4 CACHE REUSE MODELS

4.1 Overview

Processor caches are critical components of the memory hierarchy that exploit locality to keep frequently-accessed data on chip. Caches can significantly boost performance and reduce energy usage, but their benefit is highly workload dependent. In modern power and energy constrained computer systems, understanding a workload's dynamic cache behavior is important for making critical resource allocation and scheduling decisions. For example, allocating excess cache capacity to a workload wastes power, as large caches dissipate significant leakage power, while allocating insufficient cache capacity hurts performance and increases main memory power.

Caches can be made dynamically reconfigurable to enable energy savings by exploiting workload characteristics. Previous research has explored placing some or all of a cache in low-power mode [10, 67, 77] or dynamically partitioning the cache to eliminate resource contention [175, 214]. A recent Intel processor [116] can dynamically reduce its LLC's capacity to save power. Researchers [240] have also explored mechanisms to dynamically reconfigure both the number of sets and the associativity of set-associative caches.

Being able to exploit cache reconfiguration capability will enable our governors to make the system operate more efficiently if possible. But real systems today offer limited or no support for user-/OS-driven cache reconfigurations. So, in this chapter and the next one, we will study cache reconfigurations using a simulator.

Cache reconfigurations incur significant overheads, so we will not consider the expensive approach of trying out various configurations before deciding on the best configuration. Instead, we will use online predictors that estimate the temporal locality in cache accesses and predict the performance of different cache configurations. These

models will be our focus in this chapter whereas the next chapter will focus on using these models to develop governors that control both core frequency and cache size to meet SLApower.

We develop a reuse distance/stack distance based analytical modeling framework for efficient, online prediction of cache performance for dynamically reconfigurable set-associative caches that use LRU/PLRU/RANDOM/NMRU replacement policies. Our framework is inspired by two foundational works: Mattson's stack distance characterization [150] (also used later as reuse distance [29, 65]) and Smith's associativity model [103, 200] for LRU caches.

The central theme of our framework is to decouple temporal characteristics in the cache access stream from characteristics of the replacement policy. We propose a novel low-cost hardware circuit, that uses Bloom Filters and sampling techniques, to estimate cache reuse distance distributions online. These distributions are then used as inputs to analytical models of replacement policy performance to predict miss ratios. This separation of aspects brings the advantage of being able to easily refine either aspect in isolation without affecting the other.

Our work differs from prior art in being suited for online predictions [103, 200], working with practical replacement policies other than LRU [103, 174, 175, 200, 215], allowing reconfigurability in the number of sets in addition to associativity [127, 174, 175], and being very low cost [84]. Our framework unifies existing cache miss rate prediction techniques such as Smith's associativity model, Poisson variants, and hardware way-counter based schemes.

The main contributions of this chapter are:

1. We formulate an analytical framework based on generalized stochastic Binomial Matrices [212] for transforming reuse distance distributions (Sections 4.4, 4.5).

2. We formulate new miss ratio prediction models for RANDOM (Section 4.5.2), NMRU (Section 4.5.3), PLRU (Section 4.5.4) replacement policies.

3. We show that the traditional hardware way-counter based prediction [215] for varying associativity is a special instance of our unified framework (Section 4.6.2). Further, we show how way-counter data for LRU may be transformed to apply to caches with a different number of sets. (Section 4.6.2.2)

4. We propose a novel hardware scheme for efficient online estimation of reuse distance/stack distance distributions (Section 4.6.1).

The rest of this chapter is organized as follows:

Evaluation Infrastructure: Section 4.2 describes the workloads and simulators that we used for this study.

Temporal locality metrics: Section 4.3 defines reuse distributions that capture the temporal locality of address streams. Section 4.4 shows how to modify these to apply for a cache with a different number of sets.

Replacement policy models: Section 4.5 introduces the notion of cache hit-functions that, when multiplied with the per-set reuse distribution, produce expected cache hit ratios. Sections 4.5.1.1 and 4.4.3 consider optimizations for LRU hit ratio prediction. Sections 4.5.2, 4.5.3 and 4.5.4 develops new prediction models for RANDOM, NMRU, PLRU respectively. Section 4.5.5 discusses prediction accuracy and computation overheads.

Hardware Support: Section 4.6.1 presents the novel, low-cost hardware for estimating reuse distributions. We also discuss two traditional hardware mechanisms—set-counters and way-counters (Section 4.6.2).

Index Hashing: Section 4.7 describes the index-hashing scheme that we used to map addresses to cache sets.

size \ assoc.	2	4	8	16	32
2MB	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}
4MB	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}
8MB	2^{16}	2^{15}	2^{14}	2^{13}	2^{12}
16MB	2^{17}	2^{16}	2^{15}	2^{14}	2^{13}
32MB	2^{18}	2^{17}	2^{16}	2^{15}	2^{14}

Table 4.1: Relation between number of sets and associativity for different cache sizes. Assuming some cache configuration is the current configuration, there are a total of 25-1=24 possible other cache configurations. An inspection of the table reveals that *at most* 4 of these possible 24 configurations can have the same number of sets as the current configuration. For example, with 32MB 32-way as the current configuration, other configurations with the same number of sets (2^{14}) are: 2MB 2-way, 4MB 4-way, 8MB 8-way and 16MB-16-way. Thus, way-counters (Section 4.6.2) can predict for *at most* 4 of 24 possible other configurations at any time.

4.2 Infrastructure

In our study, caches are characterized by the number of sets S, associativity A, and replacement policy. We generally use S' and A' while referring to the target cache configuration for which we want to predict the miss ratio. We assume a fixed line size of 64 bytes. Table 4.1 shows the relation between S, A and cache size for the configurations of the Last-Level Cache (LLC) that we study.

Our models estimate hit ratio (hit/access). This is easily converted into other measures: miss ratio=1-hit ratio; miss rate=miss ratio*access/instruction. For evaluating prediction quality, we obtain address traces of accesses to a 32MB 32-way LLC in a simulated system (Table 4.2) for our workloads, run the traces through a standalone cache simulator (that does not model timing) and compare measured against predicted metrics.

Table 4.2 describes the 8-core CMP we use for gathering traces. We assume an 8-banked L3 cache that is dynamically re-configurable for a total of 25 configurations (see

Core configuration		4-wide out-of-order, 128-entry window, 32-entry scheduler		
Number of cores		8	**On-chip frequency**	2132 MHz
Technology Generation		32nm	**Temperature**	340K
Functional Units		4 integer, 2 floating-point, 2 mem units		
Branch Prediction		YAGS 4K PHT 2K Exception Table, 2KB BTB, 16-entry RAS		
Disambiguation		NoSQ 1024-entry predictor, 1024-entry double-buffered SSBF		
Fetch		32-entry buffer, Min. 7 cycles fetch-dispatch time		
Inclusive	**L1I Cache**	private 32KB 4-way per core, 2 cycle hit latency, ITRS-HP		
	L1D Cache	private 32KB 4-way per core, 2 cycle hit latency, ITRS-HP		
	L2 Cache	private 256KB 8-way per core, 6 cycle access latency, PLRU, ITRS-LOP		
	L3 Cache	shared, configurable 2–32 MB 2–32 way, 8 banks, 18 cycle access latency, PLRU, ITRS-LOP, serial		
Coherence protocol		MESI (`Modified`, `Exclusive`, `Shared`, `Invalid`), directory		
On-Chip Interconnect		2D Mesh, 16B bidirectional links		
Main Memory		4GB DDR3-1066, 75ns zero-load off-chip latency, 2 memory controllers, closed page, pre-stdby		

Table 4.2: System configuration.

Workload	**Time** (seconds)	**#LLC accesses** $\times 10^6$	**#unique line addresses** $\times 10^6$	**#pages in LLC accesses** $\times 10^6$
`apache`	0.562	177.764	3.829	0.136
`jbb`	0.260	35.474	5.831	0.123
`oltp`	0.410	150.126	2.401	0.138
`zeus`	0.322	10.488	1.003	0.048

Table 4.3: Workload Characteristics. Cache line size = 64 bytes. Page size = 4K bytes.

Table 4.1). The cache uses a hashed indexing scheme to map addresses to cache sets (see Section 4.7). We conservatively assume a constant access latency for all configurations.

We use 4 Wisconsin commercial workloads [4] (`apache`, `jbb`, `oltp`, `zeus`). Each workload uses 8 threads and runs for a fixed amount of work (e.g. #transactions or loop iterations [6]) that corresponds to ~4B instructions per workload. Each simulation run starts from a mid-execution checkpoint that includes cache warmup. Table 4.3 shows a summary of the characteristics of the workload executions.

Figure 4.1 shows the average miss ratios for a 32 MB 32-way LLC over the execution

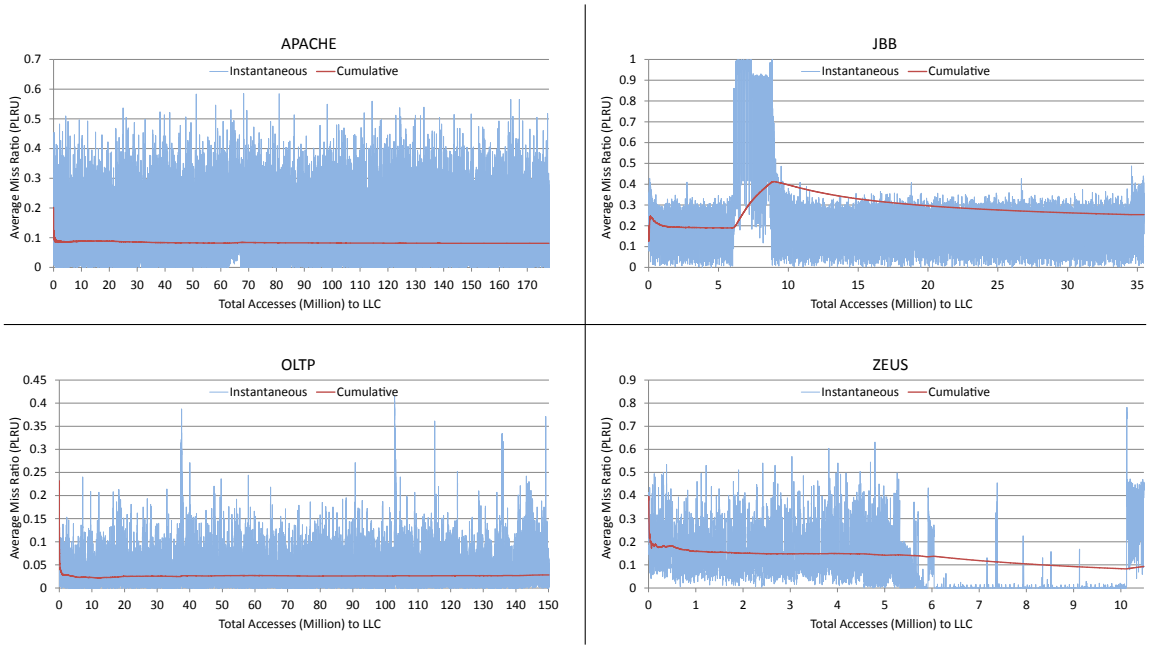


Figure 4.1: "Instantaneous" and cumulative miss ratios. Granularity is 1000 LLC accesses.

of the workloads. The "instantaneous" miss ratios are computed at the end of every 1000 accesses to the LLC and show a lot of variation for every workload. The cumulative miss ratio shows the average miss ratio of all LLC accesses in the execution till that point. It is much more stable than the "instantaneous" ratios. Its final value at the end of the execution is the overall/long-term average miss ratio. Our miss ratio prediction models aim to predict this long-term average.

Table 4.4 show why it is useful to consider reconfigurability in both the number of sets and ways (associativity) for the LLC. For a given cache size, M_* compares the maximum to minimum miss ratio for all configurations with that cache size (see Table 4.1 for the configurations). We obtain these numbers by running the access traces through our standalone cache simulator. As an example, `oltp` sees up to 57% increase in miss ratio with a suboptimal configuration for a 2MB LRU cache. M_S indicates what happens

Policy	Workload	2MB		4MB		8MB		16MB		32MB	
		M_*	M_S	M_*	M_S	M_*	M_S	M_*	M_S	M_*	M_S
LRU	apache	1.07	1.07	1.04	1.01	1.19	1.04	1.69	1.04	1.33	1.00
	jbb	1.03	1.03	1.14	1.08	1.13	1.02	1.09	1.00	1.08	1.00
	oltp	1.57	1.57	1.80	1.30	1.53	1.04	1.45	1.01	1.44	1.00
	zeus	1.13	1.13	1.09	1.03	1.07	1.01	1.08	1.00	1.07	1.00
RANDOM	apache	1.01	1.01	1.02	1.00	1.07	1.01	1.17	1.01	1.13	1.00
	jbb	1.01	1.01	1.02	1.01	1.03	1.01	1.02	1.00	1.02	1.00
	oltp	1.20	1.20	1.34	1.13	1.30	1.03	1.29	1.01	1.23	1.00
	zeus	1.02	1.02	1.03	1.01	1.02	1.00	1.02	1.00	1.02	1.00
NMRU	apache	1.03	1.00	1.02	1.00	1.02	1.00	1.07	1.01	1.06	1.05
	jbb	1.02	1.00	1.02	1.00	1.03	1.01	1.04	1.03	1.04	1.04
	oltp	1.12	1.12	1.21	1.03	1.18	1.00	1.17	1.01	1.11	1.01
	zeus	1.04	1.00	1.04	1.00	1.03	1.01	1.03	1.02	1.03	1.03
PLRU	apache	1.06	1.06	1.04	1.01	1.15	1.03	1.49	1.03	1.30	1.00
	jbb	1.03	1.03	1.10	1.05	1.11	1.02	1.07	1.00	1.05	1.00
	oltp	1.43	1.43	1.59	1.20	1.46	1.04	1.38	1.01	1.34	1.00
	zeus	1.10	1.10	1.08	1.02	1.05	1.00	1.05	1.00	1.05	1.00

Table 4.4: Relative miss ratios for difference cache sizes and replacement policies. M_* shows the max-to-min miss ratio over configurations having all possible number of sets for the given cache size. Relative ratios $\geqslant 1.05$ are shown in red. M_S shows the relative miss ratio of the configuration having the same number of sets (=2^{14}) as that of the largest cache (32 MB, 32-way) compared to the minimum miss ratio over all configurations for the given cache size. Entries with relative ratios $\geqslant 1.05$ are shaded.

if only way configurability is present. It assumes the same number of sets (2^{14}) as that for the 32MB 32-way cache. For `oltp`, this turns out to be the worst configuration for a 2MB cache (configuration: 2MB, 2-way)—it has 57% more misses than for the best configuration (2MB 32-way, number of sets = 2^{10}). Set configurability is more important at small cache sizes than at larger sizes.

4.3 Measures of Temporal Locality

In this section we develop metrics of temporal locality in the address stream that are independent of the cache configuration. These metrics will be used for estimating the miss ratios for arbitrary cache configurations. For our study, all addresses are (hashed) line addresses of cache accesses.

Consider an address trace T as a mapping of consecutive integers in increasing order, representing successive positions in the trace, to tuples (x, m) where x identifies the address and m identifies its repetition number. The first occurrence of address x in the trace is represented by $(x, 0)$. Let $t = T^{-1}$ denote the inverse function. $t(x, m)$ denotes the position of the m^{th} occurrence of address x in the trace. We now introduce a few more definitions.

Reuse Interval: The **reuse interval** (RI) is defined only when $m > 0$ and denotes the portion of the trace enclosed between the m^{th} and $(m-1)^{th}$ occurrence of x. Formally,

$$RI(x, m) = \begin{cases} \{(z, m') | t(x, m-1) < t(z, m') < t(x, m)\} & \text{if } m > 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Unique Reuse Distance: This denotes the total number of unique addresses between two occurrences of the same address in the trace. Thus,

$$URD(x, m) = \begin{cases} \left|\{z | (z, m') \in RI(x, m)\}\right| & \text{if } m > 0 \\ \infty & \text{otherwise} \end{cases}$$

Numerically, this is 1 less than Mattson's much earlier stack distance [150].

Absolute Reuse Distance: This denotes the total number of positions between two

occurrences of the same address in the trace. Thus, $ARD(x, m) =$

$$\begin{cases} \left|RI(x, m)\right| = t(x, m) - t(x, m-1) - 1 & \text{if } m > 0 \\ \infty & \text{otherwise} \end{cases}$$

As an example, in the access sequence a b b c d b a, $URD(a, 1) = 3$ and $ARD(a, 1) = 5$.

4.3.1 Reuse Distance Distributions

Our study is concerned with average-case behavior. So instead of focusing on each individual point in T, we characterize it using probability vectors that reflect average/expected distributions.

- The **unique reuse distance distribution** of trace T is a probability distribution that we denote by row vector $\mathbf{r}(T)$ such that the k^{th} component,
  $\mathbf{r}_k(T) = P(URD(x, m) = k), \forall (x, m) \in image(T)$.

- The **expected absolute distance distribution** of trace T is a row vector that we denote by $\mathbf{d}(T)$ such that the k^{th} component,
  $\mathbf{d}_k(T) = E(ARD(x, m) | URD(x, m) = k), \forall (x, m) \in image(T)$

4.3.2 $T \rightarrow \mathbf{r}(T)$ is a Lossy Transformation

The characterization is lossy in the sense that in general, T cannot be recovered from $\mathbf{r}(T)$ even up to permutation of entity identifiers.

Consider two traces T_A and T_B such that they have disjoint sets of entities and different values of reuse metrics. Let T_{AB} denote a new trace formed from concatenating, in order, sequences represented by T_A and T_B. This operation is not commutative, that is, T_{AB} and T_{BA} are distinct, yet have the same values for the reuse metrics. So the reverse

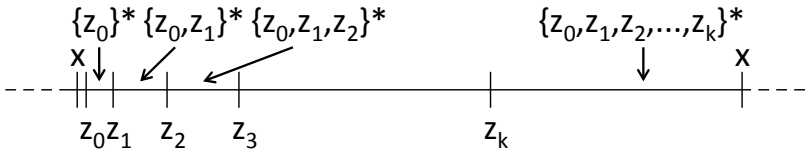


Figure 4.2: Unique elements z_0, z_1, ... z_{k-1} partition $\mathbf{d}_k(t)$ into subintervals.

mapping from $\mathbf{r}(T)$ to T is not unique. The argument can be extended to show that any trace characterization using position-agnostic metrics must be lossy.

4.3.3 $\mathbf{d}(T)$ Estimation

It is obvious that $ARD(x, m) \geqslant URD(x, m), \forall x, m$. It then follows that $\mathbf{d}_k(T) \geqslant k, \forall k$ such that $\mathbf{r}_k(T) > 0$. Also, $\mathbf{d}_0(T) = 0$. We now show how to compute (an approximation to) $\mathbf{d}(T)$ given $\mathbf{r}(T)$.

Figure 4.2 shows a schematic of a trace and organization of URDs within a reuse interval for some address x. z_0, z_1,...z_k denote distinct addresses. This is just a conceptual tool and does not constrain the actual permutation of addresses in a particular reuse interval. The immediate next access after reference address x must be something other than x (otherwise the reuse interval would immediately terminate with $k = 0$). Between this first address z_0 and the next different address z_1, the only possible URDs of accesses must be 0. Between z_1 and z_2, the only possible URDs can be 0 and 1. Extending this reasoning till z_{k-1} and z_k we observe that $\mathbf{d}_k(T)$ and $\mathbf{d}_{k-1}(T)$ differ only in the last sub-sequence which consists of a run of accesses with URDs in $\{0, 1, .., k-1\}$. We approximate the length of this run with the expected number of trials to success in a geometric distribution with success probability $\sum_{i=k}^{\infty} \mathbf{r}_i(T)$.

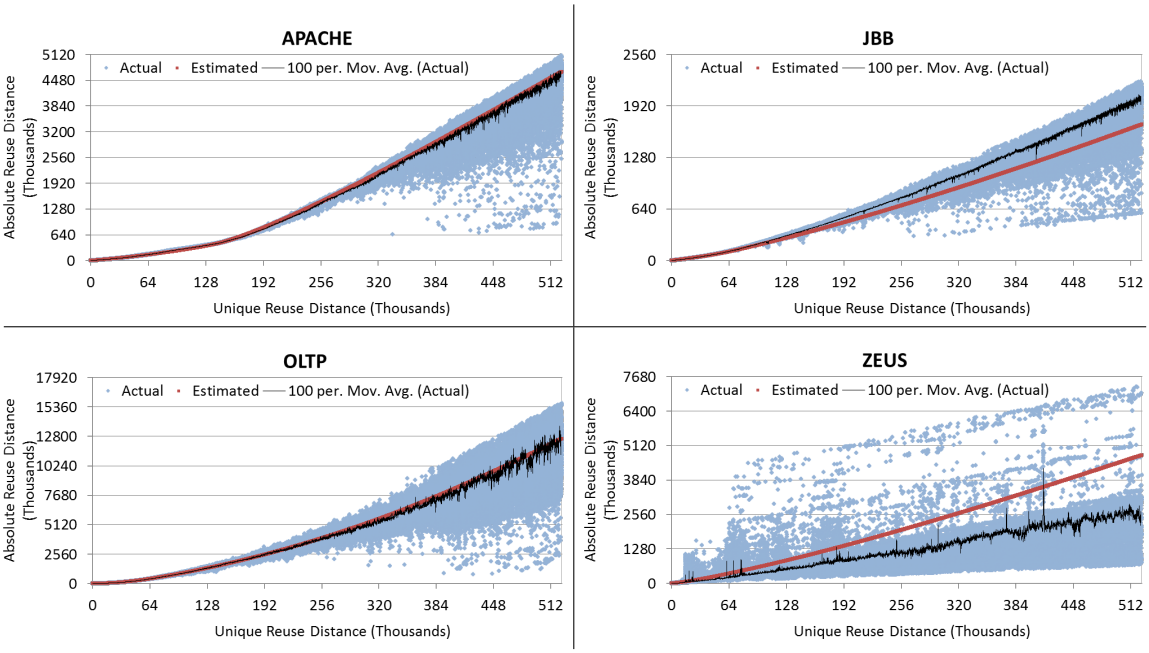


Figure 4.3: Actual, Moving Average (window size = 100) of Actual, and Estimated $\mathbf{d}(T)$.

We thus arrive at the following recurrence:

$$\begin{aligned} \mathbf{d}_0(T) &= 0 \\ \mathbf{d}_k(T) &= \mathbf{d}_{k-1}(T) + \frac{1}{\sum_{i=k}^{\infty} \mathbf{r}_i(T)} \end{aligned} \tag{4.1}$$

Expanding the recurrence gives us

$$\mathbf{d}_k(T) = \sum_{j=1}^{k} \frac{1}{\sum_{i=j}^{\infty} \mathbf{r}_i(T)} = \sum_{j=1}^{k} \frac{1}{1 - \sum_{i=0}^{j-1} \mathbf{r}_i(T)}$$

This is similar to known approximations for the coupon collector's problem assuming a given order of coupons [34]. We find good agreement in trends between observed (Moving Average) and estimated values of $\mathbf{d}(T)$ as illustrated in Figure 4.3. The moving

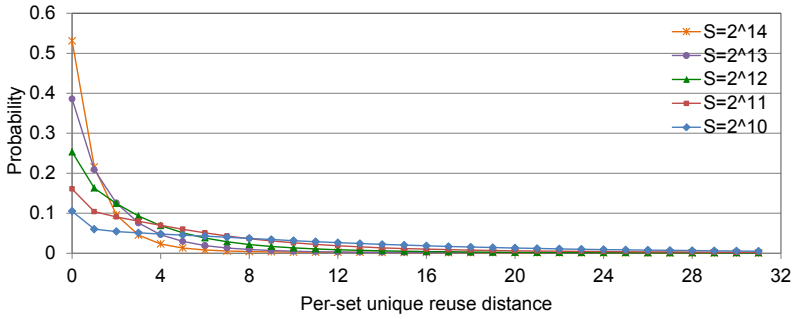


Figure 4.4: Effect of the number of sets (S) on per-set locality for `oltp`.

average calculation acts like a low-pass filter that removes short-term variation to reveal long-term trends. Our estimates are good for workloads with long traces, e.g., `apache` and `oltp`. We expect the differences to reduce for `jbb` and `zeus` with longer traces.

4.4 Per-set Locality

Replacement policy decisions (determining which cache line to evict) in traditional caches happen for each individual set. This in turn influences the miss ratio. So it is essential to determine the locality in the address stream that each individual set sees on average. We refer to the temporal locality in the per-set address stream as the per-set locality.

Per-set locality is strongly influenced by the number of sets (S) in the cache. The set of *unique* addresses in the address stream is split among the sets based on the index mapping function. The address stream that any individual set sees is the subset of the original address stream consisting of *all* accesses to the addresses mapping to that set. S thus determines the degree to which the address stream is split. Decreasing S increases URDs of the per-set address streams since addresses that hitherto mapped to other sets now get mapped to the reference set, and vice versa.

Accordingly, we extend our previous notations of locality metrics to additionally

include S as a parameter. Thus $\mathbf{r}(T, S)$ denotes the unique reuse distribution of the sub-sequence of T that a single set in the cache observes on average. $\mathbf{r}(T, 1)$ is the temporal locality of the original address stream, which is also the per-set locality of a fully-associative cache ($S = 1$). Figure 4.4 illustrates how $\mathbf{r}(T, S)$ changes with S for `oltp`. $\mathbf{d}(T)$ is adapted to $\mathbf{d}(T, S)$ similarly and can be estimated from $\mathbf{r}(T, S)$ using Equation 4.1. For brevity of notation, we will omit specifying one or more parameters when their values are clear from the context.

As Table 4.1 shows, cache configurations in our study have a range of number of sets (2^{10} to 2^{18}). For efficiently predicting miss ratios it is essential to be able to determine how $\mathbf{r}(S)$ can be transformed to $\mathbf{r}(S')$ for $S' \neq S$. The rest of this section develops a (new) methodology for this.

4.4.1 $\mathbf{r}(S')$ Estimation

For set-associative caches with $S' > 1$ we make the simplifying assumption, similar to Smith's model [103, 200], that the mapping of unique lines to cache sets are independent of each other. While this assumption does not always hold with the traditional bit selection index function, some processors use simple XOR hashing functions that increase uniformity [134]. The uniformity assumption enables both the following model and the use of uniform set-sampling techniques.

Accesses to a given set can thus be modeled as successive Bernoulli trials with the success of each trial having probability $\frac{1}{S'}$. While computing $\mathbf{r}(S')$ from $\mathbf{r}(1)$, we note that $\mathbf{r}_j(S')$ is the sum of the probability of exactly j successes (j addresses mapping to the reference set) from $\mathbf{r}_k(1)$, $\forall k$. The generalized stochastic Binomial Matrix [212] $\mathbf{B}(x, y)$ has the value ${}^kC_j y^j x^{k-j}$ in row k, column j, where ${}^kC_j$ denotes the j^{th} binomial coefficient and $x + y = 1$. This is the same as the probability of exactly j successes in k Bernoulli

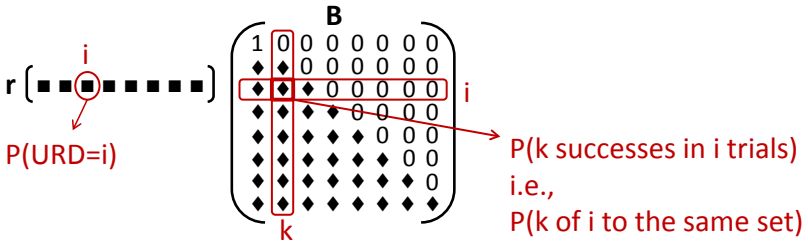


Figure 4.5: Reuse distribution transformations with stochastic Binomial Matrices.

trials with probability of each success being y. Viewing the computation of $\mathbf{r}(S')$ from $\mathbf{r}(1)$ through the lens of matrix multiplication, we recognize that the transformer is a generalized stochastic Binomial Matrix, $\mathbf{B}(1-\frac{1}{S'},\frac{1}{S'})$. Thus,

$$\mathbf{r}(S') = \mathbf{r}(1) \cdot \mathbf{B}(1-\frac{1}{S'},\frac{1}{S'}) \tag{4.2}$$

Figure 4.5 shows a schematic of the transformation. The transformer, B, is always a lower triangular matrix.

It is straight-forward to show that the transformation respects $\sum_{i=0}^{\infty}\mathbf{r}_i(S') = \sum_{i=0}^{\infty}\mathbf{r}_i(1) = 1$. Qualitatively, this transformation results in a re-distribution of mass with $\mathbf{r}(S')$ getting compressed as S' is increased and dilated as S' is decreased (see Figure 4.4).

We will now show how to compute $\mathbf{r}(S')$ from any starting cache configuration S. This shows how computations can be reused instead of always needing to start from the ground configuration ($S = 1$) and will also be useful in reasoning about way-counters (Section 4.6.2).

Binomial Matrices are invertible (when the second parameter is non-zero) and closed under multiplication within the same dimension [212]. Using identities $\mathbf{B}(x,y)\mathbf{B}(w,z) =$

$\mathbf{B}(x + yw, yz)$ and $\mathbf{B}(x, y)^{-1} = \mathbf{B}(-xy^{-1}, y^{-1})$, [212], we get

$$\begin{aligned}
\mathbf{r}(S') &= \mathbf{r}(1) \cdot \mathbf{B}(1 - \frac{1}{S'}, \frac{1}{S'}) \\
&= \mathbf{r}(S) \cdot (\mathbf{B}(1 - \frac{1}{S}, \frac{1}{S}))^{-1} \cdot \mathbf{B}(1 - \frac{1}{S'}, \frac{1}{S'}) \\
&= \mathbf{r}(S) \cdot \mathbf{B}(1 - \frac{S}{S'}, \frac{S}{S'}) \qquad (4.3)
\end{aligned}$$

Equation 4.3 is a general form of Equation 4.2. The transformer depends only on the ratio of the number of the sets in the current cache to that in the target cache. There are two cases to consider depending on the value of this ratio:

Case 1, $\mathbf{S'} \geqslant \mathbf{S}$: The transformation is always safe in that the computed probabilities are valid ($\in [0, 1]$) *even if* $\mathbf{r}(S)$ *has not been computed binomially*. Moreover, this allows intermediate steps; for example, computing $\mathbf{r}(2^{14})$ from $\mathbf{r}(1)$ is equivalent to first computing $\mathbf{r}(2^{10})$ from $\mathbf{r}(1)$ and then computing $\mathbf{r}(2^{14})$ from $\mathbf{r}(2^{10})$. This provides an opportunity to *reuse intermediate computations*. So, $\mathbf{r}(S)$ can be computed once from $\mathbf{r}(1)$ for the smallest S (2^{10} in our study, see Table 4.1) and used for all other target configurations.

Case 2, $\mathbf{S'} < \mathbf{S}$: Since $\mathbf{B}(1 - \frac{S}{S'}, \frac{S}{S'}) = (\mathbf{B}(1 - \frac{S'}{S}, \frac{S'}{S}))^{-1}$, Case 2 transforms can invert Case 1 transforms provided Case 1 results have not been truncated (see below). Otherwise, the computed probabilities may not be valid ($\notin [0, 1]$).

For an example transformation, consider $S' = 2S$. The components of the transformed reuse distribution, r', are computed as:

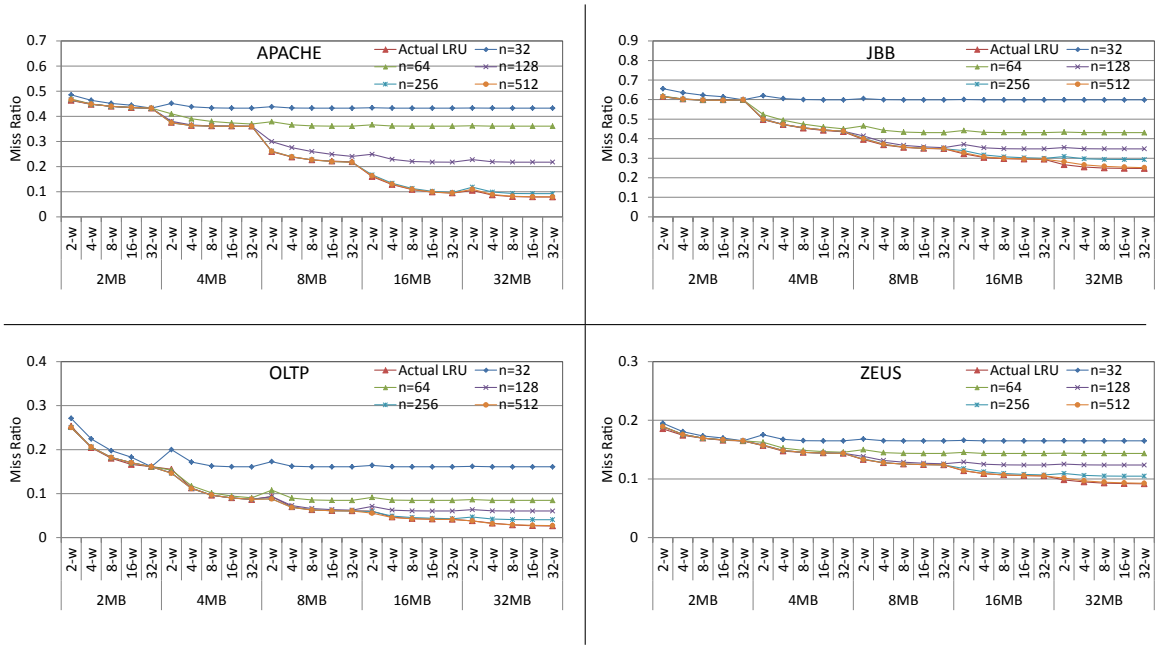


Figure 4.6: LRU prediction with reuse information limited to length n at $\mathbf{r}(2^{10})$ which is first computed from $\mathbf{r}(1)$ (Equation 4.2).

$$
\begin{aligned}
r'_0 = \;& r_0 + \;\; (1/2)\cdot r_1 + \;\; (1/4)\cdot r_2 + \;\; (1/8)\cdot r_3 + \;\; \ldots \\
r'_1 = \;& \qquad (1/2)\cdot r_1 + \;\; (2/4)\cdot r_2 + \;\; (3/8)\cdot r_3 + \;\; \ldots \\
r'_2 = \;& \qquad\qquad\quad (1/4)\cdot r_2 + \;\; (3/8)\cdot r_3 + \;\; \ldots \\
r'_3 = \;& \qquad\qquad\qquad\quad (1/8)\cdot r_3 + \;\; \ldots \\
\ldots
\end{aligned}
$$

4.4.2 Matrix dimension and Truncation of $\mathbf{r}$

The dimension of $\mathbf{B}$ is determined by the maximum (per-set) URD that we are interested to maintain to avoid large computational costs. Let n denote the length of the $\mathbf{r}$ vector

that we maintain. That is, $\mathbf{r}(S)$ is computed for $\mathbf{r}_0(S)$ through $\mathbf{r}_{n-1}(S)$, with $\mathbf{r}_\infty(S)$ adjusted so that $\mathbf{r}_\infty(S) = 1 - \sum_{i=0}^{n-1} \mathbf{r}(S)$.

Assume $\mathbf{r}(2^{10})$ is available, computed from $\mathbf{r}(1)$ using Equation 4.2. Figure 4.6 shows predicted miss ratios with $\mathbf{r}(2^{10})$ maintained for various values of n. Section 4.5.1 explains LRU prediction. Although the maximum associativity that we consider is 32, Figure 4.6 shows that n has to be much larger than that ($\geqslant 512$) for good predictions for larger caches with $S' > 2^{10}$, such as 32MB caches (see Table 4.1).

While $n = 512$ is good for $\mathbf{r}(2^{10})$, the equivalent value for $\mathbf{r}(1)$ is very large, potentially up to $512 \cdot 2^{10}$. To appreciate this, consider the $\mathbf{r}(1)$ address stream as a merger of the 2^{10} mutually exclusive per-set address streams, each of which has reuse intervals of up to 512. Determining the long-tailed $\mathbf{r}(1)$ distribution or using large matrices to compute $\mathbf{r}(2^{10})$ from $\mathbf{r}(1)$ in software is time-consuming. Section 4.6.1 proposes low-cost hardware support to approximately estimate $\mathbf{r}(2^{10})$ with $n = 512$.

4.4.3 Poisson approximation to Binomial

Cypher [55, 56] uses a Poisson approximation to binomial for reducing computational costs – when i is large and $\frac{1}{S'}$ is small, the binomial distribution can be approximated by a Poisson distribution with parameter $\lambda = \frac{i}{S'}$. Computing this is faster than the binomial coefficient.

Figure 4.7 shows pseudo-code for the `compute_per_set_r` function that computes Equation 4.3. It uses Poisson approximation to Binomial and assumes that $\mathbf{r}(2^{10})$ up to $n = 512$ is available. $num\_set\_bits \in [1, 8] = \log_2(\frac{S'}{S})$. The computation is done for $2A'$ terms (see Section 4.5).

```
void init() {
    int i;
    for(i=0;i<9;i++)
        precomputed_exp_inc[i]=exp(-1.0/(1<<i));
    for(i=1;i<64;i++)
        precomputed_v[i]=1.0/i;
}

void compute_per_set_r(int num_set_bits, int max_assoc) {
    const double *ptr=&r_histogram[0];
    double s3=precomputed_exp_inc[num_set_bits];
    double s2=1.0;
    double base_lambda=1.0/(1<<num_set_bits);
    double lambda=0;
    int i, rd;
    for(i=0;i<512;i++) {
        double s1=s2;
        for(rd=0;rd<2*max_assoc;rd++) {
            per_set_r[num_set_bits][rd]+=ptr[i]*s1;
            s1*=lambda*precomputed_v[rd+1];
        }
        s2*=s3;
        lambda+=base_lambda;
    }
}
```

Figure 4.7: Equation 4.3 pseudo-code with Poisson approximation.

4.5 Cache Hit Functions

Given a target cache organization $(S', A', \texttt{policy})$ and a trace T, our goal is to determine a vector $\boldsymbol{\phi}(\mathbf{r}(S'), S', A', \texttt{policy})$ such that the expected hit ratio for the trace is

$$h = \mathbf{r} \cdot \boldsymbol{\phi} \tag{4.4}$$

The idea is to characterize workload traces by $\mathbf{r}$ and caches by $\boldsymbol{\phi}$ so that the effect on hit ratio for changes in traces or cache configurations can be readily estimated.

We call $\boldsymbol{\phi}$ the cache hit function. The value of the k^{th} component, $\boldsymbol{\phi}_k$ is the conditional

probability of a hit for accesses x such that $\mathrm{URD}(x, m) = k$ where m is the repetition count for x at that point in the trace when the access happens. $\boldsymbol{\phi}_k$ monotonically decreases with k in this model. This is because non-eviction of a cache-resident address after accesses involving k other unique addresses implies non-eviction after accesses involving $k-1$ unique addresses *and* the remaining accesses. If there are no intervening accesses ($k = 0$), the access must be a hit. Accesses hitherto never seen ($k = \infty$) must miss. So,

$$\boldsymbol{\phi}_k = \begin{cases} 1 & \text{if } k = 0 \\ \leqslant \boldsymbol{\phi}_{k-1} & \text{if } k \geqslant 1 \\ 0 & \text{at } k = \infty \end{cases} \tag{4.5}$$

Figure 4.8 shows $\boldsymbol{\phi}$ curves for common replacement policies. We consider the well-known, but rarely-implemented[1] LRU policy as well as the practical RANDOM, NMRU, and PLRU policies. In each figure, we superimpose the $\boldsymbol{\phi}$ curves for $S' = 2^{14}$ and $A' = 2$, 4, 8, 16, and 32. These five curves appear from left to right, in that order, in each figure. $\boldsymbol{\phi}(\mathrm{LRU})$ is always a step function, with the 1-to-0 transition happening at A'. We show $\boldsymbol{\phi}(\mathrm{LRU})$ on each figure for comparison. Note that $\boldsymbol{\phi}$ for RANDOM, NMRU, PLRU are non-zero beyond A'. So, computing $\mathbf{r} \cdot \boldsymbol{\phi}$ up to A' is not sufficient for these replacement policies. For our evaluations, we compute the dot-product for $2A'$ terms; longer than that has diminishing returns for our workloads.

Apart from LRU, $\boldsymbol{\phi}$ is not independent of $\mathbf{r}$ for different replacement policies. As we shall show later, $\boldsymbol{\phi}(\mathrm{RANDOM})$ depends on $\mathbf{d}$, while $\boldsymbol{\phi}(\mathrm{PLRU})$ may need more information.

[1]LRU is typically not implemented in real caches for associativity larger than 4 due to hardware complexity.

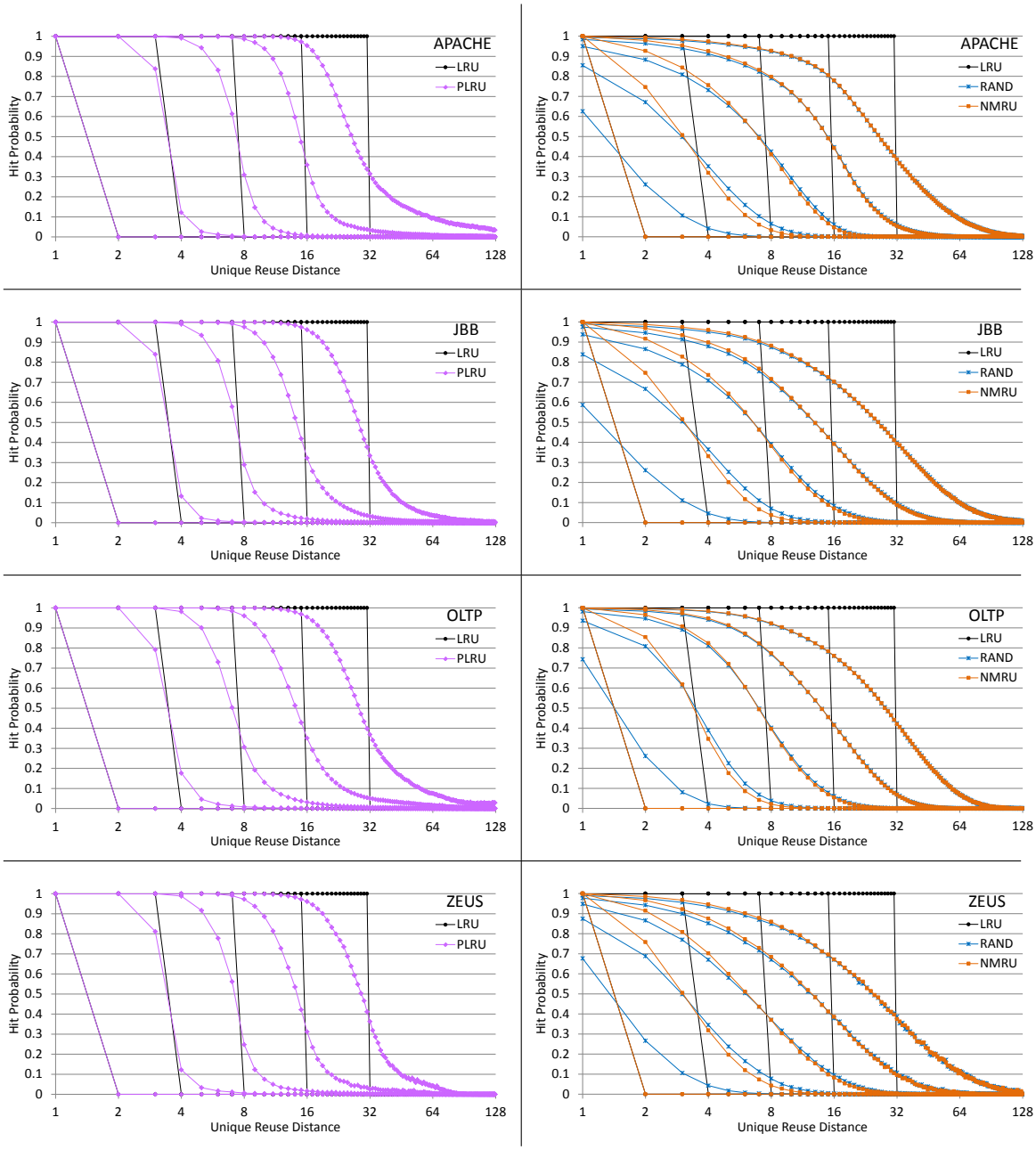


Figure 4.8: Representative hit ratio functions (Φ_k) with $S' = 2^{14}$ and $A' = 2, 4, 8, 16, 32$ with different replacement policies. Φ_0(not shown) $= 1$ for all policies. Note that the x-axes are in $\log_2$ scale. Only LRU has a step function for any associativity.

4.5.1 Estimating $\phi(\mathbf{LRU})$

For a set-associative LRU cache with associativity A', it is well known that all accesses with addresses re-appearing with less than A' unique intervening elements must hit and all other accesses must miss. This leads us to the following characterization of the LRU hit ratio function.

$$\boldsymbol{\phi}_k(\mathrm{LRU}) = \begin{cases} 1 & \text{if } 0 \leqslant k < A' \\ 0 & \text{if } k \geqslant A' \end{cases} \tag{4.6}$$

Figure 4.6 shows actual vs estimated ($n = 512$) miss ratios with LRU using Equations 4.3, 4.6 and 4.4. As observed earlier by Hill and Smith [103], increasing A' yields diminishing returns.

4.5.1.1 Optimization (Smith's Model)

A naive combination of Equations 4.4, 4.6 and 4.3 results in $\left(\sum_{i=0}^{A'-1}(n-i)\right) - 1 = nA' - \frac{A'(A'-1)}{2} - 1$ multiplications with binomial computations to estimate the hit ratio for a cache with S' sets and associativity A'. The number of multiplications can be reduced by observing that due to the step-function nature of $\boldsymbol{\phi}(\mathrm{LRU})$, some of the coefficients will sum to 1. Expanding the computation and simplifying, we get

$$h(S') = \sum_{i=0}^{A'-1} \mathbf{r}_i(1) + \sum_{i=A'}^{n-1} \mathbf{r}_i(1) \cdot \sum_{k=0}^{A'-1} {}^{i}C_k \cdot \left(\frac{1}{S'}\right)^k \cdot \left(1 - \frac{1}{S'}\right)^{(i-k)} \tag{4.7}$$

Equation 4.7 is an optimized version of Smith's associativity model [103, 200]. It requires $(n - A')A'$ multiplications which is $\frac{A'(A'+1)}{2} - 1$ less than the naive combination. But computing binomial terms is costly and n is usually much larger than A', so the gains from this optimization are small.

4.5.2 Estimating ϕ(RANDOM)

The RAND replacement algorithm [25] (also popularly called RANDOM) chooses a line (uniformly) randomly from the lines in the set for eviction on a miss.

For an A'-way set-associative cache, the probability of replacement of a given line on a miss is $\frac{1}{A'}$. Accounting for the number of misses in between successive reuses of an address is therefore needed. For expected miss rate θ, the expected number of misses for a sequence of α accesses is $\alpha \cdot \theta$. This is why $\mathbf{d}$ is important for RANDOM whereas LRU works independent of such information.

We make the simplifying assumption that miss occurrences (not specific addresses) are independent and hence amenable to be modeled as a Bernoulli process. While this may not be accurate, it allows us to make reasonably good predictions without tracking additional state.

Let $\mathbf{d}_k = \alpha$. The probability of i misses is estimated by ${}^{\alpha}C_i \cdot \theta^i \cdot (1-\theta)^{(\alpha-i)}$. The probability that a specific line is not replaced after i misses is $\left(1 - \frac{1}{A'}\right)^i$. We thus have

$$\begin{aligned} h(\text{RANDOM}) &= \mathbf{r} \cdot \boldsymbol{\phi}(\text{RANDOM}) \\ \boldsymbol{\phi}_k(\text{RANDOM}) &= \sum_{i=0}^{\alpha | \mathbf{d}_k = \alpha} {}^{\alpha}C_i \cdot \theta^i \cdot (1-\theta)^{(\alpha-i)} \cdot \left(1 - \frac{1}{A'}\right)^i \\ \theta &= 1 - h(\text{RANDOM}) \end{aligned} \tag{4.8}$$

To simplify the computation, we approximate Binomial(α,θ) by Poisson($\lambda = \alpha \cdot \theta$). Let $q = \left(1 - \frac{1}{A'}\right)$. This gives

$$\begin{aligned}
\boldsymbol{\phi}_k(\text{RANDOM}) &= \sum_{i=0}^{\alpha|\mathbf{d}_k=\alpha} {}^{\alpha}C_i \cdot \theta^i \cdot (1-\theta)^{(\alpha-i)} \cdot q^i \\
&= \sum_{i=0}^{\infty} {}^{\alpha}C_i \cdot \theta^i \cdot (1-\theta)^{(\alpha-i)} \cdot q^i \\
&\approx \sum_{i=0}^{\infty} e^{-\lambda} \cdot \frac{\lambda^i}{i!} \cdot q^i \\
&= e^{-\lambda(1-q)} \sum_{i=0}^{\infty} e^{-\lambda q} \cdot \frac{(\lambda q)^i}{i!} \\
&= e^{\frac{-\alpha\theta}{A'}} \qquad (4.9)
\end{aligned}$$

The system of equations in 4.8 can now be approximated by the following system.

$$\begin{aligned}
h(\text{RANDOM}) &= \mathbf{r} \cdot \boldsymbol{\phi}(\text{RANDOM}) \\
\boldsymbol{\phi}_k(\text{RANDOM}) &= e^{\frac{-\mathbf{d}_k\theta}{A'}} \\
\theta &= 1 - h(\text{RANDOM}) \qquad (4.10)
\end{aligned}$$

We solve the system of equations in 4.10 with the initial value $h = \mathbf{r}_0$. $\mathbf{d}$ is estimated using equation 4.1. Usually 5 or fewer iterations suffice to reach within 1% of a fix-point.

4.5.2.1 Convergence for RANDOM

First note that if a fix-point exists, the solution satisfies the general conditions of $\boldsymbol{\phi}$ (Equation 4.5). This is because $\mathbf{d}_0 = 0$ (Equation 4.1) and from Equation 4.10,

$$\begin{aligned}
\frac{\boldsymbol{\phi}_k}{\boldsymbol{\phi}_{k-1}} &= e^{\frac{-(\mathbf{d}_k-\mathbf{d}_{k-1})\theta}{A'}} = e^{-\left(\frac{\theta/A'}{\sum_{i=k}^{\infty} r_i(T)}\right)} \\
&\leqslant 1 \qquad (4.11)
\end{aligned}$$

Let H denote a fix-point and h^0, h^1, h^2, ... denote successive approximations. By

re-arranging the system of equations in 4.10 we have

$$h^{j+1} = r_0 + \sum_{i=0}^{n-1} r_i e^{\frac{d_i(-1+h^j)}{A'}} \tag{4.12}$$

Since the exponential function is monotonic, H must be unique. Since $0 \leqslant r_i \leqslant 1, \forall i$, $r_0 \leqslant H \leqslant 1$.

Also, it is easy to show that $h^j \geqslant h^{j-1} \implies h^{j+1} \geqslant h^j$. Thus, successive iterations produce a chain of values $r_0 = h^0 \leqslant h^1 \leqslant h^2$....

We will now prove that $h^j \leqslant H, \forall j$. This is true at $j = 0$. For induction, let $h^j = H - \epsilon$ with $\epsilon \geqslant 0$. Then,

$$\begin{aligned} h^{j+1} &= r_0 + \sum_{i=0}^{n-1} r_i e^{\frac{d_i(-1+h^j)}{A'}} \\ &= r_0 + \sum_{i=0}^{n-1} r_i e^{\frac{d_i(-1+H)}{A'}} \cdot e^{-\frac{d_i \epsilon}{A'}} \\ &\leqslant r_0 + \sum_{i=0}^{n-1} r_i e^{\frac{d_i(-1+H)}{A'}} = H \end{aligned} \tag{4.13}$$

This shows a convergence chain $r_0 = h^0 \leqslant h^1 \leqslant h^2 ... \leqslant H$.

4.5.2.2 Optimization

A better approximation for $A' = 2$ can be obtained by using the fact that for the reference element not to be evicted at $URD \geqslant 2$, the previous element must be evicted (since the set can hold only 2 elements). The probability of the previous element to be evicted is $1 - \phi_1$. For the reference element to hit at $URD = k$, it must hit at $URD = k - 1$ and the

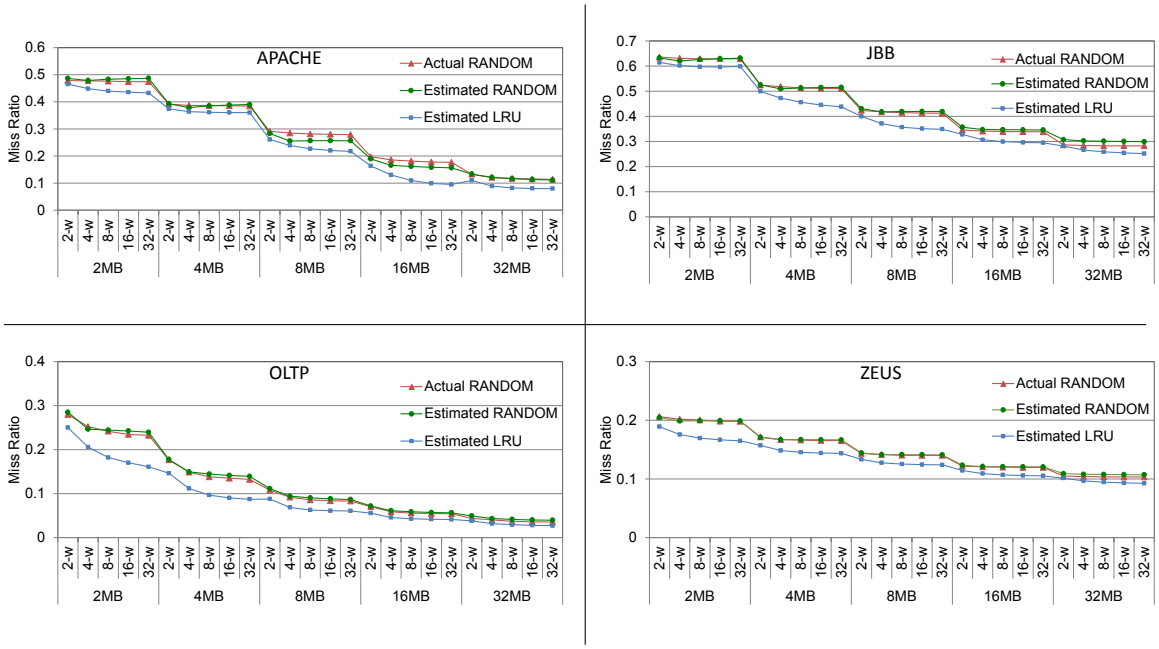


Figure 4.9: Actual vs estimated miss ratios with RANDOM replacement policy. LRU estimates are shown as reference. $\mathbf{r}(2^{10})$ is first computed from $\mathbf{r}(1)$ (Equation 4.2).

above condition must hold. This leads us to the following approximation.

$$\boldsymbol{\phi}_k \quad = \quad \boldsymbol{\phi}_{k-1} \cdot (1 - \boldsymbol{\phi}_1), \quad k \geqslant 2 \tag{4.14}$$

This approximation is possible since the model can exactly determine the set contents for URD $>= 2$. For higher associativities, exact determination of set contents is difficult.

Figure 4.9 shows actual vs estimated ($n = 512$) values of miss ratios for RANDOM with the estimates computed using Equations 4.3, 4.10, 4.14 and 4.4.

4.5.3 Estimating $\boldsymbol{\phi}$(NMRU)

The NMRU (or non-MRU) replacement algorithm differentiates the most recently accessed (MRU) line from other lines in the set [203]. On a miss, a line is chosen (uniformly)

randomly from among the $A' - 1$ non-MRU lines.

At $A' = 2$, $\boldsymbol{\phi}(\mathrm{NMRU}) = \boldsymbol{\phi}(\mathrm{LRU})$. For the rest of the cases, the framework is similar to that of RANDOM except that accesses at $\mathrm{URD} \leqslant 1$ are guaranteed to hit. Moreover, the replacement logic has $A' - 1$ possible choices for an eviction in case of a miss. This leads to a few simple modifications to the system of equations in 4.10. The modified system is shown below:

$$\begin{aligned} \boldsymbol{\phi}_1(\mathrm{NMRU}) &= 1 \\ h(\mathrm{NMRU}) &= \mathbf{r} \cdot \boldsymbol{\phi}(\mathrm{NMRU}) \\ \boldsymbol{\phi}_k(\mathrm{NMRU}) &= e^{\frac{-(\mathbf{d}_k - \mathbf{d}_1)\theta}{A'-1}} \\ \theta &= 1 - h(\mathrm{NMRU}) \end{aligned} \tag{4.15}$$

Figure 4.10 shows actual vs estimated ($n = 512$) values of miss ratios for NMRU with the estimates computed using Equations 4.3, 4.15 and 4.4.

4.5.4 Estimating $\boldsymbol{\phi}$(PLRU)

Partitioned LRU [203] (also popularly called pseudo-LRU) maintains a balanced binary tree that, at each level, differentiates between the two sub-trees based on access recency. Every internal node is represented by a single bit whose value decides which of the two subtrees was accessed more recently. The cache lines are represented by the leaves of the tree. Whenever a line is accessed, the nodes on the path from the root to the leaf flip their bit values, thus pointing to the other subtree at each level. On a miss, the subtree pointed to is chosen, recursively starting from the root. The line corresponding to the leaf reached in this way is chosen for eviction. The bit-values along this path are then flipped.

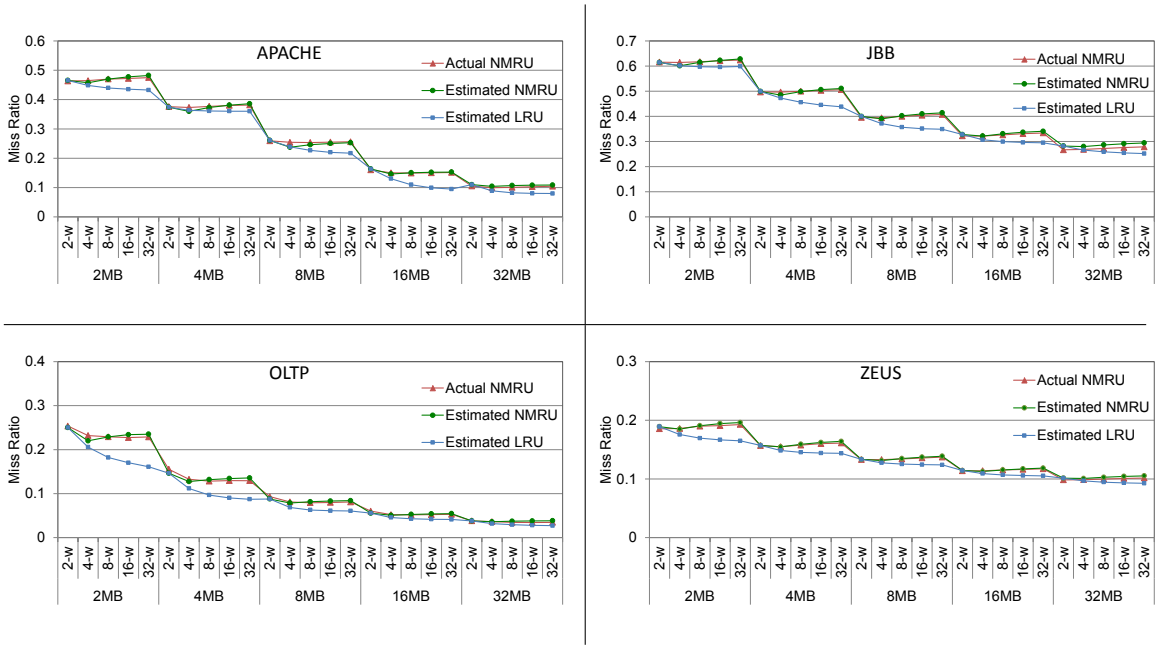


Figure 4.10: Actual vs estimated miss ratios with NMRU replacement policy. LRU estimates are shown as reference. $\mathbf{r}(2^{10})$ is first computed from $\mathbf{r}(1)$ (Equation 4.2).

In the PLRU scheme, the most recently accessed element is always known but the least recently accessed one is not. In contrast to the LRU scheme, that maintains a total access order between the lines, PLRU maintains only a partial order. Since there is no difference between partial and total orders involving 2 elements, PLRU is LRU when $A' = 2$. In contrast to LRU that guarantees exactly $A' - 1$ unique accesses before eviction, PLRU guarantees at least $\log_2(A')$ (=number of tree levels) unique accesses before the reference address is evicted.

Since the PLRU tree is symmetric, we can fix any way as reference without loss of generality. Let the immediate neighbor be denoted by Q_0, the next two neighbors be collectively denoted by Q_1 and so on with the most distant group of $A/2$ neighbors denoted by $Q_{\log_2(A)-1}$. To calculate the probability that the reference line will be evicted

on a particular miss we need to consider the immediate past sequence of accesses to that set. A necessary and sufficient condition for the reference line to be evicted is for the suffix of the trace to have accesses that match the particular regular expression described below.

$$
\begin{aligned}
A = 2 \quad &: \quad Q_0^+ \\
A = 4 \quad &: \quad Q_0 Q_1^+ \\
A = 8 \quad &: \quad Q_0 (Q_1 + Q_2)^* Q_1 Q_2^+ \\
A = 16 \quad &: \quad Q_0 (Q_1 + Q_2 + Q_3)^* Q_1 (Q_2 + Q_3)^* Q_2 Q_3^+ \\
A = 32 \quad &: \quad Q_0 (Q_1 + Q_2 + Q_3 + Q_4)^* Q_1 (Q_2 + Q_3 + Q_4)^* \\
&\qquad Q_2 (Q_3 + Q_4)^* Q_3 Q_4^+
\end{aligned}
$$

On a miss, the reference line will be evicted if and only if the immediately preceding sequence of accesses follows a particular pattern. These patterns can be described using regular expressions. In contrast to RANDOM, not only the number of misses in the reuse interval, but also the pattern of accesses determines eviction probability. It is difficult to estimate $\boldsymbol{\phi}(\mathrm{PLRU})$ by computing probabilities of the regular expressions since the distance to misses within the reuse interval as well as the ways occupied by the intervening elements are not known. Instead, we use a different approach.

First, we compute $\boldsymbol{\phi}(A' = 4, \mathrm{PLRU})$ then compute $\boldsymbol{\phi}(A' = 8, \mathrm{PLRU})$ by dividing traffic using a binomial distribution and applying $\boldsymbol{\phi}(A' = 4, \mathrm{PLRU})$ on the divided traffic. We view an 8-way tree as a composition of two 4-way trees with the top-node dividing traffic between the two subtrees. Similar observations hold between 8-way and 16-way trees and so on. This helps us to estimate $\boldsymbol{\phi}(\mathrm{PLRU})$ for successively higher associativities. We assume that the top node divides traffic evenly between its two constituent sub-trees.

4.5.4.1 Base case: $A' = 4$

Since $\log_2(4) = 2$, ϕ_k is 1 when $k \leqslant 2$. Let x denote the reference element. We will now estimate the likelihood that the second occurrence of x in the access sequence $x\ e_1$ ... e_2 ... $e_k\ x$ will hit in the cache. The elements e_1 through e_k all map to the same cache set and are distinct so that the URD of the sequence is k. $\phi_k = \phi_{k-1} \cdot$ P(x not evicted by e_k).

First, consider the case when $k \geqslant 4$. To have $\phi_k = 1$, x must be present in the cache. Moreover, both e_{k-1} and e_k will also be in the cache as PLRU guarantees that the last two unique elements seen will remain in the cache. Since $A' = 4$, there is room for one more element other than x, e_{k-1}, and e_k in the cache set. This element must be e_{k-2} as it could not have been evicted by either e_{k-1} or e_k. Therefore, e_{k-3} must have been evicted by e_k. So if e_{k-3} were to reappear instead of the second occurrence of x in the above sequence, it would miss. That is, the access sequence e_{k-3} ... e_{k-2} ... e_{k-1} ... e_k ... e_{k-3} would have caused the second e_{k-3}, with URD 3, to miss. This probability is $(1 - \phi_3)$. Thus, P(x not evicted by e_k) = P(e_{k-3} evicted by e_k) = $(1 - \phi_3)$. So, $\phi_k = \phi_{k-1} \cdot (1 - \phi_3)$.

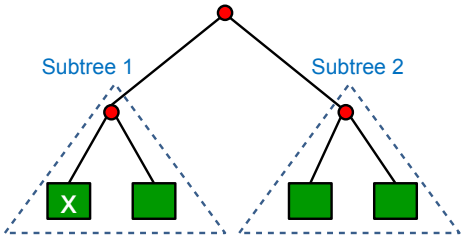


Figure 4.11: Schematic showing PLRU subtrees for $A' = 4$. Without loss of generality, we denote the subtree containing the reference element, x, as Subtree 1. Also, x is not necessarily the left-most child of Subtree 1.

The case that remains is when $k = 3$. For this case, we will refer to Figure 4.11 to

describe our estimation approach. The access sequence that we are considering is x e_1 ... e_2 ... e_3 x, with e_1, e_2, e_3 being distinct elements. The only scenario where x is evicted (by e_3) before its second occurrence occurs if all of the following conditions hold:

1. e_3 misses. e_3 has URD $\geqslant 3$. For an approximation, we just consider what would happen under LRU. It would miss for URD > 3. The probability for this happening is P(URD $> 3 \,|\, \text{URD} \geqslant 3$) = 1 - P(URD $= 3 \,|\, \text{URD} \geqslant 3$) = 1 - $\frac{r_3}{1-r_0-r_1-r_2}$.

2. The last access (to either e_2 or e_3) before the access to e_3 causes Subtree 2, not containing x, to be accessed. We assume this probability to be $\frac{1}{2}$.

3. e_1 and e_2 map to different subtrees. Since each subtree can have only 2 elements, Subtree 2 must get at least one of e_1 or e_2. Thus, the question is whether or not the other element (e_1 or e_2) maps to Subtree 1. We assume this probability to be $\frac{1}{2}$.

Thus, $\phi_3 = 1 - \left(1 - \frac{r_3}{1-r_0-r_1-r_2}\right) \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{3}{4} + \frac{1}{4} \cdot \left(\frac{r_3}{1-r_0-r_1-r_2}\right)$.
Putting everything together,

$$\phi_k = \begin{cases} 1 & \text{if } 0 \leqslant k \leqslant 2 \\ \frac{3}{4} + \frac{1}{4} \cdot \left(\frac{r_3}{1-r_0-r_1-r_2}\right) & \text{if } k = 3 \\ \phi_{k-1} \cdot (1 - \phi_3) & \text{if } k \geqslant 4 \end{cases} \tag{4.16}$$

4.5.4.2 Recurrence: $A' \geqslant 8$

Let $L = \log_2(A')$ and $\psi = \phi(A'/2)$. We will refer to Figure 4.12 to describe our estimation approach. For the first case, when $k \leqslant L$, ϕ_k must be 1. For $k > L$, consider the element e_k in the access sequence x e_1 ... e_2 ... e_3 x. There are two subcases here:

1. It maps to Subtree 2. In this case, $\phi_k = \phi_{k-1}$.

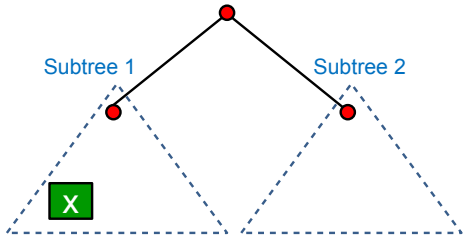


Figure 4.12: Schematic showing PLRU subtrees. Each subtree has $A'/2$ leaves. Without loss of generality, we denote the subtree containing the reference element, x, as Subtree 1.

2. It maps to Subtree 1. If $k \geqslant \frac{A'}{2} + 2$, there is at least one other element in Subtree 1 apart from x and e_k. This is because at most $\frac{A'}{2}$ elements can map to Subtree 2 before an element maps to Subtree 1. So, an element within the set $\{e_1 \ldots e_{\frac{A'}{2}+1}\}$ must map to Subtree 1. If $k \leqslant \frac{A'}{2} + 1$, the $\frac{A'}{2}$ elements other than $e_k$can all occupy Subtree 2.

   The remaining elements can map to Subtree 1 or Subtree 2. We use a Binomial distribution with success probability $\frac{1}{2}$ to estimate the likelihood of a certain number of them mapping to Subtree 1. This number plus 1 (for e_k) gives the URD for x considering only accesses to Subtree 1. We then get the hit probability for this URD from ψ.

Putting everything together,

$$\Phi_k = \begin{cases} 1 & \text{if } 0 \leqslant k \leqslant L \\ \dfrac{\Phi_{k-1}}{2} + \dfrac{1}{2}\displaystyle\sum_{i=0}^{k-3} {}^{k-3}C_i \left(\frac{1}{2}\right)^{(k-3)} \cdot \psi_{2+i} & \text{if } k \geqslant \dfrac{A'}{2} + 2 \\ \dfrac{\Phi_{k-1}}{2} + \dfrac{1}{2}\displaystyle\sum_{i=0}^{k-2} {}^{k-2}C_i \left(\frac{1}{2}\right)^{(k-2)} \cdot \psi_{1+i} & \text{otherwise} \end{cases} \tag{4.17}$$

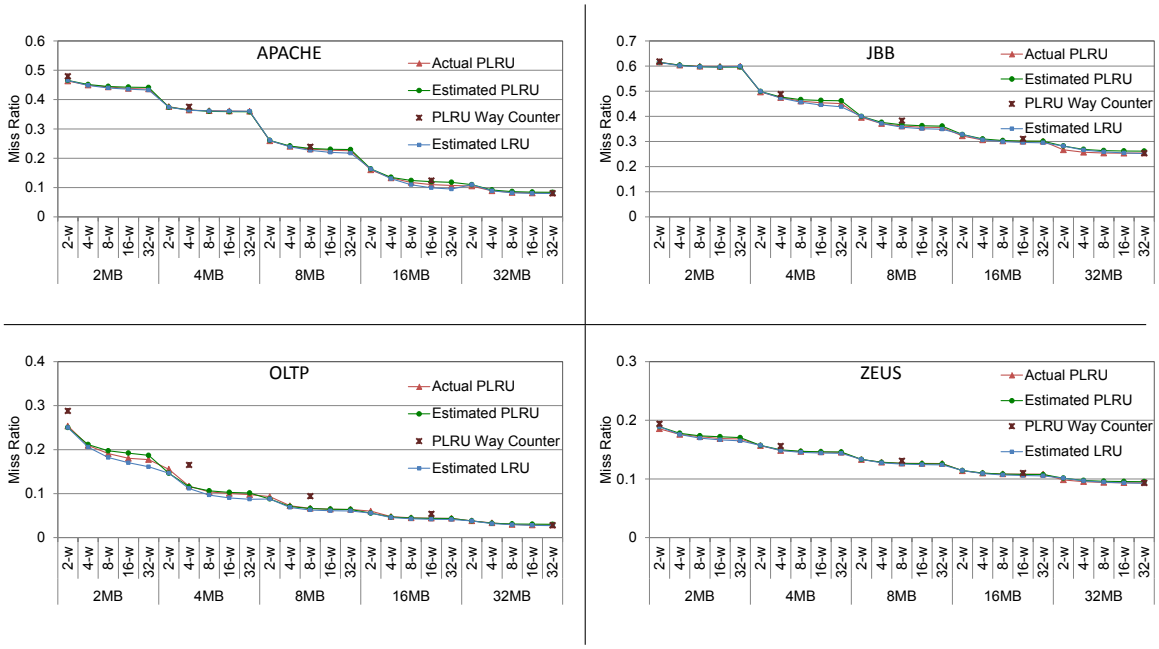


Figure 4.13: Actual vs estimated miss ratios with PLRU replacement policy. LRU estimates are shown as reference. $\mathbf{r}(2^{10})$ is first computed from $\mathbf{r}(1)$ (Equation 4.2). Section 4.6.2.1 describes PLRU Way-Counters.

In the above, we ignore the case when all of $\{e_1 \ldots e_k\}$ map to Subtree 1 along with x. This occurrence has a low probability since all of $\{e_1 \ldots e_k\}$ must have been hits (probability = P(hit)^k) and already have been present in Subtree 1 (probability = 2^{-k}).

Figure 4.13 shows actual vs estimated ($n = 512$) values of miss ratios for PLRU with the estimates computed using Equations 4.3, 4.16, 4.17 and 4.4.

4.5.5 Estimation Accuracy and Computation Time

Table 4.5 shows miss ratio prediction errors for different policies and workloads. LRU prediction is the most accurate, with relative errors < 2%, followed by PLRU with relative errors < 3%. Using the PLRU predictor instead of the LRU predictor when the actual cache uses PLRU improves prediction accuracy by ~2% for `oltp`. RANDOM and NMRU

Workload	LRU		RANDOM		NMRU		PLRU		LRU→PLRU	
	Abs.	*Rel.*	*Abs.*	*Rel.*	*Abs.*	*Rel.*	*Abs.*	*Rel.*	*Abs.*	*Rel.*
`apache`	1.23	0.81	5.12	2.67	4.68	2.27	3.41	2.31	3.40	2.23
`jbb`	3.40	1.12	7.80	2.27	6.58	1.90	5.24	1.59	4.44	1.29
`oltp`	1.59	1.85	4.04	4.90	3.77	4.05	2.88	2.97	4.88	5.18
`zeus`	0.69	0.57	2.68	1.96	1.78	1.36	1.21	0.96	1.55	1.19

Table 4.5: Average absolute values of prediction errors over all cache configurations. *Abs.* = (predicted - actual) miss ratio $\times 10^3$, *Rel.* = (predicted/actual - 1)$\times 10^2$ (to express as a percentage). LRU→PLRU shows what happens if the LRU predictor is used to predict for PLRU instead of using the PLRU predictor.

have relative errors < 5%.

A major contributor to hit ratio computation time is the determination of $\mathbf{r}$. Section 4.6.1 proposes low-cost hardware to approximate $\mathbf{r}(2^{10})$ (with $n = 512$) online. Assuming this is available, the hit ratio computation time per cache configuration on the Haswell machine (HS, at 3.9 GHz) were – LRU: $\leqslant$ 0.009 msec; PLRU: $\leqslant$ 0.011 msec; RANDOM: $\leqslant$ 0.012 msec; NMRU: $\leqslant$ 0.012 msec. On a Nehalem 2.26 GHz machine, the times were – LRU: $\leqslant$ 0.016 msec; PLRU: $\leqslant$ 0.018 msec; RANDOM: $\leqslant$ 0.023 msec; NMRU: $\leqslant$ 0.024 msec. This includes the time to compute $\mathbf{r}(S')$ from $\mathbf{r}(2^{10})$ (Equation 4.3), amortized over all configurations with the same S'.

4.6 Hardware Support

Section 4.4.2 discussed that to avoid expensive computation to determine $\mathbf{r}(1)$ or compute $\mathbf{r}(2^{10})$ from $\mathbf{r}(1)$, we need hardware support to directly estimate $\mathbf{r}(2^{10})$. Section 4.6.1 presents our proposed hardware technique to do this.

Section 4.6.2 discuss two traditional hardware mechanisms that help in cache miss ratio estimation—set-counters and way-counters.

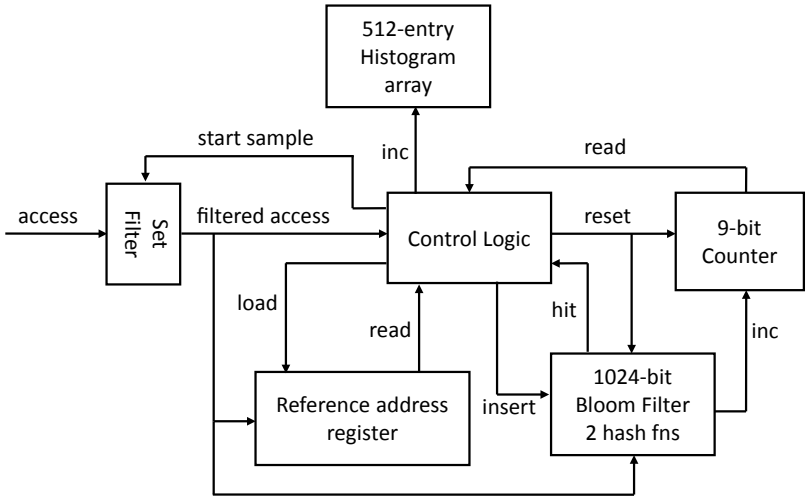


Figure 4.14: Schematic of new hardware support.

4.6.1 New hardware support to estimate reuse distributions ($\mathbf{r(2^{10}), n = 512}$)

The definition of unique reuse distance (URD) depends *only on the cardinality of the reuse interval (*RI*) and not on the contents of the set*. This suggests applicability of hardware signatures, such as Bloom filters [32], that can construct compact representations of sets. Whereas shadow tags store entire tag addresses, a Bloom filter uses only one bit per hash function to represent each address.

Our proposed hardware, shown in Figure 4.14, uses a Bloom filter (to summarize RI), a counter to determine |RI|, and set-sampling logic. We use a 1024-bit parallel Bloom filter [187] with two H_3 hash functions [42] and a 9-bit counter. The Bloom filter can be at most half-full (512 elements) before being reset. Larger Bloom filters can be used to reduce aliasing errors at the cost of more area/power overhead. The hardware uses a combination of set sampling and time sampling techniques [129, 132, 133, 172, 222]. It works as follows:

1. **Sample Initiation**: The Control Logic initializes the Set Filter to match a single set

of a cache with $S = 2^{10}$. It chooses this value by first time-sampling the incoming address stream (10% selectivity) and then choosing the set number of the chosen address as the value for the Set Filter. It also saves this address in the Reference address register.

2. **Sample Continuation**: The Control Logic inspects (see step 3) each address that passes through the Set Filter. Then it inserts the address into the Bloom Filter and increments the 9-bit Counter provided that the Bloom Filter does not return a match (already seen) for the address.

3. **Sample Termination**: This happens in one of two cases—(i) the reference address is seen again, or (ii) the maximum value (511) for the 9-bit counter is reached before inserting another new element. For case (i), the Control Logic increments the entry (whose position matches the 9-bit Counter value) in the Histogram array. For both cases, it transitions to Sample Initiation mode.

The above process is repeated. Each sequence of steps 1–3 estimates the reuse distance of a single address in the address stream. This value is between 0–511 (both inclusive) or considered as ∞ ($\geqslant 512$) otherwise. A separate counter (not shown) tracks the total number of measurements. This value, together with the histogram entries, is used to estimate $\mathbf{r}(2^{10})$.

The technique can be generalized to estimate $\mathbf{r}(2^x)$ by sizing the Bloom filter, histogram, and set filter appropriately.

4.6.1.1 Bloom Filter Analysis

The Bloom Filter is used to estimate the number of unique addresses. Here we analyze its performance by comparing three kinds of filters:

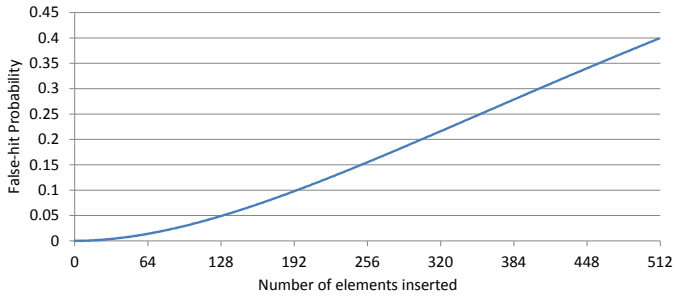


Figure 4.15: Probability of false hit in a 1024-bit Bloom filter with 2 hash functions.

1. **E** (Exact): This assumes that the addresses are fully tracked so that the estimation of the number of unique addresses is accurate.

2. **B** (Bloom): This uses traditional Bloom Filters. Addresses are not fully tracked, but represented by a few bits (2 bits in our study). Aliasing (same bits set for different addresses) may result in under-reporting of the number of unique addresses seen. This shortens the reuse distance measured.

3. **CB** (Bloom with Correction): This uses traditional Bloom Filters, but applies a correction term, based on the expected number of false aliases, to the measured reuse distance.

The aliasing probability for a traditional Bloom Filter (**B**) increases with the number of elements (addresses) inserted as more bits get set in the filter. For a Bloom Filter of size m bits and k hash functions, the probability of a false hit (alias) with n elements already inserted is given by

$$\text{P(false hit } \mid n \text{ elements inserted)} \approx (1 - e^{-kn/m})^k$$

For our study, $m = 1024$ and $k = 2$. So the aliasing probability is

$$\text{P(false hit } \mid n \text{ elements inserted)} \approx (1 - e^{-n/512})^2$$

Figure 4.15 plots this probability. We only plot till 511 elements since measurement is terminated beyond that and the reuse distance is considered as ∞.

To correct for this aliasing, the **CB** filter tracks the number of lookups for every state (number of elements already inserted) of the Bloom filter and computes an expectation of the total number of aliases. It then adds this count to the reuse distance measured. Note that the computation for the expected number of aliases may not match the number of *unique* aliases, so the correction is not exact.

For our analyses, the **E**, **B**, and **CB** filters use the same random numbers. However, the starting points of the samples can differ. This is because Sample Termination (followed by Sample Initiation) that happens when a long reuse distance ($\geqslant$ 512, equiv. ∞) is encountered, is affected by how accurately the reuse distance is calculated. So, individual samples across the three filters are not comparable. We compare the estimated miss ratios computed from sample results for the three filters.

Figures 4.16, 4.18, and 4.20 show the estimated miss ratios for LRU using **E**, **B**, and **CB** filters respectively along with the Actual LRU miss ratio. Figures 4.17, 4.19, and 4.21 show the corresponding errors—*absolute* = (estimated-actual), *relative* = (estimated/actual-1). The errors are also tabulated in Tables 4.6 and 4.7. For each analysis, we replicate the estimation hardware (except the Histogram array) to experiment with 2, 4, 8, 16, 32, and 64 Filters.

We find that **CB** filters reduce errors compared to **B** filters for some, but not all, cases. But it incurs additional complexity for applying the (approximate) corrections. **E** filters can have very low errors, e.g., for `apache` (#F=16) and `oltp` (#F=4), but are costly to implement. The **B** filters provide reasonable accuracy at low implementation cost. Surprisingly, increasing the number of filters (for any filter type) does not always increase accuracy for our experiments. We will discuss this issue shortly.

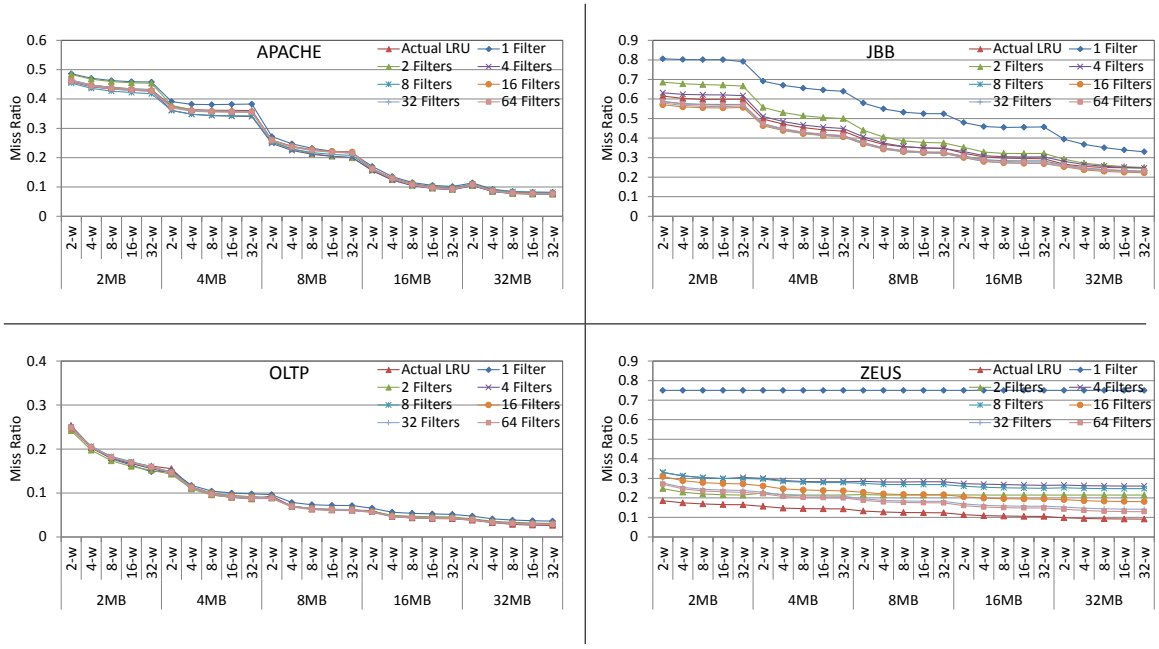


Figure 4.16: Online estimation of miss ratios using **E** filters.

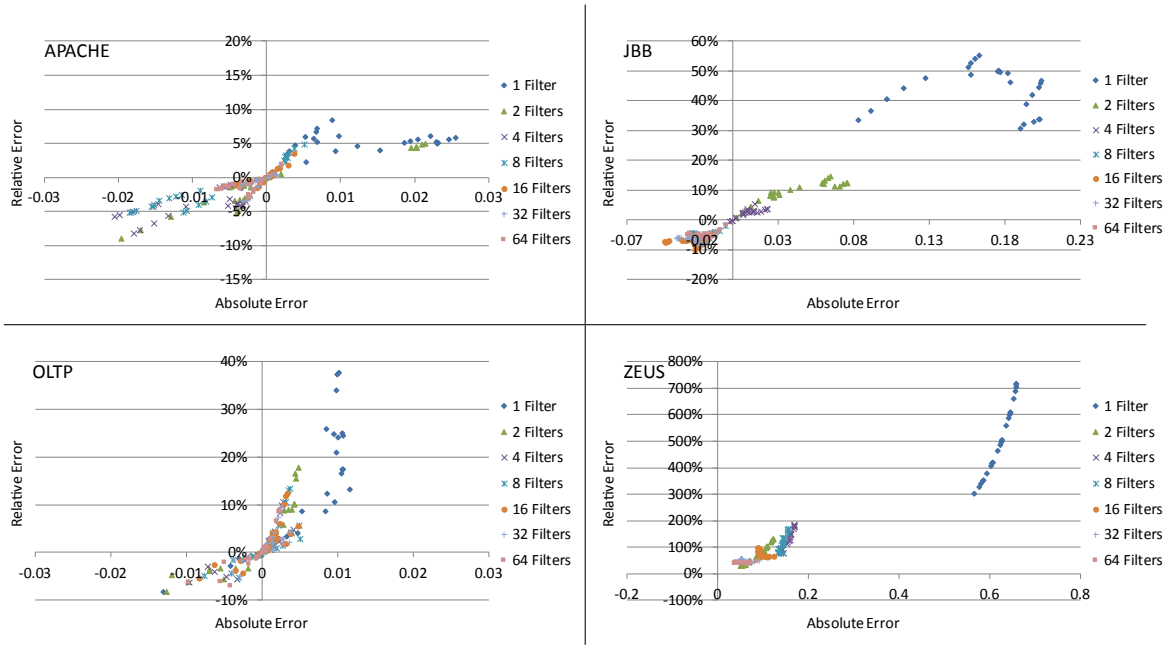


Figure 4.17: Online estimation errors with **E** filters.

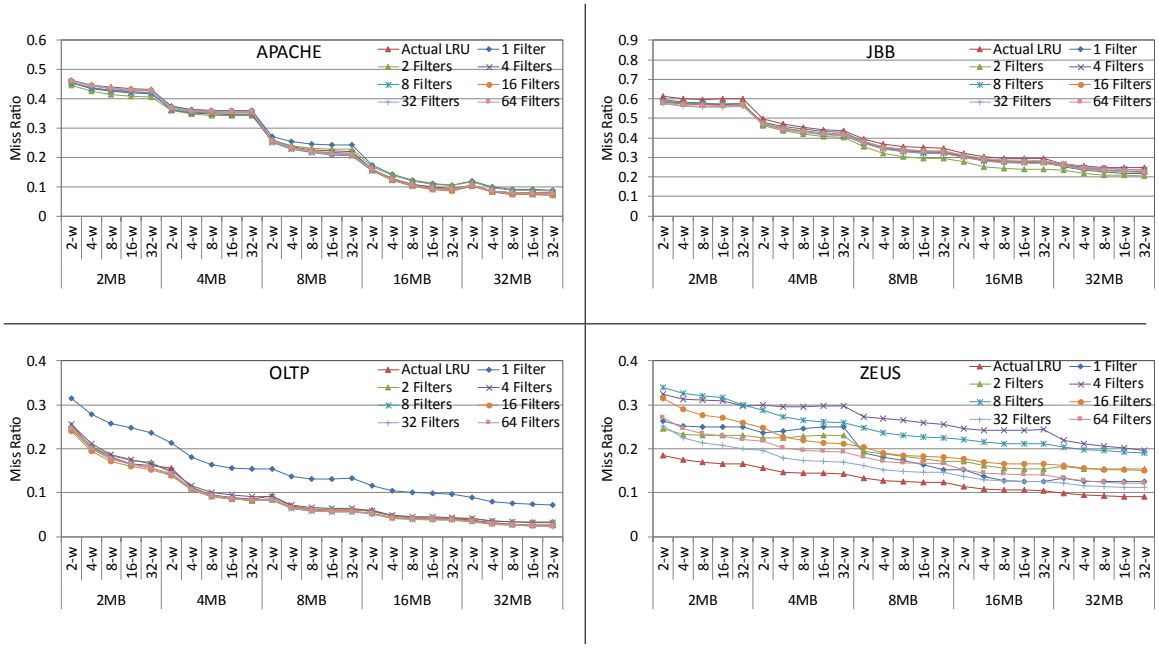


Figure 4.18: Online estimation of miss ratios using **B** filters.

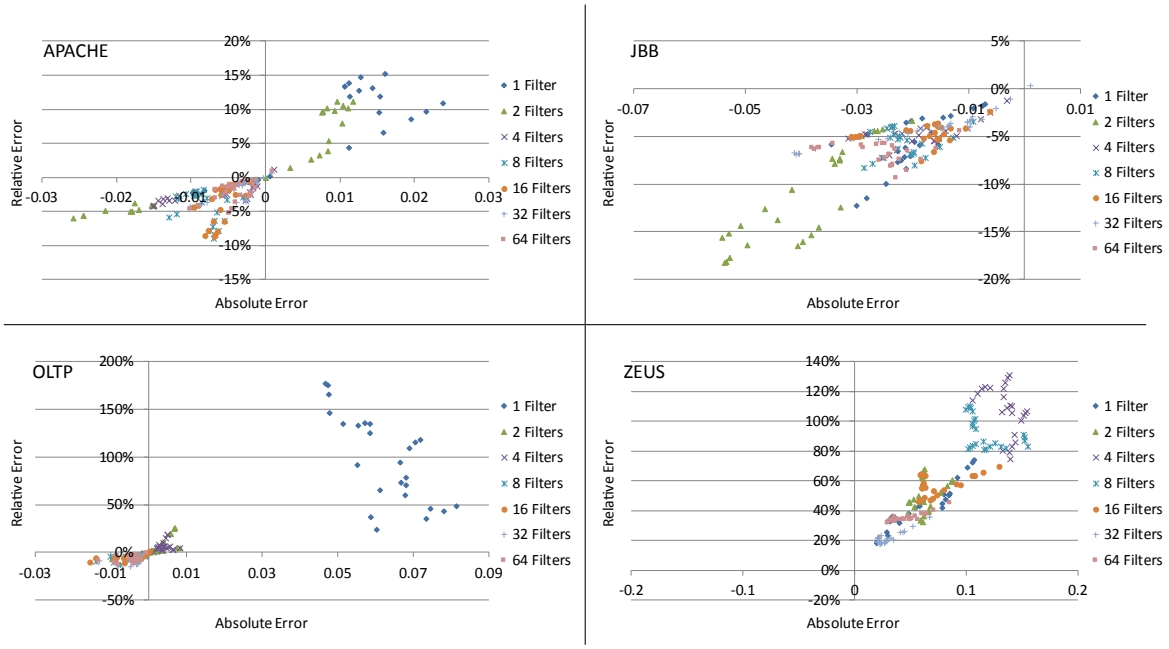


Figure 4.19: Online estimation errors with **B** filters.

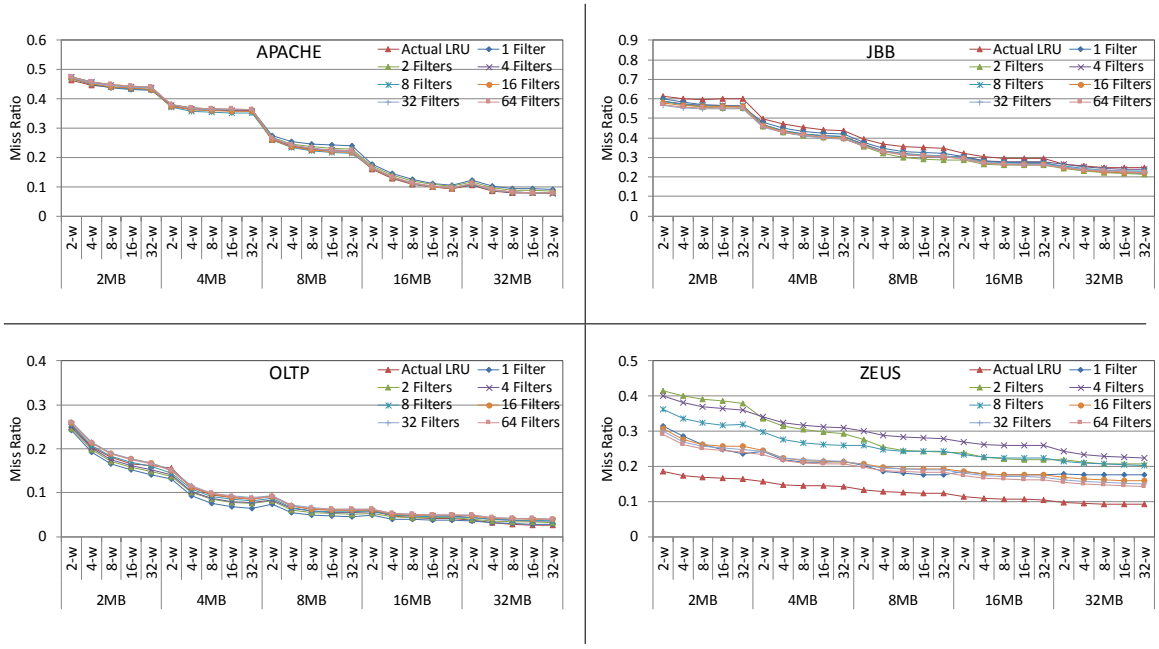


Figure 4.20: Online estimation of miss ratios using **CB** filters.

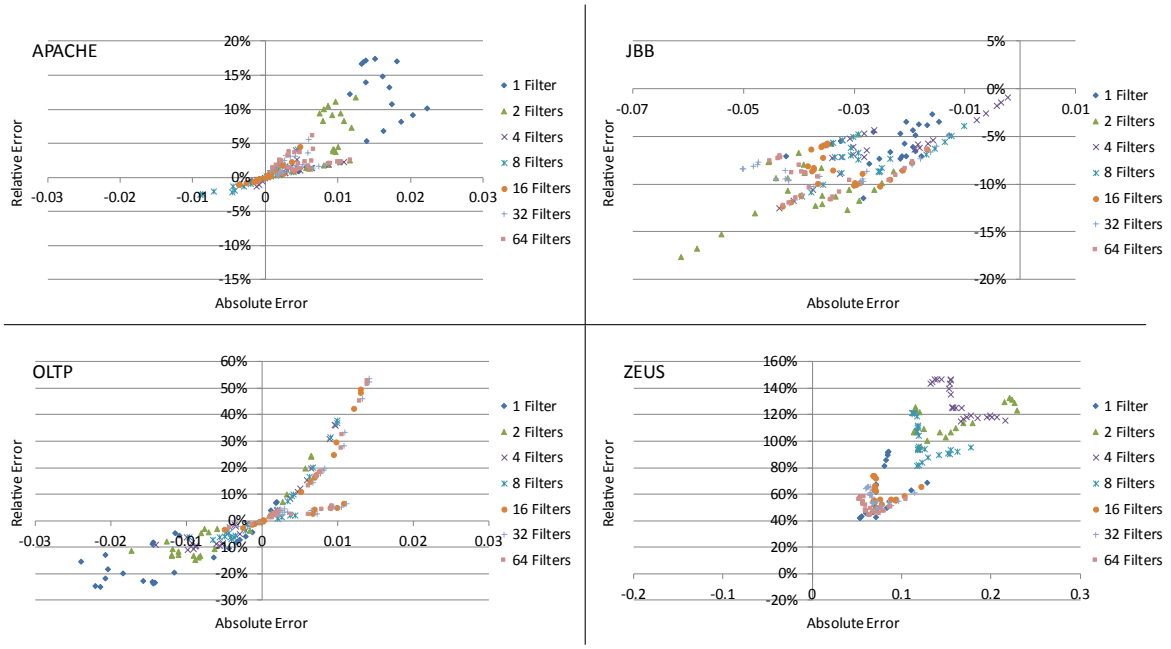


Figure 4.21: Online estimation errors with **CB** filters.

Filter	Workload	#F=1	#F=2	#F=4	#F=8	#F=16	#F=32	#F=64
E	apache	12.21	8.34	8.98	8.74	1.75	1.77	3.29
	jbb	167.64	41.22	10.46	19.57	28.84	25.60	19.68
	oltp	7.89	4.27	2.45	2.47	2.66	2.23	1.85
	zeus	620.31	87.51	152.39	143.10	96.35	61.90	52.98
B	apache	10.77	12.66	7.15	8.46	5.68	3.37	4.28
	jbb	19.87	39.38	18.83	21.51	18.53	18.26	25.84
	oltp	62.43	3.60	3.87	4.38	4.65	4.70	4.49
	zeus	56.45	62.20	134.82	117.29	73.88	29.01	46.41
CB	apache	10.80	8.38	3.14	3.13	2.00	3.88	5.21
	jbb	22.16	38.67	24.12	27.44	32.59	35.08	35.47
	oltp	11.81	7.45	6.59	4.94	5.54	6.61	5.88
	zeus	76.16	148.57	165.43	127.42	76.96	72.78	65.11

Table 4.6: $10^3\times$ Average of absolute error = abs(estimated - actual) miss ratio, over all cache configurations. Entries with values $\geqslant 50$, that is, average error $\geqslant 0.05$ are shaded.

Filter	Workload	#F=1	#F=2	#F=4	#F=8	#F=16	#F=32	#F=64
E	apache	4.94	3.44	3.89	3.42	0.86	1.16	1.44
	jbb	43.74	9.62	2.52	4.96	7.37	6.57	4.97
	oltp	15.33	6.21	3.37	3.78	4.15	3.27	2.60
	zeus	506.07	74.87	125.10	116.71	76.62	48.36	40.91
B	apache	7.25	6.22	2.62	4.44	3.67	1.95	2.16
	jbb	5.64	11.39	4.70	5.71	4.72	4.13	6.69
	oltp	97.88	6.01	6.30	5.53	4.59	7.14	6.49
	zeus	41.54	49.50	107.24	92.18	56.93	21.88	35.29
CB	apache	7.91	5.29	1.28	1.38	1.01	1.86	2.39
	jbb	6.04	10.39	5.95	7.12	8.61	8.83	9.23
	oltp	13.70	10.16	11.42	9.78	11.61	13.72	12.78
	zeus	60.83	113.93	129.96	100.55	60.52	56.96	50.81

Table 4.7: $10^2\times$ Average of relative error = abs(estimated/actual -1) miss ratio, over all cache configurations. Entries with values $\geqslant 10$, that is, average error $\geqslant 10\%$ are shaded.

Workload	apache			jbb			oltp			zeus		
#Filters & Type	**E**	**B**	**CB**	**E**	**B**	**CB**	**E**	**B**	**CB**	**E**	**B**	**CB**
1	332	235	509	48	62	88	236	89	782	4	16	17
2	725	487	1041	132	133	186	554	377	1412	28	26	29
4	1450	1060	2210	277	233	351	1154	812	2608	46	37	50
8	2748	2232	4454	596	471	720	2273	1816	5144	97	80	119
16	5581	4421	8940	1225	926	1468	4584	3544	9681	232	208	287
32	11385	8547	17627	2433	1818	2941	9397	7855	18894	542	557	603
64	22689	17042	35109	4755	3807	5936	19093	15513	38120	1142	987	1243

Table 4.8: Number of samples selected with different sampling configurations. Configurations that selected less than 385 samples are shaded (also see Section 4.6.1.3).

Table 4.8 shows the number of samples selected for reuse distance estimation in each experiment. The large prediction error for `zeus` using 1 **E** filter is due to selecting very few (4) samples for reuse estimation. We show more details for the 4 samples below. In the following, the notation $[n1, n2]$ indicates that the sample started at access number $n1$ (to the LLC) and ended at access number $n2$.

1. $[22, 54]$: Reuse distance of 0 was measured. That is, no intervening access happened that passed through the Set Filter.

2. $[84, 3299679]$: Reuse distance of ∞ was measured. That is, at least 512 intervening access happened that passed through the Set Filter. So, measurement for this sample was terminated.

3. $[3299697, 10292166]$: Reuse distance of ∞ was measured. That is, at least 512 intervening access happened that passed through the Set Filter. So, measurement for this sample was terminated.

4. $[10292169, 10488064]$: End of execution without seeing the reference address again. Reuse distance of ∞ was taken.

Workload	apache			jbb			oltp			zeus		
#Filters & Type	**E**	**B**	**CB**	**E**	**B**	**CB**	**E**	**B**	**CB**	**E**	**B**	**CB**
1	118	89	141	29	34	51	56	37	82	1	8	8
2	171	134	193	69	61	83	81	68	113	10	11	15
4	217	196	248	100	88	109	123	103	144	15	14	20
8	256	239	297	145	132	157	167	151	203	23	24	32
16	311	278	345	213	188	226	238	205	274	46	49	51
32	351	326	402	290	256	300	324	301	359	78	76	82
64	411	375	464	386	350	400	406	390	440	119	116	126

Table 4.9: Number of entries in the Histogram array (Figure 4.14) populated with different sampling configurations. The Histogram array has 512 entries (for reuse distances 0–511).

As can be seen above, measuring long reuse distances "uses up" a significant number of accesses in the trace resulting in a small number of samples for a given trace length. This is due to the fact that the absolute distance, $\mathbf{d}(T)$, increases rapidly with the reuse distance $\mathbf{r}(T)$ (see Section 4.3.3). This effect can be reduced by limiting the range of cache sizes that we want to predict for ($n = 512$ for our study due to the large range of cache sizes that we consider; see Section 4.4.2).

Table 4.9 shows how much of the Histogram array is touched by the different filter configurations. For the detailed example that we just discussed (`zeus` with 1 **E** filter), only 1 entry (for reuse distance 0) was touched. For every workload and filter type, the number of entries touched increases with the number of filters. There is thus a strictly monotonic reduction in the sparsity of the estimated reuse distribution. However, none of the filter configurations touch all 512 entries for our traces. Moreover, the reuse distances corresponding to the touched entries are not all contiguous. It is difficult to reason about accuracy of miss ratio predictions based on sparsely populated reuse distributions. Short traces tend to exacerbate the non-monotonicity in prediction error rates with the number of filters. We expect longer address traces to resolve this issue.

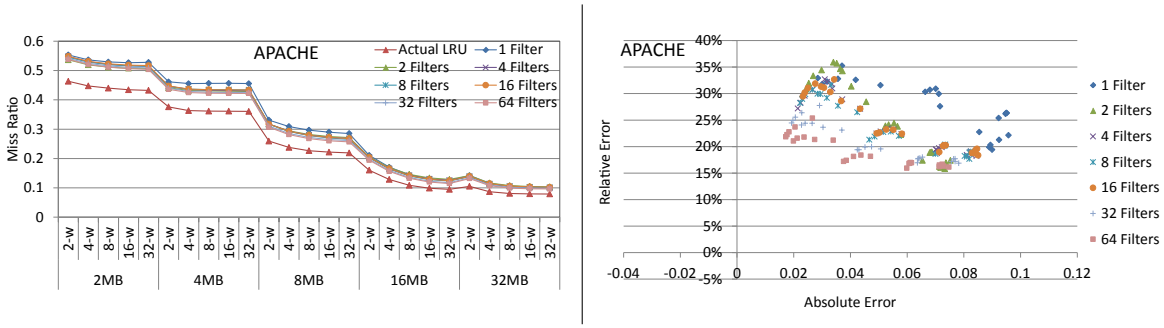


Figure 4.22: Online estimation of miss ratios using **E** filters and with a set sample randomly chosen at the start of every sample.

4.6.1.2 Time Sampling vs Set Sampling

In the Sample Initiation mode, the Control Logic time-samples the incoming address stream, then chooses the set address of the selected address for initializing the Set Filter. An alternative approach is to first choose a set address (select a sample over all possible set addresses) and initialize the Set Filter with that set address.

Figure 4.22 shows prediction errors for `apache` with this approach. The errors are larger than those when the Set Filter is initialized using time sampling (Figure 4.17). One reason is that LLC accesses are not equally distributed to all sets—some sets are accessed more often than others. Time sampling reflects these variations better than set sampling. More samples get chosen with time sampling than with set sampling, since with the latter, the Control Logic has to wait till an address accepted by the Set Filter arrives before using it as the reference address for the reuse measurement.

4.6.1.3 CLT criteria

The Central Limit Theorem (CLT) states that for large sample sizes, the distribution of the sample means approaches a Normal distribution. We will use this to develop a guideline for the minimum sample size that should be selected before the miss ratios are computed.

We will use the following notation:

- μ: Population mean.
- σ: Population standard deviation.
- n: Sample size.
- $\overline{X}$: Sample mean for a given sample.
- α: 1 - confidence level. For a confidence level of 95%, $\alpha = 1 - 0.05$.
- $\mathcal{Z}_{\alpha/2}$: The value v such that the area under the standard Normal curve between 0–v is $\alpha/2$.

Then, if the CLT theorem holds, it is well-known [33] that $\overline{X} - \mathcal{Z}_{\alpha/2}\left(\frac{\sigma}{\sqrt{n}}\right) < \mu < \overline{X} + \mathcal{Z}_{\alpha/2}\left(\frac{\sigma}{\sqrt{n}}\right)$. So,

$$\left|\overline{X} - \mu\right| < \mathcal{Z}_{\alpha/2}\left(\frac{\sigma}{\sqrt{n}}\right) \tag{4.18}$$

For a 95% confidence level, $\mathcal{Z}_{\alpha/2} = 1.96$ [33].

The metric that we are interested in this study is the average miss ratio, μ, which is the probability that an access to the cache will miss. Let M_t denote an indicator random variable such that

$$M_t = \begin{cases} 1 & \text{if the } t^{th} \text{access is a miss} \\ 0 & \text{otherwise} \end{cases} \tag{4.19}$$

We compute its expectation, $E(M_t)$, and variance, $Var(M_t)$, as follows:

$$\begin{aligned} E(M_t) =& 1 \cdot P(\text{access is a miss}) + 0 \cdot P(\text{access is a hit}) \\ =& \mu \end{aligned} \tag{4.20}$$

$$\begin{aligned} Var(M_t) =& E(M_t^2) - (E(M_t))^2 \\ =& \mu - \mu^2 \\ =& \mu(1-\mu) \end{aligned} \tag{4.21}$$

Let N denote the total number of accesses (population size). Then, the random variable $M = \frac{\sum_{t=1}^{N} M_t}{N}$ tracks the average number of misses (μ) over all accesses. Assuming that all M_t's are identically and independently distributed, we get

$$\begin{aligned} \sigma =& \sqrt{(Var(M))} \\ =& \sqrt{\left(Var\left(\frac{\sum_{t=1}^{N} M_t}{N} \right) \right)} \\ =& \sqrt{\left(\frac{\sum_{t=1}^{N} Var(M_t)}{N} \right)} \\ =& \sqrt{Var(M_1)} \\ =& \sqrt{\mu(1-\mu)} \end{aligned} \tag{4.22}$$

The expression $\mu(1-\mu)$ is maximized when $\mu = 1-\mu \implies \mu = 0.5$. Therefore, $\sigma \leqslant \sqrt{0.5 * 0.5} = 0.5$. Combining this with Equation 4.18, we can get $\left|\overline{X} - \mu\right| < 0.05$ for a

confidence level of 95%, by satisfying

$$\begin{aligned} |\overline{X} - \mu| &< 1.96 \left(\frac{\sigma}{\sqrt{n}} \right) \leqslant 1.96 \left(\frac{0.5}{\sqrt{n}} \right) < 0.05 \\ \text{that is, } 1.96 &\left(\frac{0.5}{\sqrt{n}} \right) < 0.05 \\ \implies n &\geqslant 385 \end{aligned} \tag{4.23}$$

So, for a sample size of at least 385, the absolute error of the average miss ratio of the sample from the average miss ratio of the entire trace should be less than 0.05. We call this restriction on the sample size as the CLT criteria.

Table 4.8 shows shaded entries for experiments that did not meet this criteria. Figures 4.23, 4.24, and 4.25 show estimation accuracy by the **E**, **B**, and **CB** filters only for configurations that satisfy the CLT criteria. Experiments for workloads other than `zeus` passed this criteria in the sense that the absolute errors in miss ratio were less than 0.05.

Our CLT criteria can fail to contain the maximum error within a specified bound because some assumptions may not hold in practice. For example, access are not necessarily independent, so iid assumptions may not hold. Moreover, we are sampling reuse distances whereas the derivation assumes sampling addresses to check whether they missed or not. These are not orthogonal aspects, since for LRU and the same number of sets, there is a one-to-one mapping between the reuse distance and whether or not the address missed in the cache. However, set locality predictions for a different number of sets can introduce errors.

We propose doing an additional test once the CLT criteria has been satisfied. This is to predict the miss ratio for the *current* cache configuration and compare with the actual value. If the difference is more than a threshold, additional sampling is needed before a cache configuration decision based on estimated miss ratios can be made.

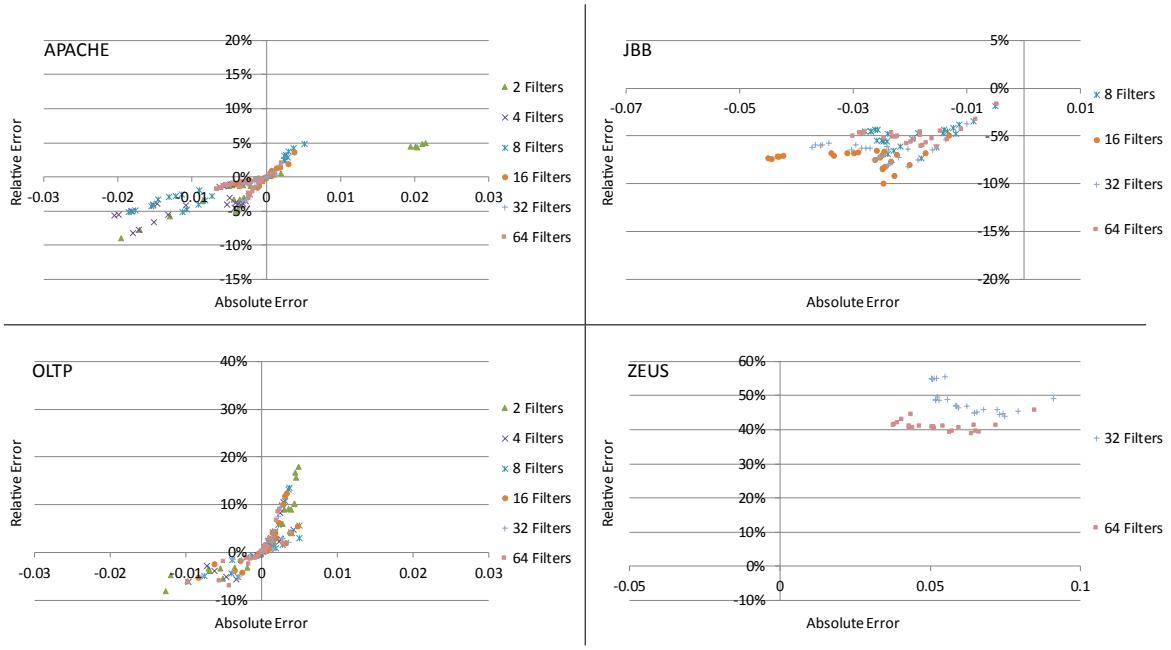


Figure 4.23: Online estimation errors using **E** filters and CLT criteria.

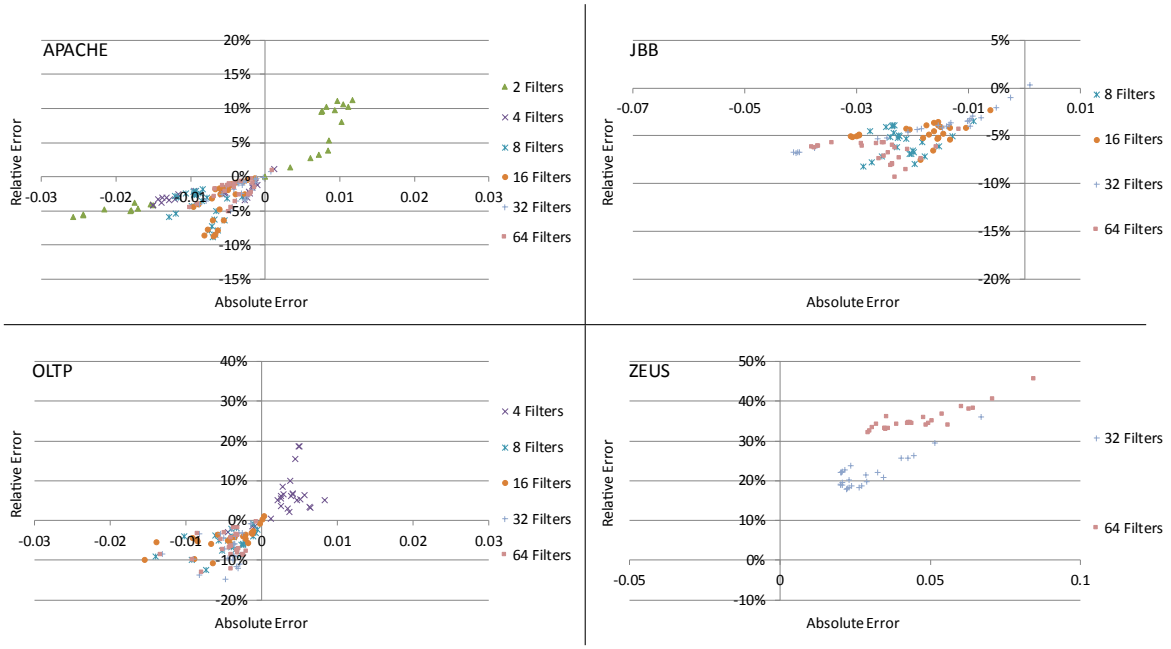


Figure 4.24: Online estimation errors using **B** filters and CLT criteria.

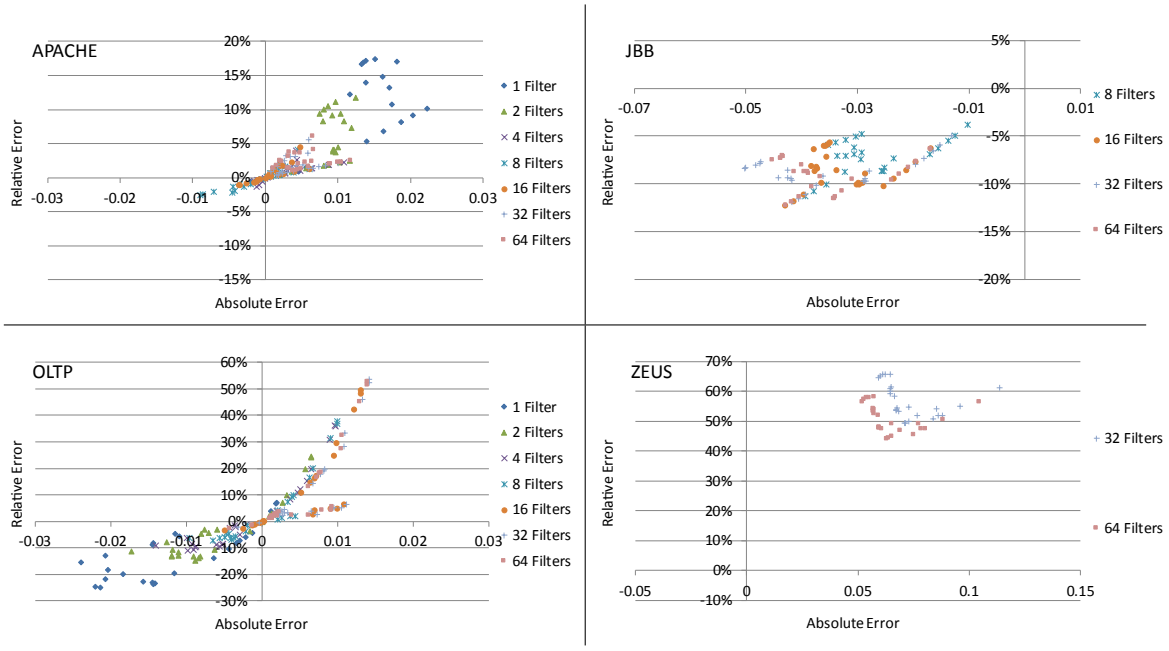


Figure 4.25: Online estimation errors using **CB** filters and CLT criteria.

4.6.2 Set-Counters, Way-Counters, and Shadow Tags

Set-counters [215] use counters that track the number of accesses per set or a group of sets. However, since they can only track changes in the number of accesses per set *but not changes in per-set locality*, they do not model the behavior shown in Figure 4.4.

Way-counters [215] increment a counter associated with each logical stack position (ordered by access recency) on every cache hit. The number of hits for associativity A' is the sum of the counter values from 0 to $A' - 1$.

The above assumes $A' \leqslant A$ where A is the associativity of the current/predicting cache (32MB 32-way in this study). In applications such as dynamic reconfiguration situations, this is problematic since the cache may need to be sized up, not only sized down. Shadow tags [172] (or auxiliary tag directories [175]) circumvent this difficulty by maintaining a copy of the tags that is not deactivated during reconfigurations. This

always maintains a stack depth to the maximum desired value and facilitates simulating the effect of hits and misses on a cache with associativity larger than that of the current cache. Qureshi et al. [174] used dynamic set sampling to reduce storage and power costs of the shadow copy.

Way-counter values, converted to probabilities, estimate $\mathbf{r}(S)$ up to length A. The estimation is exact for LRU caches. Their operation can be understood by deriving Equation 4.7 from Equation 4.3 instead of from Equation 4.2. We get

$$h(S') = \sum_{i=0}^{A'-1} \mathbf{r}_i(S) + \sum_{i=A'}^{n} \mathbf{r}_i(S) \cdot \sum_{k=0}^{A'-1} {}^{i}C_k \cdot \left(\frac{S}{S'}\right)^{k} \cdot \left(1 - \frac{S}{S'}\right)^{(i-k)}$$

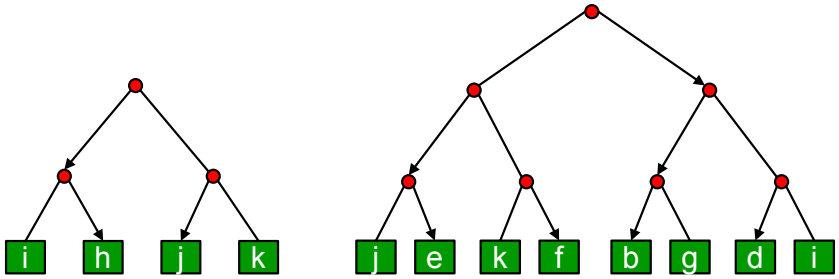
Under the assumption $S' = S$, $h(S') = \sum_{i=0}^{A'-1} \mathbf{r}_i(S)$ which is computationally extremely efficient.

4.6.2.1 Way-counters for PLRU

In PLRU, the MRU line is known with certainty but the rest of the logical ordering is not precisely known. Kedzierski et al. [127] proposed a heuristic for approximating logical stack positions for PLRU caches to enable way-counter based prediction . Let $waynum$ be the way number of the accessed line and $pathbits$ denote the bit-values of the tree nodes along the path from the root to the leaf with root bit in MSB position. Let the function $reverse(b)$ reverse bit positions in the binary representation of b. The following heuristic is used to approximate $URD(x, m)$:

$\hat{URD}(x, m) = A - 1 - (reverse(waynum) \oplus pathbits)$

This approach aims to compute $\mathbf{r} \cdot \boldsymbol{\phi}(LRU)$ with $\mathbf{r}$ approximately measured using the above mechanism. However, apart from the traditional limitations of way-counters (Section 4.6.2.2), it also ignores the fact that $\boldsymbol{\phi}(PLRU) \neq \boldsymbol{\phi}(LRU)$ for $A \neq 2$. Interestingly, it fails



(a) **4-way** (b) **8-way**

Figure 4.26: PLRU trees demonstrating non-inclusion. The 8-way tree does not include element h of the 4-way tree.

to accurately estimate the hit ratio even for a 2-way cache where $\phi(\text{PLRU}) = \phi(\text{LRU})$ *when the current configuration that does the estimation has* $A \neq 2$ (see, for example, Figure 4.13 where the current/predicting configuration has A=32). In contrast, our framework overcomes this by *decoupling hit ratio estimation from the organization of the current cache*.

4.6.2.2 Way-Counter Limitations

Way-counters (+shadow tags) have the following fundamental limitations:

Fixed number of sets: The relation ($S' = S$) that makes way-counters efficient also implies the restriction that the number of sets must be fixed. As can be observed from Table 4.1, miss ratios for only 4 of 24 configurations can be predicted at any time; other predictions must be preceded by (time-consuming) re-training for the changed S'.

However, our framework reveals that Equation 4.6.2 may be used to transform way-counter values when $S' \geqslant S$ (also see discussion for Case 1 in Section 4.4). With reference to Table 4.1, maintaining shadow tags corresponding to $S = 2^{10}$ allows conversion of values for any $S' \neq S$. But, Figure 4.6 shows that to use way-counter values for a cache with a larger number of sets, the shadow tags and counter values must be maintained for $n(> A)$ positions.

Replacement policies with stack inclusion: Way-counters exploit the stack inclusion property [150] of LRU to predict miss ratios $\forall A' \leqslant A$. For replacement policies that do not guarantee stack inclusion (PLRU/RANDOM/NMRU), this is no longer true.

For example, consider the access sequence: `a b c d e d f e g h f i j i k` simultaneously to an 8-way PLRU set and a 4-way PLRU set. Figure 4.26 shows the two sets and associated PLRU trees after the sequence. The arrows in the figures point to the less recently used subtree. Initially, both sets were empty and the eviction bits in each tree were pointing to the "left" subtrees. At the end of the sequence, the two sets together contain 9 distinct elements (`i h j k e f b g d`) whereas a policy satisfying inclusion would have exactly 8 elements. Thus, maintaining information for 8 ways is not sufficient to accurately predict miss ratios for both a 4-way and an 8-way cache *even if* $S' = S$.

Tight coupling with replacement policy implementation: Since way-counters are tightly coupled with the implementation of replacement policies that track stack positions (e.g. LRU), they are unusable with other policies such as RANDOM that can also be predicted well using reuse information. Way-counters depend on the replacement policy mimicking stack operation, so they run into trouble when the stack is absent (PLRU/RANDOM/NMRU) (see Section 4.6.2.1 for a discussion on PLRU) or reconfigured ($S' \neq S$).

Shadow Tag overhead: For very large caches, tag area and power are significant. Loh and Hill [142] propose novel tag management schemes for such caches. Maintaining additional shadow tags in those systems seem difficult.

4.7 Index Hashing

All the experiments in this chapter use an XOR-based hashed indexing scheme for the LLC. The hashing scheme is inspired by prior work [55, 56].

Let $x = \log_2(S)$. Given a 32-bit byte address b, a "plain" cache interprets the bits of b as follows:

- [0 : 5] :- block address. (Each cache line is 64 bytes.)
- [6 : 6+(x-3)-1] :- set address
- [6+(x-3) : 6+(x-1)] :- bank address. (Our LLC has 8 banks.)
- [6+x : 31] :- tag

Our "hashed" cache interprets the bits of b as follows:

- [0 : 5] :- block address. (Each cache line is 64 bytes.)
- Let k1 = bits 6 : (6+(x-3)-1) from b. (This is the set address for the "plain cache".) Let k2 = bits 20 : 31 from b. Compute $k3 = k1 \oplus k2$. Bits 0 : ((x-3)-1) from k3 form the set address for the "hashed cache".
- [6+(x-3) : 6+(x-1)] :- bank address. (Our LLC has 8 banks.)
- [6+x : 31] :- tag

Figure 4.27 shows the savings in miss ratio with hashed indexing compared to plain indexing, computed as 1 - (miss ratio with hashed indexing/miss ratio with plain indexing). While there is no guarantee that hashing will always reduce miss ratios, it reduces it in most cases. Depending on the bit patterns in the addresses, the savings can be non-trivial, e.g., up to ~27% (absolute difference in miss ratio of 0.036) for `apache`.

Hashed indexing aims to distribute the total number of *unique* addresses uniformly over all the cache sets. This is corroborated by Table 4.10 that shows the coefficient of variation of the number of unique addresses mapped to each cache set for a 32-way cache of the given size. The coefficient of variation is significantly lower with hashed

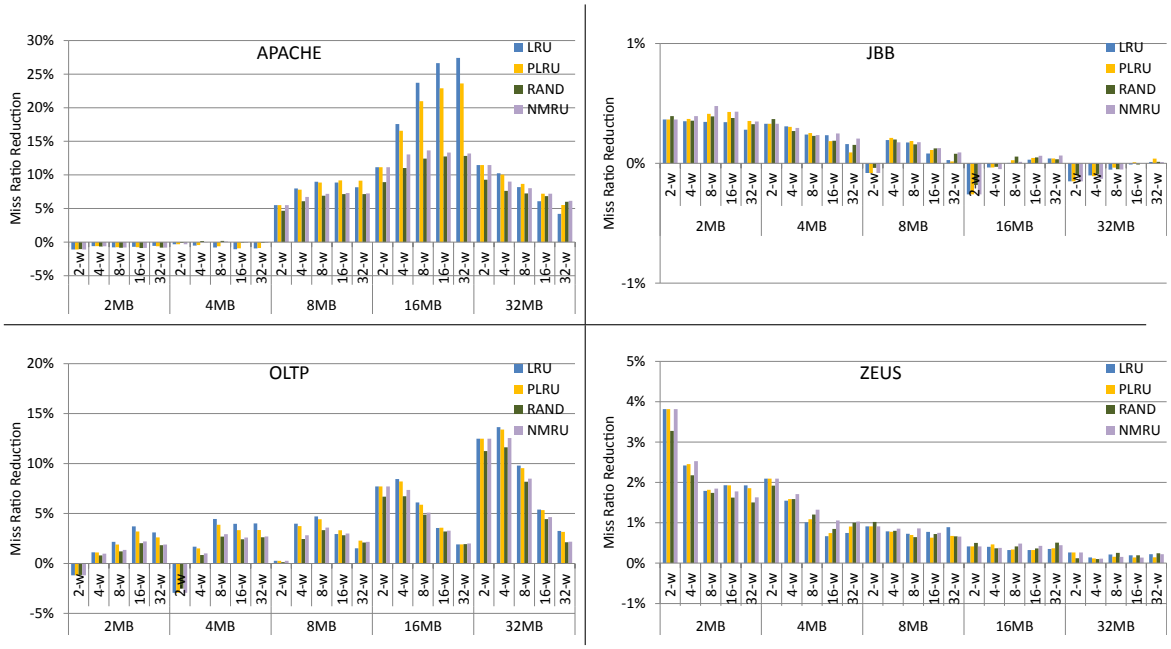


Figure 4.27: Miss ratio reduction with hashed indexing.

indexing compared to plain indexing. The Table also shows the minimum and maximum number of unique addresses mapping per set over all sets. As expected, the min–max range is higher with plain indexing than hashed indexing although the average is the same for both schemes. However, its impact on miss ratio reduction is less due to two reasons—(i) the associativity of 32 is much less than the average number of unique addresses mapping to each set, so conflict misses cannot be eliminated, and (ii) some addresses will compulsorily miss irrespective of how they map to cache sets.

Since the mapping of unique addresses to cache sets is more uniform with hashed indexing than with plain indexing, the former is also more suited for applying the models that we developed in this chapter. (Section 4.4.1 discusses the uniform mapping assumption made by our models.)

Workload	Size (MB)	Avg.	Plain			Hashed		
			Min.	Max.	Coeff.	Min.	Max.	Coeff.
apache	2	3739.28	2513	4900	0.115	3444	4033	0.028
	4	1869.64	1238	2481	0.116	1705	2027	0.025
	8	934.82	603	1283	0.117	839	1038	0.033
	16	467.41	284	658	0.120	406	530	0.042
	32	233.71	132	337	0.125	186	278	0.054
jbb	2	5694.68	5526	6259	0.013	5603	5798	0.006
	4	2847.34	2741	3143	0.015	2769	2936	0.008
	8	1423.67	1358	1578	0.017	1364	1485	0.012
	16	711.83	660	798	0.022	668	759	0.018
	32	355.92	320	400	0.029	324	392	0.025
oltp	2	2345.03	1375	8037	0.584	2213	2513	0.022
	4	1172.52	664	4094	0.585	1066	1303	0.030
	8	586.26	311	2048	0.586	504	658	0.041
	16	293.13	145	1085	0.589	246	348	0.053
	32	146.56	67	622	0.593	103	194	0.075
zeus	2	979.65	589	1311	0.107	901	1083	0.031
	4	489.82	289	681	0.110	422	559	0.043
	8	244.91	138	365	0.117	196	299	0.060
	16	122.46	66	192	0.128	89	155	0.080
	32	61.23	29	100	0.147	37	90	0.117

Table 4.10: #Unique. line addresses mapping to each set of a 32-way cache with plain and hashed indexing. Coeff. (Coefficient of Variation) = Standard Deviation/Average.

4.8 Limitations

Our models currently do not handle the following:

1. **Short-term Effects**: Our models predicts long-term averages for cache performance. Reconfiguration policies based on these models will not be able to react to high-frequency (short-term) variations in cache performance. The main reason for this is that the reuse sampling framework needs long traces, e.g., over several seconds of execution time, to get a reasonable number of samples that can be used for miss ratio

predictions. High-frequency reconfigurations for large caches may anyways not be feasible due to (i) latency and energy overheads in writing back a potentially large amount of dirty data to memory when downsizing the cache and (ii) significant cache warmup delays when upsizing the cache.

2. **Prefetching Effects**: Our models do not consider variability in the address stream that may be caused by prefetching. Currently, we assume that all demand and prefetch accesses to the LLC are included in the address stream presented to the reuse estimation hardware and that the characteristics of this address stream will remain unchanged with a reconfigured cache.

3. **Cache Hierarchy Effects**: Our models also do not consider variability in the address stream due to inclusion policies in the cache hierarchy. For example, in a strictly inclusive hierarchy, evictions at the LLC may cause evictions at L1 and/or L2. This in turn can cause additional misses at those cache levels, resulting in a different address stream arriving at the reconfigured LLC. This effect would be larger at smaller LLC sizes than at larger sizes.

4. **Other replacement policies**: We have not modeled other proposed replacement policies [117, 119] that improve upon LRU. One way to handle those could be to model their relative advantage over LRU and use that in conjunction with the LRU model described in this work to predict cache performance.

4.9 Conclusion

The central theme of this chapter is an online modeling framework, new analytical models, and efficient hardware support, to predict cache performance at runtime for a range of replacement policies and cache organizations. Our framework uses the concept of

reuse/stack distances and transformations of probability vectors with Binomial matrices. The framework unifies previous analytical models such as Smith's associativity model, Cypher's Poisson model, and hardware techniques such as way-counters. We discussed limitations of set and way-counters, gave a method to convert way-counter values for caches with a different number of sets and showed that this requires maintaining shadow tags for more than the maximum associativity. We also proposed a new predictor that is decoupled from the cache configuration, uses hardware signatures for compact representation of reuse intervals and can be used as an alternative to way-counters for miss ratio predictions.

These models will enable governors to also decide optimal cache configurations, without needing to profile numerous potential target cache configurations, in addition to configurations for other knobs. Chapter 5 demonstrates one such governor that uses these models and meets SLApower by simultaneously reconfiguring core frequency and size of the last-level cache.