SmartIndex: Learning to Index Caches to Improve Performance

Kevin Weston, Farabi Mahmud, Vahid Janfaza, Abdullah Muzahid, *Member, IEEE*

Abstract—Modern computers rely heavily on caches to achieve higher performance. Unfortunately, a cache indexing scheme can often cause an uneven distribution of addresses across cache sets resulting in many evictions of useful cache blocks. To address this issue, we propose SmartIndex, a self-optimized indexing scheme that leverages machine learning to actively learn the memory access pattern and dynamically adjust indexes to evenly distribute the cache lines across all sets in the cache, thereby reducing cache misses. Experimental results on a set of 26 memory-intensive applications show that for non-uniform applications, SmartIndex can reduce the misses per kilo instructions (MPKI) of a direct mapped cache by up to 39%, translating into an IPC speedup of 7.23% compared to the conventional power-of-two indexing scheme. Our experiments also show that SmartIndex can work with any cache associativity.

Index Terms—Cache, Machine learning, Indexing, Conflict miss.

1 Introduction

1.1 Motivation and Challenges

Cache memory is designed to hold frequently used memory references for fast accessing, bridging the gap between processor speed and main memory latency. A well-designed cache should have lower miss rate and latency. Intuitively, *conflict misses are caused by the poor indexing scheme in which addresses with incompatible access patterns are mapped to the same set.* The conventional *power-of-two modulo* function, while being easy to implement, is susceptible to cache conflicts when the memory addresses get unevenly distributed across different sets. Even more, sophisticated indexing scheme in modern caches often leads to many conflict misses in memory intensive applications [8], [10], [18]. An ideal indexing function for the cache should evenly distribute the memory references across all cache sets. We aim at finding such a function using Machine Learning (ML). Although there are existing works that use ML for improving cache performance [4], [6], [7], [12], [14], [15], [17], this is the *first* work to do it for cache indexing. It entails several challenges:

- Deterministic Index: A naive design approach would be to replace the index function with an ML model which takes a memory address and few other features (such as instruction, execution context, etc.) and produces the set index bits. However, such a design would make indexing non-deterministic i.e., the same address may result in a different set index when the features change. Such a non-deterministic index will make cache implementation complex and error-prone.

- Simplicity vs. Flexibility: Index calculation is in the critical path of cache accesses. Therefore, the indexing function should be simple to calculate. At the same time, it should be flexible enough for the ML model to tune.

- Lack of Training Labels: It is difficult to determine the training labels for input addresses, since there is no effective way to determine the optimal index for each individual address. Additionally, since the behavior of different programs can vary wildly from one another, an identical memory address of two different programs may have two different optimal indices. Therefore, we need a feedback mechanism for the ML model.

1.2 Our Solution

We propose SmartIndex, the *first* cache indexing scheme that uses an ML model to learn and infer optimal cache mapping of memory addresses. It models cache indexing as a function $f(addr, t) = index$, where $f(.)$ is a simple function consisting of boolean operations on the inputs, $index$ is the set index for an address $addr$ and t is a parameter tuned (through learning and inference) by an ML model. This formulation addresses the first two challenges in Section 1.1. Due to this formulation, the goal of the ML model boils down to optimizing t to reduce conflict misses. Finally, to address the lack of training labels (i.e., the third challenge), we choose a *Deep Reinforcement Learning* (DRL) model as our choice of ML for SmartIndex. DRL does not require any labeled data. Instead, it optimizes t based on the feedback (e.g., whether the conflict misses are reducing). Combining these components, SmartIndex works in the following way. During an execution, SmartIndex collects addresses' frequency and cache-related statistics at regular intervals. The frequency numbers provide insight into the memory access pattern of the program during the interval. The cache statistics, including usage, number of replacements, etc. represent the cache state at that time. From these two pieces of information, the DRL model determines the optimal t for the next interval. During the next interval, indices are calculated using $f(.)$ and the inferred t. DRL model receives positive or negative feedback on its choice of t based on whether the cache misses are reduced in that interval. Thus, DRL eventually predicts optimal address mapping for every interval.

In summary, we make the following contributions:

- We propose SmartIndex, the *first* ML-based cache indexing scheme that can work with any cache of any general-purpose computing systems. SmartIndex can learn and predict the optimal cache mapping during the execution of a program.
- We formulate cache indexing as an optimization problem for a tunable parameter t. We demonstrate how a DRL model can optimize t to provide optimal cache index bits.
- We show an efficient hardware for SmartIndex.
- We evaluate the efficacy of SmartIndex with 26 memory-intensive applications from the SPEC 2006 and SPEC 2017 benchmark suites. For non-uniform applications, our results indicate that SmartIndex can reduce the MPKI of a direct mapped cache by up to 39%, translating into an IPC improvement of 7.23% compared to the conventional power-of-two modulo index. Compared to prior schemes such as prime modulo [8] and CEASER [10], it improves the IPC by 3.1% and 5.0%, respectively. The results hold for higher associative caches as well.

2 Background

Prior work in this area can be divided into 2 categories [11]. Techniques in the first group focus on optimizing the hash function to achieve a more even access distribution across cache sets. Our work falls into this category. Techniques in the second group try to increase the number of possible locations to allocate a cache block by using extra hardware or extending the concept of cache associativity.

Optimizing the Hashing Function: Research in this category proposes more advanced hashing functions to replace the conventional power-of-two modulo indexing. These complex hashing functions have been used at the last level cache in some commercial processors such as the Intel i7-2600, Intel Xeon E5-2640 v2, UltraSPARC T2 [5], [8]. In some cases, these complex hashing functions may require more area and power [5]. Prime modulo indexing scheme uses the prime number nearest to the number of cache sets as the modulus [8]. For example, if the number of sets is 2048, the index of an address a is computed as $a\ mod$ 2039, since 2039 is the prime number that is closest to 2048. This hash function has been shown to be better than the conventional power-of-two modulo hashing [8]. CEASER [10] randomly changes the cache mapping during program's execution to prevent conflict-based cache attacks. Physical addresses going to the shared cache are encrypted before being used to index into the cache. The encrypted key is changed periodically to avoid the eviction patterns to be learned by adversaries.

Increasing the Number of Possible Locations: Skewed-associative cache uses a different hash function for each cache way to achieve a higher utilization rate and lower miss rate [2], [13]. Zcache extends skewed cache to supports conventional replacement policies [11]. On a hit, zcache operates similar to skewed cache. On a miss, zcache first walks the tag array to identify a set of eviction candidates, then uses the replacement policy to select the optimal one.

3 Main Idea: SmartIndex

3.1 Overview

Cache indices are calculated using an index function $f(addr, t) = I_{addr} \oplus M_t$, where I_{addr} is the index calculated for the address $addr$ using a modulo operation (i.e., $I_{addr} = addr\ \%\ Number\ of\ cache\ sets$) and M_t is a bit mask chosen using a tunable parameter t. SmartIndex augments each cache set with a counter called *Access Counter* (AC). This counter keeps track of the number of set accesses during an interval (e.g., every one million cache accesses). AC of a cache set is incremented during each access to that set and cleared after an interval elapses. At the end of an interval, ACs of all cache sets are aggregated and supplied to a DRL model as an input. This is shown in Figure 1. The model predicts some value for t. The new value of t is used to calculate cache indices starting in the next interval. SmartIndex also keeps track of cache misses during an interval. If the a particular choice of t results in a lesser number of cache misses at the end of an interval (where the particular t is used), the DRL model receives positive feedback. Otherwise, the model receives negative feedback. Based on the feedback, the DRL model is updated so that it can predict a better choice of t. As this process continues, the DRL model keeps learning and eventually, will predict the optimal t for each interval, thereby reducing the number of cache misses.

3.2 Tunable Bitmask Formulation

SmartIndex predicts a tunable parameter t which in turn chooses a bitmask M_t. By changing t, SmartIndex can effectively change the index function. M_t comprises of bits taken from the address $addr$. Let us consider a 2MB direct mapped cache with 64-bit address space and 6-bit offset. Since the total number of sets is 2^{15}, we need 15-bit index. That means the bitmask size is also 15. Since the offset is 6 bits long, the total number of bits that are available to choose from for the bitmask is 43, from bit $21st$ to $63rd$. Thus, the total number of possible bitmasks is $\binom{43}{15}$. Such a vast search space may slow down the training time significantly and negatively affect the scalability of SmartIndex. Thus, we make two modifications to limit the search space. First, we only consider up to the 47 least significant bits, with the intuition that higher-order bits rarely change, and hence, will not be useful in the address redistribution. As a result, the number of available bits to select is limited to 26, from bit $21st$ to bit $46th$. Secondly, instead of selecting each bit to form the bitmask, SmartIndex selects a sequence of consecutive bits, whose length is equal to the index bit size. For example, if the index is 15 bits long, then the first sequence is from bit $21st$ to $35th$, the second sequence is from bit $22nd$ to $36th$, and so on. The last possible sequence is from bit $32nd$ to $46th$. This results in 12 sequences in total. We then reverse these sequences to get another set of 12 sequences and add an *all-zero* sequence, which means no XOR-hashing. This is the default configuration used at the beginning of the execution. These modifications reduce the possible combinations to 25. Thus, t can be any integer value from 0 to 24. Each choice of t represents a particular bit sequence. This sequence is extracted from the address $addr$ to form the bitmask M_t.

Example: Let us consider an address $addr$ = 0xA0DF21F1. For our example 2MB direct mapped cache with 6-bit offset, I_{addr} = 0x7C87. Suppose, SmartIndex predicts $t = 1$, which chooses the sequence of bits from $22nd$ to $36th$ from $addr$ to form the bitmask. As a result, M_t = 0x60A0. Hence, the set index $f(addr = 0xA0DF21F1, t = 1)$ = 0x1C27.

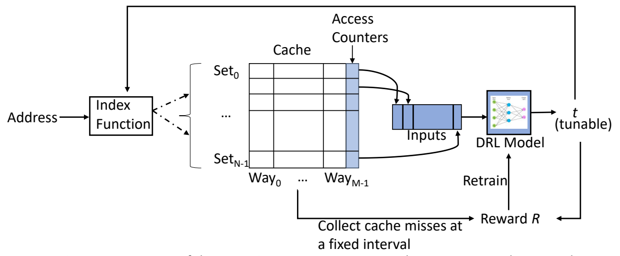


Fig. 1: Overview of how SmartIndex learns cache indexing.

3.3 Learning Formulation

To predict the optimal value of t, SmartIndex uses a DRL model, more specifically the *Deep Q Learning* (DQN) [3]. We formulate the cache indexing as a Markov Decision Process where the next state solely depends on the present state. Let us assume that the cache is in a specific state. The DQN model predicts the best possible t, moving the cache to the next state. The reward is then computed based on the miss rate of the new indices. Using this reward, the Q-value of the action is updated and the neural network model of DQN is retrained. We define the *state*, *action*, and *reward* as follows.

The ***State*** is defined as the access distribution of the cache at a given time. This cache state is represented by a vector, whose each element shows the standardized deviation of one set access against the overall mean cache access. The state vector size is equal to the number of cache sets. The ***Reward*** is computed as the negation of the cache miss rate of that interval. For example, if SmartIndex applies $t = t_i$ for interval i and observes a miss rate of 0.72%, then the reward r_i is -0.72. Intuitively, the higher the miss rate, the lower the reward. The ***Action*** is defined based on the values possible for t. For example, in the case of a 2MB cache with 6-bit offset (Section 3.2), t is an integer from 0 to 24. Hence, there will be 25 actions - one for each value of t. Finally, to speed up the learning speed, SMARTINDEX is integrated with two improvements: experience replay and iterative target update [9].

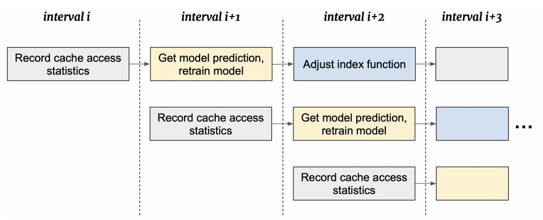

interval i
interval i+1
interval i+2
interval i+3
Record cache access statistics
Get model prediction, retrain model
Adjust index function
Record cache access statistics
Get model prediction, retrain model
...
Record cache access statistics

Fig. 2: Timeline of prediction and training for SMARTINDEX.

Figure 2 shows the general timeline of SMARTINDEX. At the beginning, SMARTINDEX starts with the traditional power-of-two modulo indexing for the cache. After N accesses to the cache (e.g., N could be 1 Million), SMARTINDEX collects the per-set ACs and use them as input to the model. These counters inform the model of the access distribution of the *first* interval. The model processes the input and predicts t for the *third* interval. When the system finishes the *third* interval, it evaluates the effectiveness of the prediction and computes the reward accordingly. This reward is used to retrain the model. Note that the *first* predicted t is applied on the *third* interval. In return, the reward of the *third* interval is used to retrain for the *first* prediction. This is because the model prediction is computed in the background while the system is passing the *second* interval. This strategy helps SMARTINDEX incur near-zero latency for the model. In general, for interval i, SMARTINDEX applies the t predicted at interval $i-2$. At the end of interval i, SMARTINDEX computes the reward and collects the access distribution of i. The access distribution is used to predict t for interval $i+2$. Meanwhile, the reward is used to retrain for the prediction done at interval $i-2$. As a result, the model gradually learns from experience the optimal indices for each execution interval.

When a new mapping due to a new t is applied to the cache at the beginning of an interval, existing valid cache blocks need to be moved to their new locations to keep them consistent with the new mapping. This transition must be handled carefully in order to avoid excessive data movement. We use the gradual remapping scheme proposed in CEASER [10].

4 IMPLEMENTATION

4.1 The DRL Module

A DRL module contains a neural network which can be implemented as a Multi-Level Perceptron (MLP). Tarsa et al. [16] have shown that an existing microcontroller in modern processors can handle all computations of an MLP in a reasonable time. Therefore, the cache controller would communicate with this microcontroller for inference and training. The microcontroller will handle three types of operations: storing samples, training, and inference. At the end of each interval, the cache controller issues an *inference* command to get the new mapping. In this case, the controller reads the ACs and sends the current state (input) to the microcontroller. The model's predicted t is then sent back and written to a register in the cache controller. When the reward is calculated at the end of an interval, the cache controller issues a *store samples* request and sends the reward and next state to the module. The microcontroller forms a new sample from the current state, next state, prediction and reward and saves it to the replay buffer. DRL model picks samples from this buffer randomly and trains itself.

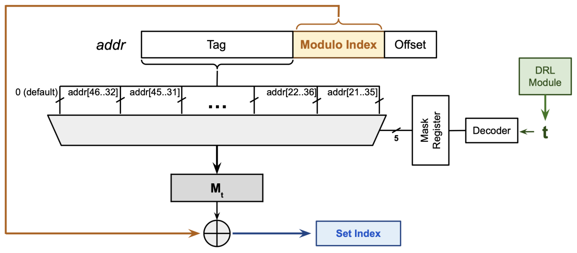


Fig. 3: Hardware implementation of SMARTINDEX.

4.2 Cache Index Resolution Logic

Since the index computation is different than the original, we need a redesigned index resolution logic. The DRL prediction is an integer t which is decoded to some bit sequence and stored in a register. These bits are used as selection bits for the MUXs (Figure 3). The MUXs select appropriate bits and form the bitmask M_t. M_t is XORed with the modulo index to calculate the actual set index. We use the Synopsys Design Compiler NXT to analyze the hardware cost of SMARTINDEX and compare it with the prime modulo indexing scheme, as shown in Table 1. We used 28nm technology. The prime modulo cannot run any faster than 1.20 GHz. Even at this frequency, it takes 2 cycles to compute the index. This is equivalent to a 5-cycle extra latency in LLC access time for our 3.0 GHz simulated hardware. Compared to prime modulo, SMARTINDEX is 5X faster in latency while consuming 40% less power and occupying 18% less area. Since the model training and inference are off the critical path, as discussed in Section 3.3, this index computation latency is the only perceivable latency for SMARTINDEX and will be used in our experiments.

Method	Max Freq. (GHz)	Latency at 3.0 GHz (cycles)	Total Power (mW)	Area
Prime [8]	1.20	5	0.544	852
SMARTINDEX	3.0	1	0.326	697

TABLE 1: Hardware cost of SMARTINDEX and Prime Modulo.

5 EVALUATION

5.1 Experimental Setup

Parameter	Value
Processor	1-core @ 3.0 GHz
L1 cache (I/D)	32KB, 2-way, 2-cycle latency
L2 Cache	128KB, 4-way, 8-cycle latency
LLC	2MB, direct-mapped, 24-cycle latency (baseline)
	2MB, 2-way, 26-cycle latency
	2MB, 4-way, 26-cycle latency
	2MB, 8-way, 26-cycle latency
	2MB, 16-way, 32-cycle latency
DRAM	tRP=tRCD=tCAS=20

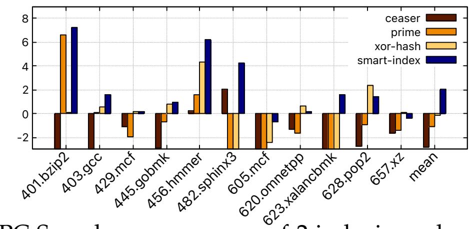
TABLE 2: Parameters of the overall simulated hardware. LLC latency numbers are obtained from CACTI 7.0.

SMARTINDEX is implemented in the LLC using the ChampSim simulator [1]. We use the SPEC 2006 and SPEC 2017 benchmarks, excluding the failed traces and those that have the MPKI less than 1 under the direct-mapped LLC baseline. We categorize the benchmark applications into two groups based on their per-set eviction distribution. An uneven eviction distribution indicates that the cache is experiencing a pathological access pattern. Assume $e_0, e_1, e_2, ..., e_n$ are the number of evictions of the cache sets $s_0, s_1, s_2, ..., s_n$, and $\bar{e}$ is the mean eviction count across all sets. If the ratio $stdev(e_i)/\bar{e}$ of the program is greater than 0.5, it is classified as *non-uniform*. Table 3 presents the complete classification of all applications in this work.

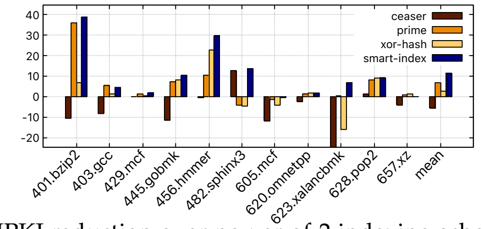
Apps	$stdev(e_i)/\bar{e}$				
	1-way	2-way	4-way	8-way	16-way
401.bzip2	**1.417**	**1.133**	**0.705**	0.184	0.114
403.gcc	**1.668**	**1.377**	**0.997**	0.482	0.170
410.bwaves	0.041	0.021	0.016	0.009	0.006
429.mcf	**0.508**	0.271	0.161	0.114	0.080
436.cactusADM	0.299	0.202	0.142	0.084	0.065
437.leslie3d	0.198	0.088	0.055	0.044	0.036
445.gobmk	**2.419**	0.392	0.116	0.053	0.031
450.soplex	0.436	0.245	0.175	0.100	0.077
456.hmmer	**1.626**	**1.417**	**1.076**	0.445	0.244
459.GemsFDTD	0.055	0.035	0.021	0.011	0.007
470.lbm	0.056	0.022	0.018	0.014	0.013
471.omnetpp	0.395	0.267	0.187	0.131	0.091
473.astar	0.499	0.338	0.238	0.184	0.154
481.wrf	0.150	0.073	0.036	0.022	0.014
482.sphinx3	**0.964**	**0.665**	0.450	0.316	0.138
602.gcc_s	0.293	0.169	0.114	0.064	0.037
605.mcf_s	**0.759**	0.480	0.267	0.146	0.090
607.cactuBSSN_s	0.490	0.122	0.025	0.016	0.013
619.lbm_s	0.045	0.022	0.018	0.015	0.014
620.omnetpp_s	**0.643**	0.292	0.187	0.127	0.091
621.wrf_s	0.302	0.162	0.121	0.095	0.061
623.xalancbmk_s	**9.301**	**8.470**	0.288	0.145	0.093
627.cam4_s	0.382	0.092	0.057	0.032	0.023
628.pop2_s	**1.208**	0.458	0.148	0.049	0.030
649.fotonik3d_s	0.355	0.028	0.007	0.005	0.004
657.xz_s	**0.747**	0.390	0.213	0.142	0.090

TABLE 3: LLC eviction uniformity of all applications. As the associativity increases, the uniformity ratio decreases, since the evictions are more evenly distributed.

For each application, the DQN model is trained for 250 episodes. Each episode is one execution of the program. Table 2 shows the system and LLC configurations. We compared our solution with CEASER [10], prime modulo [8] and the simple XOR-hash [8] schemes. For CEASER, we use the LLC latency of 2 cycles as reported in the paper. For XOR-hash, we use zero latency. For prime modulo and SMARTINDEX, we use the result from our analysis in Section 4.2. We conduct a sensitivity test and find that an interval size of *1 million* cache accesses strikes the optimal balance between the system performance and runtime overhead.



(a) IPC Speedup over power-of-2 indexing scheme (%)

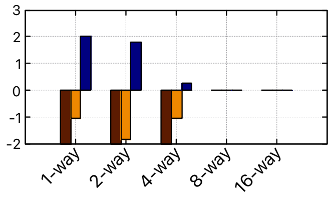


(b) MPKI reduction over power-of-2 indexing scheme (%)

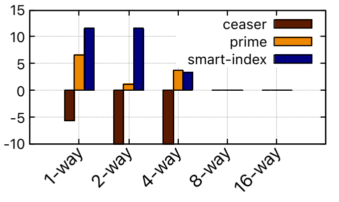
Fig. 4: Performance comparison between SMARTINDEX and other techniques for applications with **non-uniform** cache evictions on **direct-mapped cache**.

5.2 Results

Figure 4 shows the performance results of SMARTINDEX compared to other techniques when applied on a direct-mapped cache. Specifically, SMARTINDEX can reduce the MPKI by 11.52% on average. This reduction amount translates to an IPC increase of up to 7.23%, seen in *401.bzip2*. Compared to other techniques, SMARTINDEX outperforms by a huge margin in every application. CEASER gives the worst performance since the cache remapping is performed randomly. Prime modulo indexing gives a considerable MPKI reduction, but its complex hardware implementation negated all the potential performance benefit. XOR-hash does not have extra latency. But since it does not give significant MPKI reduction as well, the IPC overall impact of this scheme is minimal. Figure 5 shows the performance of different techniques on different cache associativities. In general, SMARTINDEX outperforms other techniques in every associativity. Our solution gives a significant speedup for direct-mapped and 2-way associative caches. The performance improvement decreases when the associativity increases, since the cache evictions are more evenly spread, resulting in less cache conflicts.



(a) IPC Speedup (%).



(b) MPKI reduction (%).

Fig. 5: Performance comparison between indexing techniques on different cache associativity for non-uniform applications. *1-way* is direct mapped cache. 8/16-way columns are empty since there is no non-uniform application (Table 3).

6 CONCLUSION

We propose SMARTINDEX, an adaptive cache index that leverages machine learning to minimize the number of cache conflict misses. Our simulation results show that applying SMARTINDEX can help reduce the MPKI of a direct mapped cache by up to 39%, translating into an IPC speedup of 7.23% compared to the conventional power-of-two indexing scheme.

REFERENCES

[1] CRC-2. (2017) The 2nd cache replacement championship. [Online]. Available: http://crc2.ece.tamu.edu
[2] X. Ding, D. S. Nikolopoulos, S. Jiang, and X. Zhang, "Mesa: reducing cache conflicts by integrating static and run-time methods," in *ISPASS*, 2006.
[3] L. Graesser and W. L. Keng, *Foundations of Deep Reinforcement Learning: Theory and Practice in Python*. Addison-Wesley, 2018.
[4] M. Hashemi, K. Swersky, J. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning memory access patterns," in *ICML 2018*.
[5] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Systematic reverse engineering of cache slice selection in intel processors," in *Euromicro Conference on Digital System Design*, 2015.
[6] D. Jiménez and E. Teran, "Multiperspective reuse prediction," in *MICRO 2017*.
[7] S. Khan, Y. Tian, and D. Jiménez, "Sampling dead block prediction for last-level caches," in *MICRO 2010*.
[8] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee, "Using prime numbers for cache indexing to eliminate conflict misses," in *HPCA 2004*.
[9] V. Mnih, K. Kavukcuoglu, and D. Silver, "Human-level control through deep reinforcement learning," in *Nature*, vol. 518, 2015.
[10] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *MICRO 2018*.
[11] D. Sanchez and C. Kozyrakis, "The zcache: Decoupling ways and associativity," in *MICRO 2010*.
[12] S. Sethumurugan, J. Yin, and J. Sartori, "Designing a cost-effective cache replacement policy using machine learning," in *HPCA 2021*.
[13] A. Seznec, "A case for two-way skewed-associative caches," *ACM SIGARCH Computer Architecture News*, 1993.
[14] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in *MICRO 2019*.
[15] Z. Shi, A. Jain, K. Swersky, M. Hashemi, P. Ranganathan, and C. Lin, "A hierarchical neural model of data prefetching," in *ASPLOS 2021*.
[16] S. J. Tarsa, R. B. R. Chowdhury, J. Sebot, G. Chinya, J. Gaur, K. Sankaranarayanan, C.-K. Lin, R. Chappell, R. Singhal, and H. Wang, "Post-silicon cpu adaptation made practical using machine learning," in *ISCA 2019*.
[17] E. Teran, Z. Wang, and D. Jiménez, "Perceptron learning for reuse prediction," in *MICRO 2016*.
[18] C. Zhang, "Balanced cache: Reducing conflict misses of direct-mapped caches," in *ISCA 2006*.