applied sciences

MDPI

Review

A Survey of Machine Learning-Based System Performance Optimization Techniques

Hyejeong Choi and Sejin Park *

Department of Computer Science, Keimyung University, Daegu 1095, Korea; hyejeong12311@gmail.com
* Correspondence: baksejin@kmu.ac.kr; Tel.: +82-53-580-5270

Abstract: Recently, the machine learning research trend expands to the system performance optimization field, where it has still been proposed by researchers based on their intuitions and heuristics. Compared to conventional major machine learning research areas such as image or speech recognition, machine learning-based system performance optimization fields are at the beginning stage. However, recent papers show that this approach is promising and has significant potential. This paper reviews 11 machine learning-based system performance optimization approaches from nine recent papers based on well-known machine learning models such as perceptron, LSTM, and RNN. This survey provides a detailed design and summarizes model, input, output, and prediction method of each approach. This paper covers various system performance areas from the data structure to essential system components of a computer system such as index structure, branch predictor, sort, and cache management. The result shows that machine learning-based system performance optimization has an important potential for future research. We expect that this paper shows a wide range of applicability of machine learning technology and provides a new perspective for system performance optimization.

Keywords: deep learning; machine learning; system performance; optimization

check for updates

Citation: Choi, H.; Park, S. A Survey of Machine Learning-Based System Performance Optimization Techniques. *Appl. Sci.* **2021**, *11*, 3235. https://doi.org/10.3390/app11073235

Received: 23 February 2021
Accepted: 2 April 2021
Published: 4 April 2021

1. Introduction

As the amount of data and the complexity of data structures increase, computer system performance optimization is highly required. However, traditional optimization techniques do not work adaptively because most of them are designed for specific data under a specific environment. Hybrid techniques [1–3] came out but are still time-consuming and difficult to obtain good results on complex data distribution.

Prefetching or prediction techniques have already existed concepts, but they are limited in time and space. In addition, traditional techniques only work well under certain patterns or situations. For example, stride prefetcher [4] cannot learn various data characteristics such as delta [5]. That is to say, there is a possibility of space waste and performance decline when irregular data come in. Also, depending on the data distribution, the conventional workload or pattern-based algorithm leads bad if the data distribution does not fit to the algorithm. Furthermore, existing techniques for handling large data such as database management and sorting have reached speed limits.

Therefore, since it is difficult to intuitively design a system that satisfies the increasingly complex workload and performance metrics (latency, prediction) that the system targets, Machine Learning was introduced to design the system architecture more automatically. Machine learning can be a good solution because machine learning can find the various relationship among data, such as linear, non-linear. This is the biggest advantage of a machine learning-based approach, and it can automatically explore patterns for a given workload.

The goal of this survey paper is to show the state-of-the-art machine learning techniques for various areas in terms of system performance optimization. It covers various machine learning-based approaches from CPU prefetching to basic data structure.

In the papers we reviewed, each topic or designed model is unique, but the main purpose is the same-system performance optimization using machine learning techniques.

This paper reviews 11 ML-based techniques from nine papers that have applied machine learning to optimize various systems such as traditional database management, data structure, sorting algorithms, etc. Each system has a target optimization direction, and by applying machine learning, problem-solving possibilities and future potential are confirmed. In addition, there may be various models, approaches, and available data for a given problem. Therefore, based on the challenge found in the conventional system, we explore the design space for the ML component and analyze how to apply it to the conventional system.

The paper is organized as follows. Section 3 introduces recent works that applied machine learning for system performance optimization. Section 4 describes the characteristics and architecture design of the works introduced in Section 3. Finally, Section 5 discusses the challenges and future directions of applying ML for system designs and concludes this paper.

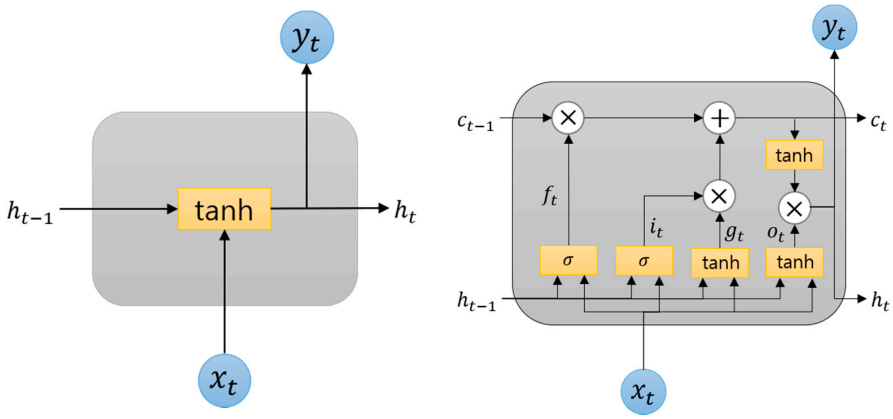
2. Background

We briefly summarize the models introduced in this survey. Most approaches applied four models.

Perceptron receives an input signal and outputs one signal. Each input signal has a weight, and this weight represents the importance of the input signal. Single-layer perceptron have only input and output stages, and multi-layer perceptron add a hidden layer between input and output.

Support Vector Machine (SVM) is an algorithm similar to perceptron and is a method of defining a decision boundary to obtain a baseline for classification and maximizing the margin between the decision boundary and the actual data.

Recurrent Neural Network (RNN) has input signals, output signals, and weights like perceptron, but has several hidden layers inside, and these hidden layers are not only affected by the current input but are cyclically connected to memorize previous input information. The left side of Figure 1a is the RNN cell. To calculate the hidden state of the current time t, it is calculated as follows using the hidden state h_{t-1} of the previous time $t-1$ and the input x_t. Next, the calculated h_t value is sent to the output layer and becomes an input value for calculating h_{t+1} again.



(**a**) RNN cell (**b**) LSTM cell

Figure 1. (**a**) The Architecture of Recurrent Neural Network (RNN) cell; (**b**) The Architecture of Long Short-term Memory (LSTM) cell.

Long Short-term Memory (LSTM) [6] is a proposed method to solve the disadvantage that the current hidden layer loses the memory of the previous input information when the input of the RNN becomes long. In order to solve the problem, unnecessary memories are deleted by adding a value called cell state, and things to remember are selected. The

right side of Figure 1b is an LSTM cell. LSTM has three gates to obtain hidden state and cell state values. The first forget gate (f_t) is a gate to delete memories. As the second input gate (i_t,g_t), it is a gate to remember the current information. Finally, the output gate (o_t) is a gate to calculate the hidden state at the current point in time. The cell state c_t can be obtained by multiplying two values (i_t,g_t) calculated at the input gate for each element and adding the memory selected at the input gate to the result of the forget gate. The hidden state h_t can be obtained by performing an entrywise product for the cell state c_t and the output gate (o_t).

3. Review of Machine Learning-Based System Performance Optimization Research

This section reviews machine learning-based system performance optimization research. Table 1 is a summary of the paper before surveying the paper.

Table 1. Summary of the reviewer searches.

Research	Model	Description
Learned Index Structures [7]	B-Tree index Single and Multi-layer Perceptron	It replaces a B-Tree with a neural network to improve query speed. A regression-based model predicts the position of the key if it is given as the input. The model is hierarchically organized for higher accuracy.
	Hash-Map index Single and Multi-layer Perceptron	It replaces a hash-function with a neural network to solve the collision problem, which is the major problem of hashing technique. When a key is given as an input value, a regression-based model predicts the position of the key. This model implemented model hierarchically to reduce the complexity of a single model.
	Bloom filter RNN	It includes a neural network to minimize the spatial overhead and false positive rate of Bloom filter. It designed a classification model that treats Bloom filter as a binary classification problem to predict 0 or 1 when a key comes in.
Pavo [8]	LSTM	It replaces a hash-function with LSTM to increase the space utilization rate of inverted indexing. The architecture consists of a total of four stages: Input Stage is pre-processing data, Disperse Stage distributes the sub-models uniformly, Mapping Stage maps the sub-data to a local hash, and the Join Stage creates a global hash table.
Learned Branch Predictor [9]	Perceptron	It replaces the traditional two-level scheme counter table with perceptron for efficient branch prediction. Given input is indexed into one of the perceptron tables, and the model classifies whether the branch will be taken or will not be taken by a weight vector.
NN-sort [10]	Multi-layer Perceptron	It is a regression-based neural network to improve sorting performance. The overall framework consists of three stages: the input phase is a step of pre-processing the input data, the sorting phase is a step of sorting the data by repeatedly inputting data into the model. Finally, there is a polish phase to correct inaccurately sorted elements to produce an accurate result.
MER-sort [11]	Perceptron	It is a machine learning sorting algorithm-based radix sort. Similar to radix sort, this model proceeds sorting by separating the unordered keys into partial buckets.
Learned Reuse Predictor [12]	Perceptron	It is a perceptron-based reuse predictor to improve the accuracy of reuse prediction. However, an actual neural network is not applied, it only brought an idea of a perceptron. The multiple inputs are given, and it is indexed each weight table using input and PC, and it conducts prediction by adding each weight.
Learned Cache Prefetching [13,14]	Perceptron	It is a perceptron-based cache prefetching technique to reduce unnecessary prefetching. The basic configuration is the traditional prefetcher that proposes the block. It keeps the latest cache miss address, and perceptron determines whether the block is accepted or not.
	LSTM	It is a neural network-based prefetcher considering data distribution. Two LSTM-based architectures are proposed. The first is embedding LSTM. If the embedded Program Counter (PC) and delta (a distance between two addresses) are given as input, the delta value is predicted. The second is Clustering + LSTM. It separates various cluster regions according to addresses and predicts deltas within the cluster.

Table 1. *Cont.*

Research	Model	Description
Learned Cache Replacement Policy [15]	LSTM, SVM	It proposed an attention-based LSTM for offline prediction and SVM for online prediction. Although there are differences in the architecture of the two models, both models work with the same input. This model classifies whether the input is cache-friendly or not.

3.1. Machine Learning Based Index Structures

Traditional data management systems use a heuristic-based algorithm, which means that they do not utilize the characteristics of specific applications and data themselves [16]. In order to improve these problems, the papers [7,8] introduce a new index structure that leverages machine learning. In paper [7], three well-known index structures (B-Tree, Hash-Map, Bloom filter) are re-designed with machine learning-based techniques. Ref. [8] also proposes an inverted index based on machine learning.

3.1.1. Learned B-Tree

B-Tree index model predicts a position of look-up key within a sorted set of keys. Original B-Tree guarantees that found the key is the first key or higher than the look-up key. In addition, B-Tree determines whether the look-up key is in the page through binary search. In this paper [7], a regression-based range index is proposed.

For the Learned B-Tree design, the Recursive Model Index (RMI) [7] was designed to increase accuracy and reduce complexity rather than a single CDF model. Because RMI trains only for each range of data, such as B-Tree, RMI can easily build with a simple model. In addition, this paper supports B-Tree. If learning data distribution does not fit for prediction, they use B-Tree.

In Figure 2, the upper-level model receives the key as input and predicts the next-level model until it is located at the lowest level. Upon reaching the lowest level model, the model predicts the location of the queried key and finds the queried key between min_err and max_err of the predicted location.

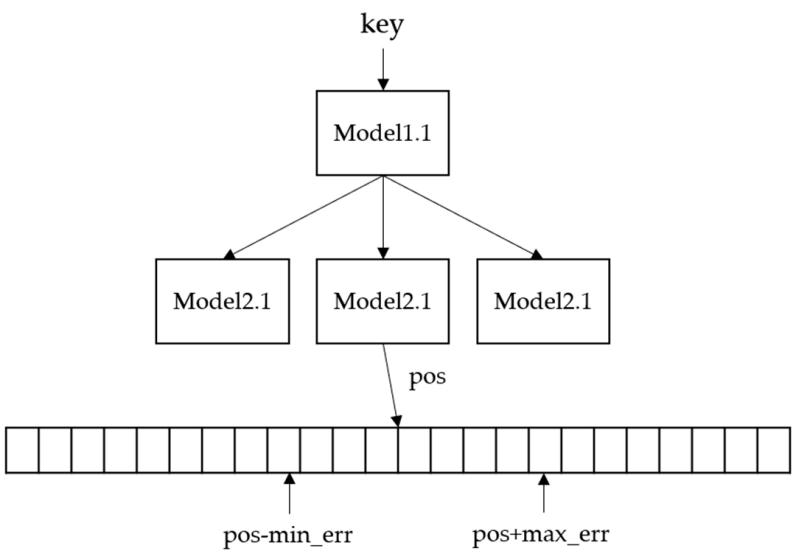


Figure 2. Learned B-Tree [7]: The top-level model receives a key as an input and it predicts the final position between min_err and max_err.

Compared with the original B-Tree under various conditions, the learning index model achieved much less memory consumption and faster look-up speed than B-Tree.

3.1.2. Learned Hash-Map

A Hash-Map uses hash-functions to map the position in the array. The goal of the learned Hash-Map is to reduce the hash conflict [7]. To address the problem, they replace the hash-function with a machine learning-based model. They used a regression-based model. The model is also modeled as CDF for the key distribution, and it also uses RMI like the range index. In Figure 3, when the trained model receives a key as an input, it predicts the position of the key in the array, similar to a hash function. When the learned hash-function was compared with the existing hash-function, the collision rate of the learned hash-function for the given dataset was reduced by up to 77%.

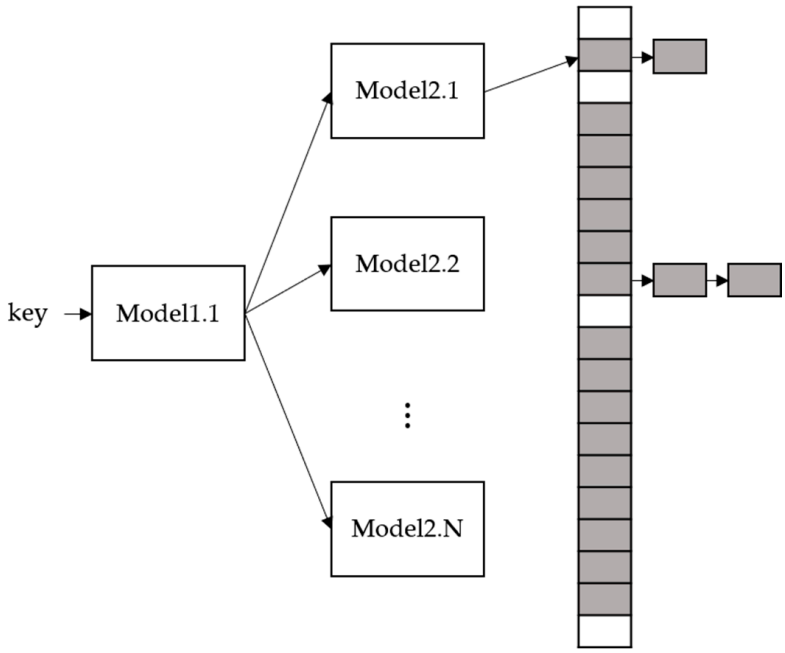


Figure 3. Learned Hash-Map [7]: The top-level model received the key as an input and it predicts the final position in the leaf model.

3.1.3. Learned Bloom Filter

A Bloom filter is a structure that determines whether the data is a member of the set or not. By design, a Bloom filter is space efficient because it checks whether the key exists at the position that K hash-function returned to the bit-array, but Bloom filter still occupies a large amount of memory. Also, it has a false-positive result because of hash function conflict. The goal of this learned existence index [7] is to minimize the space and false positive. One way to construct a Bloom filter is to think of it as a binary classification model. Therefore, a predictive model has been proposed to determine whether an element is a key or not. However, since this model has a false negative, unlike the Bloom filter, the authors add an overflow Bloom filter to keep the false-negative rate (FNR) at zero. In Figure 4, if a key is given as input to the trained model and the predicted value by the model exceeds the threshold, queried key is determined that the key exists in the set. However, if it does not exceed the threshold, the existence of the key is checked through the actual bloom filter. In an experiment to track blacklisted phishing URLs, Bloom Filter of 1% FPR needed 2.04 MB of memory, while the learned Bloom filter reduced the size of the required memory with 1.31 MB of memory.

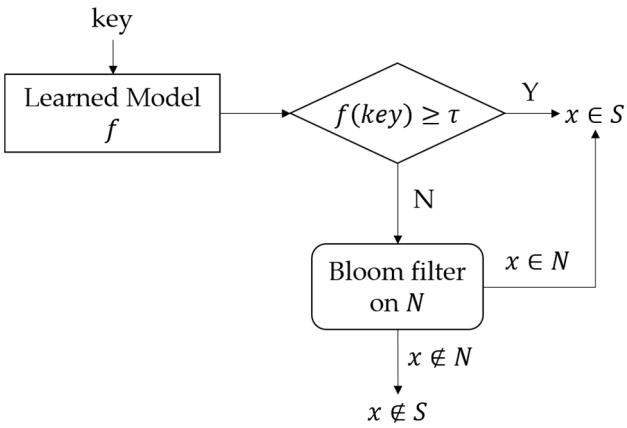


Figure 4. Bloom Filter Architecture [7]: The Learned model received a key as an input and predicts a regression value to determine if the key exists in the set or not.

3.1.4. Learned Inverted Index

An inverted index is a technique that is widely used in a large-scale text search. In contrast to forward index, words become key and documents become value (so, inverted). However, when indexing such a large-scale text, it is generally generated through the same hash function without understanding the distribution of various data. In order to increase the space utilization rate of the inverted list when the large amount of data is given, this paper presents Pavo [8], an RNN-based learned inverted index.

Pavo conducts four stages as shown in Figure 5. The first stage, the input stage, is a stage for pre-processing input data. The input data string is processed as bigram and used as the input sequence of the RNN. For example, the word "Documentaries" can be divided into do, oc, cu, . . . , es when using bigram. The second stage, the disperse stage, consists of a hierarchical model and plays a role to distribute data evenly among multiple models. By dividing the dataset into several sub-data sets, the low-level model helps to learn the mapping relationship of the dataset. The third stage, the mapping stage, maps each data in the dataset of models in the last layer to a local hash. The last stage, the join stage, is connected by a local hash to create the final hash-table. All local hashes are serially connected to create a global hash table, and as shown in Figure 5, pos1 located in the local hash of RNN 3.2 is mapped to the global hash table to finally find the location of the key. This paper introduces two learning methods in the mapping stage: supervised and unsupervised learning methods. First, supervised learning aligns keys, then sets labels in order, and learns to map keys one-to-one with hash entries. Supervised learning hash-function tends to overfitting the model when the number of labels is small, or the noise is high [17]. Therefore, unsupervised learning is introduced. The second unsupervised learning is a method of self-learning the distribution of data without labels. When each key is input to the model, the model outputs a vector of hash table length. Next, the vector is the input to the Softmax function [18]. It finds the position of the largest value in the output value of Softmax and sets the position's value to 1. At the end of the stage, to obtain a distribution, the value of each category is summed, and the model optimized the computed error through a loss function to distribute the data uniformly. When mapping size is 1000, the average number of look-up of learned hash-function from the entire dataset was approximately 1.0–1.15, which was lower than 1.5 of the existing hash-function.

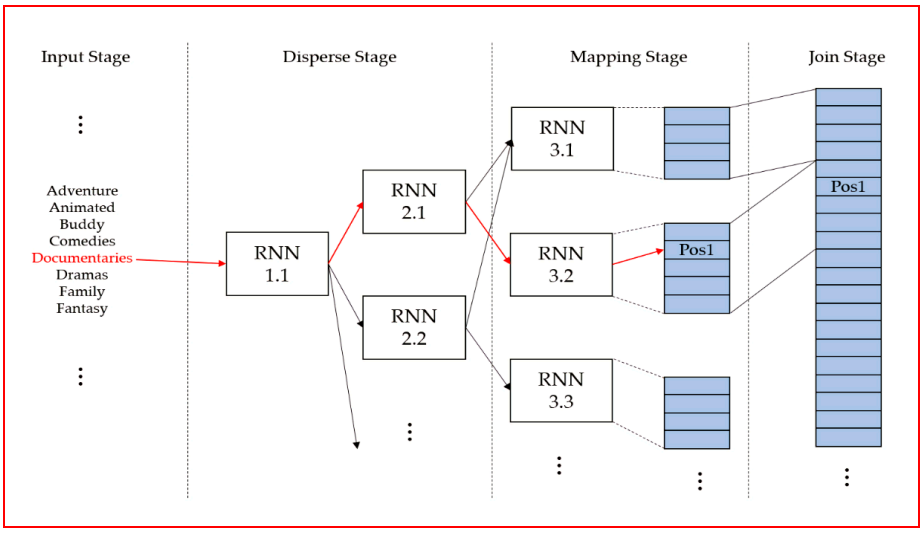


Figure 5. Learned Inverted Index [8]: The RNN model received the preprocessed data string as an input and predicts the position of the data in the leaf RNN.

3.2. Perceptron Based Branch Predictor

The existing 2-bit branch prediction [19] stores two-level scheme branch history information for branch prediction. In order to predict, a pattern history table (PHT) is used. To improve branch accuracy, a larger table is needed. In order to reduce the space overhead, it shares counters, but duplicates may occur. Because of these disadvantages, history length is limited, and the predictor is difficult to learn a long history. Perceptron is a simple neural network, but it has the advantage of learning a long history. This paper proposed a two-level scheme using perceptron instead of a 2-bit counter [9]. A detailed description of this structure is shown in Figure 6. If a branch address is given, the address is hashed to obtain an index. The index obtained by hashing the branch address indexes the entry of the Perceptron Vector Table. The entry has a weight and a bias indicating the correlation between branch addresses. The extracted weights and biases are dot products with the Global Branch History that stores the past branch address to calculate the output value y. If y is negative, the branch is non-taken, otherwise it is taken. Finally, when the actual branch result appears, the weight is updated by comparing it with the predicted branch result. When comparing the prediction rate according to the hardware budget, learned branch predictor improved 14.7% over gshare [20] and 10% over bi-mode [21].

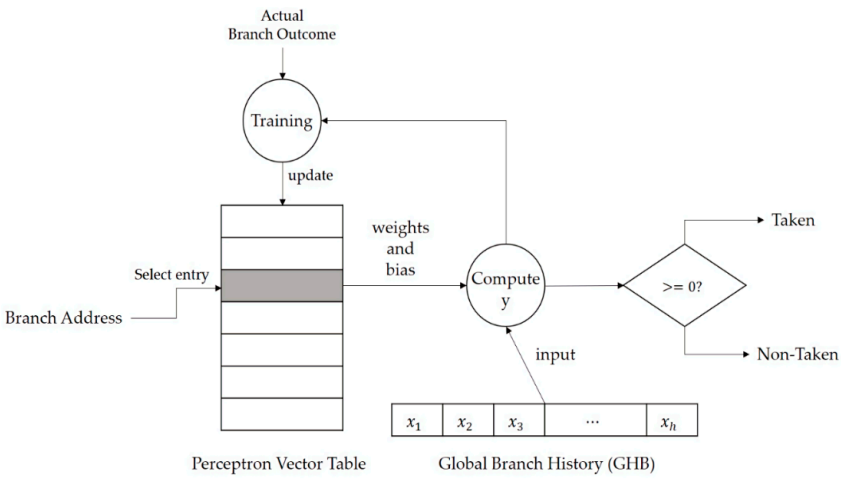


Figure 6. Perceptron Predictor Block Diagram [9]: After selecting the parameter corresponding to the branch address, compute dot product the parameter with the previous branch history, and the predictor decides if the branch is taken or not based on the result.

3.3. Machine Learning Based Sort

Sorting is one of the basic tasks of computing components used in many applications. Recently, machine learning is applied to solve the problem because the speed of the algorithm has reached the limit.

3.3.1. NN-Sort (Neural Network Sort)

In paper [10], a neural network-based sorting algorithm called NN-sort [10] is proposed. NN-sort trains the model about past data and classifies incoming data in the future. The overall structure of NN-sort is divided into three phases as shown in Figure 7: Input Phase, Sorting Phase, and Polish Phase.

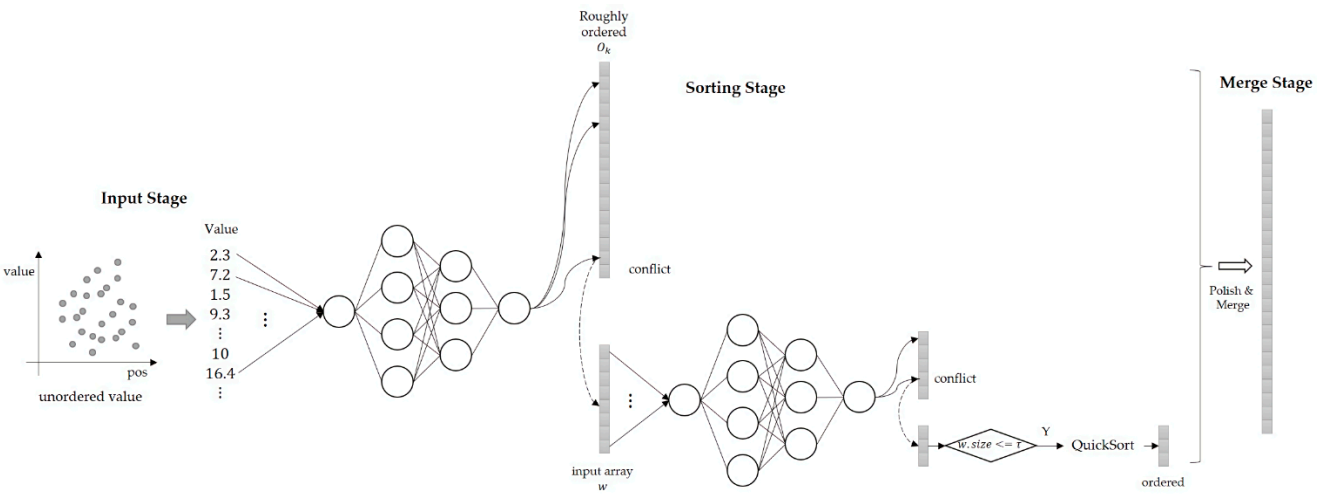


Figure 7. NN-Sort (Neural Network Sort) [10].

Input Phase serves to convert data into a vector so that NN-sort can learn the various type of data.

Sorting Phase serves to convert unsorted data into roughly sorted data by repeatedly executing the proposed model. If different input data result in the same position, NN-sort stores the input data in the conflicting array c. On the other hand, the non-conflicting key is stored in the array o_k. In this case, c is used as input of model f at the next iteration. If c is below the threshold after one iteration, c is not supplied from model f again, but c is sorted by the existing algorithm.

At Polish Phase, all arrays of $\{o_1, o_2, \ldots, o_k, \ldots, o_t\}(0 < t < \epsilon$, ϵ is the maximum number of iterations) are scanned and merged with w. If the position of an element in o_i is incorrect, NN-sort inserts the element in the correct position and if the position of the element is correct, NN-sort appends the element in the result array. Compared to the real-world dataset, NN-sort shows a sorting rate of 5950 data points per second, which is 2.72 times of std::sort [22], 7.34 times of Redis sort [23], and 58% faster than std::heap [22].

3.3.2. MER-Sort (Model-Enhanced Radix Sort)

Machine learning sorting algorithm-based radix sort with almost linear time called MER-sort has been proposed [11]. When n data are given, radix sort performs bucket sorting of k phases with time complexity of $O(kn)$ [24]. MER-sort is also similar to radix sort because MER-sort divides the bucket until the input is sorted. But MER-sort learns the CDF model, and the learned model predicts the position [11].

Figure 8 shows the structure of MER-sort. In Step 1A, the model predicts the bucket where the input will be located when the unordered input array comes in. At this time, when the bucket overflows, the input enters the spill bucket. In Step 1B, the model repeats the same mapping operation until the number of inputs per bucket is less than the threshold (typically 100). Next, for a bucket of variable size, MER-sort determines how many inputs

each bucket has. The number of inputs per bucket is stored in Count Array. In addition, a bucket of variable size can be stored directly in the output array (Step 2). Finally, the spill bucket is merged with the output array, and insert sort is used for modification (Step 3). At up to 1 billion keys of standard normal distribution, MER-sort showed only 30% of sorting speed compared to other algorithms [11].

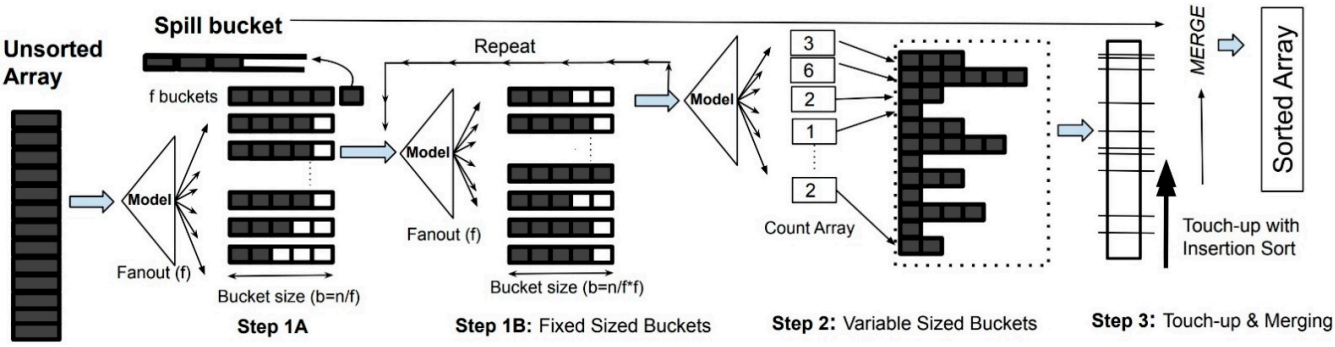


Figure 8. MER-sort (Model-Enhanced Radix Sort) [11].

3.4. Machine Learning Based Cache Management

The speed of a CPU is far ahead of the memory latency. This problem causes a bottleneck in many computer applications [25]. A hierarchical memory system was constructed with the addition of cache memory to overcome between the speed of memory access and the speed of the CPU. Since cache memory is even smaller than main memory capacity, cache memory must be managed efficiently. Therefore, various techniques such as cache replacement, prefetching, etc. were introduced to increase efficiency. Since these traditional techniques are speculative, they should ideally predict future patterns. However, predictors are almost impossible to know the pattern of the future, so one way is to learn memory patterns of the past.

3.4.1. Reuse Prediction

Cache block reuse prediction is a technique to predict whether the block currently residing in the cache is likely to be accessed again before being replaced. Previous techniques have not significantly combined various input [12]. The proposed model is possible to increase cache efficiency by weighting various inputs and analyzing correlations with each other. This paper does not use perceptron algorithm but uses a similar perceptron learning algorithm [12]. In this paper [12], when a block is accessed a predictor works. If the block was in the cache (i.e., cache hit), the predictor predicts the possibility of reusing in the near future. If the block is predicted as it will not be reused, the block will not be stored in the cache again and vice versa.

Figure 9 shows the structure of the predictor presented in the paper [12]. The predictor uses six features: the memory access trace and the PC of the memory instruction that caused the eviction, and the memory address bits. Each feature has its own table and indexes into the corresponding entry to the hash by applying the XOR gate to each feature with the PC of the instruction. This predictor is similar to perceptron predictor known as hashed perceptron [26]. Next, when the weights of the feature table are extracted through the calculated index and the weights extracted from each table are added to exceed the threshold, it is determined that the accessed block is not reused. If the sum of corresponding weights exceeds the threshold, the predictor is predicted that the prediction is incorrect.

In a single-thread workload, the average Sampling Dead Block Prediction (SDBP) [27] (3.5%), Signature-based Hit Predictor (SHiP) [28] (3.8%), and Perceptron (6.1%) improved compared to the LRU when the predictor's introduction per cycle (IPC) by the LRU's IPC.

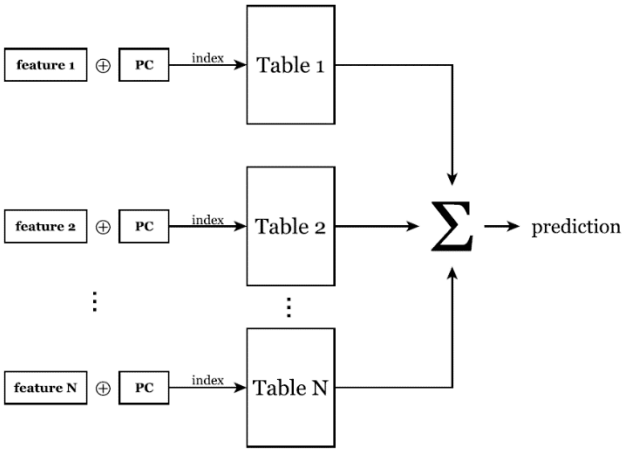


Figure 9. Perceptron Reuse Predictor [12]: After XORing the PC and each feature and adding all indexed weights, the predictor predicts the reusability of the block based on the added weights.

3.4.2. Perceptron Based Prefetcher

Prefetching is the process of fetching data from memory into the cache before the processor demands it. If the prediction is wrong, cache pollution occurs, and the efficiency may be reduced because additional work is required. To remove this unnecessary memory request, a two-level prefetching mechanism based on perceptron has been proposed [13]. The first level combines the existing prefetchers (stride prefetcher [4], Markov prefetcher [29]) and Global History Buffer (GHB) [2]. In Figure 10, if a cache miss occurs, the model pushes the block to the GHB stack, and the convolutional prefetcher proposes a block. Next, after finding the proposed block in the index table, it is hashed to the GHB to extract the features related to the proposed block and input it to the perceptron. Perceptron performs prediction based on the input features and does not prefetch the corresponding block if the predicted value is negative. Otherwise, it prefetches the proposed block. Also, depending on the result, the block is added to the Deny Table or Accept Table. Both tables are used as perceptron's training data to adjust memory access patterns. Compared with Stride, Markov Prefetcher, an average of 80.84% of proposed requests by stride prefetcher was rejected, and an average of 49.71% of proposed requests by Markov prefetcher was rejected. However, the cache hit rate was −1.67 to 2.46% because even useful proposals could be rejected.

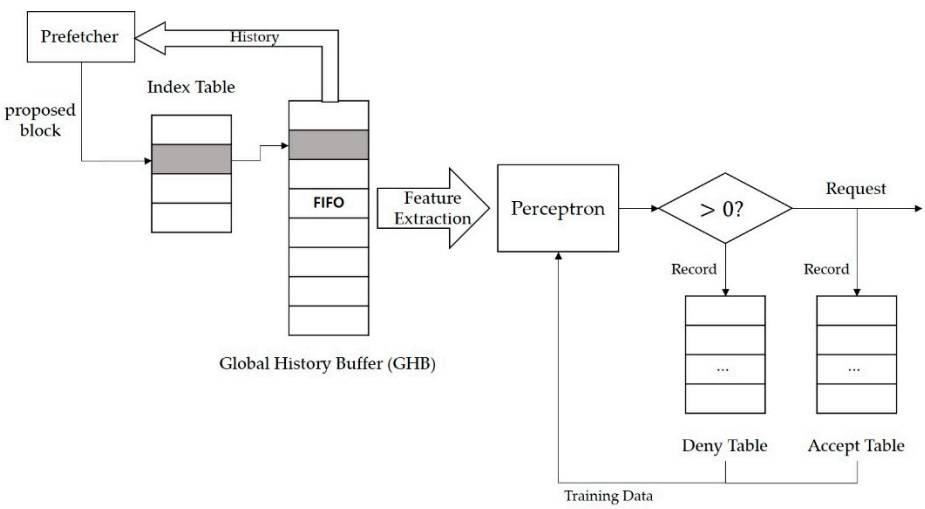


Figure 10. Two Level Prefetcher [13]: In the first level prefetcher, the feature related to the block proposed by the existing prefetcher is transferred to the perceptron, and in the second level prefetcher, the proposal is accepted or denied based on the value predicted by the perceptron.

3.4.3. LSTM Based Prefetcher

Prefetch's role is to determine what data needs to be cached to ease the memory wall [30]. For prefetch performance, many existing tasks rely on tables [2,4,5,29]. However, as the workload increases, the table also must expand, which puts pressure on the hardware. Considering that the previous long history will affect the current prefetching, this paper proposes a prefetcher using a sequence-based neural network. In this paper [14], two designs are introduced.

The first is Embedding LSTM, the second is Clustering + LSTM. In the Embedding LSTM (see Figure 11), the PC and the delta are individually embedded, and the two features are connected to become a two-layer LSTM input. Then, the model outputs the top-10 deltas for each time stamp and selects the highest probability delta among the last timestamp K.

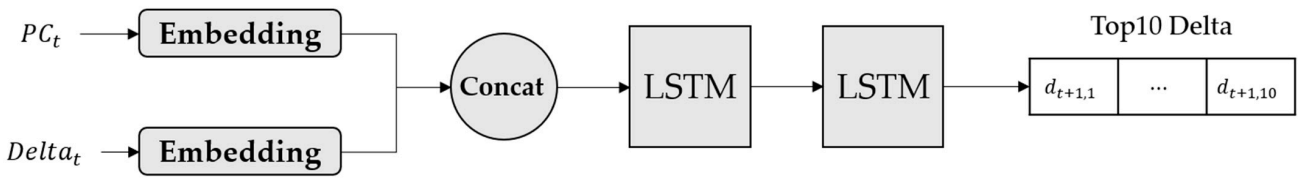


Figure 11. The Embedding LSTM Model [14].

Clustering + LSTM (see Figure 12) is built on the assumption that the interaction between addresses occurs locally. Therefore, the address space is clustered using k-means. The PC and Address are divided into each cluster, and the delta between adjacent addresses within each cluster is calculated. Assuming that there are two clusters in Figure 12, $PC_{1\sim N}$ and $Addr_{1\sim N}$ are clustered, and the PC and Delta existing in the first cluster are called $PC_{1,n}$ and $Delta_{1,m}$. This separate set of data is embedded with appropriate values, and the two embedding values are concatenated to become the input of the LSTM. At this time, to reduce the size of the model, multi-task LSTM is used to model each cluster. LSTMs are an independent model, but the weights of model tie and cluster ID provide to give a bias. Because this structure can calculate delta within the address set, the vocabulary size can be significantly reduced compared to Embedding LSTM. However, because Clustering + LSTM trains the model in a separate address region, it has a disadvantage that access to other regions cannot be handled.

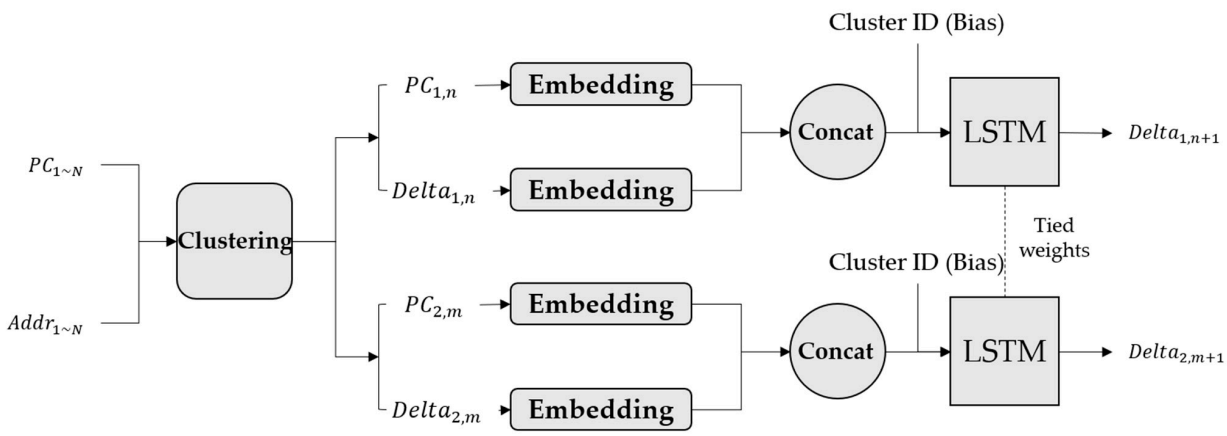


Figure 12. The Clustering + LSTM Model [14].

3.4.4. Cache Replacement Policy

Cache replacement is a problem of determining which blocks to remove when a new access is requested under the "cache capacity is full" condition. Cache replacement is important to replace the block so that cache hit ratio is not violated. Existing heuristic-

based replacement algorithms have a disadvantage that they cannot adapt to data changes. To solve this disadvantage, this paper [15] proposes two neural network models as a solution. The first model is an attention-based LSTM, which is an offline learning, and the second model is an integer Support Vector Machine (ISVM), called Glider, which enables offline and online learning. The two models are Hawkeye [31]-based prediction approach, which learns optimal caching behavior as supervised learning model. The first model, an attention-based LSTM, sees the problem as a sequence labeling problem. The detailed architecture consists of embedding, single layer LSTM, and attention layer as shown in Figure 13. The embedding layer is the task of processing the input into meaningful data, the LSTM layer learns the behavior of the cache, and the attention layer learns the correlation between PCs. The output is a binary value which predicts whether the PC sequence is cache-friendly or not.

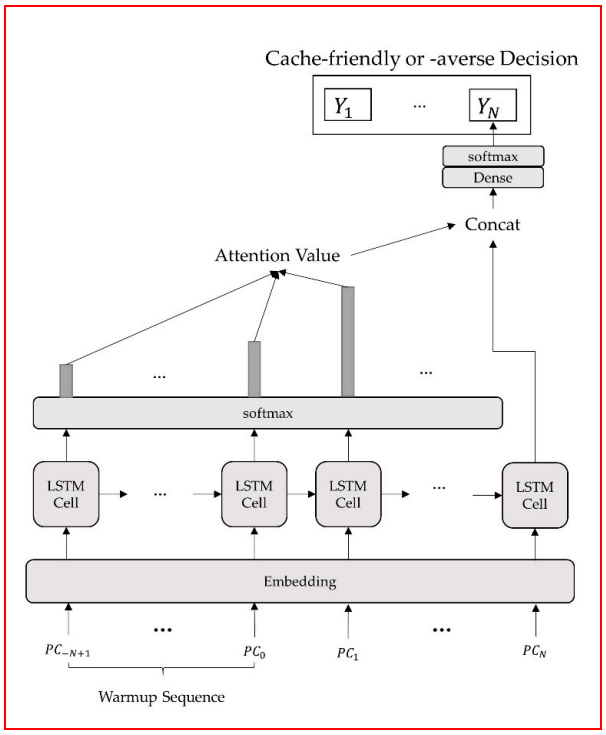


Figure 13. Attention-based LSTM Model [15]: LSTM received a sequence of PC recurrently as an input and predicts the cache priority for current PC.

The second model, Glider simplifies feature by designing the PC sequence as a k-sparse binary to simplify the model as shown in Figure 14. This feature has the advantage of not having to learn the order of the sequences. Glider consists of a PC History Register (PCHR) and an ISVM table. PCHR keeps the last 5 PCs seen from each core in any order, and is modeled as an LRU cache to keep the most recent 5 PCs. ISVM stores the weight of each PC and consists of 16 weights. When the current PC is accessed, it creates an index by hashing the current PC and finds the corresponding ISVM entry in the ISVM table. And, based on the indexed ISVM and the PC existing in the current PCHR, a 4-bit hash is created to find the five weights. Five weights are extracted through five indexes. The output is calculated by adding the extracted five weights. If the output is above the threshold, the model predicts the block as cache-friendly. In this case, it is inserted with a high-priority. If the output is less than 0, the model predicts the block as cache-averse. In this case, the block is inserted with low-priority. If the output is between 0 and the threshold, the model predicts the block as cache-friendly, but it is inserted with a mid-priority.

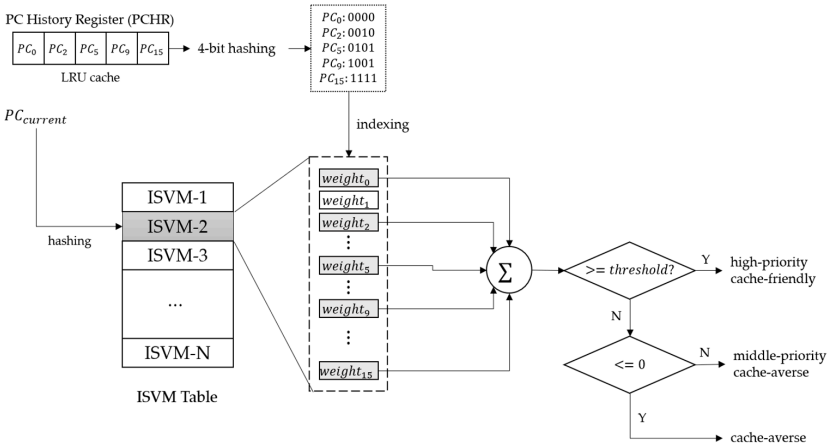


Figure 14. The Glider Predictor [15]: Glider selects the ISVM as a currently accessed PC and determines the cache priority based on the sum of the weights associated with the previously accessed PC.

When evaluating the online model, Glider showed an average accuracy improvement of 88.8 vs. 84.9% compared to the latest online model Hawkeye.

4. Evaluation and Design Consideration

This section compares the learned models explained in Section 2 in various perspectives and explores the design considerations.

4.1. Evaluation

Based on the above-mentioned studies, we evaluate the characteristics that can be observed by dividing into three. First, the machine learning model must be able to generalize not only the training data but also various data. Therefore, when applying machine learning to an existing system, we evaluate the generalization ability of a machine learning model. Second, we compares the characteristics found by applying machine learning to the existing computer system with the characteristics of the existing system. Finally, we compare the characteristics of each machine learning model applied to the studies mentioned above.

4.1.1. Data Distribution

All the above models explain in Section 2 show that they obtain a lot of benefits by learning the distribution of the data. However, since these models are supervised learning models, they rely on the distribution of the training data.

We can compare NN-sort with std::sort by data distribution. For example, NN-Sort trains well by uniform distribution data. When the target data set contains regular distribution data more than 45%, we can see that std::sort shows better performance than NN-Sort since NN-sort was trained by uniform distribution data [10].

The perceptron-based branch predictor is a model in which perceptron trains a linear relationship. When prediction data is non-linear, gshare [20] that can learn non-linear distribution by applying address and history to XOR gate showed better performance than perceptron-based branch predictor.

4.1.2. Machine Learning Model and Traditional Model

Based on the reviewed papers, the advantages and disadvantages that can be identified when replacing the traditional model with machine learning are summarized. Compared to the traditional B-Tree, Learned B-Tree showed improved results that were up to 1.5~3× faster and up to 2× less spatially [7], but since Learned B-Tree is trained with training data, it works well only in read operations. Therefore, when a new input comes in, it is necessary to retrain Learning B-Tree. This means that the trained model does not reflect the

diversity of data. Two Level Prefetcher [13] uses perceptron to deny unnecessary proposals from traditional prefetcher. Although Two Level Prefetcher increased accuracy by denying unnecessary decisions, it reduced the cache hit rate slightly because the perceptron can deny the required requests, and the prefetch may not be done during the perceptron training. Learned Prefetcher [14] designed an LSTM-based prefetcher and compared to two hardware prefetchers for experimentation. In various application traces, the LSTM-based prefetchers showed high accuracy and recall compared to two traditional prefetchers, but there were no usability experiments to see if the training model could learn different types of traces. In view of this aspect, machine learning shows an improvement in accuracy when compared to traditional models, but machine learning is limited by the training data and may not be able to guarantee timeliness due to the training time. Table 2 summarizes two pros and cons.

Table 2. Difference between machine learning model and traditional model.

	Advantage	Disadvantage
Machine Learning Model	(1) Can learn various data distribution according to the complexity or characteristics of the model.	(1) Results depends on training datasets (2) It takes time to generate a prediction model.
Traditional Model	(1) Fast result (The model is already implemented, and results are quickly out). (2) Not sensitive to data distribution.	(1) The model usually shows a consistent result, but only a certain class shows better performance.

4.1.3. Perceptron and LSTM

Perceptron and LSTM are used in the reviewed papers. It is difficult to find a suitable network structure for the technology, but it is important to select a suitable neural network. Based on the paper introduced in Section 2, Table 3 summarizes the characteristics, advantages, and disadvantages of the two models. This is assumed that all models are computed in a single layer. The model parameter means a parameter whose value changes during training, such as weight. Operation refers to the computation method of neural networks.

Table 3. Properties and Advantages and Disadvantages of Perceptron and LSTM.

		Perceptron	LSTM
Model parameter	Properties	Proportional to the number of inputs.	Parameters exist for each gate as well as input.
	Advantage	• Fast training • Small model size	The relationship of inputs can be saved in more detail.
	Disadvantage	Difficult to remember complex distribution.	• Model size increases. • Slow training.
Operation	Properties	Feed forward	Hidden layer is used as input again.
	Advantage	• Simple operation • Back-propagation algorithm isn't needed to update the weight.	Can take advantage of the previous long history.
	Disadvantage	It is not remembering long-term history because it is only affected by the current input.	Not suitable for time-variant system.
Offline& Online learning	Properties	Offline & Online learning are available	Offline learning is available
	Advantage	Even at runtime, the model can be flexibly changed according to the influx of workload.	Can give a variety of models and features.
	Disadvantage	Online learning requires limitations (storage space, data operation), so a less complex model is required.	When limited resources are given, learning becomes impossible if the amount of data increases.

4.2. Design Considerations

To design a new model with machine learning, the researcher needs to be able to think of a new way of thinking and learning. At this time, the effort of the engineer is required. This section describes design considerations by comparing with reviewed papers. We describe training and prediction for design considerations. Table 4 summarizes the reviewed papers.

Table 4. Design Summary of the Reviewed Papers.

	Summary	Input	Output	Learning Method	Prediction
Learned index [7]	Learn the index structure for efficient range request	Key	Regression (key's position)	Offline learning	RMI network + B-Tree
	Learn hash-function for efficient key lookup				RMI network
	Learn structure to learn whether a key is recorded		Classification (key's existence (0/1))		RNN + overflow Bloom filter
Pavo [8]	Learn hash-function for efficient indexing in space and speed.	Word to be used as a key	1. Regression (supervised learning) 2. Classification (unsupervised learning)	Offline learning	LSTM + fully connected layer
Learned branch predictor [9]	Learn a model that replaces a 2-bit counter in a two-level scheme	Global History Register	Classification (taken or non-taken)	Online learning (Adjust weights according to loss of predictions and actual results)	Perceptron + gshare/perceptron
NN-sort [10]	Data distribution-aware sorting algorithm	Data elements	Regression (Return position)	Offline learning	Multi-layer NN + Quick sort
MER-sort [11]	Learn Radix sort-based sorting algorithm, which is not sensitive to data	Data elements	Regression (Return position)	Offline learning	Perceptron + insert sort
Reuse predictor [12]	Determine if the access block is reusable or not	• PC • memory addr • compressed data time/reference count	Classification (Reuse or not)	Online learning (Learning by inputting part of the access to a sampler that has a partial set of cache)	Perceptron
Cache prefetcher [13]	Learning-based two-level scheme to reduce prefetching of unnecessary requests.	• prefetch distance • transition probability • address delta • frequency	Classification (Accept or Deny)	Online learning	Two-level predictor (traditional prefetcher + perceptron)
Learning memory access patterns [14]	Learn prefetcher to analyze memory access pattern	• PC • address delta	Classification (Delta with the highest probability)	Offline learning (Vocabulary generation during training for classification)	(1) Embedding LSTM (2) Clustering + LSTM

Table 4. *Cont.*

	Summary	Input	Output	Learning Method	Prediction
Glider [15]	Learn hardware cache replacement policy	• attention-based LSTM: PC sequencer • Glider: unordered unique PC sequence	Classification (cache-friendly or not)	Offline learning (attention-based LSTM, ISVM) Online learning (ISVM)	(1) attention-based LSTM (2) ISVM

4.2.1. Training

Input. Machine learning can use several features to learn patterns in data. Reuse predictor uses features associated with block access, such as certain bits of memory address, to correlate with adjacent blocks, rather than using only Program Counter (PC) that generated the current eviction [12]. Two Level Prefetcher [13] features prefetch distance, number of occurrences, address delta, etc. to prevent cache pollution. Learned Prefetcher [14] viewed memory access as a sequential trace and used past PC sequence and delta sequence as inputs. However, no matter how many features the engineer chooses, the model may fall short of the expected value. In addition, as more features are required to be implemented in real hardware, space for storage is required, and time for computation is required, so the researcher should balance between hardware overhead and accuracy. There are three feature selection methods to select meaningful data [32]. First, the filter is a method that selects a subset of variables in pre-processing as independent of the model. Second, the wrapper looks at the learning model as a black box and scores a subset of variables for performance measurement. Finally, the embedded method performs a variable selection in the training process. Also, by analyzing the meaning between the memory access pattern and natural language, such as Learning access pattern and attention-based LSTM [14,15], the model input data is analyzed and modeled by NLP (Natural Language Processing). This means that the use of domain knowledge that can interpret not only natural language but also programming language with NLP plays an important role in the interpretation of the ML model to apply ML to the system.

Regression & Classification. A supervised learning model needs to understand whether the problem is a classification or a regression before selecting. Briefly speaking, classification is the task of predicting one of the pre-defined class labels, and regression is the task of outputting a series of values.

Sorting algorithm can be thought of as regression because the algorithm is a task that predicts a continuous position. Branch prediction can be thought of as a classification because two classes are pre-defined whether taken or non-taken a branch when a branch address comes in. Prefetching can be thought of as regression because prefetcher predicts future memory access. However, since address space has a wide range of features when considered as a regression, there is a way to divide address space into a classification problem [14]. They also described a paper designing prefetching as classification problem that determines whether taken or non-taken prefetching as the auxiliary structure of prefetching [13].

Online & Offline Learning. The model can learn online or offline. Online learning is required to gradually learn model as the system changes, and offline learning is to learn the model only once and apply the model in the system. Online learning is proper for cache tasks with limited computing resources. This is because the cache needs to be adjusted for the condition of the hit or miss. Glider [15] selected perceptron that consumes fewer computing resources for cache optimization. Additional feedback is needed to learn online. The best way is to keep track of the entry in the long term, but this is not possible because it has time and space overhead. Perceptron-based Prefetching [13] uses a cache access interval to tread as an incorrect prediction if an entry in the accept table is not referenced within a certain reference period and to read as a correct prediction if an entry in deny table

is not referenced within a certain reference period. Offline learning is properly for database systems. This is because already stored data need only learn the correlation of the data. However, re-training is required to insert or delete tasks.

Consequently, online learning generally uses SVM and shallow artificial neural network (ANN), and offline learning shows the diversity of models. This is because online learning requires only limited data availability and space in order to have low complexity. However, offline learning requires a lot of data and overhead for model accuracy and generalization.

4.2.2. Prediction

Since machine learning is applied to improve performance, prediction accuracy must be guaranteed. Some paper claims that replacing the traditional technique requires auxiliary assistance of traditional technique as well as machine learning. Branch predictor [9] replaces PHT (Pattern History Table) and maintains a global history register (GHR) to have the same architecture as the traditional two-lever scheme. In the learned index [7], the learned B-Tree is supported by traditional B-Tree according to min/max-error in the last mile. The learned Bloom filter adds an overflow Bloom filter to guarantee zero of false-negative rate (FNR) in the binary classification task. NN-sort [10] uses a traditional sorting algorithm such as Quick Sort in Polish Phase to guarantee accuracy. MER-sort [11] uses an insert sort algorithm to merge Spill array and Output array. Instead of replacing all existing techniques to provide guarantees similar to the specific functions of existing techniques and to increase predictive accuracy, the engineer can make a hybrid model, which mixed the existing techniques and machine learning model.

5. Discussion

New ML schemes. Despite the achievements through the ML-based system design, a new ML-based system is still needed. The reviewed papers [7,10,11,13] proposed a combined method of traditional algorithm and machine learning-based approach. This approach improves performance by addressing the pros and cons between machine learning and traditional techniques. However, the combination with traditional techniques has a fundamental limitation of traditional techniques. Recently, there is an approach to approximating Belady′s MIN without a heuristic combination in cache replacement studies [33]. In addition, we improved the accuracy of the model by proposing a method that hierarchically combines multiple models rather than a single model [7,8,10,11]. Another promising approach that can be seen through this design is hierarchical reinforcement learning (RL) [34], a method of hierarchically learning the goals by setting multiple goals. The overhead of a single model can be reduced because the behavior can be coordinated between layers.

Scalability & New Application. The ML-based system must be applicable to the existing architecture and new systems and needs to be continuously developed. Some designs are limited to specific workloads, and generalization can be compromised when faced with new workloads. For that reason, new ML learning methods can be used. For example, LSTM shows good performance in a lot of labeled data, but because it requires a lot of iteration for optimization, it is difficult to optimize if only a few labeled examples are used for learning [35]. Therefore, the proposed method for model generalization with a few data is Meta-learning [36], and it is possible to adapt to new tasks or environments not encountered during fast weight update and training time.

Implementation Improvement. An efficient strategy is needed for an ML-based system to be actually implemented with acceptable overhead. In terms of the model, Glider [15] reduced the hardware budget by removing redundant PCs and proposing an integer value ISVM. Perceptron Reuse Prediction [12] reduces the memory overhead for the training dataset by applying the concept of a sampler that stores separate cache metadata. In addition, promising methods that can be proposed are network pruning, memory sharing method [37], and 16-bit float representation [38], which can reduce the

number of operations and the model size. DNN acceleration can be used to improve computational performance versus power consumption. Recently, there is a DNN inference acceleration method that caches the output of the DNN's hidden layer and consumes only the computation required for inference [39].

This paper surveys recent papers applying machine learning to optimize various computer system. We evaluated the machine learning applied system and analyzed the machine learning design method in several aspects. Most reviewed papers use a well-known neural network like perceptron, LSTM. By just using a simple model, the proposed methods show better performance than traditional techniques. Of course, machine learning techniques are not a panacea. They face its own problems such as over-fitting, scalability, calculation overhead, and memory footprint, etc. However, in order to overcome these problems, a more practical implementation is possible while proposing quantization, pruning, and inference acceleration methods. Therefore, as the network introduced in this paper, as well as the more expressive and complex model, is sought, a wide range of applications will be possible. These optimization opportunities will act as a positive feedback loop between ML and system design as computer architecture/system advances for ML acceleration and ML for computer architecture/system optimization are pursed. In a near future, we believe that innovative machine learning-based system optimization techniques will appear.

Author Contributions: Conceptualization, H.C.; writing—original draft preparation, H.C.; supervision and review, S.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2019R1G1A1100305).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Lee, D.; Noh, S.H.; Mim, S.L.; Cho, Y. LRFU: A block replacement policy which exploits systems and theory. *IEEE Trans. Comput.* **2001**, *50*, 1352–1361.
2. Nesbit, K.J.; Smith, J.E. Data cache prefetching using a global history buffer. In Proceedings of the Tenth Symposium on High-Performance Computer Architecture, Madrid, Spain, 14–18 February 2004.
3. Musser, D.R. Introspective Sorting and Selection Algorithms. *Softw. Pract. Exp.* **1997**, *27*, 983–993. [CrossRef]
4. Fu, J.W.; Patel, J.H.; Janssens, B.L. Stride directed prefetching in scalar processors. *ACM SIGMICRO Newsl.* **1992**, *23*, 102–110. [CrossRef]
5. Kim, J.; Pugsley, S.H.; Gratz, P.V.; Reddy, A.N.; Wilkerson, C.; Chishti, Z. Path confidence based lookahead prefetching. In Proceedings of the 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Taipei, Taiwan, 15–19 October 2016.
6. Hochreiter, S.; Schmidhuber, J. Long short-term memory. *Neural Comput.* **1997**, *9*, 1735–1780. [CrossRef] [PubMed]
7. Kraska, T.; Beutel, A.; Chi, E.H.; Dean, J.; Polyzotis, N. The case for learned index structures. In Proceedings of the International Conferernce on Management of Data, Houston, TX, USA, 10–15 June 2018; pp. 489–504. [CrossRef]
8. Xiang, W.; Zhang, H.; Cui, R.; Chu, X.; Li, K.; Zhou, W. Pavo: A RNN-Based Learned Inverted Index, Supervised or Unsupervised? *IEEE Access* **2019**, *7*, 293–303. [CrossRef]
9. Jimnez, D.A.; Lin, C. Dynamic branch prediction with perceptrons. In Proceedings of the HPCA Seventh International Symposium on High-Performance Computer Architecture, Monterrey, Mexico, 19–24 January 2001; pp. 197–206.
10. Zhu, X.; Cheng, T.; Zhang, Q.; Liu, L.; He, J.; Yao, S.; Zhou, W. NN-sort: Neural Network based Data Distribution-aware Sorting. *arXiv* **2019**, arXiv:1907.08817.
11. Kristo, A.; Vaidya, K.; Çetintemel, U.; Misra, S.; Kraska, T. The case for a learned sorting algorithm. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, Portland, OR, USA, 14–19 June 2020; pp. 1001–1016.
12. Teran, E.; Wang, Z.; Jimnez, D.A. Perceptron learning for reuse prediction. In Proceedings of the 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Taipei, Taiwan, 15–19 October 2016; pp. 1–12.
13. Wang, H.; Luo, Z. Data cache prefetching with perceptron learning. *arXiv* **2017**, arXiv:1712.00905.

14. Hashemi, M.; Swersky, K.; Smith, J.; Ayers, G.; Litz, H.; Chang, J.; Kozyrakis, C.; Ranganathan, P. Learning Memory Access Patterns. In Proceedings of the 35th International Conference on Machine Learning, in PMLR, Stockholmsmässan, Stockholm SWEDEN, 15 July 2018; Volume 80, pp. 1919–1928.
15. Shi, Z.; Huang, X.; Jain, A.; Lin, C. Applying deep learning to the cache replacement problem. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, Columbus, OH, USA, 12–16 October 2019; pp. 413–425.
16. Kraska, T.; Alizadeh, M.; Beutel, A.; Chi, E.; Kristo, A.; Leclerc, G.; Madden, S.; Mao, H.; Nathan, V. SageDB: A learned database system. In Proceedings of the 9th Biennial Conference on Innovative Data Systems Research, CIDR '19, Asilomar, CA, USA, 13–16 January 2019.
17. Wang, J.; Kumar, S.; Chang, S.-F. Semi-supervised hashing for largescale search. *IEEE Trans. Pattern Anal. Mach. Intel.* **2012**, *34*, 2396–2406.
18. Nwankpa, C.; Ijomah, W.; Gachagan, A.; Marshall, S. Activation Functions: Comparison of trends in Practice and Research for Deep Learning. *arXiv* **2018**, arXiv:181103378.
19. Yeh, T.-Y.; Patt, Y. Two-level adaptive branch prediction. In Proceedings of the 24th ACM/IEEE Int'l Symposium on Microarchitecture, Albuquerque, NM, USA, 18–20 September 1991; pp. 51–61.
20. McFarling, S. *Combining Branch Predictors*; Technical Report TN-36; Digital Western Research Laboratory: Palo Alto, CA, USA, June 1993.
21. Lee, C.-C.; Chen, C.C.; Mudge, T.N. The bi-mode branch predictor. In Proceedings of the 30th Annual International Symposium on Microarchitecture, Research Triangle Park, NC, USA, 3 December 1997.
22. C++ Resources Network. Available online: http://www.cplusplus.com/ (accessed on 3 April 2021).
23. Redis. Redis is an Open Source (BSD Licensed), In-Memory Data Structure Store, Used as a Database, Cache and Message Broker. Available online: https://redis.io/ (accessed on 10 May 2009).
24. Andersson, A.; Hagerup, T.; Nilsson, S.; Raman, R. Sorting in linear time? In Proceedings of the 27th Annual ACM Symposium on the Theory of Computing, Las Vegas, NV, USA, 29 May–1 June 1995; pp. 427–436.
25. Manegold, S.; Boncz, P.A.; Kersten, M.L. Optimizing database architecture for the new bottleneck: Memory access. *VLDB J.* **2000**, *9*, 231–246. [CrossRef]
26. Tarjan, D.; Skadron, K. Merging path and gshare indexing in perceptron branch prediction. *ACM Trans. Archit. Code Optim.* **2005**, *2*, 280–300. [CrossRef]
27. Khan, S.M.; Tian, Y.; Jim´enez, D.A. Sampling dead block prediction for last-level caches. In Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, Atlanta, GA, USA, 4–8 December 2010; pp. 175–186.
28. Wu, C.-J.; Jaleel, A.; Hasenplaugh, W.; Martonosi, M.; Steely, J.S.C.; Emer, J. SHiP: Signature-based hit predictor for high performance caching. In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-44, New York, NY, USA, 3–7 December 2011; pp. 430–441.
29. Joseph, D.; Grunwald, D. Prefetching using markov predictors. *ACM SIGARCH Comput. Archit. News* **1997**, *25*, 252–263. [CrossRef]
30. Wulf, W.; McKee, S. Hitting the wall: Implications of the obvious. *ACM SIGARCH Comput. Archit. News* **1995**, *23*, 20–24. [CrossRef]
31. Jain, A.; Lin, C. Back to the future: Leveraging Belady's algorithm for improved cache replacement. In Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA), Seoul, Korea, 18–22 June 2016; pp. 78–89.
32. Guyon, I.; Elisseeff, A. An introduction to variable and feature selection. *J. Mach. Learn. Res.* **2003**, *3*, 1157–1182.
33. Song, Z.; Berger, D.S.; Li, K.; Lloyd, W. Learning relaxed belady for content distribution network caching. In Proceedings of the 17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20), Santa Clara, CA, USA, 25–27 February 2020; pp. 529–544.
34. Kulkarni, T.D.; Narasimhan, K.R.; Saeedi, A.; Tenenbaum, J.B. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. *arXiv* **2016**, arXiv:1604.06057.
35. Ravi, S.; Larochelle, H. Optimization as a model for few-shot learning. In Proceedings of the 5th International Conference on Learning Representations (ICLR), Toulon, France, 24–26 April 2017; pp. 1–11.
36. Hospedales, T.; Antoniou, A.; Micaelli, p.; Storkey, A. Meta-learning in neural networks: A survey. *arXiv* **2020**, arXiv:2004.05439.
37. Chen, T.; Li, M.; Li, Y.; Lin, M.; Wang, N.; Wang, M.; Xiao, T.; Xu, B.; Zhang, C.; Zhang, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv* **2015**, arXiv:1512.01274.
38. Courbariaux, M.; Bengio, Y.; David, J.-P. Low precision arithmetic for deep learning. *arXiv* **2014**, arXiv:1412.7024.
39. Balasubramanian, A.; Kumar, A.; Liu, Y.; Cao, H.; Venkataraman, S.; Akella, A. Accelerating Deep Learning Inference via Learned Caches. *arXiv* **2021**, arXiv:2101.07344.