Customizing Cache Indexing through Entropy Estimation

Kevin Weston, Avery Johnson, Vahid Janfaza, Farabi Mahmud, Abdullah Muzahid
Texas A&M University
College Station, USA
{kevin.weston, averyjohnson, vahidjanfaza, farabi, abdullah.muzahid}@tamu.edu

***Abstract*—Modern computers heavily rely on caches as one of the means to achieve higher performance. As a result, cache management has been the topic of extensive research. Compared to cache replacement and prefetching, cache indexing has received far less interest over the years. Being in the critical path, a good cache index function must exhibit a high performance while having a minimal computational delay. Previous indexing schemes fall short of these requirements, having either moderate performance or a prohibitively expensive delay. We propose ENTROPYINDEX, an entropy-based cache indexing scheme that can deliver superior performance while maintaining a minimal computational cost. ENTROPYINDEX is based on the idea of constructing the index function dynamically at runtime using the address bits with the highest entropy (randomness) to maximize the balance of the cache access distribution. The entropy of the address bits is measured by determining which bits change between two subsequent cache misses. ENTROPYINDEX periodically compares the entropy of different bits and selects the ones that change the most. This dynamic selection scheme allows ENTROPYINDEX to adapt to different types of applications.**

Our experimental results show that ENTROPYINDEX outperforms previous indexing schemes both with and without hardware prefetching. For SPEC 2006, SPEC 2017, PARSEC 3.0 and GAP benchmarks without prefetching, ENTROPYINDEX delivers a geometric mean IPC improvement of 3.39% (with the highest being 52.2%), compared to a 1.74% improvement of the state-of-the-art index function (PRIME) and a 1.76% improvement of a commercialized indexing scheme (XORHASH) over the baseline power-of-two modulo scheme. With prefetching, ENTROPYINDEX is the only indexing scheme with a substantial performance gain of 1.42% (with the highest being 30.1%), compared to a 0.41% improvement of PRIME and a 0.49% improvement of XORHASH over the same baseline. For non-uniform applications and no-prefetching, ENTROPYINDEX gives an IPC speed up of 5.58%, compared to a 2.26% speed up of PRIME and a 2.23% speed up of XORHASH. For non-uniform applications with prefetching, the IPC speed up of ENTROPYINDEX is 2.08%, compared to a 0.35% speed up of PRIME and a 0.53% speed up of XORHASH. For CVP workloads without prefetching, ENTROPYINDEX delivers a speed up of 3.04% over the baseline compared to a 1.52% of PRIME and a 2.04% of XORHASH. For CVP workloads with prefetching, ENTROPYINDEX improves the IPC by 1.60%, compared to 0.63% of PRIME and 1.07% of XORHASH.

***Index Terms*—last level cache, indexing, hashing, entropy.**

I. INTRODUCTION

A. Motivation

Cache memory is designed to hold frequently used memory references for fast accessing, bridging the gap between the processor speed and the main memory latency. As modern programs are becoming increasingly memory intensive, the cache system shows an even greater impact on the overall system performance. To improve the cache performance, the common tendency is to increase the cache size. However, such an approach may not always be effective due to the manufacturing challenges and increased latency associated with the bigger cache. Additionally, many modern data center applications' working set size is now far beyond the LLC capacity [19]. Therefore, it is important to improve the caches through innovative microarchitectural research.

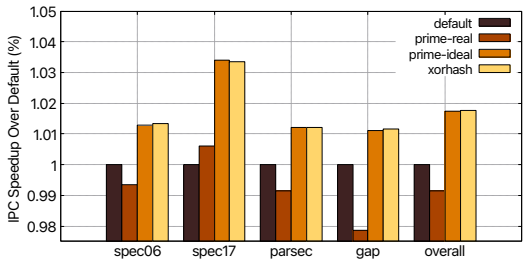
When it comes to cache memory research, the two dominant techniques are replacement [16], [18], [20], [22], [35], [39], [41] and prefetching [11], [12], [17], [24], [26], [27], [30], [37], [43]. Such impressive research efforts in both areas have improved the cache performance significantly, but, at the same time, come closer to their own eventual limit. For instance, recent studies in cache replacement show that they have reached an IPC speed up of 5.7%, being 0.3% away from the 6.0% unrealizable IPC improvement of the oracle replacement policy [38]. Hence, it is crucial to find other avenues to keep pushing the cache performance forward.

Indexing is one of the design aspects of cache memory that has received relatively less attention. Cache indexing is essentially a hashing problem where every memory address is an input and the corresponding set index is the output. A better index function may help lower the number of conflict misses. Unfortunately, prior work in conflict misses reduction mostly focuses on proposing alternative cache organizations, including skewed-associative caches [6], [34], [36], column-associative caches [4], or pseudo-associative caches [3]. Despite being the most commonly used cache organization in modern processors [13], there are limited studies on improving the index function of set-associative cache to reduce conflict misses, especially in recent years. This paper aims to change that by revisiting the indexing problem of set-associative caches.

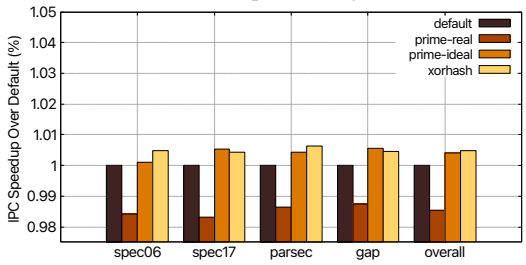
B. Formulation of Cache Index Function

Finding an ideal hash function for set-associative caches is a non-trivial task. A good cache index function must possess two qualities: (1) it must exhibit a high performance against different workloads even under pressure (such as hardware prefetching), and (2) it must have a minimal computation latency. Generally, a high quality hash function should have a more even distribution in its output, thus leading to

fewer cache conflicts (collisions). The conventional *Power-of-2 modulo* function (DEFAULT) has a simple computation, but pathological patterns found in several applications cause a significant increase in hashing collisions [23].



(a) No prefetching



(b) With prefetching

Fig. 1: Geometric mean IPC improvement of various Last Level Cache (LLC) index functions normalized to the baseline (prefetchers used: Next-Line in L1D, Signature Path Prefetching (SPP) [24] in L2). The baseline is *power-of-two modulo* index function (DEFAULT).

Prime-modulo function (PRIME) holds unique mathematical characteristics that help it to have a better output distribution, therefore providing a better performance. In fact, PRIME has been considered to be one of the functions with the lowest number of conflicts [33]. The main problem of PRIME is that it requires complicated computations in hardware. Our analysis using the Synopsys Design Compiler NXT [1] shows that the prime-modulo calculation based on the Polynomial method proposed in [23] requires 5 cycles to complete for a system running at 3.0GHz. To understand the impact of this latency, let us consider Figure 1. The figure shows the performance of different index functions with and without prefetching. Hypothetically, PRIME index with zero latency (PRIME-IDEAL) has better performance than DEFAULT. However, the 5-cycle increased latency eliminates all the potential benefits (PRIME-REAL). Therefore, we set out to find a solution that is capable of providing a high performance hashing capability while being hardware friendly.

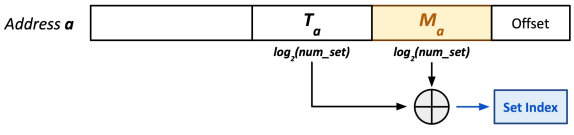
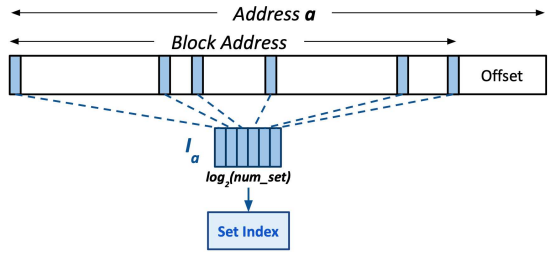


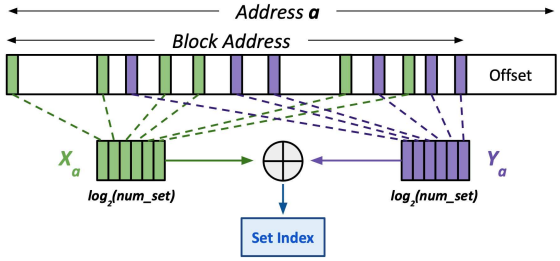
Fig. 2: Example of one of the most commonly used *XOR-hash* indexing schemes [23].

Towards that end, we investigate a family of hash functions based on the XOR operation (XORHASH). XOR-hashing techniques have been extensively studied [6], [32], [42] and implemented in commercial processors [2], [28], [29]. XORHASH is built on the idea that the XOR operation will increase the randomness of the output, leading to a more balanced distribution. XORHASH performs comparable to PRIME-IDEAL (Figure 1) while having a far more efficient hardware implementation (Figure 2).

Inspired by these results, we introduce ENTROPYINDEX, a dynamic indexing scheme based on the idea of maximizing randomness. We hypothesize that *a more balanced distribution of the set indices can be achieved by increasing the randomness of the set index bits.* As a result, at runtime, if we can identify the bits in the cache access addresses with the most random behavior and use them to form the index function, we will get better performance.



(a) Uni-sequence index function formulation



(b) Dual-sequence index function formulation

Fig. 3: Index function formulations for ENTROPYINDEX. Assume I_a is the set index of address a. For the Uni-sequence index function, I_a is the sequence of N highest entropy block address bits. For the Dual-sequence index function, $I_a = X_a \oplus Y_a$ (X_a and Y_a are the sequences of N highest and N next highest entropy bits selected from the block address ($N = log_2(num\_set)$). The Dual-sequence index function results in significantly better performance (Figure 4).

We come up with two generalized index hashing functions under the umbrella of ENTROPYINDEX, as shown in Figure 3. The first index function (Figure 3a) uses the highest random bits from the block address directly as index. We refer to this as the *Uni-sequence Index*. The second function (Figure 3b) performs an XOR operation between the highest (green color) and the next highest (purple color) random bits from the block address to get the final set index. With the use of more bits and an XOR operation, this formulation (referred to as the *Dual-sequence Index*) adds even more randomness to the set index. As a result, it delivers significantly better performance over the Uni-sequence, as shown in Figure 4 (more detail in Section II-A), so we opt to use the Dual-sequence as the final index formulation of ENTROPYINDEX.

C. Maximizing Randomness: ENTROPYINDEX

Entropy is a measure of randomness [44]. In our indexing problem, the most random address bits are those that have the highest entropy. To measure the entropy of individual bits, each address bit is associated with an *Entropy Counter (EC)*. These counters are updated upon every miss to the cache. The address bits of the current cache miss are compared with those of the previous one. The bits that differ between these two addresses exhibit a tendency of being more random than the other bits. Therefore, the EC corresponding to those bits are incremented. ENTROPYINDEX evaluates these counters at regular intervals during the program execution, selects the bits with the highest entropy, and uses them to form the index function for the subsequent interval. When the new index function is installed, the CEASER-based gradual remapping process [31] is triggered to reallocate existing cache lines to their new set indices. Additionally, since these updates and the bit selection processes can be done off the critical path, ENTROPYINDEX does not have any perceivable impact to the cache access latency and cycle time.

Our empirical results indicate that ENTROPYINDEX significantly outperforms previous indexing schemes in both prefetching and no-prefetching scenarios. For SPEC 2006, SPEC 2017, PARSEC 3.0 and GAP benchmarks without prefetching, ENTROPYINDEX provides a geometric mean IPC improvement of 3.39% over DEFAULT (with the highest being 52.2%), compared to a 1.76% improvement of XORHASH and a 1.74% improvement of PRIME-IDEAL. In the presence of hardware prefetching, ENTROPYINDEX delivers a substantial performance gain of 1.42% (with the highest being 30.1%), compared to a 0.49% improvement of XORHASH and a 0.41% improvement of PRIME-IDEAL. For non-uniform applications from the SPEC 2006, SPEC 2017, PARSEC 3.0 and GAP benchmark suites, ENTROPYINDEX gives an IPC speed up of 5.58%, compared to a 2.23% speed up of XORHASH and a 2.26% speed up of PRIME-IDEAL (no-prefetching). With prefetching, these numbers are 2.08% for ENTROPYINDEX, 0.53% for XORHASH and 0.35% for PRIME-IDEAL. For CVP benchmarks without prefetching, ENTROPYINDEX delivers a geometric mean speed up of 3.04%, compared to a 2.04% of XORHASH and a 1.52% of PRIME-IDEAL. with prefetching, ENTROPYINDEX improves the IPC of CVP workloads by 1.60%, compared to 1.07% of XORHASH and 0.63% of PRIME-IDEAL.

We also conduct a head-to-head performance comparison between a 16-way set-associative LLC with ENTROPYINDEX and a 16-way skewed-associative LLC (SKEWEDCACHE). Simulation results show that the set-associative cache with ENTROPYINDEX outperforms SKEWEDCACHE by a substantial margin: 3.39% versus 1.44% IPC improvement over DEFAULT (no-prefetching), and 1.42% versus 0.30% IPC improvement over DEFAULT (with prefetching). Our solution works seamlessly with single and multiprogram workloads.

Compared to previous indexing schemes, ENTROPYINDEX exhibits the ideal qualities for a good index function. First, ENTROPYINDEX provides a consistently high performance against a variety of applications both with and without hardware prefetching. Second, ENTROPYINDEX does not incur extra latency to the cache since all of the counters updating and bit selection computations can be taken off the critical path. ENTROPYINDEX also has a straightforward hardware implementation.

The generalized XOR-based index formulation in Figure 3 also provides us with a systematic approach to find an optimal or near-optimal performing XOR-hashing configuration for each application. Particularly, we use the Genetic Algorithm to search for a near-optimal XOR-hashing configuration for each application (NEAR-OPTIMAL). These NEAR-OPTIMAL index configurations found by the Genetic Algorithm represent an empirical estimation of the overall performance potential of the XOR-based cache indexing schemes.

D. Contributions

In summary, we make the following major contributions:

- We formulate cache indexing as a general XOR-hashing problem. Our formulation provides a systematic approach to find a high performance XOR-hashing configuration for an application dynamically at runtime.
- With the XOR-hashing formulation, we use the Genetic Algorithm to find the NEAR-OPTIMAL index function for each application. These NEAR-OPTIMAL index configurations serve as an empirical estimation of the overall performance gain potential of the XOR-based cache indexing schemes.
- We propose a robust, adaptive cache indexing scheme based on the notion of entropy, called ENTROPYINDEX. At runtime, ENTROPYINDEX tracks the entropy of multiple bits in the addresses of cache accesses, then selects the highest entropy bits to form the index function. This is the *first* work that looks at entropy to design an XOR-based cache indexing scheme. We provide an efficient and detailed implementation of our solution in hardware.
- We provide the first performance analysis of different cache indexing schemes in the presence of hardware prefetching.
- We conduct an in-depth performance analysis of ENTROPYINDEX using various benchmark suites. For SPEC 2006, SPEC 2017, PARSEC 3.0, and GAP benchmarks without prefetching, ENTROPYINDEX delivers a geometric mean IPC improvement of 3.39% over DEFAULT (with the highest being 52.2%), compared to a 1.74% improvement of the state-of-the-art index function (PRIME-IDEAL) and a 1.76% improvement of a commercialized indexing scheme (XORHASH). With prefetching, ENTROPYINDEX delivers a performance gain of 1.42% over DEFAULT (with the highest being 30.1%), compared to a 0.49% improvement of XORHASH and a 0.41% improvement of PRIME-IDEAL. For non-uniform applications and no-prefetching, ENTROPYINDEX provides an IPC speed up of 5.58%, compared to a 2.26% speed up of PRIME-IDEAL and a 2.23% speed up of XORHASH.

For non-uniform applications with prefetching, the IPC speed up of ENTROPYINDEX is 2.08%, compared to a 0.35% speed up of PRIME and a 0.53% speed up of XORHASH. ENTROPYINDEX also outperforms SKEWED-CACHE: 3.39% versus 1.44% IPC improvement over DEFAULT (no-prefetching), and 1.42% versus 0.30% IPC improvement over DEFAULT (with prefetching). For CVP workloads, ENTROPYINDEX delivers a speed up of up to 6.79% without prefetching and 4.13% with prefetching compared to the baseline.

II. MAIN IDEA: ENTROPYINDEX

At the high level, ENTROPYINDEX works by continuously monitoring the entropy of multiple individual address bits during the execution. Periodically, ENTROPYINDEX evaluates the entropy of these bits and selects the highest entropy bits to form the new index function. In this section, we will go through the detailed design of ENTROPYINDEX. Each subsection corresponds to a question about a design choice or trade-off that we made during the design process.

Particularly, Section II-A explains *(1) how* ENTROPYINDEX *keeps track of the entropy of the individual address bits* and *(2) how many bits we should track*. Section II-B demonstrates *(3) how* ENTROPYINDEX *manages the existing cache blocks in the event of an index function change*. Section II-C explains *(4) how* ENTROPYINDEX *minimizes the cost of the index function changes*. We then put them all together with the explanation on the general workflow of ENTROPYINDEX in Section II-D.

Finally, Section II-F gives a detailed demonstration on how we use the Genetic Algorithm to search for the NEAR-OPTIMAL XOR-hashing index configuration for each program.

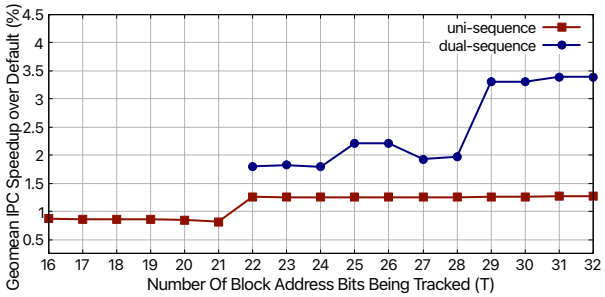


Fig. 4: The impact of the number of tracked block address bits (T) to the performance of ENTROPYINDEX for a 2MB 16-way set-associative cache.

A. The Cost-Effective Per-Bit Entropy Tracking Mechanism

At its core, ENTROPYINDEX maintains an array of T Entropy Counters (ECs) to measure the entropy of the block address bits. Each counter corresponds to a single bit of the block address. As a result, there are T block address bits being tracked. Figure 4 shows the impact of T on the overall performance of ENTROPYINDEX. For a 2MB 16-way set-associative cache, the index size is 11 bits. Thus, $T \geq 22$ for the Dual-sequence index formulation and $T \geq 11$ for the Uni-sequence index formulation. In general, when T is greater, it adds more flexibility to ENTROPYINDEX, therefore leading to more performance. The simulation results also show that the Dual-sequence index formulation is superior regardless of the value of T. Thus, we opt to use the Dual-sequence index formulation in the design of our solution. We do not see any more substantial performance benefit when T gets higher than 32. At the same time, increasing T over 32 would require a larger multiplexer, which will lead to more power and higher delay (more in Section III). Thus, we use $T = 32$ in our design.

ENTROPYINDEX updates the ECs upon every cache miss. Figure 5 demonstrates an EC update process. Assume blk_addr_{curr} is the block address of the current cache miss, blk_addr_{last} is the block address of the last cache miss, then:

For each bit i in blk_addr_{curr} $(i = 0 \rightarrow T)$:

$$EC[i] \text{ += } (blk\_addr_{curr}[i] \oplus blk\_addr_{last}[i]).$$

The idea behind this update policy is that the more frequent a bit flips, the higher the entropy of that bit. Thus, the counter $EC[i]$ is incremented upon every time the bit $i-th$ flips.

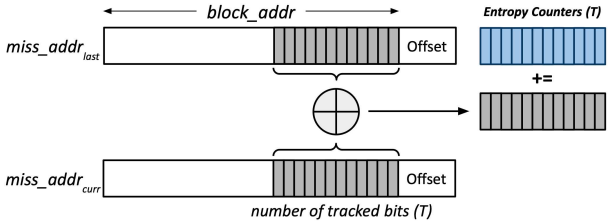


Fig. 5: ENTROPYINDEX updates the Entropy Counters upon every cache miss. T is the number of block address bits that are being tracked.

B. Cache Remapping Strategy

When a new index function 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 [31], as shown in Figure 6.

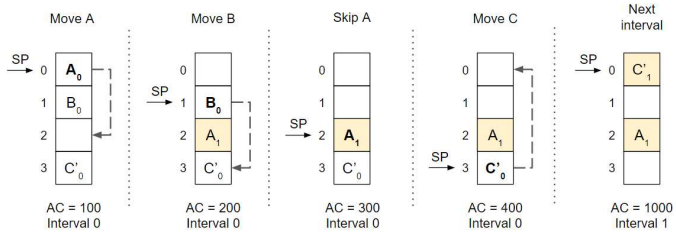


Fig. 6: Example of the gradual cache remapping. Whenever the Access Counter (*AC*) reaches a threshold *R*, the set pointed at by Set Pointer (*SP*) is remapped, and *SP* is incremented. After *AC* reaches the interval size *M*, we enter the next interval, and *SP* is reset. Dirty lines are denoted by the prime notation. To minimize the number of writebacks, if the source line is clean and the destination line is dirty, we evict the source line (example: B_0 and C'_0).

We maintain an *Access Counter (AC)* and a *Set Pointer (SP)* to keep track of the number of accesses to the cache

and the next set to remap, respectively. After every cache access, the counter *AC* is incremented. The parameter *R* is the pre-configured remapping rate of the system. When *AC* reaches *R*, the set pointed to by *SP* is remapped using the new index function. In Figure 6, *R* is set to 100, therefore when *AC* reaches 100, set 0 is remapped (A_0 from set 0 to set 2). After all cache lines in set 0 have been reallocated, *SP* is incremented. Similarly, when *AC* reaches 200, all cache lines in set 1 are reallocated (B_0 from set 1 to set 3). At this point, there is a conflict between B_0 and C'_0. We use the dirty bit as the tie-breaker. We will keep the dirty lines. If both lines are dirty, we keep the line at the source set and evict the line at the destination set. This process continues until every cache set is remapped.

When there is a cache request during the remapping process, we compute the set index using the current index function. If this set index is greater or equal to the *SP*, we know that this set has not yet been remapped, so we serve the cache request with this set index. Otherwise, we use the new index computed by the new index function.

C. Minimizing The Cache Remapping Cost

The expected extra miss rate due to the gradual cache remapping is a function of the remapping rate R and the cache associativity (let us call it W) [31]. Specifically, the average miss rate increase due to the remapping process is W/R. For example, in a 2MB 16-way set-associative cache, if $R = 1600$, the extra miss rate generated by the gradual remapping process is $W/R = 16/1600 = 1\%$. This implies that the new index function must improve the cache miss rate by at least the same amount to break even the cost. Thus, we should only switch to the new index function if its *potential miss rate reduction* is higher than the remapping cost. Assume the *potential cache miss rate reduction* of an index function is P, then the index function change is justified if $P \geq W/R$.

The problem is that there is no easy way to determine P at runtime. As a result, we use the total entropy of the index function to approximate P. Assume f_i is the current index function, f_{i+1} is the new index function, E_i and E_{i+1} are the entropy sum of all bits in f_i and f_{i+1} respectively, then:

$$P_{approx} = \frac{E_{i+1} - E_i}{E_i}$$

In our design, since our remapping rate $R = 80$, and the associativity $W = 16$, therefore the expected extra miss rate due to cache remapping is $W/R = 16/80 = 20\%$. Thus, to justify the remapping cost, ENTROPYINDEX only switches from f_i to f_{i+1} when $P_{approx} \geq 20\%$. We dedicate Section IV-B4 and Figure 15 to discuss about the index function change frequency observed in our experiments.

D. General Workflow of ENTROPYINDEX

Figure 7 demonstrates the overall workflow of ENTROPYINDEX at runtime. During the program execution, ENTROPYINDEX continuously tracks the entropy of T block address bits by updating the ECs upon every cache miss. At the end of an interval i, ENTROPYINDEX ranks the tracked bits based on their EC value, then selects the top N bits as the X_{i+1} sequence and the next N bits as the Y_{i+1} sequence. As such, the new index function candidate for the next interval could be represented as $f_{i+1} = X_{i+1} \oplus Y_{i+1}$. ENTROPYINDEX then computes E_i and E_{i+1} (the entropy sum of all bits in f_i and f_{i+1}). If $\frac{E_{i+1} - E_i}{E_i} \geq 20\%$, ENTROPYINDEX will apply the new index function f_{i+1} to the cache and trigger the cache gradual remapping process.

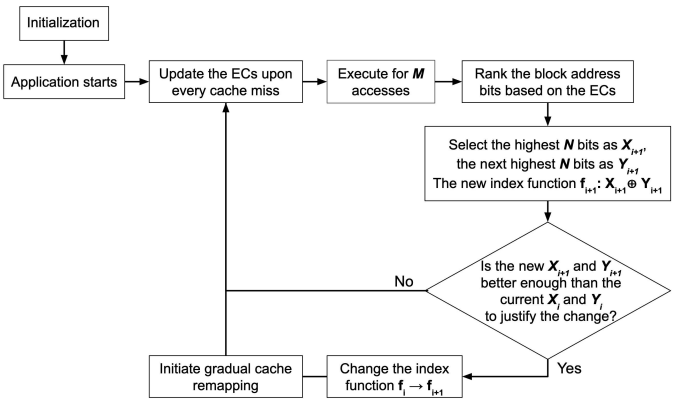


Fig. 7: General workflow of ENTROPYINDEX. M is the interval size, measured in terms of cache accesses.

E. ENTROPYINDEX For Banked Shared Cache

In this work, we assume a monolithic cache with uniform timing across all locations. In modern multi-core architectures, the LLC is split into several banks and distributed across the chip. This banked LLC design uses some bits from the block address to index to the cache banks. For this scheme, we have two possible implementations for ENTROPYINDEX.

The *bank-agnostic* implementation of ENTROPYINDEX will treat the cache as if it is monolithic. Thus, the implementation is similar to Figure 3b. In this design, the set index function will also determine the bank index. During remapping, cache lines can be relocated from one bank to another.

The *bank-aware* implementation of ENTROPYINDEX will ignore the bank index bits, as shown in Figure 8. In this scheme, we only modify the set index function, while keeping the bank index function untouched. As a result, $X_a \oplus Y_a$ determines the set index within a bank, and $N = log_2(num\_sets\_per\_bank)$. In some modern architectures, the slice index function may consist of a large amount of address bits [29], leaving too few remaining bits for the set index function formulation, therefore indirectly affecting the performance of ENTROPYINDEX. For these architectures, we can consider a hybrid design in which some slice index bits can be reused in the set index function. One possible design could be tracking a preset number of highest entropy bits from the slice index, along with the unused address bits for the set index function formulation.

F. Estimating The Full Potential of ENTROPYINDEX Using the Genetic Algorithm

In microarchitectural research, it is often useful to have a theoretical or empirical standard upper bound as a reference.

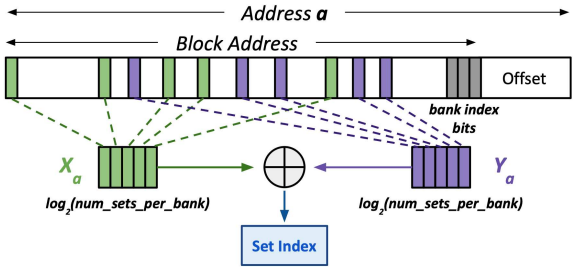


Fig. 8: *Bank-aware* index function formulation. Address bits that are used to index to the banks are ignored (gray colored boxes).

This performance standard gives an estimate of the full potential of an optimization technique, as well as how much benefit we have realized. For example, the Belady algorithm serves as the theoretical performance standard for all replacement policy research. Unfortunately, the problem of finding the optimal set of index bits has been demonstrated to be NP-complete [33]. As such, we attempt to find an empirical near-optimal indexing solution using the Genetic Algorithm.

1) Genetic Algorithm Problem Formulation: The Genetic Algorithm works by first generating a population of random candidates (i.e., potential solutions of an optimization problem). A fitness function is designed to estimate the quality of each candidate. Higher quality candidates should have higher fitness scores. In every training iteration, each candidate in the population is evaluated and ranked based on its fitness score. The highest quality candidates are retained in the population. Mutation and Crossover are then performed on these quality candidates to produce the next generation of candidates. The Genetic Algorithm uses this process to evolve the candidates from one generation to the next until it discovers a near-optimal solution to the problem at hand. To apply the Genetic Algorithm to our indexing problem, we need to define two key components: (1) the candidate representation, (2) the definition of the fitness function and fitness score.

Let us revisit the formulation shown in Figure 3b. Here, X_a and Y_a are sequences of bits extracted from the block address of a. We can represent X_a as:

$$X_a = a[X_{N-1}]\ a[X_{N-2}]...a[X_1]\ a[X_0]$$

where $a[X_i]$ denotes the $X_i - th$ bit of the address a, and $N = log_2(num\_set)$. Similarly, Y_a can be represented as:

$$Y_a = a[Y_{N-1}]\ a[Y_{N-2}]...a[Y_1]\ a[Y_0].$$

As such, a candidate G can be represented as:

$$G = [X_{N-1}, X_{N-2}, ..., X_0, Y_{N-1}, Y_{N-2}, ..., Y_0].$$

An optimal XOR-hashing would choose the bit indices $X_{N-1}, ..., X_1, X_0$ and $Y_{N-1}, ..., Y_1, Y_0$ such that the number of cache misses are minimized for a given application. The lower the number of cache misses is, the better the candidate should be. Therefore, we use the negative of total cache misses as the fitness score.

2) Genetic Algorithm Training Process: To evaluate a candidate, we execute an application with a cache indexing scheme based on that particular index function candidate and count the total number of cache misses. The fitness score of the candidate will be the negative of the total misses.

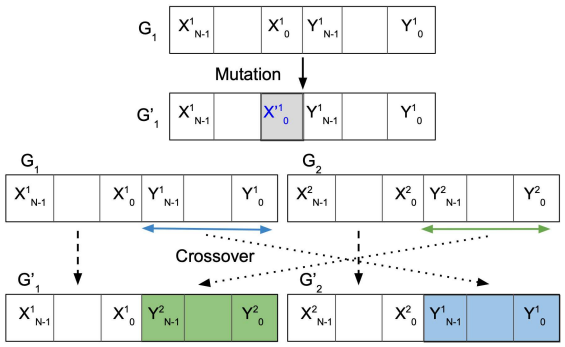


Fig. 9: An example of how Mutation and Crossover can be done with two candidates G_1 and G_2.

In every iteration of the Genetic Algorithm, the candidates are evaluated to determine their quality. They are then sorted based on the fitness score. A number of top scoring candidates are kept and mutated and crossed over in an attempt to produce better quality candidates for the next iteration. Let us consider two candidates - $G_1 = [X^1_{N-1}, ..., X^1_0, Y^1_{N-1}, ..., Y^1_0]$ and $G_2 = [X^2_{N-1}, ..., X^2_0, Y^2_{N-1}, ..., Y^2_0]$. Mutating G_1 can be done by randomly choosing an index inside the candidate and changing it to a different valid index as shown in Figure 9. Here index X^1_0 is changed to a different index X'^1_0. Crossover can be done by swapping half of one candidate G_1 with similar half of another candidate G_2.

For every application, we use the Genetic Algorithm to find the set of bits for X_a and Y_a that has the lowest number of cache misses. We refer to the solutions found by the Genetic Algorithm as NEAR-OPTIMAL and include them in our experiments as a comparison reference.

G. Quantitatively Measuring the Balance of The Cache Access Distribution

A high-performance hash function would provide a more even output distribution, leading to less conflicts. Thus, the balance of the cache access distribution is one of the metrics to evaluate the quality of the cache index function. We adopt the ratio $stdev/mean$ from [23] to quantitatively measure the balance of the cache access distribution. Particularly, assume $a_0, a_1, ..., a_n$ are the number of accesses to the sets $s_0, s_1, ..., s_n$, and $\bar{a}$ is the mean access count across all sets, then the ratio $stdev(a_i)/\bar{a}$ represents the balance of the cache access distribution. Additionally, previous studies show that sophisticated indexing schemes work better for applications with an uneven cache access distribution between cache sets, since there are more opportunities to reduce the cache conflicts [23], [45].

In Figure 10, *473.astar* with ENTROPYINDEX has a more even LLC access distribution compared to DEFAULT, represented by a smaller standard deviation. In this example, the LLC access $stdev/mean$ ratio of *473.astar* reduces from 0.171 with DEFAULT to 0.086 with ENTROPYINDEX. This improvement in the distribution leads to an IPC speed up

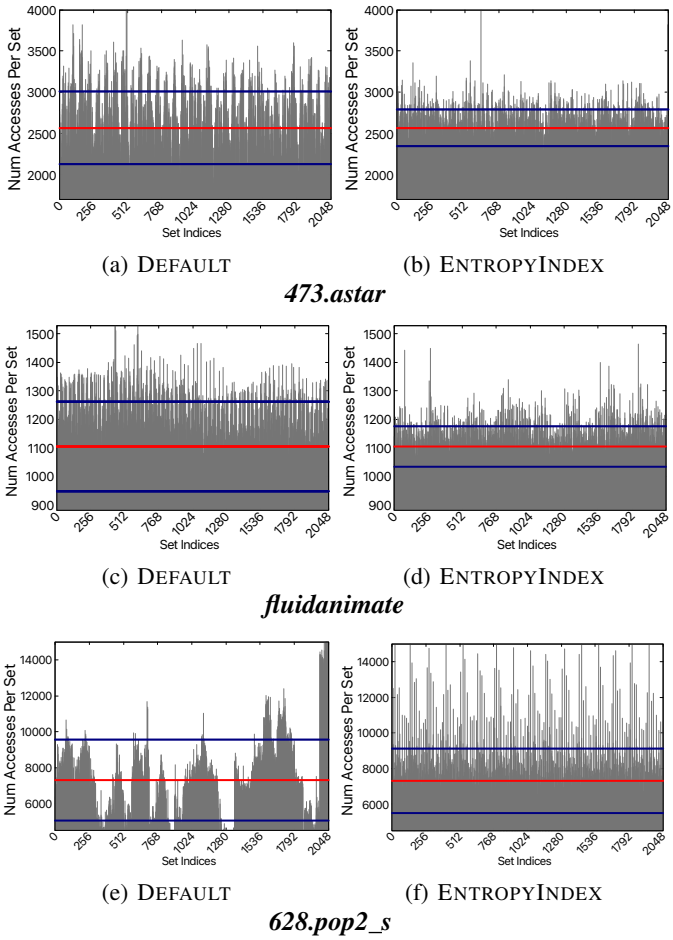


Fig. 10: LLC access distribution of ENTROPYINDEX and DEFAULT in no prefetching scenario. The red line is the mean accesses per set, the blue lines are the upper and lower standard deviation of the cache accesses.

of 1.75%. Similarly, the LLC access $stdev/mean$ ratio of *fluidanimate* and *628.pop2_s* reduces from 0.142 to 0.065 and 0.307 to 0.247, leading to a performance gain of 18.7% and 2.09%, respectively. In Section IV-B3, we evaluate the performance of ENTROPYINDEX on the subset of workloads with imbalanced access distribution, measured by the $stdev/mean$ ratio.

III. IMPLEMENTATION

In this section, we demonstrate the ENTROPYINDEX implementation for a 2MB 16-way set-associative cache. The hardware design of ENTROPYINDEX consists of two parts: the *Entropy Tracking Module (ETM)* and the modified *Set Index Resolution Logic (SIR)* of the cache controller. The ETM can be implemented in a separate circuit, lying off the critical path. The ETM is responsible for updating the Entropy Counters upon every cache miss, and determining the bits with the highest entropy to form the index function f for the next interval. The SIR logic is part of the cache controller circuitry and is responsible for calculating the set index of every cache request.

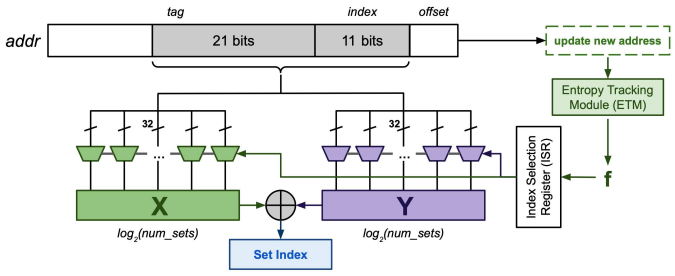


Fig. 11: Hardware implementation of the SIR logic for a 2MB 16-way set-associative cache.

A. Entropy Tracking Module (ETM)

The ETM consists of an array of 32 Entropy Counters and a register holding the block address of the most recent cache miss. Every time there is a new cache miss, the new missed address is compared with the most recent missed address and the Entropy Counters are updated accordingly. At the end of an interval, the ETM will find the 22 block address bits with the highest Entropy Counters and return them to the cache controller. Since the ETM can be implemented in a separate circuitry, it has no impact on the cache access latency as well as the cache cycle time.

B. Set Index Resolution Logic (SIR)

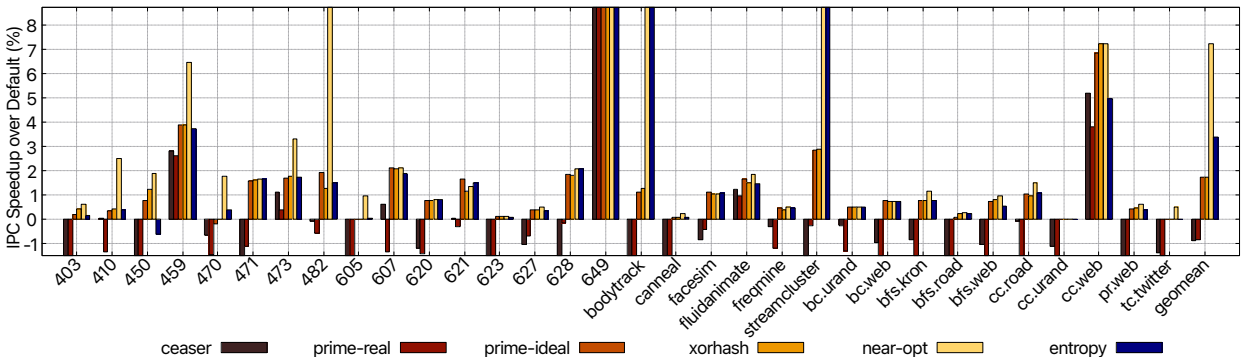
The SIR logic is implemented as a parallel array of 32:1 multiplexers, as shown in Figure 11. The index function configuration f from the ETM is decoded to some bit sequence and saved in the *Index Selection Register (ISR)*. This bit sequence is used as selection bits for the multiplexers. Appropriate bits are selected to form the bitmasks X_i and Y_i. The set index is then computed as $X_i \oplus Y_i$.

32:1 multiplexers can be implemented using transmission gates [21]. Prior work shows that a 16:1 multiplexer implemented using transmission gates can achieve a delay of 19.8 picoseconds [33]. As such, the delay of a 32:1 multiplexer using transmission gates would be approximately 39.6 picoseconds. As a reference, the cycle time of a system running at 3.0 GHz is 334 picoseconds. In Table II, we provide the performance of ENTROPYINDEX in both cases: zero delay and 1-cycle delay. For power consumption, our analysis using the Synopsys Design Compiler NXT [1] shows that the dynamic and static combined power of ENTROPYINDEX is only 1.064 mW. We use the 28nm technology in our synthesis.

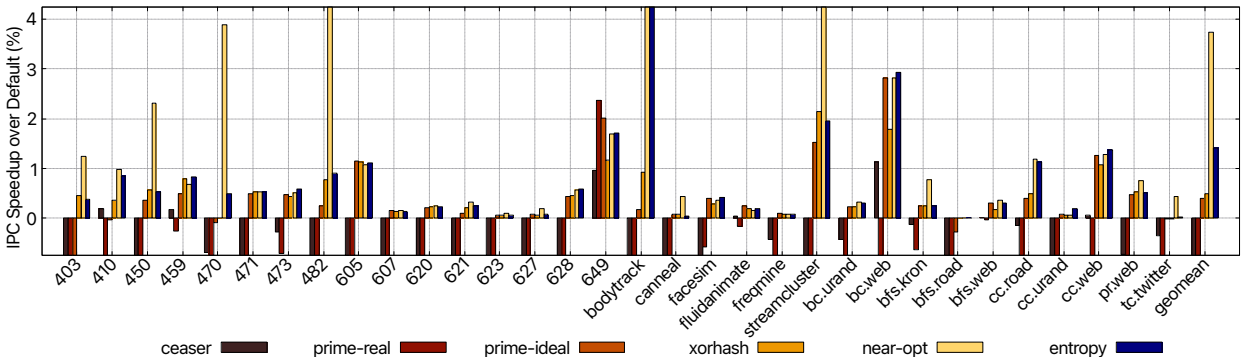
IV. EVALUATION

Parameter	Value
Processor	1 and 4-core @ 3.0 GHz, FetchWidth=6, DecodeWidth=6, ExecWidth=6, RetireWidth=4, 352-entry ROB, 128-entry LQ, 72-entry SQ.
L1 cache (I/D)	32KB (per-core), 2-way, 2-cycle latency.
L2 cache	128KB (per-core), 4-way, 8-cycle latency.
LLC (shared)	2MB and 8MB, 16-way, 32-cycle latency.
Prefetchers	L1D: Next-Line Prefetching, L2: Signature-Path Prefetching [24].
Replacement	Least Recently Used (LRU).
DRAM	tRP=tRCD=tCAS=24.

TABLE I: Simulated hardware parameters. LLC latency number is obtained from CACTI 7.0 [14].



(a) No prefetching



(b) With prefetching

Fig. 12: Single-core performance gain of EntropyIndex compared to other schemes.

A. Methodology

1) Simulator: We evaluate EntropyIndex using the ChampSim simulation framework [7]. ChampSim is widely used in cache microarchitecture research and competitions [7], [10], [16], [17], [25], [31], [35], [38]–[40]. Parameters of the simulated hardware are shown in Table I.

2) Benchmarks: We test our solution using a diverse set of memory-intensive applications from SPEC 2006, SPEC 2017, PARSEC 3.0, and GAP [5]. For SPEC 2006 and 2017, we reuse the traces from the 2nd Cache Replacement Championship (CRC-2) [7] and the 3rd Data Prefetching Championship (DPC-3) [10]. For PARSEC 3.0, we profile the applications in single-threaded mode, since ChampSim does not support multi-threaded program simulation. For all benchmark suites, we exclude the failed traces and those that have the misses-per-kilo-instruction (MPKI) less than 1. We end up using a total of 32 applications in our evaluation. We also evaluate EntropyIndex on the workloads from the Championship Value Prediction (CVP) benchmark suite provided by Qualcomm [8]. Since the entire suite has over 1000 traces, we choose the top 10 workloads with the highest $stdev/mean$ ratio from each of the compute-int and compute-fp benchmark suites.

For each benchmark, we warm up the cache for 2 million LLC accesses, then collect the simulation results of the next 1 billion instructions. For the single-core experiments, we use an interval size of 1 million LLC accesses.

3) Multiprogram Setup: We evaluate the performance of EntropyIndex when 4 different workloads are run on 4 different cores simultaneously. We randomly generate 100 sets of 4 workloads from the set of 32 benchmarks that we have. For this 4-core experiment, we use an interval size of 4 million LLC accesses. We first warm up the cache for 8 million LLC accesses, then run each mix until each application in the mix has executed at least 250 million instructions. If an application reaches the end of its trace, the corresponding core repeats the simulation of that trace from the beginning until every other benchmark in the set has executed 250 million instructions. This multiprogram evaluation methodology is similar to prior work [16], [35], [39].

4) Baseline and Comparison Work: We use the *power-of-2 modulo* index (Default) as the baseline. We compare our solution against different hash functions for set-associative caches, including Xorhash, Prime, and Ceaser. For Xorhash, we implement the XOR scheme presented in Figure 2. For Prime, we compare against both the theoretical zero-delay case (Prime-Ideal) and the realistic, 5-cycle delay, Polynomial Method-based prime modulo proposed in the paper [23] (Prime-Real). For Ceaser, we implement the proposed 4-stage Feistel Network cipher with dynamic remapping scheme from the original work [31]. We also use the latency of 2 cycles mentioned in the paper. The Near-Optimal index function configurations found by the Genetic Algorithm serve as the reference for our experiments, indicating the estimated overall potential performance gains of EntropyIndex.

For deeper analysis, we also give a performance comparison between a 16-way set-associative cache with ENTROPYINDEX index and a 16-way *skewed-associative* cache (SKEWEDCACHE). The 16-way SKEWEDCACHE uses a total of 16 hash functions, one per cache way. These hash functions are generated from the perfect shuffle operations proposed in the original paper [6].

B. Results

1) Overall Single-core: Figure 12 shows the IPC improvement of different indexing schemes compared to the DEFAULT index function in single-core applications. Without prefetching, ENTROPYINDEX provides a 3.39% speed up over DEFAULT, outperforming both XORHASH and PRIME-IDEAL significantly. The highest speed up is 52.2%, achieved in *bodytrack*. With prefetching, ENTROPYINDEX provides an IPC improvement of 1.42%, compared to a 0.49% improvement of XORHASH, and a 0.41% improvement of PRIME-IDEAL. PRIME-REAL has a negative overall performance gain in both scenarios due to the 5-cycle delay of the prime division computation. Similarly, CEASER degrades performance in almost all applications because of the 2-cycle extra delay caused by the index decryption calculation. Additionally, the index function selection in CEASER is completely random, therefore resulting in unpredictable performance.

Index Function	No prefetching	With prefetching
CEASER	-0.88%	-2.12%
PRIME-REAL	-0.84%	-1.46%
PRIME-IDEAL	1.74%	0.41%
XORHASH	1.76%	0.49%
ENTROPYINDEX	3.39%	1.42%
ENTROPYINDEX (worst-case)	2.85%	0.97%
NEAR-OPTIMAL	7.23%	3.73%

a) IPC improvement over DEFAULT

Index Function	No prefetching	With prefetching
CEASER	-4.22%	-5.50%
PRIME-REAL	0.42%	0.38%
PRIME-IDEAL	0.42%	0.29%
XORHASH	0.59%	0.58%
ENTROPYINDEX	3.08%	1.79%
ENTROPYINDEX (worst-case)	3.08%	1.80%
NEAR-OPTIMAL	8.39%	5.71%

b) MPKI reduction over DEFAULT

Index Function	No prefetching	With prefetching
CEASER	-3.39%	-4.56%
PRIME-REAL	0.41%	0.52%
PRIME-IDEAL	0.41%	0.15%
XORHASH	0.53%	0.42%
ENTROPYINDEX	2.80%	1.30%
ENTROPYINDEX (worst-case)	2.80%	1.40%
NEAR-OPTIMAL	8.23%	4.99%

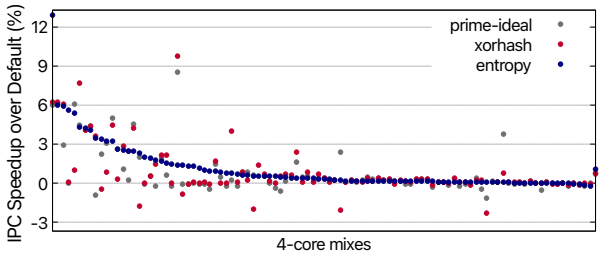
c) Estimated uncore energy reduction over DEFAULT

TABLE II: Geometric mean speed up, MPKI reduction and estimated uncore energy saving of different set-associative cache indexing schemes for all 32 benchmarks in the study. To be on the safe side, we also evaluate the *worst-case* implementation of ENTROPYINDEX which has a 1-cycle delay.

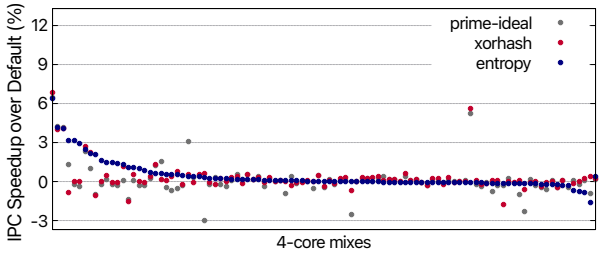
The per-app NEAR-OPTIMAL index configurations found by the Genetic Algorithm provide an overall speed up of 7.23% (no-prefetching) and 3.73% (with prefetching) over the baseline. However, there are some applications where NEAR-OPTIMAL underperforms ENTROPYINDEX. There are two possible explanations for these results. First, the Genetic Algorithm could have converged to a local maxima, resulting in a sub-optimal indexing configuration. Second, previous studies have shown that the set indices that suffer from conflicting accesses can change from one execution phase to another within the same application [33]. ENTROPYINDEX has a more fine-grained control over the index function, being able to change it during the execution. Hence, ENTROPYINDEX has more flexibility to adapt to any change in the program phases, leading to better performance in some applications.

Table II presents the overall performance gain, MPKI and estimated uncore energy reduction rate of different set-associative cache indexing schemes evaluated in this study. Adding one cycle delay reduces the performance of ENTROPYINDEX by approximately half a percent, even though the MPKI is not affected. Nevertheless, this worst-case scenario implementation of ENTROPYINDEX still substantially outperforms previous index functions.

To estimate the energy consumption, We make two assumptions: (1) every LLC access consumes 1 unit of energy and (2) every DRAM access consumes 25 unit of energy on average. These estimation numbers have been used in prior work [38], [46]. Overall, ENTROPYINDEX can reduce the uncore energy by 2.80% (no prefetching) and 1.30% (with prefetching), thanks to the reduction in DRAM traffic.



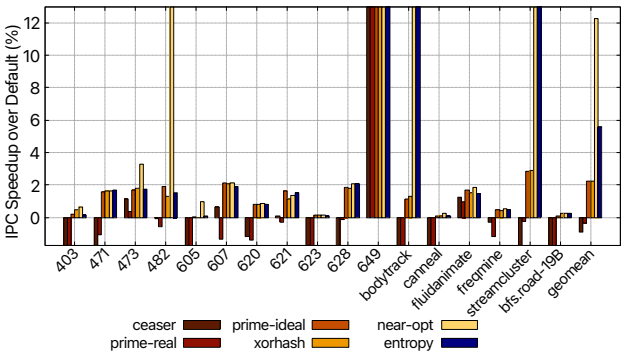
(a) No prefetching.



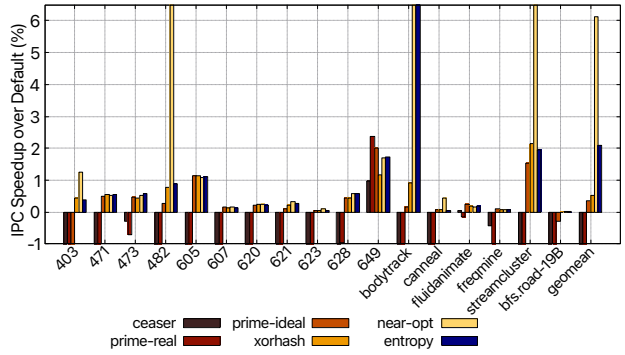
(b) With prefetching.

Fig. 13: Performance comparison between ENTROPYINDEX and other indexing schemes in 100 mixes of 4-core workloads.

2) Overall Multi-core: Figure 13 shows the performance of different indexing schemes in the multi-core setup. When there is no prefetching, the geometric mean performance gain of ENTROPYINDEX is 1.07% over the baseline, compared to a 0.76% improvement of XORHASH and a 0.69% improvement of PRIME-IDEAL. With prefetching, the geometric mean performance gain of ENTROPYINDEX is 0.42% over the baseline, compared to a 0.25% improvement of XORHASH and a 0.14% improvement of PRIME-IDEAL.



(a) No prefetching.

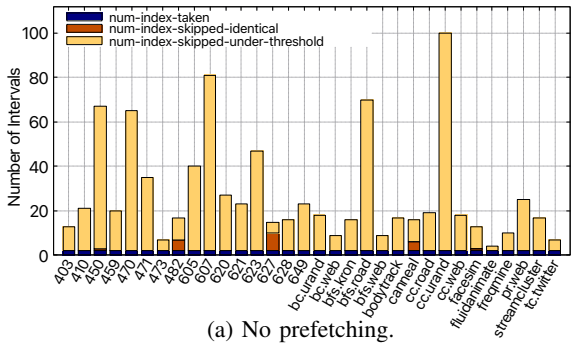


(b) With prefetching.

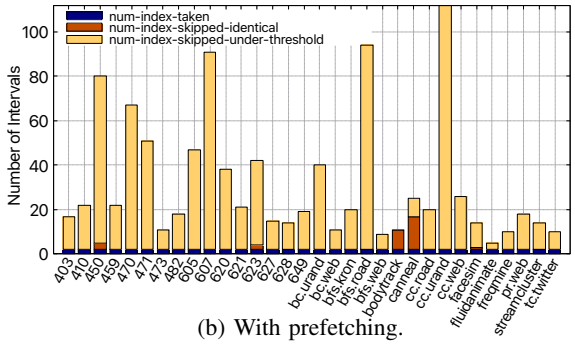
Fig. 14: Performance of ENTROPYINDEX compared to other indexing schemes in *non-uniform* workloads.

3) Non-uniform Applications: In this study, we define *non-uniform* applications as those that have the LLC access $stdev/mean$ ratio to be more than 0.1 under the DEFAULT index function. The experimental results show that ENTROPYINDEX significantly outperforms other indexing schemes for non-uniform applications, with and without prefetching, as shown in Figure 14. In the no-prefetching scenario, ENTROPYINDEX provides an IPC improvement of 5.58%, compared to a 2.23% improvement of XORHASH, a 2.26% improvement of PRIME-IDEAL, and a 12.25% improvement of NEAR-OPTIMAL. With prefetching, the geometric mean speed up of ENTROPYINDEX is 2.08%, compared to a 0.53% improvement of XORHASH, a 0.35% improvement of PRIME-IDEAL, and a 6.12% improvement of NEAR-OPTIMAL. These results indicate that the non-uniform applications generally do get more benefits from complex indexing schemes, but not all cases. There are uniform applications, such as *459* and *cc.web*, still showing considerable performance gains.

4) Number of Index Function Changes: Figure 15 shows the frequency of index function changes observed in our study. For many applications, ENTROPYINDEX only switches the index function a few times before it enters the stable phase. During this phase, new index functions found in later intervals are mostly skipped because either they are the same as the one in place, or they do not pass the preset entropy difference threshold (i.e. $\frac{E_{i+1}-E_i}{E_i} < 20\%$). In our study, we see that during the stable phase, the entropy sum difference between



(a) No prefetching.



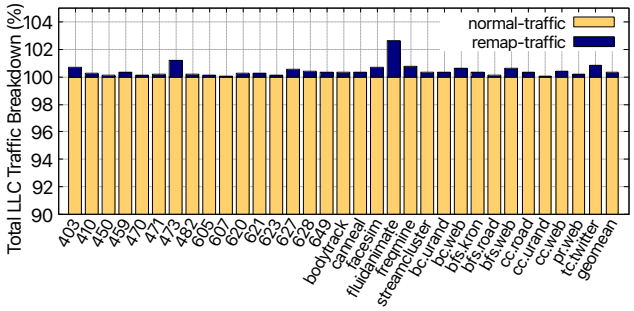
(b) With prefetching.

Fig. 15: Number of index function changes during execution of different applications. ENTROPYINDEX will skip the new index function if it is either (1) the same as the current function (*num-index-skipped-identical*), or (2) the difference $\frac{E_{i+1}-E_i}{E_i} < 20\%$ with E_{i+1} and E_i are the entropy sum of all bits in the new function f_{i+1} and current function f_i (*num-index-skipped-under-threshold*).

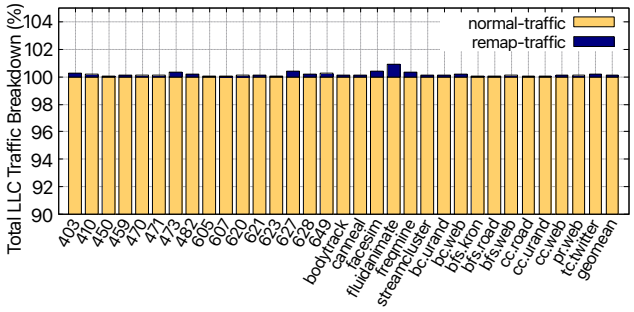
the new index functions and the current function being used is usually less than 2%.

5) LLC Traffic Cost Due to Cache Remapping: Figure 16 shows the impact of the gradual remapping on the overall LLC traffic. We measure the total extra cache reads and writebacks caused by ENTROPYINDEX and divide by the total number of LLC accesses. Without hardware prefetching, ENTROPYINDEX only generates an extra 0.29% traffic to the LLC. When hardware prefetching is on, the total LLC traffic increases significantly due to the prefetching requests. Thus, the extra traffic due to gradual remapping accounts for just 0.12% of the total LLC traffic. In both scenario, the remapping cost stays below 3.0% for all applications. Setting the threshold $\frac{E_{i+1}-E_i}{E_i} \geq 20\%$ helps reducing the number of unnecessary index function changes, minimizing the remapping cost.

6) Compared with SKEWEDCACHE*:* Figure 17 shows that a 16-way set-associative LLC with ENTROPYINDEX outperforms the same-sized 16-way SKEWEDCACHE with and without hardware prefetching. In the no-prefetching scenario, ENTROPYINDEX yields an overall IPC speed up of 3.39% over DEFAULT, compared to a 1.44% speed up of SKEWEDCACHE. In the presence of hardware prefetching, ENTROPYINDEX achieves a mean speed up of 1.42% over DEFAULT, compared to a 0.30% improvement of SKEWEDCACHE. Overall, SKEWEDCACHE performance is extremely polarized. Specifi-



(a) No prefetching.

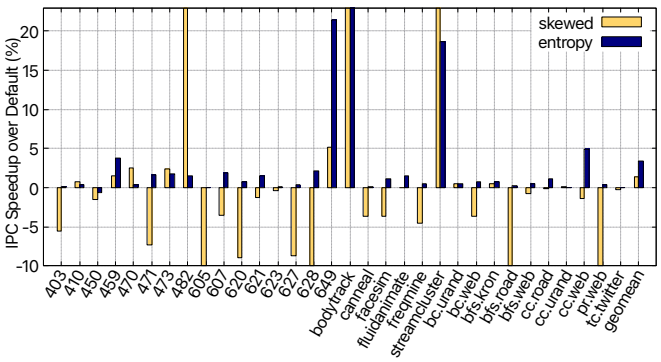


(b) With prefetching.

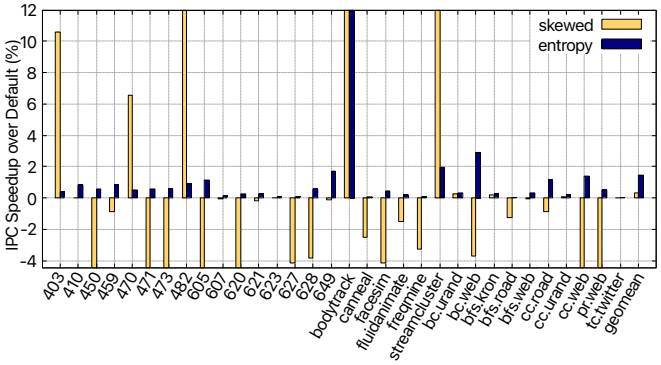
Fig. 16: Extra LLC traffic cost due to cache remapping (%). LLC traffic is the sum of all LLC accesses of all types (demand/prefetch/writeback).

cally, in the no-prefetching scenario, most of the performance gain of SKEWEDCACHE comes from 3 applications: *482, bodytrack, streamcluster*. Similarly, with prefetching, most of the SKEWEDCACHE performance gain comes from 5 applications: *403, 470, 482, bodytrack, streamcluster*. The rest of the applications do not benefit from the multi-index-function scheme of SKEWEDCACHE, resulting in performance losses compared to the DEFAULT indexing scheme. On the contrary, ENTROPYINDEX provides a more uniform performance improvement across all applications, therefore having a better overall result.

7) Performance In The CVP Workloads: Figure 18 shows the single-core results of ENTROPYINDEX in the CVP workloads compared to other schemes. Without prefetching, ENTROPYINDEX provides a geometric mean speed up of 3.04% (with the highest being 6.79%) over the baseline, compared to a 2.04% of XORHASH, 1.52% of PRIME-IDEAL, and 5.93% of SKEWEDCACHE. With prefetching, the geometric mean IPC gain of ENTROPYINDEX is 1.60% (with the highest being 4.13%), compared to a 1.07% of XORHASH, 0.63% of PRIME-IDEAL, and 3.42% of SKEWEDCACHE. We observe that the compute-fp applications in our study generally have a much higher LLC miss rate and also get more benefits from the advanced indexing schemes compared to the compute-int applications. Despite having the highest overall IPC gain, SKEWEDCACHE performance is highly polarized for CVP workloads, similar to other benchmarks. Most of SKEWEDCACHE improvement is concentrated in only five compute-fp benchmarks. In several compute-int workloads, SKEWEDCACHE gives negative IPC gain over the baseline.

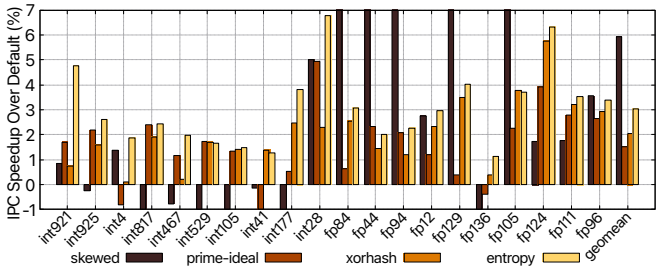


(a) No prefetching.

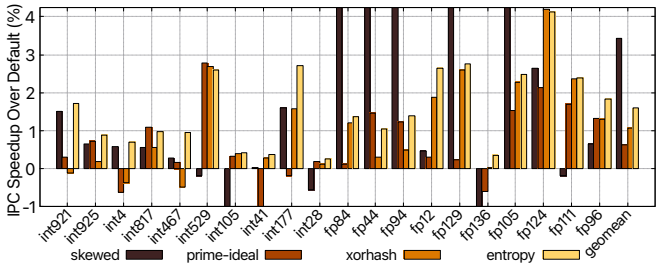


(b) With prefetching.

Fig. 17: Performance comparison between a 16-way set-associative LLC with ENTROPYINDEX and a 16-way skewed-associative LLC (SKEWEDCACHE).



(a) No prefetching.



(b) With prefetching.

Fig. 18: Performance comparison between different cache indexing schemes in CVP workloads.

V. BACKGROUND AND RELATED WORK

A. Index Functions for Set Associative Caches

In many workloads, the conventional *power-of-two modulo* indexing scheme gives an uneven cache access distribution, leading to increased conflict misses. Thus, shared LLCs often

use more advanced hash functions to better spread out cache blocks [15]. The caveat here is that these advanced hash functions require more power and area. XOR-based index functions (XORHASH) should increase the randomness of the index selection, thereby decreasing the likelihood of conflict misses. XORHASH has been utilized in conflict reduction [42] as well as skewed [6] and multi-banked [32] caches.

Alternatively, PRIME [23] uses the function: $set\ index = (address)\ mod\ \ (prime)$, where $prime$ is the largest prime that is less than the number of cache sets. PRIME is shown to improve cache access distribution and is considered the lowest-conflict function due to it's minimal number of divisors [9]. The drawback of PRIME is that it is difficult to implement in a way that is time efficient. One such implementation is Arbitrary Modulus Indexing [9], which uses the binary reciprocal array multiplication mechanism to compute the index for PRIME and other non power-of-2 modulo schemes. In our experiments, we compare to both PRIME as well as a configuration with no latency called PRIME-IDEAL to gauge the full potential of this scheme in a zero-delay scenario.

B. Alternative Cache Organization Methods

The hash-rehash method [3] attempts to increase cache access distribution with the use of two independent hash functions. The first function is used until there is a conflict miss, at which point the second hash function will be used to place the conflicting block elsewhere. By using two hash functions, this scheme is mimics two-way set associative behavior in a direct-mapped cache. The problem with this approach is that all accesses must check two index locations, increasing hit time, and possibly undoing any improvement in IPC. Much like hash-rehash, the column-associative cache [4] utilizes an additional index function to circumvent conflict misses. The inclusion of a "rehashed" bit for each cache line indicates whether or not that line was placed using the first or second function. Like the hash-rehash method, the column-associative cache will experience increased hit time over a direct-mapped cache with one hash functions. The skewed-associative cache [6], [36] takes the opposite approach to hash rehash and column-associative cache by attempting to mimic direct-mapped behavior in a cache with higher associativity. Each way in the cache has it's own hash function, providing the increased distribution of using multiple index functions along side the inherent conflict avoidance of set associative caches. B-Cache [47] attempts to reduce conflict misses by using a programmable decoder instead of multiple hash functions. Two fields of the cache block address are selected, one is used in standard operation and the other is dynamically set to select the bits that correspond to an empty cache line in the event of a conflict miss.

Z-cache [34] expands upon the skewed-associative cache idea of using separate hashing functions for each way by increasing the number of replacement candidates when a conflict miss occurs. First-level blocks are cache blocks whose index directly conflicts with that of the incoming block and second-level blocks are cache blocks that conflict with the first-level blocks. First-level and second-level blocks are all considered candidates for replacement, and the block that was least recently used is evicted to make room for the incoming block. The candidates then have to be relocated so that the incoming block can be placed in a line that matches its index. The number of levels to consider is arbitrary and can include all of the cache blocks; however, each additional layer increases the miss penalty of the cache.

VI. CONCLUSION

Cache index optimization has received far less attention compared to other cache-related research topics over the years. We identify the two most important characteristics of a good cache index function: (1) high performance, and (2) minimal computational latency. Existing cache indexing schemes fall short of these two requirements, having either moderate performance or an impractical, expensive delay. To address this challenge, we propose ENTROPYINDEX, an adaptive entropy-based cache indexing scheme that provides superior performance while having a minimal computational cost. ENTROPYINDEX provides a cost-effective mechanism to track the entropy of multiple address bits during the program execution, then selects the highest entropy bits to form the index function. When the new index function is installed, ENTROPYINDEX triggers the CEASER-based gradual remapping process [31] to reallocate existing cache lines to their new set indices. These processes are done dynamically at runtime, allowing ENTROPYINDEX to adapt to different applications.

We then conduct an in-depth analysis to evaluate the performance of ENTROPYINDEX. For SPEC 2006, SPEC 2017, PARSEC 3.0, and GAP benchmarks without prefetching, ENTROPYINDEX delivers an IPC improvement of 3.39% (with the highest being 52.2%), compared to a 1.74% improvement of the state-of-the-art index function (PRIME) and a 1.76% improvement of a commercialized indexing scheme (XORHASH) over the baseline power-of-two modulo scheme. With prefetching, ENTROPYINDEX is the single indexing scheme with a substantial performance gain of 1.42% (with the highest to be 30.1%), compared to a 0.41% improvement of PRIME and a 0.49% improvement of XORHASH over the same baseline. For non-uniform applications from the SPEC 2006, SPEC 2017, PARSEC 3.0, and GAP benchmark suite and no-prefetching, ENTROPYINDEX provides an IPC speed up of 5.58%, compared to a 2.26% speed up of PRIME and a 2.23% speed up of XORHASH. With prefetching on the same workloads, the IPC speed up of ENTROPYINDEX is 2.08%, compared to a 0.35% speed up of PRIME and a 0.53% speed up of XORHASH. For CVP workloads, ENTROPYINDEX delivers a speed up of up to 6.79% without prefetching and 4.13% with prefetching compared to the baseline, outperforming previous schemes.

ACKNOWLEDGMENTS

We thank the members of PALab group for valuable discussions. This work was supported in part by NSF under grants CCF 2301334 and TAMU HPRC center.

REFERENCES

[1] "Synopsys design compiler," https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/design-compiler-graphical.html, synopsys Design Compiler NXT.

[2] "Ultrasparc t2 supplement to the ultrasparc architecture 2007," Sun Microsystems, Tech. Rep., August 2007.

[3] A. Agarwal, J. Hennessy, and M. Horowitz, "Cache performance of operating system and multiprogramming workloads," *ACM Trans. Comput. Syst.*, vol. 6, p. 393–431, nov 1988.

[4] A. Agarwal and S. D. Pudar, "Column-associative caches: A technique for reducing the miss rate of direct-mapped caches," in *Proceedings of the 20th annual international symposium on Computer architecture*, 1993, pp. 179–190.

[5] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," in *arXiv*, 2017.

[6] F. Bodin and A. Seznec, "Skewed associativity enhances performance predictability," *ACM SIGARCH Computer Architecture News*, vol. 23, no. 2, pp. 265–274, 1995.

[7] CRC-2. (2017) The 2nd cache replacement championship. [Online]. Available: http://crc2.ece.tamu.edu

[8] CVP-1. (2018) The 1st championship value prediction. [Online]. Available: https://microarch.org/cvp1/

[9] J. R. Diamond, D. S. Fussell, and S. W. Keckler, "Arbitrary modulus indexing," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 140–152.

[10] DPC-3. (2019) The 3rd data prefetching championship. [Online]. Available: https://dpc3.compas.cs.stonybrook.edu

[11] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal instruction fetch streaming," in *2008 41st IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2008, pp. 1–10.

[12] D. A. J. P. V. Gratz and G. C. N. Gober, "Barca: Branch agnostic region searching algorithm," *The First Instruction Prefetching Championship*, 2020.

[13] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*, 6th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.

[14] HP-Labs. An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model. [Online]. Available: https://www.hpl.hp.com/research/cacti

[15] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Systematic reverse engineering of cache slice selection in intel processors," in *2015 Euromicro Conference on Digital System Design*. IEEE, 2015, pp. 629–636.

[16] A. Jain and C. Lin, "Back to the future: Leveraging belady's algorithm for improved cache replacement," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 78–89.

[17] ——, "Rethinking belady's algorithm to accommodate prefetching," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 110–123.

[18] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*, 2010.

[19] D. Jimenez, "Addressing challenges of core microarchitecture research," https://hpca-conf.org/2023/keynotes/.

[20] D. Jiménez and E. Teran, "Multiperspective reuse prediction," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017, pp. 436–448.

[21] V. K. Kamal Chaudhary, Philip Costello, "Programmable circuit optionally configurable as a lookup table or a wide multiplexer," 2004, u.S. Patent 7075333. [Online]. Available: https://patents.google.com/patent/US7075333B1/en

[22] T. A. Khan, D. Zhang, A. Sriraman, J. Devietti, G. Pokam, H. Litz, and B. Kasikci, "Ripple: Profile-guided instruction cache replacement for data center applications," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[23] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee, "Using prime numbers for cache indexing to eliminate conflict misses," in *10th International Symposium on High Performance Computer Architecture (HPCA'04)*. IEEE, 2004, pp. 288–299.

[24] J. Kim, S. H. Pugsley, P. V. Gratz, A. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *MICRO*, 2016.

[25] J. Kim, E. Teran, P. V. Gratz, D. A. Jiménez, S. H. Pugsley, and C. Wilkerson, "Kill the program counter: Reconstructing program behavior in the processor cache hierarchy," *ASPLOS*, 2017.

[26] R. Kumar, B. Grot, and V. Nagarajan, "Blasting through the front-end bottleneck with shotgun," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 30–42, 2018.

[27] R. Kumar, C.-C. Huang, B. Grot, and V. Nagarajan, "Boomerang: A metadata-free architecture for control flow delivery," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 493–504.

[28] M. Lipp, V. Hadžić, M. Schwarz, A. Perais, C. Maurice, and D. Gruss, "Take a way: Exploring the security implications of amd's cache way predictors," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '20, 2020.

[29] J. McCalpin, "Mapping addresses to l3/cha slices in intel processors," in *ACELab Technical Report TR-2021-03*, 2021.

[30] J. Pierce and T. Mudge, "Wrong-path instruction prefetching," in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*. IEEE, 1996, pp. 165–175.

[31] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 775–787.

[32] B. R. Rau, "Pseudo-randomly interleaved memory," *SIGARCH Comput. Archit. News*, vol. 19, no. 3, p. 74–83, apr 1991. [Online]. Available: https://doi.org/10.1145/115953.115961

[33] A. Ros, P. Xekalakis, M. Cintra, M. E. Acacio, and J. M. Garcıa, "Adaptive selection of cache indexing bits for removing conflict misses," *IEEE Transactions on Computers*, vol. 64, no. 6, pp. 1534–1547, 2015.

[34] D. Sanchez and C. Kozyrakis, "The zcache: Decoupling ways and associativity," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010, pp. 187–198.

[35] S. Sethumurugan, J. Yin, and J. Sartori, "Designing a cost-effective cache replacement policy using machine learning," in *2021 IEEE 27th International Symposium on High Performance Computer Architecture (HPCA)*, 2021, pp. 291–303.

[36] A. Seznec, "A case for two-way skewed-associative caches," *ACM SIGARCH computer architecture news*, vol. 21, no. 2, pp. 169–178, 1993.

[37] ——, "The fnl+ mma instruction cache prefetcher," in *IPC-1-First Instruction Prefetching Championship*, 2020, pp. 1–5.

[38] I. Shah, A. Jain, and C. Lin, "Effective mimicry of belady's min policy," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

[39] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, p. 413–425.

[40] Z. Shi, A. Jain, K. Swersky, M. Hashemi, P. Ranganathan, and C. Lin, "A hierarchical neural model of data prefetching," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2021, 2021, p. 861–873.

[41] E. Teran, Z. Wang, and D. Jiménez, "Perceptron learning for reuse prediction," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[42] N. Topham, A. Gonzalez, and J. Gonzalez, "The design and performance of a conflict-avoiding cache," in *Proceedings of 30th Annual International Symposium on Microarchitecture*, 1997, pp. 71–80.

[43] G. Vavouliotis, G. Chacon, L. Alvarez, P. V. Gratz, D. A. Jiménez, and M. Casas, "Page size aware cache prefetching," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022.

[44] A. Wehrl, "General properties of entropy," *Rev. Mod. Phys.*, vol. 50, pp. 221–260, Apr 1978. [Online]. Available: https://link.aps.org/doi/10.1103/RevModPhys.50.221

[45] K. Weston, F. Mahmud, V. Janfaza, and A. Muzahid, "Smartindex: Learning to index caches to improve performance," *IEEE Computer Architecture Letters*, vol. 22, 2023.

[46] H. Wu, K. Nathella, J. Pusdesris, D. Sunwoo, A. Jain, and C. Lin, "Temporal prefetching without the off-chip metadata," in *MICRO '52*, 2019, pp. 996–1008.

[47] C. Zhang, "Balanced cache: Reducing conflict misses of direct-mapped caches," in *33rd International Symposium on Computer Architecture (ISCA'06)*, 2006, pp. 155–166.