A Cost-Effective Dueling Framework for Set-Associative Cache Indexing

Kevin Weston
Texas A&M University
College Station, USA
kevin.weston@tamu.edu

Vahid Janfaza
Texas A&M University
College Station, USA
vahidjanfaza@tamu.edu

Avery Johnson
Texas A&M University
College Station, USA
averyjohnson@tamu.edu

Abdullah Muzahid
Texas A&M University
College Station, USA
abdullah.muzahid@tamu.edu

Abstract

Pathological program behavior may cause a non-uniform access distribution in set associative caches, leading to an increase in conflict misses. To address this challenge, prior works profile the program patterns and propose different index functions to avoid these conflict misses [11, 18]. However, as we analyze the prior work on set-associative cache indexing, we identify two major issues. First, there is no single index function that is guaranteed to perform well for every application. Second, advanced indexing schemes typically have sophisticated implementation and prohibitively long computation latency. In this paper, we propose Duelhash, a dynamic *N-way* indexing framework for set associative caches, which provides an effective dueling mechanism for multiple index functions at runtime with a simple and efficient hardware implementation. At runtime, the performance of the index functions are evaluated periodically, and the best performer is applied to the cache.

To evaluate the performance of Duelhash, we conduct a case study on a 16-way set-associative LLC using a diverse set of benchmarks, including SPEC 2006, SPEC 2017, PARSEC 3.0, CVP and GAP. Our empirical results show that without prefetching, Duelhash provides an IPC speed up of 2.8% (with the highest being 23%) over the conventional *power-of-two modulo* (Default) index, compared to a 1.6% speed up of a commercialized indexing scheme (Xorhash). When pattern-based prefetchers are turned on in the L1 data and L2 caches, Duelhash can provide up to 5.8% single-core speedup over Default. Duelhash also provides a 6.2% MPKI reduction over Default for non-uniform applications, the highest among all schemes tested in the study. The reduced LLC miss rate results in an uncore energy saving of 4.9% (without prefetching) or 4.1% (with prefetching). Compared to the state-of-the-art cache indexing scheme, Duelhash offers a comparable performance in most applications while using 93% less power.

Keywords

last level cache, index hashing, index dueling

ACM Reference Format:
Kevin Weston, Vahid Janfaza, Avery Johnson, and Abdullah Muzahid. 2025. A Cost-Effective Dueling Framework for Set-Associative Cache Indexing. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3721145.3729513

ICS '25, Salt Lake City, UT, USA

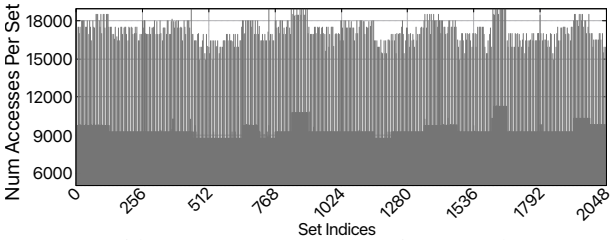
ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3729513

1 Introduction

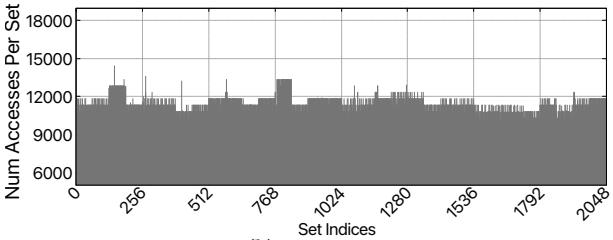
The cache memory has a great impact on the overall system performance. To keep up with the ever growing demand of modern applications, the cache memory is being consistently expanded in size. Conversely, such an approach may not always be effective due to the increasing latency and manufacturing cost. As a result, it is crucial to improve the cache performance through microarchitectural innovations.

Conflict misses are a well-known performance bottleneck in cache memory. This is because the Default index function is susceptible to cache conflicts when facing pathological program behaviors [18]. Figure 1 shows an example of this issue. The Default index function causes an uneven cache access distribution in *streamcluster*, as some of the cache sets receive more than twice as many accesses as other sets (Figure 1a). A more advanced indexing scheme, such as Xorhash, can provide a more even distribution (Figure 1b), significantly reducing the conflict misses. In this example of *streamcluster*, Xorhash reduces the misses-per-kilo-instructions (MPKI)

by 7.22%, leading to an IPC increase of 2.91%. That said, as we analyze the prior work on set-associative cache indexing, we identify two important challenges. Firstly, finding a single, high-performance cache index function for all applications is an extremely difficult task. This is because, similar to other fields, there is not a perfect hash function that is guaranteed to work for all types of input patterns. Secondly, since the set index computation lies in the critical path, it is crucial for the index function to have minimal latency [39]. There are advanced hash functions [6, 18] that offer near-optimal results in theory, but the latency of their computations erases all of the performance benefits [39].



(a) Power-of-two Modulo (Default)



(b) Xorhash

Figure 1: Per-set access distribution of a 2MB 16-way Last Level Cache (LLC) between the Default index function baseline and Xorhash in *streamcluster*.

To address these challenges, we propose an adaptive indexing framework called Duelhash for set-associative caches. At its core, Duelhash maintains a set of N hash functions, each of which is associated with a *Conflict Tracking Counter* to keep track of the number of hashing conflicts in the cache at runtime. These counters are updated with every access to the cache, and during program execution, Duelhash periodically evaluates these counters and applies the index function with the lowest number of conflicts. As such, Duelhash is effectively an *N-way dueling indexing scheme*, in which the best index function is selected for the cache dynamically at runtime based on their performance, measured by the number of conflicts generated. We should note that there are other competition-based adaptive mechanisms in microarchitectural research, one of which is *set-dueling* [17, 24]. Set-dueling is intended to dynamically compare multiple cache policies. Set-dueling uses the set index to determine whether a cache request would go to the dedicated or follower sets. As we have N index functions in Duelhash, there would be cases in which a cache request belongs to multiple dedicated sets, or follower sets at the same time. As a result, set-dueling is inapplicable in cache indexing studies. Hence, we propose Duelhash, a novel N-way dueling framework specifically designed for cache indexing. To the best of our knowledge, this is the first work on dynamic cache indexing using a simple and cost-effective dueling mechanism.

We conduct a case study on a shared 16-way set-associative LLC to illustrate the performance of our proposed framework compared to different indexing solutions. In the study, we gather a set of five XOR-based lightweight index functions, as well as the Default index function to set up a 6-way dueling indexing scheme for the LLC. Simulation results show that Duelhash outperforms Prime and Xorhash in both prefetching and no-prefetching scenarios. By using multiple hash functions, we give the cache indexing scheme an opportunity to dynamically adapt to different program patterns. Moreover, Duelhash is the only indexing scheme that does not degrade the single-core performance of any testing application over Default, both with and without hardware prefetching. This is because Duelhash keeps the Default index function as one of its dueling functions. Hence, Duelhash will fall back to the Default index function if it detects that switching to a more sophisticated hash function would not yield any better performance.

Duelhash also reduces the LLC MPKI by 6.2% in both with and without prefetching scenarios, outperforming all other indexing schemes. This amount of LLC MPKI reduction translates into an uncore energy saving of 4.9% (without prefetching) or 4.1% (with prefetching). Compared to EntropyIndex [39], the state-of-the-art dynamic indexing scheme, Duelhash provides a comparable performance in most applications while consuming 93% less power. Duelhash also works seamlessly with single and multiprogram workloads. Additionally, Duelhash has a surprisingly efficient hardware implementation based on a single 8:1 multiplexer and some rewiring.

In summary, we make the following contributions:

- We propose Duelhash, the *first* dueling-based adaptive indexing framework for set-associative caches that shows substantial performance gain while being extremely hardware-efficient.
- We provide a detailed and efficient hardware implementation of Duelhash.
- We conduct a case study on a shared 16-way set associative LLC using a diverse set of benchmarks including SPEC 2006/2017, PARSEC, CVP and GAP to evaluate the performance of Duelhash. Without prefetching, Duelhash delivers an IPC speed up of 2.8% (with the highest being 23%) over Default. When pattern-based prefetchers are turned on in the L1 data and L2 caches, Duelhash can provide up to 5.8% single-core speedup

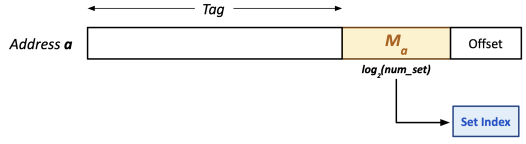
over Default. For non-uniform applications, Duelhash gives an MPKI reduction of 6.2%, outperforming all other indexing schemes in the study. Compared to the state-of-the-art adaptive cache indexing scheme, Duelhash provides a comparable performance in most applications while consuming 93% less power.

2 Background and Related Work

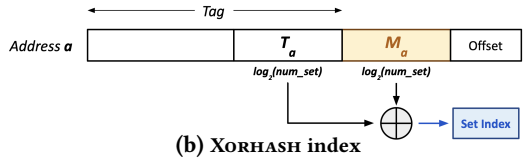
When it comes to cache index hashing, previous studies focus on two directions: improving the indexing scheme for set-associative caches [11, 18], or proposing alternative cache organizations [1, 2, 5, 27, 41]. In this work, we focus on the set-associative cache and its index function as it remains the most commonly used cache organization in processors [12].

2.1 Index Function for Set Associative Caches

Power-of-two modulo (Default) is the simplest hash function used for cache indexing. In this technique, some bits of the cache block address are used to directly identify the index (Figure 2a). For shared LLCs, more advanced hash functions are proposed to evenly distribute the cache accesses [14].



(a) Conventional *power-of-two* (Default) index



(b) Xorhash index

Figure 2: Example of a commonly used Xorhash compared to the Default index function [39].

XOR-based index hashing (Xorhash) has been proposed for skewed caches [5], conflict reduction [36], and multi-banked caches [26]. The main idea of Xorhash is that the XOR operation should increase the randomness of the set index, which could potentially lower the number of conflicts. Figure 2b shows one prominent example of Xorhash. instead of being just M_a as in Default, the set index of an address in Xorhash is computed as $T_a \oplus M_a$, with M_a being the original index in Default, and T_a being a sequence of $\log_2(num\_set)$ bits from the tag part. Xorhash has been reported to be used in commercial processors [14, 21, 22]. EntropyIndex [39] leverages the same idea as Xorhash, looking to maximize the randomness in the set index computation to improve performance. Particularly, EntropyIndex proposes a light-weight strategy to approximate the randomness of each address bits during program execution. The XOR-based index function is then constructed using the address bits with the highest randomness and is used in the following execution interval.

Prime[18] uses the function: $set\ index = (address)\ mod\ (prime)$ with $prime$ being the prime number that is lower and closest to the number of cache sets. Prime has long been considered to be the lowest-conflict index function since it has the smallest number of divisors [9]. Prime shows improvements to the balance in the cache access distribution, leading to performance gain. However, the real challenge is to find a timing-wise practical implementation of Prime. Arbitrary Modulus Indexing [9] addresses this issue by proposing efficient methods based on the binary reciprocal array multiplication mechanism to compute indices for non power-of-2 modulo indexing schemes, including Prime. In our experiments, we include a zero-delay Prime-Ideal configuration as a comparison work to evaluate the full potential of Prime indexing without latency.

2.2 Alternative Cache Organizations

The hash-rehash method [1] addresses this problem by using two sequential hash functions, where when a miss occurs, a second hash function is used, which allows the direct mapped cache to mimic two-way associative behavior. One problem with this approach is that the use of two simultaneous hashing functions will increase hit time, potentially negating any IPC improvement. Similar to the hash-rehash method, column-associative caches [2] attempt to address this issue by using a separate hashing function in the event of a miss to place the cache block at a separate index. Additionally, a 'rehashed' bit is added to the cache lines to indicate whether or not a given entry is using the alternative hashing function. Due to the use of additional hashing functions in a direct-mapped structure, column-associative cache will, like hash-rehash, have an increased hit time.

In the reverse of the hash-rehash and column-associative caches, skewed-associative caches [5, 29] attempt to exhibit direct-mapped behavior in higher associative caches. The skewed-associative cache has a separate hash function for each cache way, providing the increased conflict avoidance of higher associativity caches combined with the increased distribution of cache entries with multiple hashing functions. Z-cache [27] takes the basic idea behind skewed-associative cache, that being separate hashing functions for each way, and extends it by expanding the number of replacement candidates in the event of a conflict. When a miss occurs, the first-level replacement candidates are the blocks located at conflicting indexes to the incoming data. The second-level candidates are the blocks located at conflicting indexes to the first-level candidates. Of all of these blocks, the least recently used is evicted, and the remaining candidates are relocated to accommodate the incoming block. The number of candidate

levels can be expanded to include all of the blocks in the cache at the cost of an increased miss penalty.

Instead of using multiple hashing functions to increase cache distribution, B-Cache [41] attempts to address the conflict miss problem in direct-mapped caches by using a programmable decoder which can dynamically change the indexing of the cache rows when a conflict occurs. This is done by taking two index fields from the address, one corresponding to the standard cache index, and the second corresponding to the programmable decoder, which will, in the event of a conflict miss, find the index of an unused cache space, and allocate that to the cache block.

3 Main Idea: DUELHASH

In this section, we first explain how DUELHASH works (Section 3.1). Next, we describe how DUELHASH handles the transition from one index function to the next (Section 3.2). Finally, we explain how DUELHASH minimizes the cost of the index function transition process (Section 3.3).

3.1 The Efficient Index Function Dueling Mechanism for Cache

Assume the set of dueling index function is H, consisting of N index functions. DUELHASH is equipped with an array of N *Conflict Tracking Counters (CTC array)*. Each counter tracks the number of conflicts of an index function h_i in H. DUELHASH updates these CTCs upon every access to the cache. Algorithm 1 describes the CTC update process.

Algorithm 1: How the CTC array is updated at runtime.

```
Variables:
addr_new: the incoming address to the cache;
H: set of N dueling index functions of DUELHASH;
CTC: array of N Conflict Tracking Counters (CTC);
/* Iteratively update the CTC of each hash function */
if addr_new ≠ addr_prev then
    for every hash function h_i in H do
        if h_i(addr_new) == h_i(addr_prev) then
            CTC[i] ++;
```

When a program begins its execution, DUELHASH starts out with the DEFAULT index function. This prevents any index function to have any initial advantage. After every access to the cache, DUELHASH compares the new address $addr_{new}$ with the address $addr_{prev}$ of the most recent access to the cache. For each index function h_i in H, if the new address $addr_{new} \neq addr_{prev}$, but $h_i(addr_{new}) == h_i(addr_{prev})$, the corresponding counter of h_i in the CTC array is incremented. The idea behind this policy is that if the hash function h_i produces the same output for two distinct inputs $addr_{new}$ and $addr_{prev}$, it means that there is a hashing conflict, therefore we need to increment the corresponding CTC of h_i.

Figure 3 demonstrates the general workflow of DUELHASH. After every execution interval of I cache accesses, DUELHASH examines the CTC array and finds the index function with the lowest conflict counter. If this index function is better than the current index function by a certain amount (more detail on Section 3.3), the index function change is justified and the new index function will be applied to the cache in the next interval. This process takes very little time and can easily be completed between two LLC accesses, or during a cache miss. Thus, the lowest-conflict index function can be applied to the cache in the *next* interval. This is different from previous studies on microarchitectural reconfiguration [35, 40], in which the predicted configuration is applied in the *interval after the next*, as the prediction models could take a long time to compute a prediction.

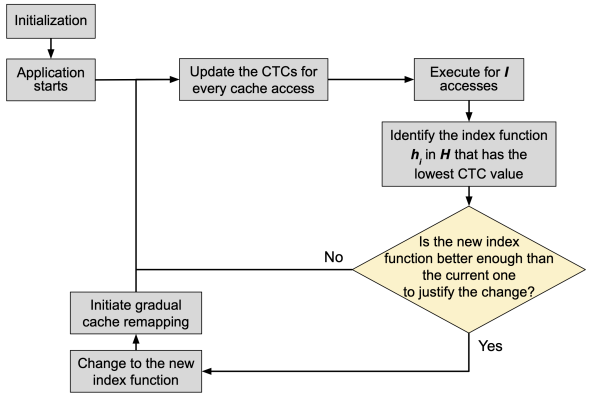


Figure 3: Overall workflow of DUELHASH. I is the interval size, measured in number of cache accesses.

3.2 Cache Remapping Strategy

When we apply a new index function to the cache, it is essential to relocate current valid cache lines to match the new function, ensuring the consistency of the cache. This transition requires careful management to prevent unnecessary data traffic, including extra cache reads, writes, and evictions. We opt to use the gradual cache remapping scheme proposed in CEASER [25]. Figure 4 shows an example of the gradual remapping scheme at work.

We use an *Access Counter (AC)* to keep track of the total number of accesses to the cache, and a *Set Pointer (SP)* to keep track of the next set to remap. After every cache access, *AC* is incremented. The predefined parameter R is the remapping rate of the system. Whenever the *AC* reaches R, the remapping process is triggered and the set pointed at by *SP* is remapped. During this process, we move every cache block in the set pointed by *SP* to its new index, computed by the new index function. If the destination set is full at that time, we trigger the cache replacement policy to find a victim. If this victim is dirty and the remapping cache block is clean, we evict the remapping block (similar to the example

in Figure 4). This policy helps minimize the number of extra writebacks. In all other cases, we replace the victim block by the remapping block.

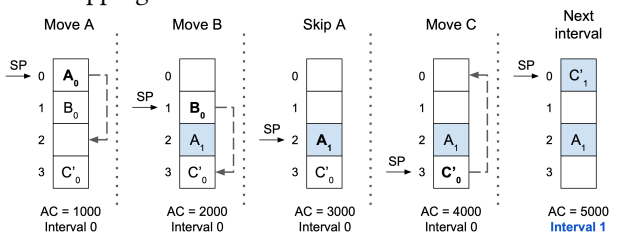


Figure 4: Example of the gradual cache remapping. Blocks marked with a prime notation are dirty. To reduce writebacks, if the source block is clean but the destination block is dirty, we remove the source block (example: B0 and C'0) [39].

The system maintains two index functions at all times: the new index function of the current interval and the last index function from the last interval. When there is an incoming cache request, we compute two set indices: the new index using the new index function and the last index using the last index function. If the last index is greater or equal to the *SP*, it indicates that the set has not been remapped yet, and so cache request is served using the last index. Otherwise, the new index is used.

If I is the interval size, S is the total number of cache sets, then R can be set anywhere between 1 and I/S. R should not be set too low, since there is a risk of the cache being overwhelmed by a sudden surge of read and write requests at the beginning of an interval, potentially stalling other cache demand accesses.

3.3 Minimizing The Cache Index Function Transition Cost

During the remapping process, some cache blocks might be evicted due to conflict when the destination set is full. The miss rate increase due to this behavior can be estimated by the ratio W/R (%), with R being the remapping rate and W being the cache associativity [25]. For example, with a 2MB 16-way LLC, if $R = 1000$, then the potential extra miss rate caused by the remapping process is $16/1000 = 1.6\%$.

To compensate for this performance loss, the new index function must manage to reduce the miss rate by the same amount in the next interval. As a result, an index function change is justified only if the miss rate reduction rate of the new index function is *higher* than the extra miss rate caused by the transition process. We adopt the optimization strategy in EntropyIndex [39]. Specifically, Duelhash only switches to the new index function when $P \geq W/R$, with P being the *potential miss rate reduction* of the new index function. As it is non-trivial to determine P for each index function, we will use the difference between the CTCs of two index functions to estimate P. Particularly, assume f_i and f_{i+1} are the index functions of interval i and $i+1$, then Duelhash will take the switch from f_i to f_{i+1} if and only if:

$$\frac{CTC_i - CTC_{i+1}}{CTC_i} \geq W/R$$

In our study, we use a remapping rate $R = 80$, therefore $W/R = 16/80 = 20\%$. Thus, we only switch to the new index function if the CTC of the new index function is at least 20% lower than the CTC of the current index function.

4 Case Study: 16-way Set-Associative LLC

We conducted a case study on a 16-way set-associative cache to illustrate the performance of our proposed indexing scheme. In this section, we first describe how the index functions for Duelhash (Section 4.1) were selected. Next, we provide the hardware implementation and a cost analysis of Duelhash for the cache configuration in this case study (Section 4.2). Finally, we explain the evaluation methodology (Section 4.3) and then present the experimental results (Section 4.4).

4.1 Selecting Index Functions for Duelhash

Duelhash offers a cost-effective N-way dueling framework for caches. However, if N is excessively large, the hardware implementation cost will become prohibitively expensive, regardless of the efficiency of the dueling mechanism. Unfortunately, the search space for cache index functions is infinite. Hence, at design time, we need to identify a manageable set of top-performance index functions. Then, at runtime, Duelhash will dynamically choose and apply the highest-performing function from this set.

One simple strategy is through pre-training. For this case study, to determine which index functions to include in Duelhash, at design time, we gather a set of training applications from five benchmark suites: SPEC 2006/2017, PARSEC 3.0, GAP, and XSBench. The index functions will be tested against this set of training applications. The highest-performing index function for each benchmark suite is selected for Duelhash. This training strategy is not perfect. However, it is simple and practical. In Section 4.3.2, we will discuss in more detail about how we split the benchmarks into the training set and testing set. The goal is to avoid having the same application in both training and testing sets.

For the index functions, we investigate a family of XOR hash functions generated by the *Perfect Shuffle* operation [33]. Perfect shuffle functions have been shown to carry decent hashing properties that potentially help them achieve a low conflict rate [5]. They are also extremely hardware efficient, consisting of just the re-wiring of the input bits.

Let us call the sub-sequences formed by the tag field as T_i (Figure 5), the modulo index bits of an address as M, and the perfect shuffle operation as σ. For each sub-sequence T_i, the perfect shuffle functions are formed as follows:

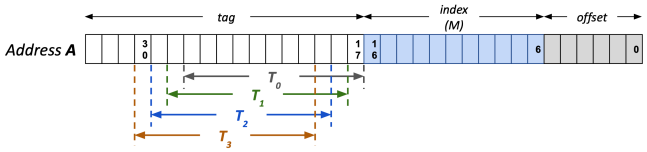


Figure 5: Example of how we form the sub-sequences T_i for the perfect shuffle functions. Assume a 2MB 16-way set-associative cache with 64B block size.

- $h_0^i = T_i \oplus M$
- $h_1^i = \sigma(T_i) \oplus M$
- $h_2^i = \sigma^2(T_i) \oplus M$
- $h_3^i = \sigma^3(T_i) \oplus M$
- $h_4^i = \sigma^4(T_i) \oplus M$
- $h_5^i = \sigma^5(T_i) \oplus M$

For each sub-sequence T_i, we can generate 6 perfect shuffle functions. As we use 4 sub-sequences from T_0 to T_3, we end up with 24 hash functions in total. We include all of them to the initial set of index functions H. We then evaluate each index function against all the applications in the training set, picking the overall best for each benchmark suite, using the IPC improvement over the DEFAULT index function as the metric. We end up with five highest performance index functions, as shown in Table 1.

Benchmark Groups	Best Index Function
SPEC 2006	h_0^2
SPEC 2017	h_1^1
PARSEC 3.0	h_1^3
GAP	h_4^0
XSBench	h_1^5

Table 1: Overall best index function per suite found with the training set. These functions will be implemented in DUELHASH and tested on the testing set.

We notice that some applications do not benefit from index hashing. For these applications, using any kind of hash functions may even degrade the performance compared to DEFAULT. Hence, we add the DEFAULT index function to the mix, making it 6 index functions in total for DUELHASH.

4.2 Implementation

The hardware design of DUELHASH consists of three separate components: the *Set Index Resolution Logic (SIR)*, the *Conflict Tracking Module (CTM)* and the CEASER-based cache remapping. The CTM is implemented as a separate circuit and is responsible for updating the conflict counters as well as determining the optimal index function for the upcoming interval. The SIR is part of the LLC circuitry and is responsible for computing the set index for every incoming address to the cache. In the following sections, we provide the hardware budget for each components for the LLC setup we investigate in our case study. We use 6 hash functions for DUELHASH.

4.2.1 Conflict Tracking Module (CTM). The CTM consists of an array of 6 CTCs (one per index function), one register to store the previous address to the cache, and another set of 6 registers to store the previous indices computed by these functions. Whenever a new access comes in, the CTM compares the new address with the previous address, and the new set of indices with the previous set of indices to determine which function has a conflict, then updates the corresponding CTC in the CTC array. When we reach the end of an interval, the CTM reads the values of the CTC array, returns the index of the lowest counter, and stores it in the Index Selection Register (ISR). This is the index of the hash function in H that will be used in the next interval.

4.2.2 Set Index Resolution Logic (SIR). Figure 6 demonstrates the hardware implementation of the SIR logic. The tag field is passed through the 6 hash functions in parallel. Each of the hash functions h_i is just a fixed re-wiring of the tag bits. The output of the hash functions is the same size of the index field of the address. An 8:1 multiplexer then selects the appropriate hash output based on the index selection decision stored in the *Index Selection Register (ISR)*. The selected hash output is then XORed with index field from the address to form the final set index. The address and the 6 hash function outputs are put together to form a new history entry. This new entry is sent to the CTM to update the CTC array.

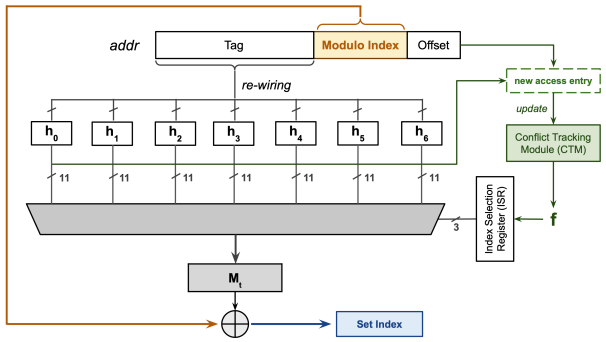


Figure 6: Hardware implementation of the SIR logic for a 2MB 16-way set-associative cache.

4.2.3 Hardware Cost. For the CTM, we need 7 registers to store the previous address and previous indices. we also need 6 registers for the CTCs. The CEASER-based cache remapping requires 24 bytes of storage [25]. So the total hardware budget is 128 bytes, assuming we use 64-bit registers.

The latency of DUELHASH comes from the latency of the SIR and CTM. Fortunately, the CTM computations are not in the critical path, thus do not propagate any extra delay to the cache access latency. As such, the perceivable latency of DUELHASH is the delay of the SIR circuit, which is the sum of the wire delay of the index hashing function, and the gate delay of the 8:1 multiplexer. In Table 2, we provide a detailed

analysis of the resource usage, area, and power consumption of DUELHASH compared to PRIME using Synopsys Design Compiler NXT [34]. For PRIME, we implemented the Polynomial Method proposed in the original work as it offers greater performance and hardware efficiency. We use the 28nm technology in our synthesis.

DUELHASH also has no impact on the cache cycle time, since the SIR logic can run at 4.15GHz, higher than the frequency of 3.0GHz for the cache. The CTM runs at slower speed, but it is implemented in a separate circuit, thereby not increasing the cache cycle time.

Method	Max Frequency (GHz)	Latency at Max Freq. (cycles)	Latency at 3.0 GHz (cycles)	Total Power (mW)	Area (um^2)
PRIME	1.25	2	5	0.602	877
DUELHASH	4.15	1	less than 1	0.071	153

Table 2: DUELHASH hardware cost breakdown. The cache system is set to run at 3.0 GHz.

In terms of power usage, DUELHASH consumes a total of 0.071mW. This is 88% lower than PRIME, which uses 0.602mW. For the same cache configuration, the power usage of ENTROPYINDEX is 1.064mW, as mentioned in the original paper [39]. Thus, DUELHASH uses 93% less power than ENTROPYINDEX. Both DUELHASH and ENTROPYINDEX use the similar CEASER-based cache remapping strategy, which increases overall system power by approximately 0.2% [25].

4.3 Evaluation Methodology

4.3.1 Simulator. We evaluate DUELHASH using the ChampSim simulator [7]. ChampSim has been widely used in cache memory research, including replacement [15, 20, 28, 30, 31], prefetching [4, 16, 32, 38], indexing [25, 39, 40], as well as other cache-related competitions [7, 10]. Parameters of the simulated hardware are shown in Table 3.

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: Berti Prefetcher [23]
	L2: Signature-Path Prefetching [19]
Replacement	Least Recently Used (LRU)
DRAM	tRP=tRCD=tCAS=24

Table 3: Simulated hardware parameters. LLC latency number is obtained from CACTI 7.0 [13].

4.3.2 Benchmarks. We use a diverse set of memory intensive applications from SPEC 2006, SPEC 2017, PARSEC 3.0, GAP [3] and XSBench [37] in our evaluation. For SPEC 2006 and 2017, we use the traces from the CRC-2 and DPC-3 competitions [7, 10]. For PARSEC 3.0, we profile the applications in single-threaded mode using the profiling tool that comes with ChampSim. For all benchmark suites, we define *memory intensive applications* as those that have the LLC misses-per-kilo-instruction (MPKI) greater than 1. This is similar to [15, 28, 31, 32].

We also evaluate our solution against the set of benchmarks provided by Qualcomm for the CVP1 Championship [8]. Since there are more than 1000 total applications in this benchmark suite, we use the top 10 most non-uniform applications with MPKI greater than 1 from each compute-fp and compute-int suite. For each application, we warm up the cache for 500 million instructions, then collect the simulation results of the next 1 billion instructions. For DUELHASH, we use a remapping rate $R = 80$ for both single and multiprogram scenarios.

With $R = 80$ (remapping one cache set per 80 LLC accesses), it takes $80 * 2048 = 163840$ accesses to completely remap all cache sets in a 16-way 2MB LLC. As such, the interval size has to be at least that same amount to allow the system to finish the cache remapping process before going to the next interval. We conducted a sensitivity test to investigate the impact of interval sizes to the performance of DUELHASH. The result is shown in Figure 7. Generally, the overall performance increases with the interval size. This is because as the interval size increases, there are less noise, leading to better index function selections. Larger interval size also results in less frequent index function change, reducing the cache remapping cost. We use the interval size of 1 million LLC accesses for the singleprogram experiments, and 4 million LLC accesses for the 4-core experiments.

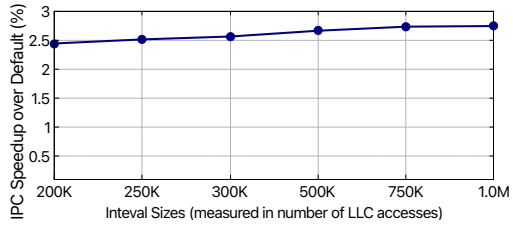


Figure 7: The impact of the interval size to the performance of DUELHASH.

In the proposed design, DUELHASH only compares the current cache address to one preceding address to check for conflict and update the CTC array. Figure 8 shows the performance of DUELHASH when we increase this comparison window from 1 to 96 LLC accesses (6X the LLC associativity). The performance gain barely changes as we vary the history size. We observe that for each application, there is an index function that has a significantly lower conflicts compared to others, so even when the comparison window changes, that same function is still eventually selected.

We split the benchmarks in to two groups: training set and testing set. The training set consists of one third of applications from each of these suites: SPEC 2006, SPEC

2017, PARSEC and GAP. For XSBench, as we only have six applications in this suite, we decide to use all of them for training, since using a third of them (two applications) would be insufficient to represent the suite in the training set. As a result, the training set contains 24 applications from SPEC 2006, SPEC 2017, PARSEC, GAP and XSBench. The testing set has 27 applications from SPEC 2006, SPEC 2017, PARSEC, GAP as well as 20 applications from the CVP compute-int and compute-fp benchmark suites, making it 47 applications in total. No training application appears in the testing set, avoiding the overfitting issue.

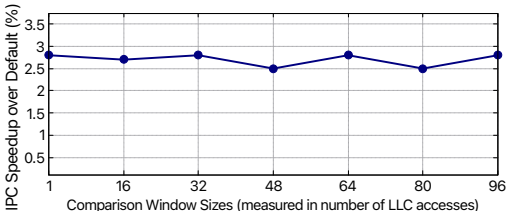


Figure 8: DUELHASH performance when we vary the number of previous addresses to which we compare the current address.

4.3.3 Multiprogram Setup. We evaluate the performance of our solution when 4 different workloads are simultaneously run on 4 different cores. We randomly generate 100 mixes of 4 workloads from the testing applications. For each mix, we warm up the cache until each application has been run for at least 200 million instructions. We then run the mix until each application has been executed for at least 250 million instructions. If an application finishes early, the corresponding core repeats the simulation of that trace until every other benchmarks in the mix has been executed for 250 million instructions. This methodology is similar to prior work [15, 28, 31, 39].

4.3.4 Baseline and Comparison Indexing Schemes. The baseline is the *power-of-two modulo* function (DEFAULT). We compare DUELHASH to CEASER, PRIME, XORHASH, ENTROPYINDEX, and SKEWEDCACHE. All indexing schemes are applied on a 2MB 16-way LLC, with Berti prefetcher in L1 data cache and SPP prefetcher in L2 cache.

For CEASER, we implement the 4-stage Feistel Network cipher with gradual remapping scheme proposed in the original paper. We also include in our simulation the 2-cycle extra latency mentioned in the original work. For PRIME, we compare our scheme against both the theoretical ideal case (PRIME-IDEAL) in which the extra latency is zero, and the realistic case (PRIME-REAL), where the latency is 5 cycles, as discussed in Section 4.2.

We also conduct a direct performance comparison between a 16-way set-associative cache with DUELHASH and a 16-way *skewed-associative* cache (SKEWEDCACHE). In the base case, a 16-way SKEWEDCACHE would have one index function for each cache way, therefore having a total of 16 functions. For better evaluation, we also test with different number of skewing functions, varied from 2 to 16 (a 16-way SKEWEDCACHE with 2 skewing functions means each function is used for 8 cache ways, etc.) We use the perfect shuffle operation to generate these functions, as proposed in the original paper [5].

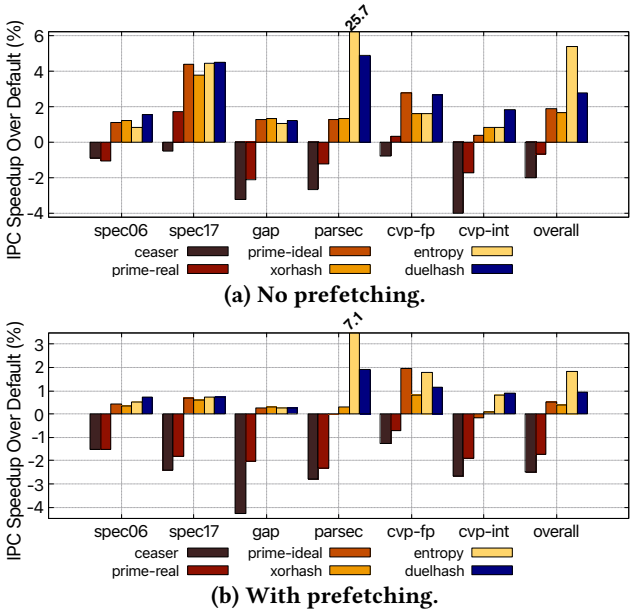
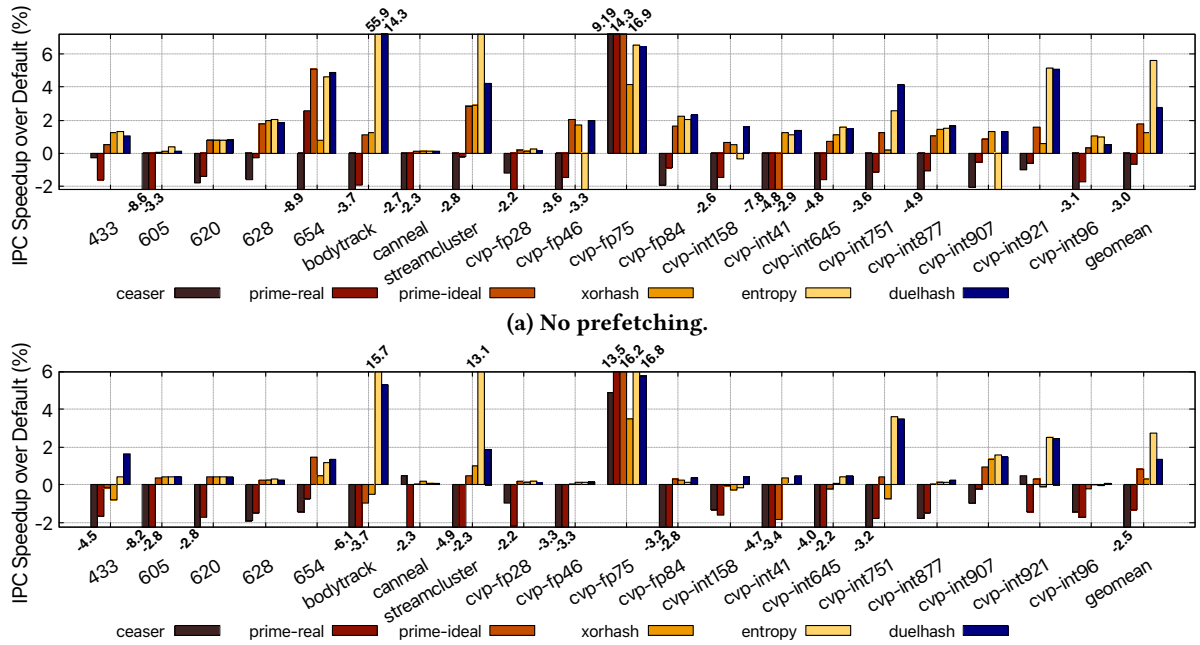


Figure 9: Geometric mean single-core performance gain of DUELHASH compared to other schemes in all applications in this study, grouped by benchmark suites.

4.4 Evaluation Results

4.4.1 Overall Single-Core. Figure 9 shows the IPC speedup of DUELHASH and the other indexing schemes with respect to the DEFAULT index function for single-core workloads. Without prefetching, DUELHASH improves the performance by 2.8%, outperforming both XORHASH and PRIME-IDEAL. The highest IPC improvement is 23%, achieved in *649.fotonik3d_s*. In the presence of hardware prefetching, DUELHASH improves the performance over DEFAULT by 1.0%, compared to a 0.4% improvement of XORHASH and 0.5% improvement of PRIME-IDEAL. With prefetching, the highest performance gain of DUELHASH is 5.8%, achieved in *cvp-fp75*.

PRIME index provides a positive improvement of 1.8% (no prefetching) and 0.5% (with prefetching) in the ideal case (PRIME-IDEAL), but a negative performance of −0.7% (no prefetching) and −1.7% (with prefetching) in the realistic case (PRIME-REAL). This result indicates the importance of hardware efficiency in designing a cache index hashing function. A complex hash function may even degrade the system performance even though it has improvement on paper.



(a) No prefetching.

(b) With prefetching.

Figure 10: Performance of DUELHASH compared to other schemes for non-uniform applications.

CEASER shows negative performance gain in almost all applications, with an average of −2.0% (no prefetching) and −2.5% (with prefetching). This is because CEASER changes the index functions randomly, without any consideration on performance. The Feistel cipher algorithm also incurs 2 extra cycles delay in the critical path, therefore degrading the overall performance.

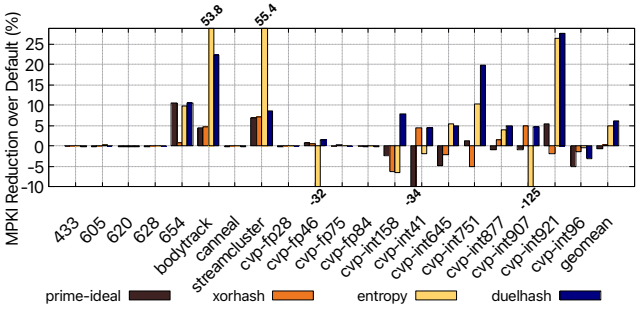
DUELHASH also provides a comparable performance to ENTROPYINDEX on most benchmarks. Without prefetching, DUELHASH outperforms ENTROPYINDEX in all benchmark suites except from PARSEC. With prefetching, DUELHASH loses to ENTROPYINDEX in PARSEC and cvp-fp applications, while presenting comparable performance in all other benchmark suites. DUELHASH achieves all these numbers while using 93% less power than ENTROPYINDEX.

4.4.2 Non-uniform Applications. Non-uniform applications are those that exhibit an uneven distribution of accesses among cache sets. Previous studies show that this type of applications benefits the most from sophisticated indexing schemes [18, 39, 40]. The balance of cache access distribution is qualitatively measured by the ratio: $stdev/mean$ of the cache accesses [18, 39, 40]. If this ratio is high, that means the cache access distribution is uneven, since the standard deviation is being far from the mean. In this work, we define non-uniform applications as those with $stdev/mean > 0.20$. With this definition, there are 20 out of 47 applications in the testing set are classified as *non-uniform*. Figure 10 shows the performance of our solution on the subset of non-uniform applications within our testing set.

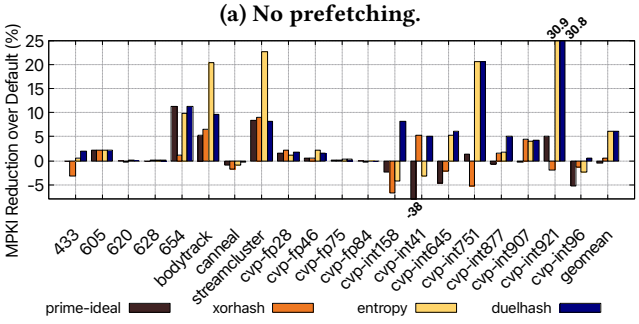
The simulation results show that DUELHASH outperforms PRIME and XORHASH for these applications, either with or without prefetching. In the no prefetching scenario, DUELHASH provides an IPC improvement of 2.7%, compared to a 1.2% improvement of XORHASH, a 1.7% improvement of PRIME-IDEAL, and a −3.0% degradation of CEASER. With prefetching, the mean speedup of DUELHASH is 1.3%, compared to a speed up of 0.3% of XORHASH, 0.8% of PRIME-IDEAL, and a −2.5% of CEASER. As oppose to prior findings, we generally do not see a clear benefit of using sophisticated indexing schemes for non-uniform applications in our experiments.

Compared to ENTROPYINDEX, DUELHASH presents a more consistent performance across different benchmarks, despite having a lower overall result. There are a few cases where ENTROPYINDEX significantly outperforms DUELHASH, skewing the overall result. In these applications, the functions formed by ENTROPYINDEX are completely different from the set of index functions in DUELHASH, highlighting the advantage of ENTROPYINDEX. ENTROPYINDEX has the full flexibility to pick individual address bits in both sequences of its XOR-based index function, theoretically resulting in better tailored index function for every applications. That said, DUELHASH has the power efficiency advantage. It performs comparably to ENTROPYINDEX in most of the applications while using 93% less power. Additionally, ENTROPYINDEX results in negative performance in a few cases. DUELHASH does not have any single-core performance degradation in all applications both with or without prefetching. This is because DUELHASH keeps the DEFAULT in the set of dueling index functions and

lets it compete normally with the others. As a result, for those applications that do not work well with advanced indexing schemes, DEFAULT would show better hashing conflict rate and get selected by DUELHASH, therefore avoid IPC degradation as well as unnecessary remapping traffic (Figure 14).

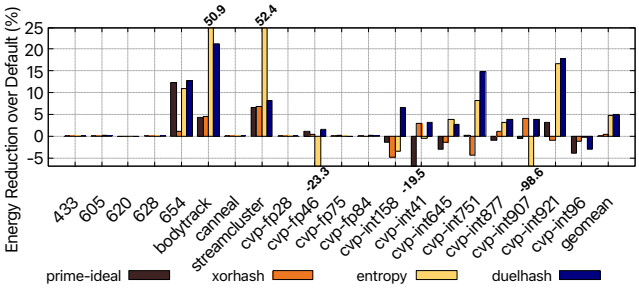


(a) No prefetching.

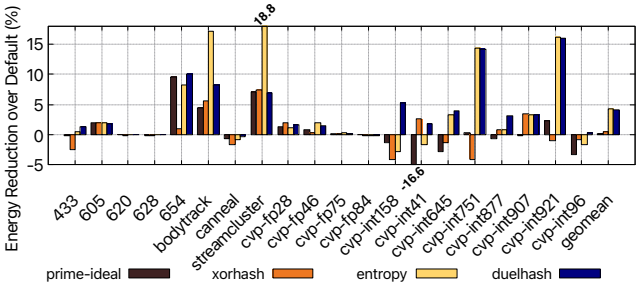


(b) With prefetching.

Figure 11: DUELHASH has the highest MPKI reduction for non-uniform applications.



(a) No prefetching.

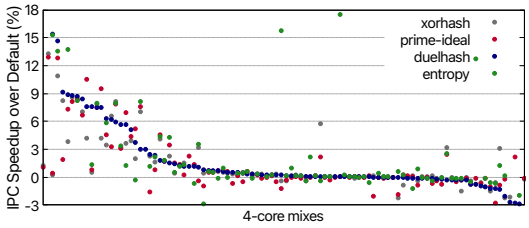


(b) With prefetching.

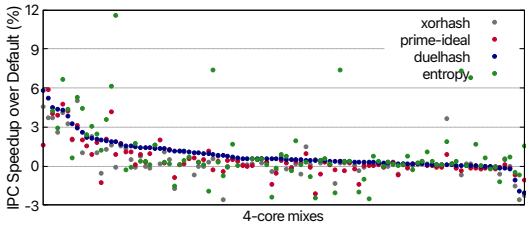
Figure 12: Uncore energy saving of DUELHASH compared to other schemes for non-uniform applications.

Figure 11 presents the MPKI reduction of different indexing schemes for non-uniform applications. DUELHASH gives the highest MPKI reduction at around 6.2% in both with and without prefetching scenario. DUELHASH numbers are on par to ENTROPYINDEX, and signifcantly higher than XORHASH and PRIME. These results show the huge potential of the dynamic indexing strategies compared to the conventional static indexing schemes for set-associative caches.

A lower MPKI in LLC also has a positive impact on the uncore energy consumption, since it reduces the DRAM traffic, as shown in Figure 12. On average, DUELHASH reduces uncore energy consumption by 4.9% (no prefetching) and 4.1% (with prefetching). These numbers are on the same level as ENTROPYINDEX. For XORHASH and PRIME, the energy savings are less than 1%. To measure the energy impact, we estimate each LLC access consumes 1 unit of energy and each DRAM access consumes 25 unit of energy. This approximation strategy has been used in prior work on cache replacement and indexing [30, 39].



(a) No prefetching.

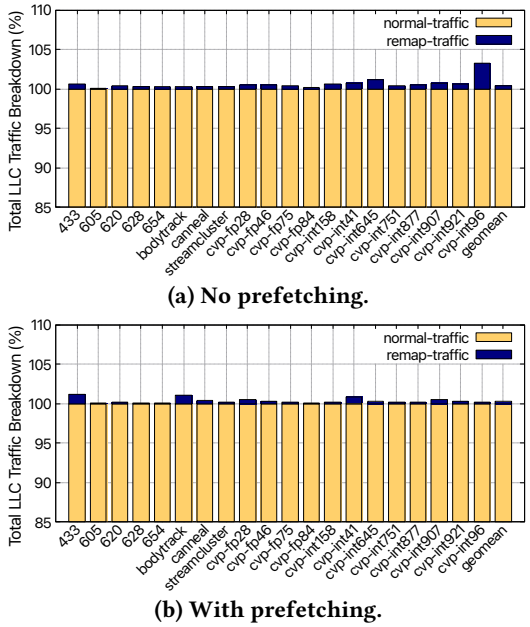


(b) With prefetching.

Figure 13: Performance comparison of different indexing schemes in the multiprogram workloads.

4.4.3 Overall Multi-Core. Figure 13 shows the performance of different indexing schemes in the 4-core setup. When there is no prefetching, DUELHASH improves the performance over the DEFAULT by 1.9% overall, while ENTROPYINDEX, XORHASH and PRIME-IDEAL improvements are 1.4%, 1.1% and 1.1%, respectively. With prefetching, the overall performance gains of DUELHASH is 0.9%, which is similar to ENTROPYINDEX. Meanwhile, the overall speed up of XORHASH and PRIME-IDEAL is 0.5%. ENTROPYINDEX has more control of selecting the bits, thus it usually finds a high performing index function for the mix. The highest speed up for ENTROPYINDEX is 42%, compared to 25% of DUELHASH. However, since the access stream in more complicate in a multi-program scenario, there is a higher risk of false positive. In some mixes,

EntropyIndex still finds the highest randomness index function, but the performance is significantly degraded, with the lowest being −28%, compared to −3.3% of Duelhash. Being less robust turns out to be an advantage for Duelhash in this scenario.



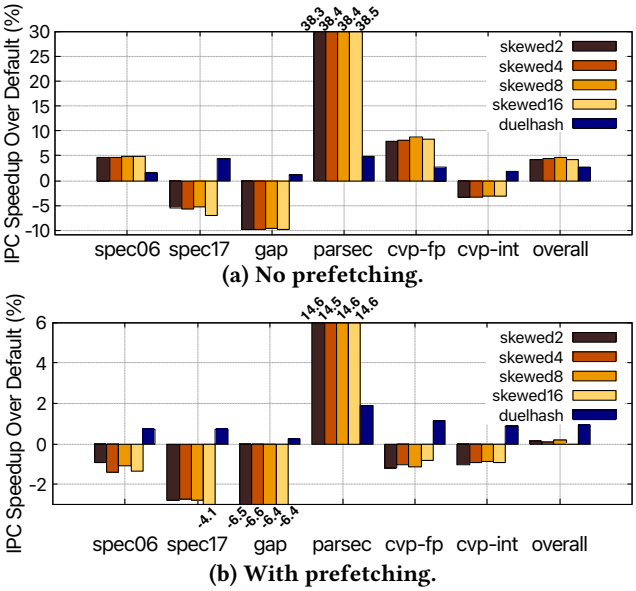
(a) No prefetching.

(b) With prefetching.

Figure 14: Extra LLC traffic cost due to cache remapping (%). LLC traffic is the sum of all LLC accesses of all types (demand/prefetch/writeback).

4.4.4 Remapping Cost. Figure 14 shows the impact of cache gradual remapping on the LLC traffic. We measure the extra cache reads and writebacks caused by Duelhash and divide by the total number of LLC accesses. On average, Duelhash only generates 0.5% extra traffic to the cache in the no prefetching scenario. When hardware prefetching is turned on, the total LLC traffic increases significantly because of the prefetcher's requests. Hence, the extra traffic due to remapping accounts for just 0.2% of the LLC normal traffic. The highest remapping cost is 3.3%, seen in cvp-int96. For some applications, this cost is zero because Duelhash does not change the index function for the entire execution. Setting the sensitivity threshold $\frac{CTC_i - CTC_{i+1}}{CTC_i} \geq 20\%$ helps reducing the number of unnecessary index function changes, therefore bringing down the total cost of remapping.

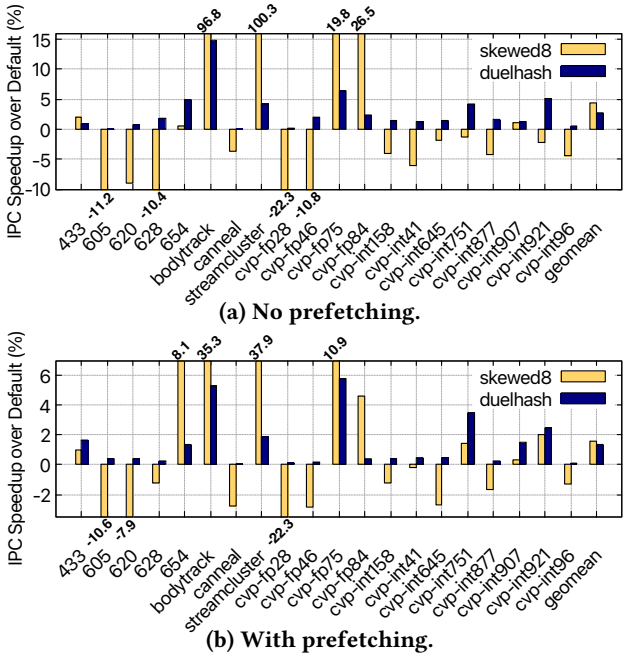
4.4.5 Compared with SkewedCache. Figure 15 show the performance of Duelhash against SkewedCache for all applications in this study. Varying the number of skewing functions does not have a big impact. The 8-skewing-function SkewedCache (*skewed8*) performs better than other skewing setups. SkewedCache is also unstable across different applications, particularly with hardware prefetcher. In the no



(a) No prefetching.

(b) With prefetching.

Figure 15: Geometric mean single-core performance comparison between a 16-way set-associative LLC with Duelhash and a 16-way SkewedCache with varied skewing functions across all applications in this study.



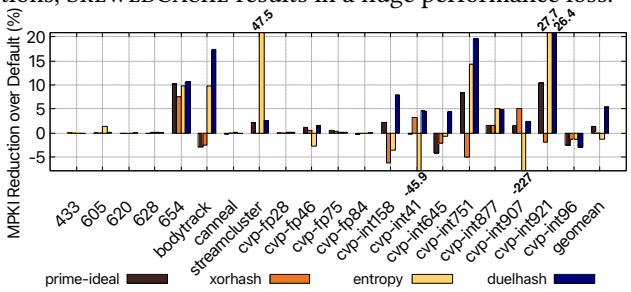
(a) No prefetching.

(b) With prefetching.

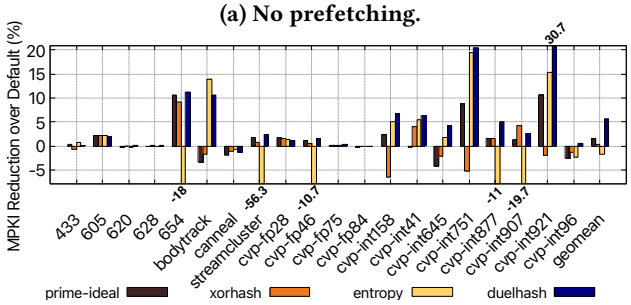
Figure 16: Performance of a 16-way set-associative LLC with Duelhash versus a 16-way SkewedCache (8 skewing functions) in non-uniform applications.

prefetching scenario, SkewedCache provides the maximum speed up of 4.5% (in *skewed8*). In the presence of hardware prefetching, the overall speed up of SkewedCache is 0.2% (also in *skewed8*). Conversely, Duelhash provides a more uniform IPC improvement.

Figure 16 shows the performance of DUELHASH and SKEWEDCACHE for the subset of non-uniform applications. We use *skewed8* as it is the highest performing skewing setup. Without prefetching, DUELHASH yields a mean speed up of 2.7% over DEFAULT, compared to a 4.4% speed up of SKEWEDCACHE. With prefetching, DUELHASH speed up is 1.3%, compared to a 1.6% improvement of SKEWEDCACHE. SKEWEDCACHE forms their index functions from a limited number of address bits (22 bits). DUELHASH expands to 32 bits, thus generating better functions. SKEWEDCACHE is also static scheme, lacking the capability of switching to a different index function to adapt to the running application. In both scenarios, the overall performance gain of SKEWEDCACHE is biased by the high speed up of 5 applications: *654.roms, bodytrack, streamcluster, cvp-fp75*, and *cvp-fp84*. In many other applications, SKEWEDCACHE results in a huge performance loss.



(a) No prefetching.



(b) With prefetching.

Figure 17: MPKI reduction in non-uniform applications with a 4-bank LLC setup.

4.4.6 DUELHASH for Banked LLC. We extended ChampSim to evaluate DUELHASH on a banked LLC. We use a 4-bank 2MB LLC setup (512KB per bank). As we could not get the simulator to fully support the parallel accessing to the banks, we only report the MPKI (Figure 17) instead of the IPC results. The banks are indexed using the lowest two bits of the block address. The set index within the bank is then computed using different indexing schemes that we are studying.

In this experiment, DUELHASH provides a 5.4% (no prefetching) and 5.6% (with prefetching) MPKI reduction compared to the baseline, slightly less than the monolithic setup (Figure 11). This is because there is less opportunity to optimize as the bank index function is static. XORHASH and ENTROPYINDEX exhibit the same behavior. For ENTROPYINDEX, we used the index formulation for banked LLC proposed in the original paper [39]. In that formulation, the bank index are excluded from the set index function. In our experiment, the two lowest address bits are the bank index bits. Excluding these bits causes a huge degradation in ENTROPYINDEX performance, because for most of the applications, these two bits are the highest entropy bits. ENTROPYINDEX could be tuned further for banked caches, but it is out of the scope of this study. PRIME-IDEAL shows a better performance in the banked setup compared to the monolithic setup. Most of the improvements come from the compute-int benchmarks. These benchmarks do not work well with PRIME originally. Thus having the lowest two bits as the bank index (effectively the DEFAULT index function) alleviates some negative impact of PRIME on these workloads.

5 Conclusion

Current set-associative cache indexing schemes have two major issues. First, there is no single index function that is guaranteed to perform well for every application. Second, advanced indexing schemes typically have sophisticated implementation and prohibitively long computation time.

To address these challenges, we propose DUELHASH, an efficient multi-way index dueling framework for set-associative caches. At a high level, DUELHASH is an adaptive indexing scheme that maintains a set of different index functions competing with each other at runtime to select the optimal index function for an application. We conducted a case study on a shared 16-way set-associative LLC to illustrate the efficacy of DUELHASH. In the study, we used a set of XOR-based index functions generated by the Perfect Shuffle operations for DUELHASH and evaluated their performance using a wide range of benchmarks, including SPEC 2006, SPEC 2017, PARSEC 3.0, CVP and GAP. Experimental results show that without prefetching, DUELHASH provides a mean IPC speed up of 2.8% (with the highest being 23%) over DEFAULT, compared to a 1.6% speed up of XORHASH. When pattern-based prefetchers present in the L1 data and L2 caches, DUELHASH can provide up to 5.8% single-core speedup over DEFAULT. DUELHASH also provides a 6.2% MPKI reduction over DEFAULT for non-uniform applications, the highest among all schemes tested in the study. The reduced LLC miss rate results in an uncore energy saving of 4.9% (without prefetching) or 4.1% (with prefetching). Compared to the state-of-the-art cache indexing scheme, DUELHASH offers a comparable performance in most applications while using 93% less power.

Acknowledgments

We thank the members of PALab group and the reviewers for valuable discussions. This work was supported in part by NSF Award No 2301334 and the TAMU HPRC center.

References

[1] Anant Agarwal, John Hennessy, and Mark Horowitz. 1988. Cache Performance of Operating System and Multiprogramming Workloads. *ACM Trans. Comput. Syst.* 6 (nov 1988), 393–431.

[2] Anant Agarwal and Stephen D Pudar. 1993. 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*. 179–190.

[3] Scott Beamer, Krste Asanović, and David Patterson. 2017. The GAP Benchmark Suite. In *arXiv*. arXiv:1508.03619

[4] Eshan Bhatia, Gino Chacon, Seth Pugsley, Elvira Teran, Paul V. Gratz, and Daniel A. Jiménez. 2019. Perceptron-Based Prefetch Filtering. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*.

[5] F. Bodin and A. Seznec. 1995. Skewed associativity enhances performance predictability. In *Proceedings 22nd Annual International Symposium on Computer Architecture (ISCA)*.

[6] J.Lawrence Carter and Mark N. Wegman. 1979. Universal classes of hash functions. *J. Comput. System Sci.* 18, 2 (1979), 143–154. doi:10.1016/0022-0000(79)90044-8

[7] CRC-2. 2017. *The 2nd Cache Replacement Championship*. http://crc2.ece.tamu.edu

[8] CVP-1. 2018. *The 1st Championship Value Prediction*. https://microarch.org/cvp1/

[9] Jeffrey R Diamond, Donald S Fussell, and Stephen W Keckler. 2014. Arbitrary modulus indexing. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 140–152.

[10] DPC-3. 2019. *The 3rd Data Prefetching Championship*. https://dpc3.compas.cs.stonybrook.edu

[11] Antonio González, Mateo Valero, Nigel Topham, and Joan M. Parcerisa. 1997. Eliminating Cache Conflict Misses through XOR-Based Placement Functions. In *ICS*.

[12] John L. Hennessy and David A. Patterson. 2017. *Computer Architecture, Sixth Edition: A Quantitative Approach* (6th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

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

[14] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. Systematic reverse engineering of cache slice selection in Intel processors. In *2015 Euromicro Conference on Digital System Design*. IEEE, 629–636.

[15] Akanksha Jain and Calvin Lin. 2016. 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, 78–89.

[16] Akanksha Jain and Calvin Lin. 2018. Rethinking belady's algorithm to accommodate prefetching. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 110–123.

[17] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*.

[18] Mazen Kharbutli, Keith Irwin, Yan Solihin, and Jaejin Lee. 2004. Using prime numbers for cache indexing to eliminate conflict misses. In *10th International Symposium on High Performance Computer Architecture (HPCA'04)*. IEEE, 288–299.

[19] Jinchun Kim, Seth H. Pugsley, Paul V. Gratz, A.L. Narasimha Reddy, Chris Wilkerson, and Zeshan Chishti. 2016. Path confidence based lookahead prefetching. In *MICRO*.

[20] Jinchun Kim, Elvira Teran, Paul V. Gratz, Daniel A. Jiménez, Seth H. Pugsley, and Chris Wilkerson. 2017. Kill the Program Counter: Reconstructing Program Behavior in the Processor Cache Hierarchy. *ASPLOS* (2017).

[21] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. 2020. 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 (ASIA CCS '20)*.

[22] Sun Microsystems. 2007. *UltraSPARC T2 supplement to the UltraSPARC architecture 2007*. Technical Report.

[23] Agustín Navarro-Torres, Biswabandan Panda, Jesús Alastruey-Benedé, Pablo Ibáñez, Víctor Viñals-Yúfera, and Alberto Ros. 2022. Berti: an Accurate Local-Delta Data Prefetcher. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 975–991. doi:10.1109/MICRO56248.2022.00072

[24] M. Qureshi, A. Jaleel, Y. Patt, S. Steely, and J. Emer. 2007. Adaptive Insertion Policies for High Performance Caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07)*.

[25] Moinuddin K. Qureshi. 2018. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 775–787. doi:10.1109/MICRO.2018.00068

[26] B. Ramakrishna Rau. 1991. Pseudo-Randomly Interleaved Memory. *SIGARCH Comput. Archit. News* 19, 3 (apr 1991), 74–83. doi:10.1145/115953.115961

[27] Daniel Sanchez and Christos Kozyrakis. 2010. The ZCache: Decoupling Ways and Associativity. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 187–198. doi:10.1109/MICRO.2010.20

[28] S. Sethumurugan, J. Yin, and J. Sartori. 2021. Designing a Cost-Effective Cache Replacement Policy using Machine Learning. In *2021 IEEE 27th International Symposium on High Performance Computer Architecture (HPCA)*. 291–303.

[29] André Seznec. 1993. A case for two-way skewed-associative caches. *ACM SIGARCH computer architecture news* 21, 2 (1993), 169–178.

[30] Ishan Shah, Akanksha Jain, and Calvin Lin. 2022. Effective Mimicry of Belady's MIN Policy. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

[31] Z. Shi, X. Huang, A. Jain, and C. Lin. 2019. Applying Deep Learning to the Cache Replacement Problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 413–425.

[32] Z. Shi, A. Jain, K. Swersky, M. Hashemi, P. Ranganathan, and Calvin Lin. 2021. A Hierarchical Neural Model of Data Prefetching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*. 861–873.

[33] H.S. Stone. 1971. Parallel Processing with the Perfect Shuffle. *IEEE Trans. Comput.* C-20, 2 (1971), 153–161.

[34] Synopsis. [n. d.]. Synopsys Design Compiler NXT. https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/design-compiler-graphical.html.

[35] Stephen J. Tarsa, Rangeen Basu Roy Chowdhury, Julien Sebot, Gautham Chinya, Jayesh Gaur, Karthik Sankaranarayanan, Chit-Kwan Lin, Robert Chappell, Ronak Singhal, and Hong Wang. [n. d.]. Post-Silicon CPU Adaptation Made Practical Using Machine Learning. In *ISCA 2019*.

[36] N. Topham, A. Gonzalez, and J. Gonzalez. 1997. The design and performance of a conflict-avoiding cache. In *Proceedings of 30th Annual International Symposium on Microarchitecture*. 71–80. doi:10.1109/MICRO.1997.645799

[37] John Tramm, Andrew Siegel, Tanzima Islam, and Martin Schulz. [n. d.]. XSBench - The development and verification of a performance abstraction for Monte Carlo reactor analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*.

[38] Georgios Vavouliotis, Gino Chacon, Lluc Alvarez, Paul V. Gratz, Daniel A. Jiménez, and Marc Casas. 2022. Page Size Aware Cache Prefetching. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[39] Kevin Weston, Avery Johnson, Vahid Janfaza, Farabi Mahmud, and Abdullah Muzahid. 2024. Customizing Cache Indexing Through Entropy Estimation. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[40] Kevin Weston, Farabi Mahmud, Vahid Janfaza, and Abdullah Muzahid. 2023. SmartIndex: Learning to Index Caches to Improve Performance. *IEEE Computer Architecture Letters* 22 (2023).

[41] Chuanjun Zhang. 2006. Balanced Cache: Reducing Conflict Misses of Direct-Mapped Caches. In *33rd International Symposium on Computer Architecture (ISCA'06)*. 155–166. doi:10.1109/ISCA.2006.12