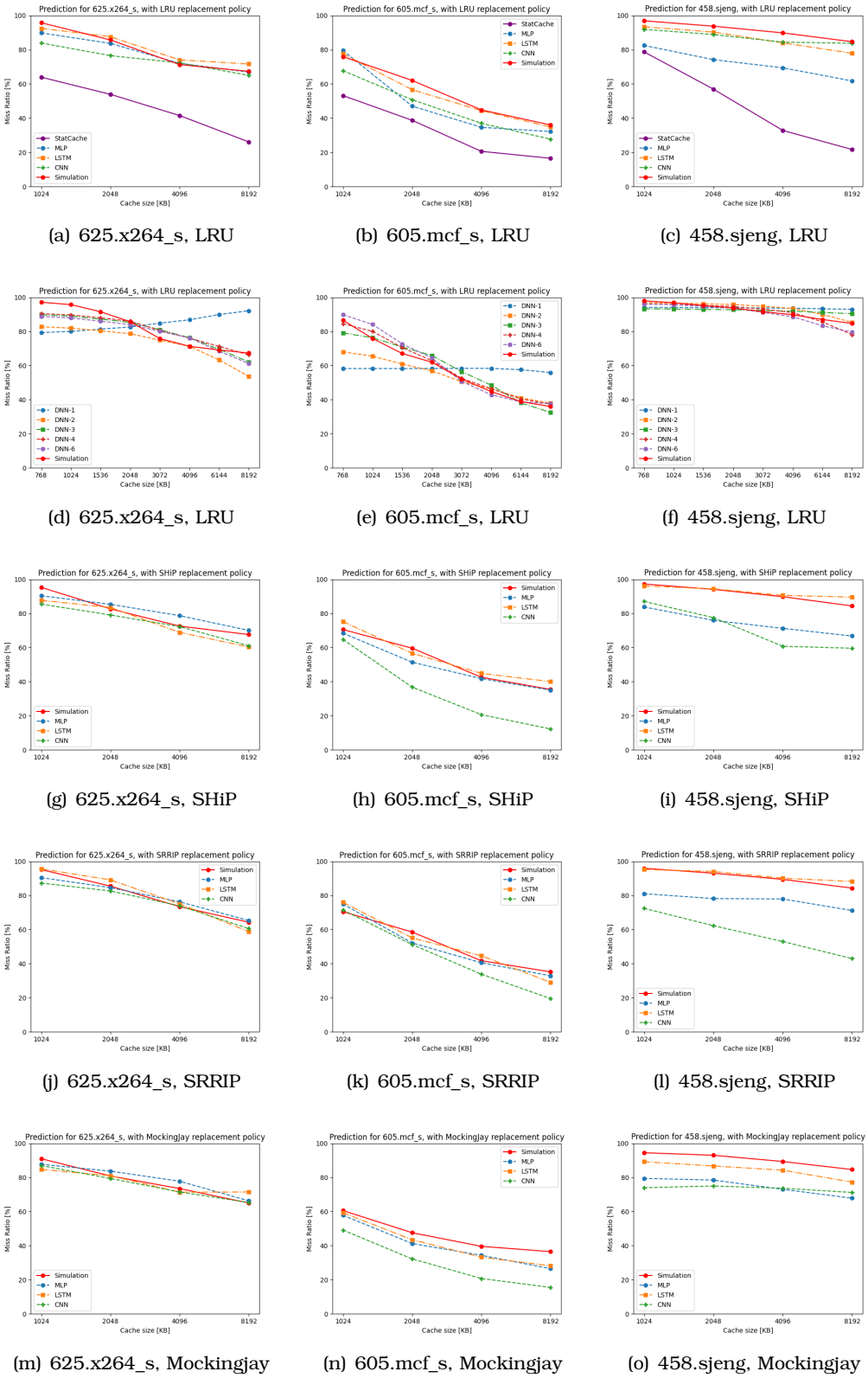


(n) 401.bzip2, Mockingjay



(o) 456.hmmer, Mockingjay

Figure 4.10. *Some of the worst benchmark to predictions.*



(a) 625.x264_s, LRU (b) 605.mcf_s, LRU (c) 458.sjeng, LRU

(d) 625.x264_s, LRU (e) 605.mcf_s, LRU (f) 458.sjeng, LRU

(g) 625.x264_s, SHiP (h) 605.mcf_s, SHiP (i) 458.sjeng, SHiP

(j) 625.x264_s, SRRIP (k) 605.mcf_s, SRRIP (l) 458.sjeng, SRRIP

(m) 625.x264_s, Mockingjay (n) 605.mcf_s, Mockingjay (o) 458.sjeng, Mockingjay

Figure 4.11. *Some of the best benchmark predictions.*

Some of the best predicted files are shown in figure 4.11. This time, the predictions from the networks are very close to the simulation's cache miss ratios. The three benchmarks that we show have some of the best predictions from our networks. These are

obviously not all the benchmarks with good predictions, just some that are indicative. It is interesting how, for almost all of these predictions, the best prediction is the one from the LSTM network. It understands these problems almost perfectly, having minimal deviations from the real values. Furthermore, it really shows how a better understanding of the problem applies to the prediction.

The CNN and MLP overall seem to have small deviations from the simulation's values, larger than the ones of the LSTM network. The DNN network seems to understand the problem as well, and with more data, the predictions get better as well. Again, we have the untrained DNN-1 plot that visibly doesn't understand the size parameter.

Lastly, StatCache is outperformed significantly on those predictions. The predictions of StatCache are not bad for all of those benchmarks. They may be bad for the 458.sjeng benchmark, but for the other ones, the predictions from our networks are far more accurate.

4.1.5 Conclusion

In conclusion, the LSTM network consistently predicts the most precise results of the networks we proposed. As we can see in figure 4.12 its absolute prediction errors are just as good as for any replacement policy. Its results showed that it can predict any cache size that it is trained on consistently with very low geometric means of prediction errors. It also showed that it is very consistent in its predictions, as the geometric means of prediction errors in table 4.6 don't vary too much on different replacement policies.

Replacement policy	1MB	2MB	4MB	8MB
LRU	3.6%	3.9%	5.3%	6.4%
SHiP	2.7%	4.1%	3.7%	7.6%
SRRIP	3.1%	4.6%	5.5%	5.7%
Mockingjay	4.9%	3.6%	5.3%	6.2%

Table 4.6. *Geometric mean of LSTM network's absolute prediction errors*

A simpler and less costly alternative to it is the CNN network, which has the capability to predict very accurately and is much faster to train, with about 20% of the training time of the LSTM. But it is also inconsistent for some replacement policies, with similar predictions to the MLP, which is not optimal. The DNN network shows the most potential, since it has the best predictions for lower cache sizes within the LRU replacement policy, but still has room for improvement on the larger caches. Overall, these results emphasize the different intricacies of the problem and show that it is indeed possible to predict it accurately.

4.2 Predicting the miss ratio of a known application

Lastly, there is another usage that we can have with this network. Suppose we have a few machines in place that have a specific architecture. We want to predict how a

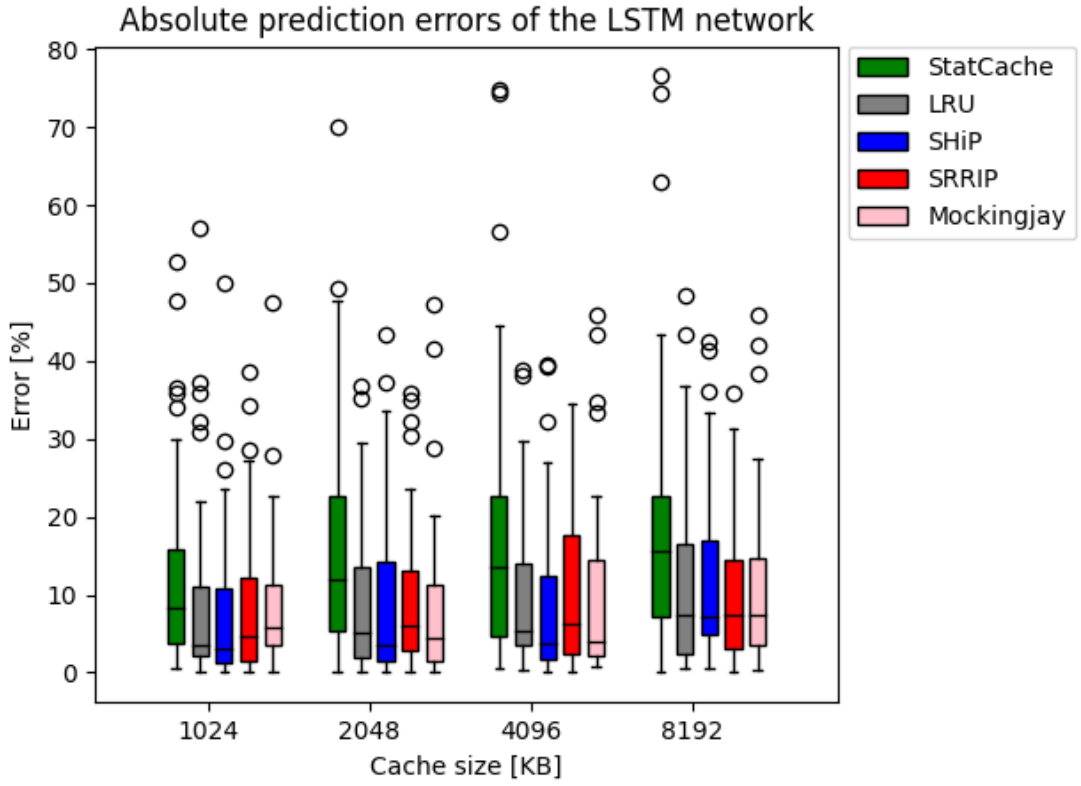


Figure 4.12. *LSTM Network's absolute prediction errors*

program that has been executed on these machines will behave when being executed by machines with the same replacement policy and other LLC sizes. To model this use case, we will use the simulations of the benchmark suite for some cache sizes and move the others to the test set. This way, it will be like having machines with the LLC cache sizes of the training set and LRU replacement policy and predicting the miss ratios for the cache sizes within the test set.

For this problem we will be keeping all the benchmarks for some cache sizes within the training set. Then, the network will be trained on these cache sizes and try to predict the miss ratios of all other cache sizes.

Firstly, we moved the data for one cache size from the train to the test set to see if and how well the network can predict it based on the rest of the benchmarks. More precisely, the test set consists of the column for cache size 3MB and the train set of the other 7 cache sizes [768, 1024, 1536, 2048, 4096, 6044, 8192] KB.

The predictions are way better than any other network could make. The geometric mean of the above errors is 0.9%, as shown in the last row of table 4.7. The geometric mean of LSTM's absolute prediction errors previously predicted for LRU are 3.9% and 5.3% for the two neighboring cache sizes of 2MB and 4MB, respectively. This means that this prediction is undeniably far better than the best prediction we had until now. It was expected that this result would be good, since the network has a very large dataset to train on and can understand the problem very well.

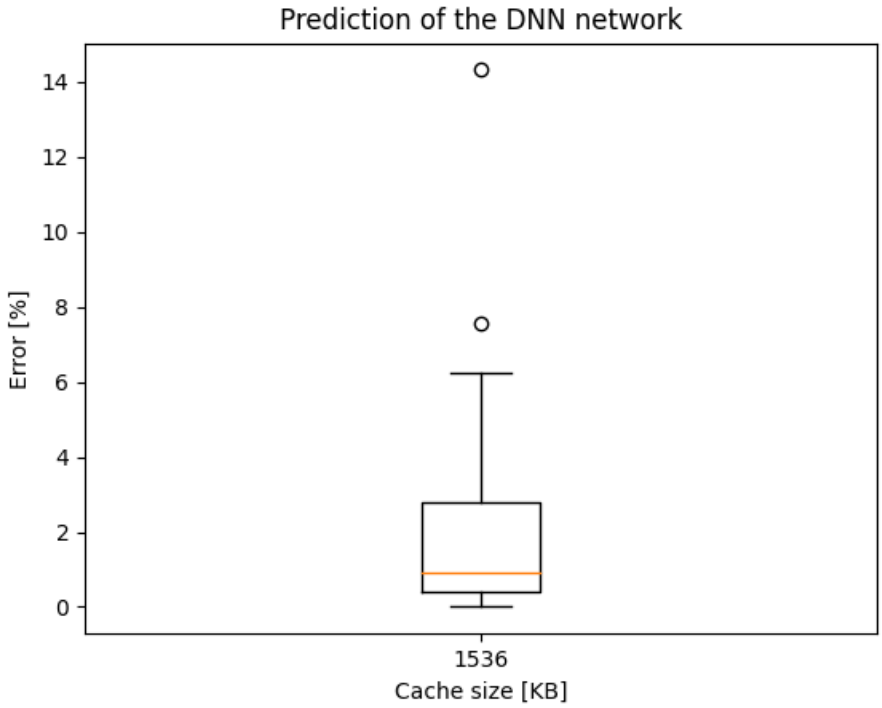


Figure 4.13. *DNN absolute prediction errors for 1 cache size, trained with 7 cache sizes' miss ratios*

This result is a good indicator that our network is suitable to predict this kind of problem. It also begged the question of how few cache sizes are needed in the training set to have an accurate prediction, since simulating seven cache sizes to predict the eighth is not a realistic problem.

Train-set cache sizes [MB]	Cache sizes [KB]							
	768	1024	1536	2048	3072	4096	6144	8192
1, 2, 4, 8	1.4%	-	0.8%	-	0.9%	-	1.1%	-
0.75, 1, 2, 4, 6, 8	-	-	0.8%	-	0.8%	-	-	-
0.75,1,1.5,2,4,6,8	-	-	-	-	0.9%	-	-	-
LSTM Network	-	3.6%	-	3.9%	-	5.3%	-	6.4%

Table 4.7. *Geometric mean of DNN network's absolute prediction errors.*

The immediate next step towards answering this question is to remove another cache size and observe how the predictions' error behaves.

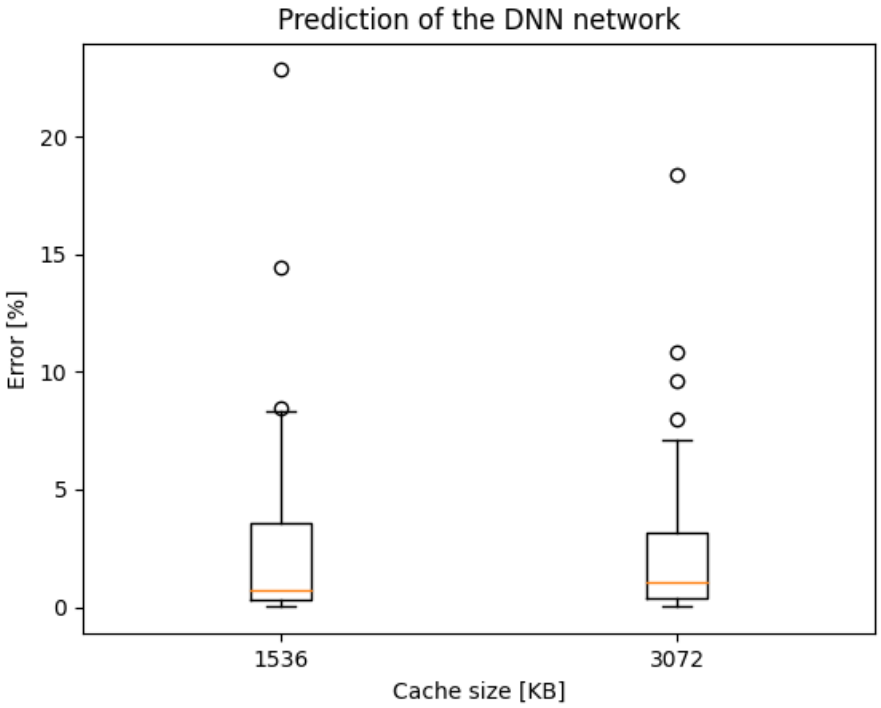


Figure 4.14. *DNN absolute prediction errors for 2 cache sizes, trained with 6 cache sizes' miss ratios*

The geometric mean for the two columns of sizes 1.5MB and 3MB is 0.8% and 0.8%, respectively. We can see that these geometric means are just as good as the ones we observed before.

It is clearly visible that, for each cache size, there is one by far worst predicted benchmark. This is the prediction for the benchmark 416.gamess; its miss ratio is always predicted to be higher, and it will be miss-predicted in the future as well, for good reason. This benchmark consists of one trace, which only has 17k accesses to the LLC, which means that the network reads one window of reuse distances with 17k accesses and tries to predict the miss ratio of it. The majority of reused distance windows that our network is trained on consist of 512k accesses. Thus, the network sees the low number of reuse distances and assumes lots of accesses that only happen once and therefore don't

have reuse distances. These result in misses (cold misses), and therefore the network computes a higher miss ratio. We can safely ignore this output since, with a 30x longer simulation, the output would most likely get closer to the real value.

Lastly, we keep the four most likely LLC caches that a real architecture will have and try to predict the other four values. Namely, we keep the LLC caches of 1024, 2048, 4096, and 8192 KB in the training set and try to predict the behavior of the benchmarks that were simulated with 768, 1536, 3152, and 6044 KB sized LLCs. With this input, we compute similar results as previously. The scale of the geometric means in the third row of table 4.7 is similar to the one for the six cache sizes in the training set.

We can observe in 4.15 that the outliers' errors still persist. It is also worth mentioning that the prediction for 768 KB LLC is clearly, but not a lot worse than the other three predictions. This is a result of it not being in between the cache sizes that have been seen by the network. For example, the network knows the output for 1MB and 2MB sized caches, and therefore it can better understand how a machine would behave for a 1536 KB sized cache. For the 768KB, it is more difficult to extrapolate the behavior of the program, and therefore the errors rise. This error is still much lower than the best error we had by predicting the 1MB-sized cache previously.

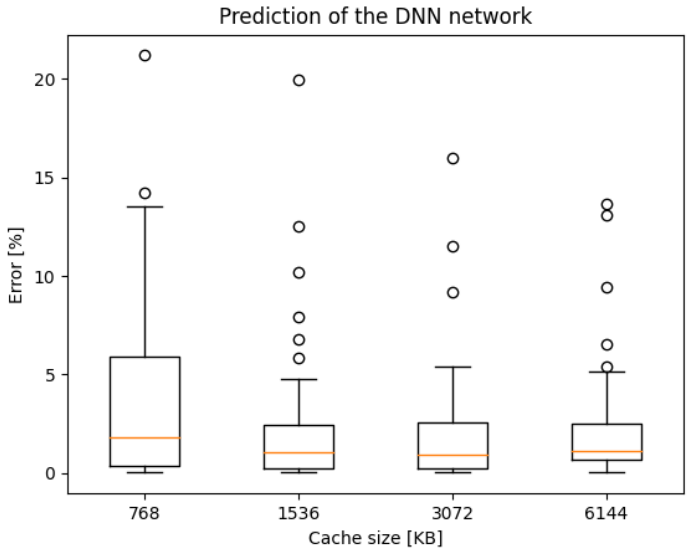


Figure 4.15. *DNN absolute prediction errors for 4 cache sizes, trained with 4 cache sizes' miss ratios*

Again, having less cache sizes in the training set doesn't make sense, therefore we won't show them.

Overall, it is clear that with this usage, the network predicts the problem with utmost precision. The predictions are undeniably better than any other predictions made before without knowing the problem. The downside of this network is that it needs to be executed and trained on all of these different cache architectures. So it is application-specific if this is worth it.

Chapter 5

Conclusion

5.1 Concluding thoughts

In this thesis, we introduce a number of machine learning networks that offer solutions for the problem of predicting miss ratios on different cache architectures. Knowing the cache miss ratios of a program on different architectures is important since it lets the user know what is the best machine to execute said program on. Simulating a process is a difficult and time-consuming process, and skipping it through a prediction is a very efficient solution to this problem.

We introduce four machine learning networks that understand the problem in their own unique way. Inspired by StatCache's already-existing probabilistic approach to this question, we decided to use the same data that it uses for its predictions to train and evaluate a machine learning network. First, a simple MLP network shows that it is quite simple to predict the network as well as the StatCache probabilistic method. This network is based on the reuse distances of one execution of the program and applies to any cache size and any replacement policy. It predicts the miss ratios of any program with any replacement policy with a geometric mean similar to the one that StatCache introduces.

The path to understanding the input data as accurately as possible led us to introduce two new networks. Together with the NLP embeddings of the programs as input, these try to understand the reuse distances differently and thus compute more accurate predictions of the problem. The first one is the LSTM network, which, as the name suggests, understands reuse distances via an LSTM layer. This network is the most time-consuming to train, but it produces the best results of all our networks by far. It produces stable, accurate predictions of miss ratios for any replacement policy and cache size that it is trained on. The CNN network takes the reuse distances as an input to a CNN layer that has the purpose of understanding the reuse distances via a convolution of the values within proximity. This network is significantly less costly to train but has the downside of not being as accurate as the LSTM network. Its predictions are mostly slightly less accurate than the LSTM network's predictions, but for some replacement policies, such as the SHiP, they are even worse, similar to the MLP network's predictions. These two networks are the essential steps that this thesis takes towards understanding this problem accurately.

Lastly, we introduce a deep neural network that works for any cache size. This one,

just like the previous ones, uses reuse distances and the embeddings of the code to understand the problem. The difference is that this one uses the cache size as input and, as we have shown, is able to understand the problem for any cache size. This network that we propose is very strong for smaller cache sizes, but has a slight weakness for larger ones. This network also produces highly accurate results if it already knows the real value of a problem for some cache sizes, and then it can predict any cache size's miss ratio with unseen accuracy. This network can be a stepping stone for further research in the direction of similar architectures that understand the problem better.

5.2 Discussion

The preceding result section, introduced a comprehensive analysis of the empirical data gathered in this study. This discussion section aims to get a deeper understanding of the significance of these results in connection with the broader context of machine architectures. By examining these findings through various lenses, this section seeks to elucidate their meaning, significance, and potential impacts within the realm of computer architectures. Furthermore, this discussion will critically analyze the implications of these results in relation to existing literature, aiming to contribute new perspectives and avenues for further exploration within the field.

The models that we propose in this thesis challenge the outputs of previous prediction mechanisms that have been the best estimations of this problem. They prove to be better than the predictions of StatCache by a significant margin, and also predict the output of any replacement policy. This means that the prediction of cache misses on a machine can now be described as a matter of training a machine learning model on a machine's outputs and letting it predict the miss ratios. The introduction of such a network would not only simplify the process of deciding what is the best architecture for a problem; it would also skip lots of hours of processing or simulation time.

To further contextualize these findings, it's imperative to consider the broader factors that might influence our networks. The introduction of custom-made traces for the benchmarks, and together with that, the embeddings of the specific part, could influence our results a lot. A network's prediction, that knows what part of the program is executing in each trace, can improve the predictions from these networks significantly. Additionally, avenues for further exploration could be made towards data that results from parallel processing or data from real machines. Predicting the miss ratio of processes executing on multiple cores within a machine will introduce much higher difficulty to the problem and should be a subject for further exploration. The data that we used for this thesis is the data produced by a simulation program. It would be interesting to see how these results compare with the networks trained on a real machine, since each has its own difficulties.

Additionally, there is potential for improving our networks. As discussed previously, the LSTM and CNN networks show potential to have even better results. An introduction of more benchmarks to the dataset or a significant change in their architecture could improve their predictions even more. The DNN network is a network that hasn't been explored to the same depth as the other networks by us. Different network architectures

and implementations could reduce its errors significantly. It could even be expanded to have an additional input, the cache miss ratio of one execution on a cache, and then make estimations for other cache sizes.

In summary, the in-depth analysis and interpretation of the findings presented in this discussion underscore the complexities inherent in predicting cache misses. Ultimately, this study stands as a stepping stone in the continual quest to unravel the complexities of computer architecture, encouraging further scholarly inquiry to advance our understanding of this field.

Bibliography

[1] E. Berg και E. Hagersten. *StatCache: a probabilistic approach to efficient and accurate data locality analysis*. *IEEE International Symposium on - ISPASS Performance Analysis of Systems and Software*, 2004.

[2] Ian Goodfellow, Yoshua Bengio και Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[3] N. Beckmann και D. Sanchez. *Modeling cache performance beyond LRU*. *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[4] X. Liu Q. Wang και M. Chabby. *Featherlight Reuse-distance Measurement*. *IEEE International Symposium on High Performance Computer Architecture*, 2019.

[5] Maurice V. Wilkes. *Slave Memories and Dynamic Storage Allocation*. *IEEE Trans. Electron. Comput.*, 14(2):270–271, 1965.

[6] Carole Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely και Joel Emer. *SHiP: Signature-based Hit Predictor for high performance caching*. *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, σελίδες 430–441, 2011.

[7] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely και Joel Emer. *High performance cache replacement using re-reference interval prediction (RRIP)*. ISCA '10, σελίδα 60–71, New York, NY, USA, 2010. Association for Computing Machinery.

[8] Ishan Shah, Akanksha Jain και Calvin Lin. *Effective Mimicry of Belady's MIN Policy*. *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, σελίδες 558–572, 2022.

[9] Alex Krizhevsky, Ilya Sutskever και Geoffrey E Hinton. *ImageNet Classification with Deep Convolutional Neural Networks*. *Advances in Neural Information Processing Systems*F. Pereira, C.J. Burges, L. Bottou και K.Q. Weinberger, επιμελητές, τόμος 25. Curran Associates, Inc., 2012.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren και Jian Sun. *Deep Residual Learning for Image Recognition*. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, σελίδες 770–778, 2016.

[11] Miltiadis Allamanis, Earl T. Barr, Christian Bird και Charles Sutton. *Learning Natural Coding Conventions*. FSE 2014, New York, NY, USA, 2014. Association for Computing Machinery.

[12] Miltiadis Allamanis και Charles Sutton. *Mining Idioms from Source Code*. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, New York, NY, USA, 2014. Association for Computing Machinery.

[13] Amy McGovern. *Building a Basic Block Instruction Scheduler with Reinforcement Learning and Rollouts*. σελίδες 141–160, 2002.

[14] C. Cummins, P. Petoumenos, Z. Wang και H. Leather. *End-to-End Deep Learning of Optimization Heuristics*. *26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017.

[15] Nan Wu και Yuan Xie. *A Survey of Machine Learning for Computer Architecture and Systems*. *ACM Comput. Surv.*, 55(3), 2022.

[16] Xiangyu Dong, Norman P. Jouppi και Yuan Xie. *A circuit-architecture co-optimization framework for exploring nonvolatile memory hierarchies*. *ACM Trans. Archit. Code Optim.*, 10(4), 2013.

[17] Ramazan Bitirgen, Engin Ipek και Jose F. Martinez. *Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach*. *2008 41st IEEE/ACM International Symposium on Microarchitecture*, σελίδες 318–329, 2008.

[18] Daniel Nemirovsky, Tugberk Arkose, Nikola Markovic, Mario Nemirovsky, Osman Unsal και Adrian Cristal. *A Machine Learning Approach for Performance Prediction and Scheduling on Heterogeneous CPUs*. σελίδες 121–128, 2017.

[19] Rosenblatt F. *The perceptron: a probabilistic model for information storage and organization in the brain*. *CCM Information Corporation. First working Deep Learners with many layers, learning internal representations*.

[20] A. G. Ivakhnenko και V. G. Lapa. *Cybernetic Predicting Devices*. *CCM Information Corporation. First working Deep Learners with many layers, learning internal representations*.

[21] Alex Graves, Marcus Liwicki, Santiago Fernández, Roman Bertolami, Horst Bunke και Jürgen Schmidhuber. *A Novel Connectionist System for Unconstrained Handwriting Recognition*. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(5):855–868, 2009.

[22] Hasim Sak, Andrew W. Senior και Françoise Beaufays. *Long short-term memory recurrent neural network architectures for large scale acoustic modeling*. *INTERSPEECH*, σελίδες 338–342, 2014.

[23] Qianji Zhao και Zequn Shang. *Deep learning and Its Development*. *Journal of Physics: Conference Series*, 1948(1):012023, 2021.

[24] Sunitha Basodi, Chunyan Ji, Haiping Zhang και Yi Pan. *Gradient amplification: An efficient way to train deep neural networks*. *Big Data Mining and Analytics*, 3(3):196–207, 2020.

[25] Joshua Peeples, Weihuang Xu και Alina Zare. *Histogram Layers for Texture Analysis*. *IEEE Transactions on Artificial Intelligence*, 3(4):541–552, 2022.

[26] M.V. Valueva, N.N. Nagornov, P.A. Lyakhov, G.V. Valuev και N.I. Chervyakov. *Application of the residue number system to reduce hardware costs of the convolutional neural network implementation*. *Mathematics and Computers in Simulation*, 177:232–243, 2020.

[27] Hamid Sadeghi και Abolghasem A. Raie. *HistNet: Histogram-based convolutional neural network with Chi-squared deep metric learning for facial expression recognition*. *Information Sciences*, 608:472–488, 2022.

[28] Ronan Collobert και Jason Weston. *A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning*. New York, NY, USA, 2008. Association for Computing Machinery.

[29] Dan C. Cireșan, Ueli Meier, Jonathan Masci, Luca M. Gambardella και Jürgen Schmidhuber. *Flexible, High Performance Convolutional Neural Networks for Image Classification*. *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, σελίδα 1237–1242. AAAI Press, 2011.

[30] Dominik Scherer, Andreas Müller και Sven Behnke. *Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition*. σελίδες 92–101, 2010.

[31] Y Lan Boureau, J. Ponce και Yann Lecun. *A Theoretical Analysis of Feature Pooling in Visual Recognition*. σελίδες 111–118, 2010.

[32] S. VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta και Y. N. Srikant. *IR2VEC: LLVM IR Based Scalable Program Embeddings*. *ACM Trans. Archit. Code Optim.*, 17(4), 2020.

[33] Yulei Sui, Xiao Cheng, Guanqin Zhang και Haoyu Wang. *Flow2Vec: Value-Flow-Based Precise Code Embedding*. *Proc. ACM Program. Lang.*, 4(OOPSLA), 2020.

[34] John L. Henning. *SPEC CPU2006 Benchmark Descriptions*. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

[35] James Bucek, Klaus Dieter Lange και Jóakimv. Kistowski. *SPEC CPU2017: Next-Generation Compute Benchmark*. *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, σελίδα 41–42, New York, NY, USA, 2018. Association for Computing Machinery.

[36] Nathan Gober, Gino Chacon, Lei Wang, Paul V. Gratz, Daniel A. Jimenez, Elvira Teran, Seth Pugsley και Jinchun Kim. *The Championship Simulator: Architectural Simulation for Education and Competition*, 2022.

[37] Geoffrey Webb, Claude Sammut, Claudia Perlich, Tamás Horváth, Stefan Wrobel, Kevin Korb, William Noble, Christina Leslie, Michail Lagoudakis, Novi Quadrianto, Wray Buntine, Lise Getoor, Galileo Namata, Jiawei Jin, Jo Anne Ting, Sethu Vijayakumar, Stefan Schaal και Luc De Raedt. *Leave-One-Out Cross-Validation*. 2010.

[38] Diederik P. Kingma και Jimmy Ba. *Adam: A Method for Stochastic Optimization*. *CoRR*, abs/1412.6980, 2014.

[39] Cem Subakan, Mirco Ravanelli, Samuele Cornell, Mirko Bronzi και Jianyuan Zhong. *Attention Is All You Need In Speech Separation*. *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, σελίδες 21–25, 2021.

List of Abbreviations

MSE	Mean Squared Error
IR	Intermediate representation
NLP	Natural Language Processing
LRU	Least Recently Used
MLP	Multi-Layered Perceptron
RNN	Recurrent Neural Network
LSTM	Long Short-Term Memory
CNN	Convolutional Neural Network
DNN	Deep Neural Network
RL	Reinforcement Learning