Blast from the Past: Least Expected Use (LEU) Cache Replacement with Statistical History

Sayak Chakraborti[*]
University of Rochester
Rochester, NY, USA
schakr11@cs.rochester.edu

Zhizhou Zhang[*]
UC Santa Barbara
Santa Barbara, CA, USA
zhizhouzhang@ucsb.edu

Noah Bertram[†]
Cornell University
Ithaca, NY, USA
nbertram@cs.cornell.edu

Chen Ding
University of Rochester
Rochester, NY, USA
cding@cs.rochester.edu

Sandhya Dwarkadas[†]
University of Virginia
Charlottesville, VA, USA
sandhya@virginia.edu

Abstract

Cache replacement policies typically use some form of statistics on past access behavior. As a common limitation, however, the extent of the history being recorded is limited to either just the data in cache or, more recently, a larger but still finite-length window of accesses, because the cost of keeping a long history can easily outweigh its benefit.

This paper presents a statistical method to keep track of instruction pointer-based access reuse intervals of arbitrary length and uses this information to identify the Least Expected Use (LEU) blocks for replacement. LEU uses dynamic sampling supported by novel hardware that maintains a state to record arbitrarily long reuse intervals. LEU is evaluated using the Cache Replacement Championship simulator, tested on PolyBench and SPEC, and compared with five policies including a recent technique that approximates optimal caching using a fixed-length history. By maintaining statistics for an arbitrary history, LEU outperforms previous techniques for a broad range of scientific kernels, whose data reuses are longer than those in traces traditionally used in computer architecture studies.

CCS Concepts: • **Theory of computation → Caching and paging algorithms**; • **Computer systems organization → Processors and memory architectures**.

Keywords: Caching, Reuse Interval, Statistical history

[*]Equal contribution
[†]Research was conducted while at University of Rochester

ISMM '23, June 18, 2023, Orlando, FL, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0179-5/23/06.
https://doi.org/10.1145/3591195.3595267

ACM Reference Format:
Sayak Chakraborti, Zhizhou Zhang, Noah Bertram, Chen Ding, and Sandhya Dwarkadas. 2023. Blast from the Past: Least Expected Use (LEU) Cache Replacement with Statistical History. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management (ISMM '23), June 18, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3591195.3595267

1 Introduction

If future reuses can be predicted, the optimal solution to cache replacement, invented by Belady [4], is to replace the data block that will be reused furthest in time. The algorithm is also known as MIN [6] and OPT [21].

The phrase "History repeats itself" has been used for centuries to depict the recurrence of events across time, and has consistently been used for prediction and speculation in computer architecture. Many cache replacement policies were designed to emulate Belady by using history to predict the future. LRU, for example, predicts the order of future reuse based on the last access time. Many techniques have improved over LRU by using additional information in the history beyond the last access time. LIRS [15] uses the last reuse distance computed from the history of recent accesses to a data block. To emulate Belady, hardware techniques including protection distance [8], NUcache [20], and the Shepherd cache [27], record history for data stored in the cache. A recent technique, Hawkeye [11], showed superior performance by recording a window of history that is 8 times the *cache size* and using the history to more precisely emulate Belady.

While Hawkeye results in significant improvements over LRU, there are important reasons to explore alternative solutions:

- Hawkeye's time and space costs are linearly proportional to the length of its history window.
- Hawkeye's prediction accuracy is limited by the size of its history window. Important workloads have long-distance

reuses that may exceed the limit of any constant-size history window. In particular, scientific kernels such as tensor computations, which are an important component in widely used machine learning algorithms today, demonstrate such long reuses.

In this paper, we present a statistical approach to predicting probable future access times by recording access Reuse Intervals (RIs) associated with instruction pointers. Past observed information on reuse intervals is used to determine expected (rather than precise) future access intervals. A cache replacement policy can then victimize the data block with the farthest expected access (least expected use (LEU)) at eviction time.

LEU's statistical approach enables an implementation that is independent of the length of the history on which statistics are collected. LEU's association of reuse distributions with instruction pointers helps condense the metadata for multiple data blocks. To reduce implementation overhead, we leverage multiple techniques to retain statistics for the common case with finite and low space overhead. Intuitively, tracking the most frequent reuse interval can provide the best approximation of expected reuse in constant space. However, a naive implementation of keeping a record of all distinct intervals and sorting them requires $O(n \log n)$ time complexity in the worst case and $O(n)$ space, where n is the length of the history. Instead, we apply a streaming[1] algorithm[16] to reduce cost to linear time and constant space, $O(n)$ and $O(1)$ respectively.

We summarize our contributions below:

- We propose the Least Expected Use (LEU) cache replacement policy, which evicts the cache line expected to be accessed furthest in the future when replacement is necessary. The next access time for each cache line is predicted using the reuse interval distribution of past accesses associated with instruction pointers, resulting in statistical rather than precise predictions.
- We make LEU practical for hardware implementation using techniques that include a streaming algorithm to reduce the space and time overhead of tracking statistics to linear time and constant space, $O(n)$ and $O(1)$ respectively, where n is the length of the history.
- We evaluate LEU using the Cache Replacement Championship Simulator[18] using both crc2 [1] traces (which are generated from SPEC2006 [2] benchmarks) and PolyBench [25] (which is representative of important scientific kernels).
- We show that LEU's ability to capture long reuse intervals allows it to improve performance over both LRU and Hawkeye for applications that demonstrate such reuses.
- Comparing implementations, we eliminate on average ∼ 20% of LRU's MPKI and ∼ 6% of Hawkeye's MPKI for PolyBench [25].

[1]Note: LEU uses a streaming technique to store RI distribution, this is different than the streaming access patterns that benchmarks have.

Section 2 describes the new replacement policy. Section 3 and Section 4 present the hardware design and evaluation, respectively. We discuss related work in Section 5 and summarize with the discussion of future work in Section 6.

2 LEU Replacement

2.1 Notations

The abbreviation and notation used in the paper can be found in Table 1.

Table 1. Abbreviations and notations used in the paper.

Notation	Meaning
b	A particular cache block
t	Current logical time
$NAT_b(t)$	Next Access Time of block b relative to time t
$LAT_b(t)$	Last Access Time of block b relative to time t
$NAD_b(t)$	Next Access Distance: $NAT_b(t)$ - t
$tesla_b(t)$	t - $LAT_b(t)$

2.2 Least Expected Use

The LEU policy leverages instruction pointer (IP) reuse interval (RI) statistics to select a victim victim. More specifically, LEU requires as input reuse interval distributions for each instruction pointer.

LEU incorporates two aspects of dynamic access behavior into its prediction of future accesses for each block in the cache.

- **Likelihood dynamism** The elapsed time, i.e. *tesla*, naturally rules out shorter RIs so in predicting the next reuse, LEU considers only RIs longer than the current *tesla*.
- **Last-reference-point dynamism** A data block may be accessed by multiple reference points (instruction pointer (IP) or program counter making the access). LEU uses the RI distribution of the last reference point.

The next-access prediction by LEU, shown in Eq. 1, incorporates the likelihood dynamism by calculating the prediction using only the RIs longer than *tesla*, and the last-reference-point dynamism by using the RI distribution of the *last* reference point that accessed b. In order to accomplish the latter, we associate *RI* distributions with *instruction pointers* (IP). We keep track of the IP that last accessed a data block and predicts its reuse behavior by the *RI* distribution of this IP.

We now define the policy. Provided with a reuse interval (RI) distribution for each instruction pointer, denoted P_{ip} for instruction pointer ip, the cache will victimize a cache block with the largest expected next access distance (NAD). The formula for expected NAD for data block b accessed with instruction pointer ip at logical time t is

$$\mathbb{E}(NAD_b(t)) = \sum_{\substack{ri \in \text{ris}(P_{\text{ip(b)}}),\\ ri > tesla_b(t)}} \left(P_{\text{ip(b)}}(ri) \cdot (ri - tesla_b(t)\right) \quad (1)$$

where $ip(b)$ is the instruction pointer from which the last access to b was made, and $\text{ris}(P_{\text{ip(b)}})$ is the set of RIs with support in the distribution $P_{\text{ip(b)}}$. For more formalism, see the appendix.

To illustrate the workings of the policy, Listing 1 shows an example program. The loop nest computes a 5-point stencil. For ease of presentation, we simplify the problem as follows. In this example, we consider data reuses at element granularity instead of block granularity. There is just one level of cache. Finally, all a accesses happen in cache, i.e., we do not use registers.

```
double a[1026][1026], b[1026][1026];
for (i = 1; i <= 1024; i++) {
   for (j = 1; j <= 1024; j++) {
      b[i][j] = a[i][j]+a[i][j+1]+a[i][j-1]
                +a[i-1][j]+a[i+1][j];
   }
}
```

Listing 1. 5-point stencil program

Examining the program code, it can be inferred from the inner loop that an array element is first accessed by reference $a[i][j+1]$, reused one iteration later by $a[i][j]$ and then another iteration later by $a[i][j-1]$. The cache should keep the data element for two iterations and evict it after the last access.

LEU captures the equivalent knowledge through last-reference-point dynamism. Table 2 shows the RI distribution for these references. For the first access, $a[i][j+1]$, the RI is 5. For the last access, $a[i][j-1]$, the RI is 6139 (the next reuse is $a[i-1][j]$ in the next i iteration).

Table 2. The RI distributions $P(ri)$ for 4 references in the stencil (assuming element granularity to simplify presentation)

a[i][j]		a[i][j+1]		a[i][j-1]		a[i+1][j]	
ri	$P(ri)$	ri	$P(ri)$	ri	$P(ri)$	ri	$P(ri)$
8	99.9%	5	100%	6139	100%	6135	99.9%
6147	0.1%	-	-	-	-	6140	0.1%

The example also shows how likelihood dynamism improves prediction accuracy. Consider the two RIs of $a[i][j]$. The last element out of every row of the array, i.e. $a[i][1024]$ is reused toward the end of the j-loop in the next i-loop. After its access, LEU uses both RIs and predicts reuse (in 8 accesses). This first prediction is wrong because the next reuse is $a[i-1][j]$ (not $a[i][j-1]$ as is the case for the other elements in the same row). After 8 accesses, however, the next prediction is correct by using only the longer RI 6147 (based on likelihood dynamism).

Likelihood dynamism becomes more important when we consider cache-block granularity. A 64-byte cache block contains 16 4-byte numbers. In a contiguous traversal, the RI distribution has a short reuse for 15/16 of all accesses and a long reuse for 1/16 of the accesses. With likelihood dynamism, LEU keeps the data block in cache for the duration of the (spatial) reuse and evicts it afterward. It is important to note the use of RIs of the data blocks referenced by an IP. The usage of IP-based cache block RI tracking helps reduce the space overheads which would have been expensive otherwise.

3 Hardware Design

In this section, we explain the mechanism of Least Expected Use (LEU) cache replacement policy and its hardware implementation.

3.1 LEU Implementation

3.1.1 LEU Data. LEU requires two main tables described below and additional metadata with the cached data:

- **Last Access Time Table (LATT)** stores a set of sampled data blocks, which may or may not be in the cache, and is used to record information that allows prediction of reuse intervals. For each data block, LATT stores the data block address, its last instruction pointer (IP), and the last access time (LAT). Time is based on logical time, implemented as a counter incremented on every access, and is used as the basis for determining the Reuse Interval (RI).
- **Reuse Interval Table (RIT)** stores the RI distributions for instruction pointers (IPs). A distribution is a histogram represented by a set of (RI, Freq) pairs, where Freq is the number of times the RI has been added (observed). It is used to calculate the expected reuse. Each entry in the RIT consists of the instruction pointer and multiple pairs of RI and their respective Freq.
- **Metadata per Cache Block** stores the last instruction pointer (IP) and the last access time (LAT). The IP is required to use the RI histogram in RIT, and LAT is required to calculate *tesla* (time elapsed since the last access).

3.1.2 LEU Update. The LATT records the time of the latest access (LAT) and the instruction pointer (IP) for sampled cache accesses (both HITS and MISSES). On a hit in the LATT, the stored LAT, along with the current logical access time, is used to calculate the RI, and the previous IP (stored in the LATT) is used to insert the calculated RI into the RIT. Both the IP and LAT are then updated to the current values. On a miss in the LATT, a new entry for the cache block address is

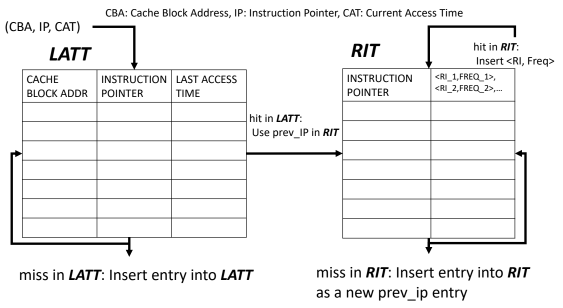
CBA: Cache Block Address, IP: Instruction Pointer, CAT: Current Access Time
(CBA, IP, CAT)
LATT
CACHE BLOCK ADDR
INSTRUCTION POINTER
LAST ACCESS TIME
hit in LATT: Use prev_IP in RIT
RIT
INSTRUCTION POINTER
<RI_1,FREQ_1>, <RI_2,FREQ_2>,…
hit in RIT: Insert <RI, Freq>
miss in LATT: Insert entry into LATT
miss in RIT: Insert entry into RIT as a new prev_ip entry

Figure 1. LEU update operation flow (Algorithm 1)

inserted into the LATT using the current IP and logical access time. Additionally, the metadata in each cache metadata block (IP, LAT) is also updated. While the LATT has similar capabilities of storing IP and LAT as the cache metadata block, it also contains information about cache lines that are currently not in the cache. This is useful for calculating RI of cache addresses that have not been in cache continuously and helps to keep track of a more accurate (and longer) RI history. Figure 1 depicts a basic flow diagram of an LEU Update operation and Algorithm 1 presents the pseudo-code for the update operation.

Algorithm 1 LEU Update

```
procedure LEU_Update(LATT, RIT, addr, IP, clock)
    if Cache hit then
        update LAT and IP
        update MRU
    end if
    if access beyond sampling rate then
        return
    end if
    if addr ∉ LATT then
        LATT.insert({addr, ip, clock})
        return
    else
        RI = clock - LATT[addr].LAT
        prev_ip = LATT[addr].ip
        update LATT[addr] = {ip, clock}
    end if
    if prev_ip ∉ RIT then
        ri_freq = {RI, 1}
        RIT.insert({prev_ip, ri_freq})
    else
        if RI ∈ RIT[prev_ip] then
            update Freq of RI by 1
        else
            RIT[prev_ip].insert({RI, 1})
        end if
        call KSP(RIT[prev_ip])
    end if
end procedure

procedure KSP(RIT, prev_ip)
    if RIT[prev_ip].size > HIST_SIZE then
        for entry ri_freq ∈ RIT[prev_ip] do
            ri_freq.Freq = ri_freq.Freq - 1
            if ri_freq.Freq ≤ 0 then
                RIT[prev_ip].remove(ri_freq)
            end if
        end for
    end if
end procedure
```

Algorithm 2 LEU Eviction

```
procedure LEU Victim(RIT, set, clock)
    for way ∈ valid ways do
        IP = block[set][way].ip
        LAT = block[set][way].la
        tesla = clock - LAT
        if IP ∈ RIT then
            for entry RI ∈ RIT[IP] and RI > tesla do
                exp_dist += RI × Freq[RI]
                cnt += Freq[RI]
            end for
            if cnt ≠ 0 then
                exp_dist=(exp_dist/cnt)-tesla
            end if
        else
            exp_dist=tesla
        end if
    end for
    victim_way = Max(exp_dist) way
    call MRU_trend(victim_way)
end procedure

procedure MRU_trend(victim_way)
    if victim_way == mru_way then
        trend_count = trend_count + 1
        if trend_count ≥ SET LIMIT then
            Use MRU victim way
        end if
        if trend_count ≥ UNSET LIMIT then
            trend_count = 0
        end if
    else
        confirm_count = confirm_count + 1
        if confirm_count ≥ CONFIRM LIMIT then
            confirm_count = 0
            trend_count = 0
        end if
    end if
end procedure
```

3.1.3 LEU Victimization.

When replacement in the cache is necessary, the least-expected-use way, or equivalently, the maximum-expected-distance way will be evicted. The intuition is to evict the way with the furthest expected reuse based on statistics from the past.

We calculate *tesla* for each candidate block in the cache set using the LAT stored in the block's metadata. We use the IP stored in each cache block metadata to look up the RIT and use the RI distribution in conjunction with *tesla* to calculate the Expected Next Access Distance (ENAD) using Eq. 2. When all the ENAD for all the ways has been calculated, the way with the highest ENAD will be evicted from the cache. Figure 2 depicts the operation flow of eviction and Algorithm 2 describes the eviction mechanism.

When a reference point has no history data either because the instruction has not been executed or because it has been removed from the RIT due to its finite size, we are unable to compute ENAD. If we are unable to compute ENAD, we fall back to *tesla*. We compare *tesla* with the expected reuse from other ways and victimize the block with the largest expected reuse distance. When a comparison is made only based on *tesla*s then it is equivalent to LRU and hence ensures that we perform at least as well as LRU in such scenarios. The eviction procedure is given in Algorithm 2.

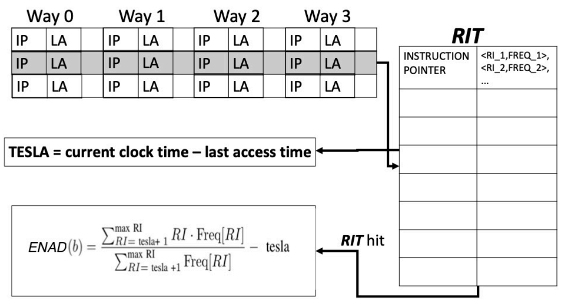
Way 0
Way 1
Way 2
Way 3
IP
LA
RIT
INSTRUCTION POINTER
<RI_1,FREQ_1>, <RI_2,FREQ_2>, ...
TESLA = current clock time – last access time
$ENAD(b) = \frac{\sum_{RI=\text{tesla}+1}^{\text{max RI}} RI \cdot \text{Freq}[RI]}{\sum_{RI=\text{tesla}+1}^{\text{max RI}} \text{Freq}[RI]} - \text{tesla}$
RIT hit

Figure 2. LEU eviction operation flow (Algorithm 2)

$$\text{ENAD}\ (b) = \frac{\sum_{\text{RI}=\text{ tesla}+1}^{\text{max RI}} \text{RI} \cdot \text{Freq}[\text{RI}]}{\sum_{\text{RI}=\text{ tesla}+1}^{\text{max RI}} \text{Freq}[\text{RI}]} - \text{tesla} \tag{2}$$

3.1.4 MRU Trend. When applications exhibit streaming behavior, they would benefit from the default use of MRU replacement rather than LRU replacement when LEU is not possible. We introduce a single MRU trend bit for the entire cache, which is set when streaming behavior is detected. In order to determine when to set the MRU trend bit, we use a Most-Recently-Used (MRU) bit per cache block to keep track of the MRU way. We also have an "MRU evict count" counter for the entire cache that is incremented when the evicted way matches the MRU way upon victimization. When "MRU evict count" reaches the $SET\ LIMIT$ threshold, the MRU trend bit is set. When the "MRU evict count" reaches the $UNSET\ LIMIT$ threshold, the MRU evict count is reset, as is the MRU trend bit, in order to reevaluate the application's need for MRU. Table 3 shows the different MRU threshold configurations used.

Table 3. MRU configurations

Config	SET LIMIT	UNSET LIMIT	CONFIRM LIMIT
MRU1	5000	10000	None
MRU2	18000	20000	100
MRU3	20000	21000	100

In order to escape out of MRU trend mode early, another counter called "confirm count" is incremented when in MRU trend mode and the evicted (LEU) way is not equal to the MRU way. When the value reaches the $CONFIRM\ LIMIT$ threshold, both counters and "*MRU trend*" are reset.

The victim way is selected as follows:

$$victim\ way = \begin{cases} LEU\ \ way, & \text{if } \exists \text{ b, for which } ENAD(b) \text{ exists,} \\ & \text{ where b is a cache block} \\ LRU\ \ way, & \text{if } \nexists \text{ b, for which } ENAD(b) \text{ exists,} \\ & \text{and } \textit{MRU trend} \text{ has } \textbf{NOT} \text{ been set} \\ MRU\ \ way, & \text{if } \nexists \text{ b, for which } ENAD(b) \text{ exists,} \\ & \text{and } \textit{MRU trend} \text{ has been set} \end{cases}$$

3.2 Keeping the Histogram Small

RIT is responsible for storing the reuse interval histogram specific to a reference. To reduce the space overhead we only keep track of the most frequent RIs. In fact, LEU keeps track of a history of any length with a constant space cost. We use a modified version of the algorithm given by Karp, Shenker, and Papadimitriou [16], which we refer to as the KSP algorithm.

3.2.1 KSP Algorithm. The KSP algorithm is an online algorithm for identifying elements that occur in the stream at a rate that is higher than a specified rate. For a general case, let θ be the threshold and the goal to find all the elements whose occurrences are over θ. It is easy to see there will be $1/\theta$ such elements at most, so we only need a size of $1/\theta + 1$ dictionary d. With the arrival of every element, we check whether the element is in d or not. If yes, we increase the count of occurrence by one; otherwise, we insert it into d with count 1. Then we check whether the size of d is beyond the threshold. If yes, the occurrence count will decrease by one for every element in the dictionary and the entry will be erased if the count reaches zero. Therefore, we can detect frequent symbols efficiently in both space and time.

3.2.2 Our Modifications. We modify the algorithm to directly keep a fixed number of histogram sizes (*HIST_SIZE*) instead of a relative rate, shown in line 30 of Algorithm 1.

The array starts empty, then iterates through the stream. If the current RI is not in the array, it is added with a counter of 1 in line 24. If the current RI is in the array, the RI's counter is increased by 1 in line 22. When the size of the histogram is larger than the threshold in line 30, we decrease the counter of every RI entry by one (lines 31-32). When the counter of a RI reaches zero, we remove that RI entry from the histogram (lines 33-34). This linear algorithm allows us to identify and store the top hist_size-th frequent $RI's$ for every IP entry in RIT while limiting space overhead.

We also apply approximation on the RIs to further reduce space overhead. Rather than recording the exact RI, values are stored within the closest bin given a fixed bucket size. For example, if the bucket size is 5, the RIs of 7 and 10 will be collocated with 8. Based on our evaluation, the resulting approximation of the ENAD calculation is not likely to have a significant impact on the selection of a victim.

4 Evaluation

We evaluate LEU on the Cache Replacement Championship simulator (CRC [1]). CRC simulates a 4-wide out-of-order processor with an 8-stage pipeline, a three-level cache hierarchy, and a 128-entry reorder buffer. The configuration parameters are shown in Table 4. We choose 50 million instructions as warmup and measure the performance of another 250 million instructions. We compare against LRU and other state-of-the-art replacement policies: Hawkeye [11], SRRIP, DRRIP [13], and SHiP [30]. We change the ways from 16 to 8 ways for a fair comparison.

Table 4. Baseline Configuration

L1 I-Cache	32 KB 4-way 1-cycle latency
L1 D-Cache	32 KB 4-way 1-cycle latency
L2 Cache	256KB 8-way 10-cycle latency
Last-level Cache	2MB 8-way 20-cycle latency
DRAM	200 cycles

The benchmarks we test include **PolyBench** [10, 26] and **SPEC2006** traces that come with the 2nd Cache Replacement Championship (CRC2) [1]. We use Intel PIN tool version 3.11 [19] to collect the traces for PolyBench. The SPEC2006 traces are collected on an Intel XEON E5-2420 CPU with 16 registers. According to CRC2, the traces are collected using the following command for perl, one of the benchmarks: "valgrind –tool=exp-bbv –interval-size=1000000000 –bb-out-file=perlbench_bbv.out –pc-out-file=perlbench_bbv.pc.out ./perlbench -Ilib checkspam.pl 2500 5 25 11 150 1 1 1 1"[2]. PolyBench is collected on an i7-3960X CPU with 16 registers and the compile command is: "g++ -std=c++11 -g -O3".

PolyBench is a collection of polyhedral benchmarks for scientific computing. We include all tests in the current version except for 2mm and doitgen (for which we were not able to collect traces). We also include a test from an earlier version of Polybench, convolution_3d, because it is the only case that our technique shows a degradation.

Modern hardware cache studies use trace samples with a limited length, e.g. 2 billion instructions, because of the high cost of architectural simulation. The methodology, however, imposes a limit on the type of reuses we see in these traces. In particular, we see only reuses no longer than the length of the trace sample. We call this the *simulation bias* for cache studies. As a result, past techniques may not fully address the problem of long reuses, while LEU is designed to use (limited statistics of) an unlimited history. PolyBench allows us to examine these techniques without the potential simulation bias.

Section 4.1 outlines the different implementation configurations we evaluate. Section 4.2 investigates the reuse interval characteristics of PolyBench and CRC2 and then explain some insights from our evaluation. We compare LEU's relative performance in Section 4.3 and show that LEU can detect and capture longer *RI*s and access pattern which include complicated stride patterns and global streaming patterns better than other replacement policies and hence outperform them in such scenarios (PolyBench). We analyze the space overheads introduced by this policy in Section 4.4. Sections 4.5 and 4.6 analyze the additional power, area, and overall (including computational) latency introduced by LEU.

[2]Complete information can be found at: https://crc2.ece.tamu.edu/

Section 4.7 evaluates the impact of providing LEU's additional metadata storage as excess data storage capacity in the LRU cache. We also analyze the sensitivity of our results to different parameters configurations in Section 4.8.

4.1 Configurations

We use multiple LEU configurations in our experiments to provide a holistic view of how changes in parameters and dimensions affect the performance of our policy. The configurations depicted in Table 6 read in conjunction with Table 5 give us an idea about the size of entries in each configuration and the size of structures and metadata used.

Our configurations can be segregated based on dimensions of precision (size of entries, for example, IP, LAT) for the LATT, RIT, and per cache-line metadata record, the number of entries in the LEU structures, and the MRU capabilities. The configurations can be categorized based on the precision due to assigned storage overheads of the LATT, RIT, and Metadata components as depicted by overheads O1, O2, and O3. Thus, we showcase the impact of the various dimensions (sometimes orthogonal) of the LEU policy and how it impacts performance.

Table 5. Structures size overhead, CBA = Cache block address, HIST_SIZE = 1

Overhead	Structure	Entry (Bytes)	Entry Size (Bytes)
O1	LATT	$< CBA(4), IP(4), LAT(8) >$	16
	RIT	$< IP(4), (HIST\_SIZE * (RI(2), Freq(1))) >$	7
	Metadata	$< IP(4), LAT(8) >$	12
O2	LATT	$< CBA(4), IP(1), LAT(3) >$	8
	RIT	$< IP(1), (HIST\_SIZE * (RI(2), Freq(1))) >$	4
	Metadata	$< IP(1), LAT(3) >$	4
O3	LATT	$< CBA(4), IP(1), LAT(3) >$	8
	RIT	$< IP(1), (HIST\_SIZE * (RI(2), Freq(1))) >$	4
	Metadata structure	$< IP(1), LAT(3), way + set(2), lru(2) >$	8

Table 6. LEU Configurations, RIT, LATT, and *HIST* (*HIST_SIZE*) are the number of entries, Sampling Ratio = 100% for all configurations, for MRU config refer table 3.

Configuration	RIT	LATT	*HIST*	Bucket	Overhead	Total Table Size	Metadata overhead
LEU config1	256	64	1	1	O1	2.75 KB	384KB
LEU config2-MRU1	256	64	7	10	O1	7.25 KB	384KB
LEU config4	256	64	1	1	O2	1.5KB	128KB
LEU config5-MRU3	256	64	1	1	O2	1.5KB	128KB
LEU config6	512	128	7	10	O2	12KB	128KB
LEU config7	256	64	1	1	O3	1.5KB	8KB

4.2 Reuse Interval and Access Pattern Characteristics

We use two methods to understand the characteristics of PolyBench and CRC2 (SPEC2006) and their impact on our replacement policy. First, we break down the Reuse Interval of PolyBench and CRC2 benchmark suites. Computation of *RI* in our policy is based on the LAT entry in LATT (Algorithm 1). Figures 3 and 4 show the breakdown of *RI*s gathered from

LLC accesses. From the figures, it is clear that there are more benchmarks in PolyBench that have longer RI as a percentage of occurrence (dominant) compared to SPEC2006. There are at least 8 benchmarks that have 60%+ non-shortest RI (longer than 9) in PolyBench (Figure 4) compared with only 2 in SPEC2006 (gcc_56B and povray_711B in Figure 3). In particular, roughly all the *RI*s for *gemm* is the longest group (190+). On the other hand, the majority *RI*s of SPEC2006 are extremely short. Most of the programs have more than 90% short (0-9) *RI*s. LEU as a policy is able to capture the more dominant longer *RI*s (apart from dominant shorter *RI*) and make accurate predictions. Using these predictions, LEU helps in reducing the MPKI and thus improves performance for PolyBench compared to other policies.

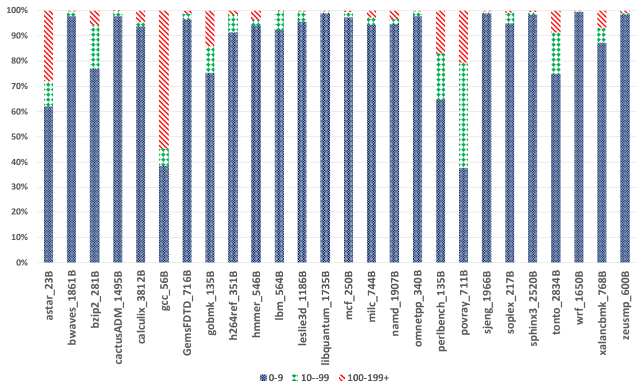

Figure 3. RI breakdown on CRC2 (SPEC2006).

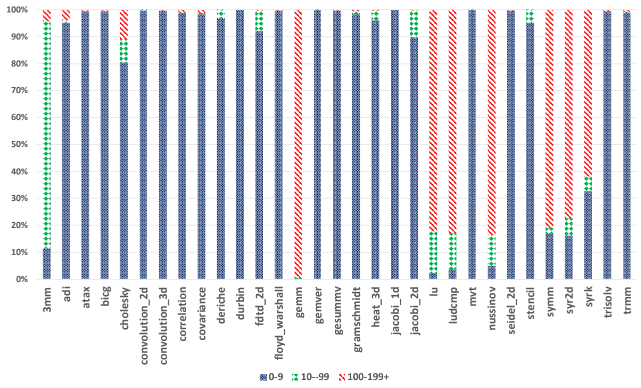

Figure 4. RI breakdown on PolyBench.

Second, we analyze the access pattern with instruction pointer (IP) based classification on most of the benchmarks from the two suites[3]. We adopt the techniques from Bouquet [23] , which is the state-of-the-art IP-classifier based prefetcher. Figure 5 and 6 show the IP classification results on LLC. It is clear from the figures that the memory behavior of CRC2 (SPEC2006) is more complicated, including much higher portion of unclassified and complex stride patterns.[4] An instruction is classified as complex stride if it has multiple frequent strides. Such pattern, however, can be recognized by LEU since it captures the most frequent reuse intervals. In comparison, PolyBench contains a greater portion of regular accesses including constant stride and global streaming. LEU captures these patterns by identifying the most frequent reuse intervals and is able to outperform LRU.

[3]We couldn't get all traces run with the tool

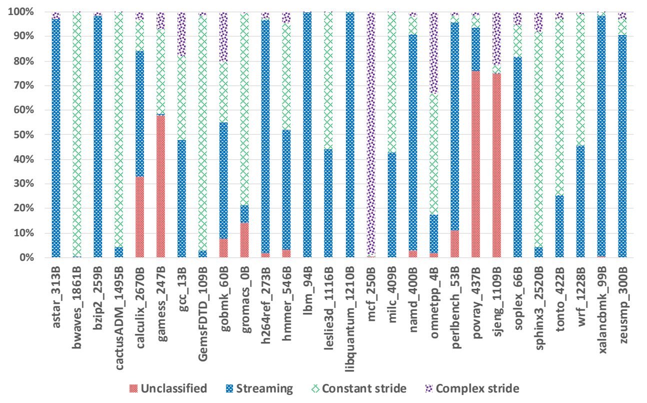

Figure 5. Access pattern breakdown on CRC2 (SPEC2006).

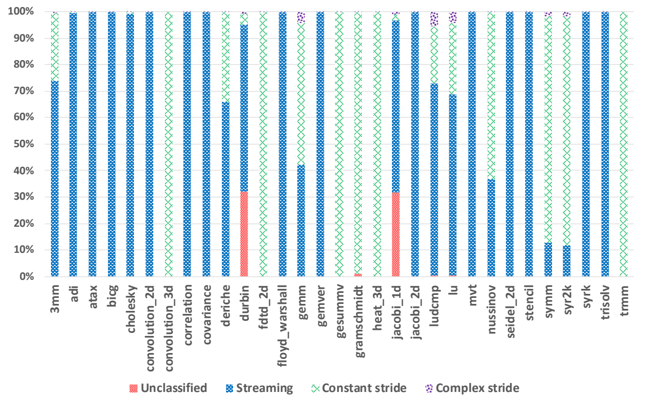

Figure 6. Access pattern breakdown on PolyBench.

4.3 Relative Performance

In this section, we discuss the performance relative to LRU (on single-core simulation) of state-of-the-art replacement policies (including Hawkeye, SRRIP, DRRIP, and SHiP) and LEU on CRC2 (SPEC2006) and PolyBench. We use "LEU-config5-MRU3" for our comparison against other replacement policies. Details about this configuration can be found in Table 6 read in conjunction with Tables 5 and 3. Table 7 compares the overheads of the policies. We observe

[4]Complex strides can be a mixed stride pattern like 1, 2, 1, 2, 1, 2. A simple constant stride cannot cover all the cases. For the details about each IP class, please refer to the Bouquet paper.

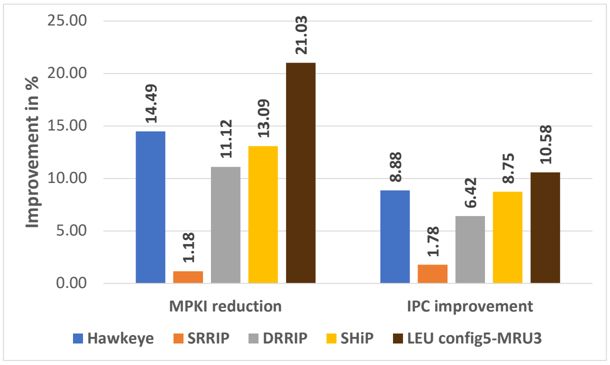

Figure 7. Geometric Means on PolyBench over LRU.

that LEU performs much better on PolyBench compared to SPEC2006. LEU reduces MPKI by up to ~75% for nussinov in PolyBench traces over LRU and improves IPC up to ~71% for ludcmp in the same benchmark suite. We can reduce MPKI for CRC2/SPEC2006 up to ~58% and improve IPC by ~17% for sphinx3, but with a higher overhead LEU configuration. We show the geometric means for both MPKI reduction and IPC improvement for both PolyBench and CRC2/SPEC2006 in Figures 7 & 8 respectively.

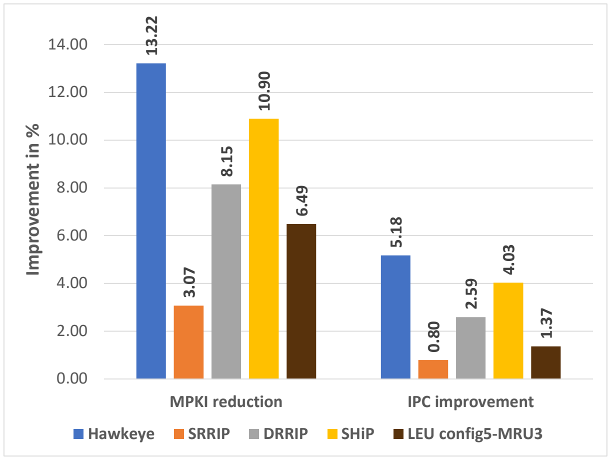

Figure 8. Geometric Means on SPEC2006 over LRU.

LEU has a better performance for benchmarks with longer dominant *RI*s. For lu, ludcmp, mvt, symm, and syr2d in PolyBench, LEU can generate 58%, 72%, 56%, 14%, and 19% IPC improvement and 64%, 63%, 15%, 14% and 24% MPKI reduction correspondingly. It is clear that LEU performs well over LRU on PolyBench and generates considerable improvement on SPEC2006. Combining the performance on both benchmark suites, LEU can achieve a similar performance compared with the state-of-the-art replacement policies, generating about 14% MPKI reduction and 6% IPC improvement. We show the combined geometric means for PolyBench and SPEC2006 in Figure 9. In summary, LEU's performance is comparable to that of Hawkeye and other state-of-the-art cache replacement policies across both benchmark suites, and can outperform in specific workloads like PolyBench.

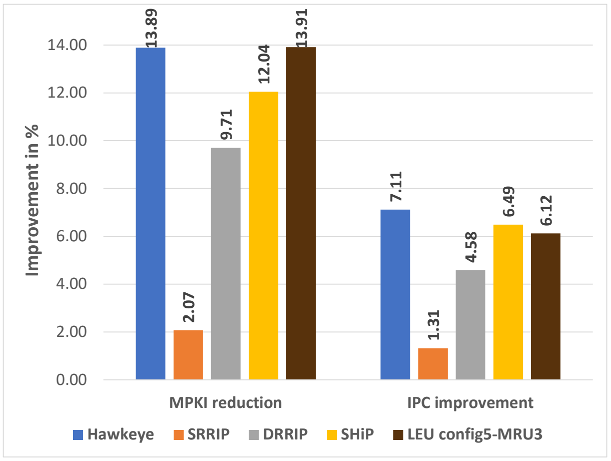

Figure 9. Geometric Means on SPEC2006 and PolyBench over LRU.

4.3.1 LEU Performance Discussion.

The filtering effect by higher-level caches most of the time eliminates small reuse intervals. LEU is able to capture long reuse intervals at the LLC and predict those accesses well. We don't get similar performance benefits for the SPEC2006 traces, because of their short reuses[8], and the phase and other temporal variation in access, which is difficult to capture at the LLC with constant-size data structures. For SPEC2006 traces, evicted lines have a very low reuse percentage at LLC[12]. Thus it is quite evident that the outer reuse for these benchmarks doesn't exist. As discussed at the beginning of this Section 4, this may be a result of sampling bias.

Most PolyBench benchmarks have $O(n^2)$ and $O(n^3)$ memory operations[33]. For example, *gemver*, a kernel from the basic linear algebra subprograms (blas), does multiple matrix-vector multiplications, *lu* performs lower and upper triangular decomposition of a matrix without pivoting. *ludcmp* is a solver for a system of linear equations that uses *lu* for decomposition. *mvt* is a matrix-vector multiplication kernel that uses two matrices, one of which is transposed. *nussinov* is used for predicting RNA folding and uses dynamic programming with data being stored in tables that are computed based on adjacent cells. All of these benchmarks have the commonality of either matrix multiplication operations, or vector operations and use nested loops. LEU at LLC can capture the outer reuse of the outer loops. The extent of being able to capture these long `RI`s also depends on the number of entries in our finite structures and the benchmark characteristics. By limiting its overhead to constant, LEU loses information either by the KSP algorithm or the approximation using buckets which may cause adverse effects in some cases.

Table 7. Comparison of hardware overheads for a 8-way 2MB LLC.

Policy	Predictor Structure	Cache Meta-data	Hardware Budget
LRU	None	12KB	12KB
DRRIP	8 bytes	8KB	8KB
SHiP	4KB SHCT 2KB PC tags	8KB	14KB
Hawkeye	12KB sampler 1KB OPTgen 3KB predictor	12KB	28KB
Hawkeye 16x	24KB sampler 1KB OPTgen 3KB predictor	12KB	40KB
LEU config5-MRU3	1.5KB LATT+RIT 4KB MRU	128KB	133.5 KB
LEU config7	1.5KB LATT+RIT	8KB	9.5 KB

4.4 Space Overhead

To reduce the space overhead of our structures, we use multiple techniques. Some of these techniques are trade-offs between space and precision. Apart from the space overhead of our LATT and RIT structures, we have a constant overhead per cache block.

4.4.1 Signature Based Metadata. We use 8-bit, 24-bit, and 32-bit signatures generated from a shift and multiplication operation-based hash function to reduce the cost of storage of 64-bit components such as the IP, LAT, and cache block address. This reduces the space required in the RIT, LATT, and the cache metadata.

4.4.2 Reducing the Number of Bits Required. The size of entries for RI and Freq depends on how quickly they are saturated. We use only a few bits to represent RI and Freq in our method. The longer an IP entry stays in RIT, the more likely it is that the reuse frequency and reuse intervals are saturated to maximum values. In our configurations, we assume a higher probability of entries getting evicted before getting saturated, which reduces the chances of inaccuracies.

Table 7 shows the space overheads of the replacement policies. Though our predictor structures have low overhead compared to other policies, our cache metadata incurs higher overhead. We observe that for LEU config 7 the overheads are lower than LRU and on average we reduce MPKI by 5% over LRU on PolyBench.

4.4.3 Comparison with Hawkeye. The original Hawkeye [11] design for SPEC2006 uses eight times the cache size as the *window length*. To capture longer reuse history (for PolyBench), we double the *window size* to 16x. However, LEU can still outperform Hawkeye by a noticeable margin, around 2% in MPKI reduction and in IPC improvement, as the statistical approach captures reuse intervals whose length is not limited by the *window size*.

4.5 Power, Area, and Latency Overheads

We use CACTI[22] to estimate the power, area, and latency overheads for the three additional storage structures introduced by our policy: LATT, RIT, and per-cached line Metadata (IP, LAT). We use three different per-entry configurations in our experiments: O1, O2, and O3, where O1 is the most expensive and O3 is the least expensive in terms of storage requirements. Table 5 shows the size of a single entry in each structure for each configuration. O1 and O2 store per-cached line Metadata with each cache block, while O3 uses a separate Metadata structure (with 1024 entries) to cache a subset of the Metadata that is maintained using full associativity and LRU.

For the configurations outlined in Table 6, we show the area and power overheads in Table 8. We discuss the latency overheads in Section 4.6. We evaluate and discuss whether adding the significant overhead of the cache metadata to the cache itself helps improve the miss ratio or not in Section 4.7.

Table 8. Power and Area consumption as percentage of 2MB LLC for LEU configurations

Config	RIT		LATT		metadata	
	Area	Power	Area	Power	Area	Power
LEU config1	0.3%	0.6%	0.3%	0.5%	16.7%	18%
LEU config2-MRU1	0.5%	1.5%	0.3%	0.5%	16.7%	18%
LEU config4	0.3%	0.5%	0.2%	0.4%	6.7%	7.4%
LEU config5-MRU3	0.3%	0.5%	0.2%	0.4%	6.7%	7.4%
LEU config6	0.7%	1.7%	0.3%	0.5%	6.7%	7.4%
LEU config7	0.3%	0.5%	0.2%	0.4%	2%	1.57%

4.6 Computational Overheads

We envision the addition of computational logic in the cache controller of the desired cache hierarchy (at LLC) and use off-the-shelf latency overheads[9] of generic ALU operations to estimate the latency of our operations.

Latency for Victimization/Eviction and Update. The computation can be broken down into logical operations as shown in Figure 10. For configurations LEU config2-MRU1, and LEU config6, where the histogram size is greater than 1 as indicated in Table 6, these overheads are justified. For the rest of the configurations, only the subtraction operation is required, which reduces the latency and power consumption for operations tremendously. Taking into account the longest sequential and dependent path, the computation overhead is around 36-37 cycles (considering histogram size greater than 1). We expect to hide the maximum ENAD computation latency for victimization within the DRAM latency (LLC Miss penalty lower bound) of 200 cycles.

Look up operations for the RIT, LATT, and the cache metadata can take from 1-2 cycles depending on the size of the structures as suggested by experiments performed on CACTI[22]. Updating the cache metadata block on a cache

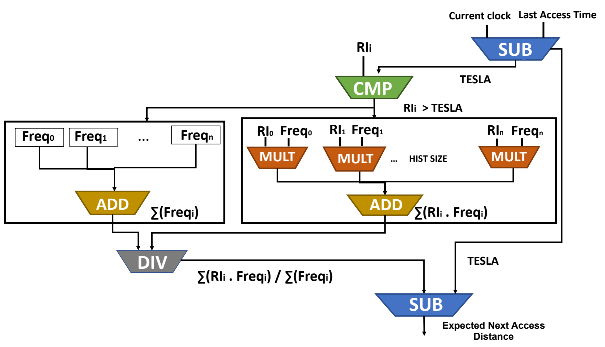

Figure 10. LEU operations required for a single way

hit happens in parallel without adding any extra latency overhead to the hit latency. For the KSP [16] algorithm, occasional iterative decrement of frequencies can occur in the non-critical path without affecting the LLC hit latency.

4.7 Space Overhead Amortization

The most significant chunk in our space overhead is the 12 Bytes for O1 overheads and 4 Bytes for O2 (see Table 5) of cache block metadata for each cache block, which amounts to 384KB for O1 and 128KB for O2 for a 2 MB cache. For O3 overhead, the size of the metadata structure is 8 KB (considering 1024 entries in the structure, used in LEU config7). In order to make a fair comparison, we add this extra space overhead as an additional cache to the LRU policy as a comparison point. We compared the reduction in MPKI against our LEU implementation and over LRU for a 2MB LLC. We observed that for both PolyBench[25] and CRC2 [1], an LLC cache size of 2MB+384KB with LRU replacement policy didn't show any improvement over LRU with 2MB LLC or over LEU with 2MB LLC (which outperforms LRU with 2MB LLC).

4.8 Sensitivity to Histogram and Bucket Size, Sampling Ratio, and Number of Ways

We perform sensitivity experiments on PolyBench to understand the impact of the number of entries in the histogram, the bucket size, the sampling ratio, and the set associativity of the cache (the number of ways).

Histogram and Bucket size. We observe that the performance impact is not huge (within two decimal places) and saturates after a point when we increase the number of histogram entries. Thus, we have limited histogram size in our configurations which also saves space and latency overheads. Regarding the bucket size, a smaller bucket size leads to higher precision but occupies more slots in the histogram. A larger bucket size compromises precision but can accommodate more entries. We set the bucket size based on our experiments.

Sampling ratio and associativity. The major factors contributing to the sensitivity of the sampling ratio for our policy are the benchmark memory access pattern, the memory footprint, the size of higher-level caches, and the filtering effect introduced by higher-level caches. The inclusivity/exclusivity property of the caches in the hierarchy also affects the access pattern to the LLC and is benchmark/trace specific. A lower sampling ratio doesn't worsen the performance of PolyBench and it also saves energy and cost. Cache associativity affects the performance of our policy given the size of our structures. For our cache configuration, 8 ways lead to the best performance for LEU with the given LEU structure configurations. We henceforth perform all our experiments with 8-way cache associativity.

4.9 Other Results

4.9.1 Comparison among Configurations. Figure 11 shows the effect of different LEU configurations on overall improvement.

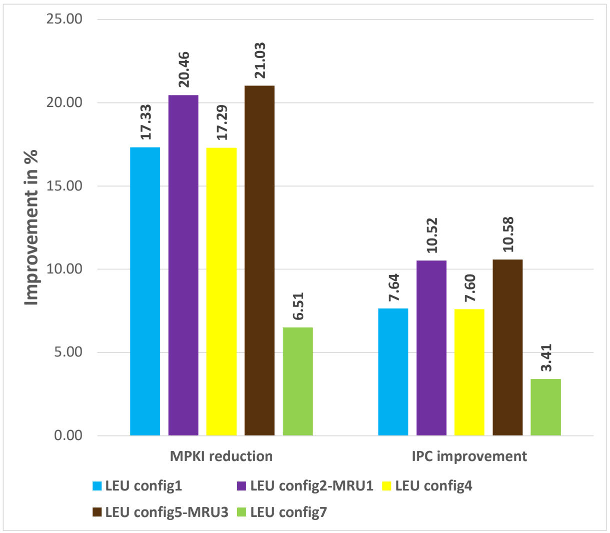

Figure 11. Geometric Means on PolyBench across different configurations.

4.9.2 Multi-Core Results. We run a 4-core simulation of LEU and LRU with a shared 8MB LLC and report the accumulated LLC misses. We choose 4 groups of benchmarks from PolyBench and the settings can be found in Table 9. We choose LEU-config6 from Table 6 as the parameter settings. LEU can significantly reduce the number of misses and miss ratios for multi-core systems. The number of misses can be reduced by 3%, 1.5%, 24%, and 5% respectively.

4.9.3 Prefetching Effect. Prefetching is widely used to hide memory latency. To test the compatibility of prefetching, we evaluate LEU config1 and LRU with next line and PC stride as L1D and L2C prefetchers respectively on PolyBench. Compared with LEU without prefetchers, LEU with prefetchers further reduces 4% in terms of MPKI over LRU from

Table 9. 4-core group setting for PolyBench.

Group	P1	P2	P3	P4
Group 1	covariance	gramschmidt	convolution_3d	convolution_3d
Group 2	adi	nussinov	fdtd_2d	trmm
Group 3	jacobi_2d	seidel_2d	symm	syr2d
Group 4	adi	syr2k	syr2k	seidel_2d

17.33% to 20.02%. The IPC improvement also rises slightly from 7.64% to 7.77%. The reason can be that prefetchers filter out some accesses to LLC and allow LEU to better analyze stride patterns and predict the expected usages more precisely.

5 Related Work

There have been a series of techniques developed to imitate Belady. RRIP [14] classifies the reuse as near, long, or distant. Protection Distance (PD) [8] identifies a time to keep a data block in cache so that the cache does (1) not evict it too early, and (2) not keep it too long to cause cache pollution. NUcache [20] sets aside part of the cache for delinquent loads (load instructions responsible for a significant fraction of cache misses) and selects one or more delinquent loads to use the extra cache based on their next-use histogram. Shepherd Cache [27] uses FIFO replacement to record accesses so it can simulate Belady in the main cache. A common theme is the efficient representation of the access history, by quantizing the reuse into a few categories, limiting the analysis to a data subset (data in the cache) or an execution window bounded by the cache size. SHiP [30] adopts a heuristic predictor to identify instructions that load streaming accesses. Hawkeye [11] uses a backward window 8 times in length the *cache-size window*, which accesses the amount of data equal to the cache size. It simulates Belady, records the decision at each reference, and replays the decision.

Hawkeye is limited in its ability to simulate Belady by the size of the past access trace it uses. To record arbitrarily long `RIs`, LEU uses sampling and records RIs for all data accessed from a specific IP, thereby increasing the sample size. Sampling has been used by Keramidas et al. [17, 24], based on the success of effective sampling in techniques such as StatCache [5]. To manage the cache, Keramidas et al. used a single RI for all references. However, the single RI for each PC cannot capture patterns with complex stride. Instead, LEU uses an RI distribution for each reference, which is necessary for its dynamism discussed in Section 2.2.

Both Hawkeye and LEU are designed to imitate MIN. The crucial difference is that Hawkeye keeps an ordered history and incurs a linear cost, while LEU uses statistical (unordered) history and can record unlimited history (however partial) under a constant cost.

Reuse intervals have been extensively used in offline analysis, including recent models Higher Order Theory of Locality [31] and the one by Beckmann et al. [3]. PRP [7] predicts the hit probability and prioritizes lines with higher probability. Reuse intervals were routinely used in the working-set theory as discussed recently [32]. Shi et al.[29] proposed a Support Vector Machine (SVM) like mechanism for online prediction based on observations from LSTM. Unlike offline modeling, the entire process of LEU prediction is on-the-fly and does not require expensive offline training.

Shah et al. [28] present Mockingjay, which uses a sampled history of past accesses to predict future reuse intervals per PC. Their solution predicts the reuse time of accesses from a given PC based on its previously observed RIs and evicts the cacheblock which is the furthest from its predicted reuse. Mockingjay updates its RI predictions differently from LEU. In Mockingjay, when a new sample has a different RI than its previously predicted value, the new RI prediction is a linear interpolation of the old prediction and the new sample RI, which is biased towards the old prediction.

6 Summary

We introduce a novel method of using instruction pointer-based reuse interval histograms to mimic Belady's algorithm for cache replacement by victimizing the cache line with the least expected use in the future when replacement is necessary. LEU's use of statistical information about past reuses on an instruction pointer basis eliminates the limitations of fixed history windows and in-cache residence of prior approaches. LEU is able to capture dominant long reuse intervals and can achieve up to 71% improvement for IPC and 75% for MPKI reduction for a single benchmark and 6% IPC improvement and 14% MPKI reduction of the two benchmark suites combined. Our results show that LEU has better performance over Hawkeye for PolyBench but that the reverse is true for SPEC2006, resulting in similar average overall performance. The choice of replacement prediction is therefore workload dependent.

Acknowledgement

The authors wish to thank Benjamin Reber for the discussion on the related work, Dong Chen for early participation, and Timothy Sherwood for advice on approximation `RI` storage and generous funding support. The authors also appreciate the anonymous reviewers and the shepherd for their feedback. This work was supported in part by the National Science Foundation (Contract No. CNS-1618497, CNS-1900803, CCF-2217395, CCF-2114319, NSF-1909099, CCF-2006542, and CCF-1763699). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding organizations.

Appendix

A Formal Policy Definition

Let S be a cache set and let T be the length of a memory access trace.

Definition 1. A **cache state**, C_t^S, is the set of cache blocks in S, at time t, where $t \in \{1, 2, \ldots, T\} = [T]$, in which each cache block is a tuple $(b, I_b(t))$, where b is the block address and I_b is some usage information about b at time t.

For example, LRU cache stores the last access time (LAT), i.e. $I_b(t) = LAT_b(t)$, which is the logical time of the most recent access of data block b before time t.

Definition 2. A cache replacement policy is a **ranking policy** if for a current cache state C_t^S, there exists a function, rank : $C_t^S \times [T] \to \mathbb{R}$, such that the victim of the cache, b_{victim} is:

$$b_{\text{victim}} \in \arg\min_{b \in C_t^S} \text{rank}(b, t)$$

The rank function for LRU replacement is the reciprocal of the time since the last access:

$$rank_{LRU}(b, t) = \frac{1}{t - LAT_b(t)} \tag{3}$$

Another example is Belady or MIN, which evicts the block with the furthest future reuse [4]. MIN uses the next access time (NAT), i.e. $I_b = NAT_b(t)$. The ranking function is then

$$rank_{MIN}(b, t) = \frac{1}{NAT_b(t) - t} \tag{4}$$

LEU imitates MIN and attempts to evict the block with the furthest future access. The key is to approximate $NAD_b(t) = NAT_b(t) - t$ (in Eq. 4), the next access distance.

LEU maintains $I_b = \{P_{\text{ip(b)}}(ri), tesla_b(t)\}$, where $P_{\text{ip(b)}}(ri)$ is the distribution of RIs of all accesses made from the instruction pointer used for the last access to block b and *tesla* is the time elapsed since last access, i.e. $tesla_b(t) = t - LAT_b(t)$.

Since there are multiple possible RIs, NAD is probabilistic. We compute its expectation. This expectation depends on two factors: the distribution of RIs and the current *tesla*:

$$\mathbb{E}\left[NAD_b(t) \middle| P_{\text{ip(b)}}(ri), \text{tesla}_b(t)\right]$$

RIs may range from 1 to T-1. At time t for block b, RIs less than or equal to $tesla_b(t)$ are not possible. Therefore, the expectation calculation considers only RIs that are longer than $tesla_b(t)$.

$$\mathbb{E}\left[NAD_b(t)\right] = \sum_{ri=\text{tesla}_b(t)+1}^{T-1} \left[P_{\text{ip(b)}}(ri) * (ri - \text{tesla}_b(t))\right]$$

$$= \sum_{ri=\text{tesla}_b(t)+1}^{T-1} \left[P_{\text{ip(b)}}(ri) * ri\right] - \sum_{ri=\text{tesla}_b(t)+1}^{T-1} \left[P_{\text{ip(b)}}(ri) * \text{tesla}_b(t)\right] \tag{5}$$

The ranking function is then:

$$rank_{LEU}(b, t) = \frac{1}{\mathbb{E}\left[NAD_b(t)\right]} \tag{6}$$

Comparing the above with MIN in Eq. 4, we see that LEU uses a statistical prediction of MIN.

References

[1] [n. d.]. The 2nd cache replacement championship – co-located with isca june 2017. https://crc2.ece.tamu.edu/

[2] [n. d.]. Standard Performance Evaluation Corporation. https://www.spec.org/cpu2006/

[3] Nathan Beckmann and Daniel Sanchez. 2016. Modeling cache performance beyond LRU. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 225–236.

[4] L. A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal* 5, 2 (1966), 78–101.

[5] Erik Berg and Erik Hagersten. 2004. StatCache: A probabilistic approach to efficient and accurate data locality analysis. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software* (Austin, Texas). 20–27.

[6] Edward Grady Coffman and Peter J Denning. 1973. *Operating systems theory*. Vol. 973. prentice-Hall Englewood Cliffs, NJ.

[7] Subhasis Das, Tor M Aamodt, and William J Dally. 2015. Reuse distance-based probabilistic cache replacement. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 4 (2015), 1–22.

[8] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum. 2012. Improving Cache Management Policies Using Dynamic Reuse Distances. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*. 389–400. https://doi.org/10.1109/MICRO.2012.43

[9] Agner Fog. 1996-2019. Instruction Tables,Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs. (1996-2019), 179–180. https://www.agner.org/optimize/instruction_tables.pdf

[10] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *2012 Innovative Parallel Computing (InPar)*. Ieee, 1–10.

[11] Akanksha Jain and Calvin Lin. 2016. Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement. In *Proceedings of the International Symposium on Computer Architecture*. 78–89. https://doi.org/10.1109/ISCA.2016.17

[12] Aamer Jaleel. [n. d.]. Memory Characterization of Workloads Using Instrumentation-Driven Simulation, A Pin-based Memory Characterization of the SPEC CPU2000 and SPEC CPU2006 Benchmark Suites. ([n. d.]). http://http://www.glue.umd.edu/~ajaleel/workload/

[13] Aamer Jaleel, Joseph Nuzman, Adrian Moga, Simon C Steely Jr, and Joel Emer. 2015. High Performing Cache Hierarchies for Server Workloads. In *High-Performance Computer Architecture (HPCA)*.

[14] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *ACM SIGARCH Computer Architecture News*, Vol. 38. ACM, 60–71.

[15] S. Jiang and X. Zhang. 2002. LIRS: an efficient low inter-reference recency set replacement to improve buffer cache performance. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*. Marina Del Rey, California.

[16] Richard M Karp, Scott Shenker, and Christos H Papadimitriou. 2003. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems (TODS)* 28, 1 (2003), 51–55.

[17] Georgios Keramidas, Pavlos Petoumenos, and Stefanos Kaxiras. 2007. Cache replacement based on reuse-distance prediction. In *Proceedings*

of the Proceedings of the International Conference on Computer Design (ICCD). 245–250. https://doi.org/10.1109/ICCD.2007.4601909

[18] 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. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 737–749.

[19] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, Vol. 40. ACM, 190–200.

[20] R. Manikantan, Kaushik Rajan, and R. Govindarajan. 2011. NUcache: An efficient multicore cache organization based on Next-Use distance. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. 243–253. https://doi.org/10.1109/HPCA.2011.5749733

[21] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. 1970. Evaluation techniques for storage hierarchies. *IBM Systems journal* 9, 2 (1970), 78–117.

[22] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. 2009. CACTI 6.0: A tool to model large caches. *HP laboratories* 27 (2009), 28.

[23] Samuel Pakalapati and Biswabandan Panda. 2020. Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 118–131.

[24] Pavlos Petoumenos, Georgios Keramidas, and Stefanos Kaxiras. 2009. Instruction-based reuse-distance prediction for effective cache management. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. 49–58. https://doi.org/10.1109/ICSAMOS.2009.5289241

[25] Louis-Noël Pouchet. [n. d.]. PolyBench/C 4.0. http://polybench.sourceforge.net.

[26] Louis-Noël Pouchet. 2012. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench* (2012).

[27] Kaushik Rajan and Ramaswamy Govindarajan. 2007. Emulating Optimal Replacement with a Shepherd Cache. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture*. 445–454. https://doi.org/10.1109/MICRO.2007.25

[28] 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)*. 558–572. https://doi.org/10.1109/HPCA53966.2022.00048

[29] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. 2019. Applying Deep Learning to the Cache Replacement Problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 413–425.

[30] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C Steely Jr, and Joel Emer. 2011. SHiP: Signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. 430–441.

[31] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. 2013. HOTL: a higher order theory of locality. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 343–356.

[32] Liang Yuan, Chen Ding, Wesley Smith, Peter Denning, and Yunquan Zhang. 2019. A Relational Theory of Locality. *ACM Transactions on Architecture and Code Optimization (TACO)* 16, 3 (2019), 33.

[33] Tomofumi Yuki and Louis-No¨el Pouchet. 2015. POLYBENCH 4.0. (2015).

Received 2023-03-03; accepted 2023-04-24