Fast Miss Ratio Curve Modeling for Storage Cache

XIAMENG HU, XIAOLIN WANG, and LAN ZHOU, Peking University
YINGWEI LUO, Peking University; Shenzhen Key Lab for Cloud Computing Technology & Applications, SECE, Peking University, SHENZHEN
ZHENLIN WANG, Michigan Technological University
CHEN DING, University of Rochester
CHENCHENG YE, Huazhong University of Science and Technology

The reuse distance (least recently used (LRU) stack distance) is an essential metric for performance prediction and optimization of storage cache. Over the past four decades, there have been steady improvements in the algorithmic efficiency of reuse distance measurement. This progress is accelerating in recent years, both in theory and practical implementation.

In this article, we present a kinetic model of LRU cache memory, based on the average eviction time (AET) of the cached data. The AET model enables fast measurement and use of low-cost sampling. It can produce the miss ratio curve in linear time with extremely low space costs. On storage trace benchmarks, AET reduces the time and space costs compared to former techniques. Furthermore, AET is a composable model that can characterize shared cache behavior through sampling and modeling individual programs or traces.

CCS Concepts: • **Computing methodologies** → *Modeling methodologies*; • **Information systems** → Hierarchical storage management;

Additional Key Words and Phrases: Cache system, data locality, miss ratio curve

ACM Reference format:
Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Zhenlin Wang, Chen Ding, and Chencheng Ye. 2018. Fast Miss Ratio Curve Modeling for Storage Cache. *ACM Trans. Storage* 14, 2, Article 12 (April 2018), 34 pages.
https://doi.org/10.1145/3185751

1 INTRODUCTION

Memory system is a multi-level structure where the upper level of memory often plays the role of cache for the lower level of storage. This design is motivated by a simple fact of *program locality*: in any time period, only a small fraction of data in a program will be frequently used. This behavior used to be modeled by the working set locality theory (Denning 1980), where data locality is characterized by working set size (WSS) (Denning 1968; Denning and Slutz 1978).

The research is supported in part by the National Science Foundation of China (Grants No. 61232008, No. 61472008, No. 61672053, and No. U1611461); Shenzhen Key Research Project No: JCYJ20170412150946024; the National Science Foundation (Contracts No. CSR-1618384, No. CSR-1422342, No. CCF-1717877, No. CCF-1629376, and No. CNS-1319617); and an IBM CAS Faculty Fellowship.
Authors' addresses: X. Hu, X. Wang, L. Zhou, and Y. Luo, Peking University, No.5 Yiheyuan Road Haidian District, Beijing, P.R.China 100871; emails: {hxm, wxl, lanzhou, lyw}@pku.edu.cn; Z. Wang, Michigan Technological University, 1400 Townsend Drive, Houghton, MI 49931-1295; email: zlwang@mtu.edu; C. Ding, University of Rochester, 500 Joseph C. Wilson Blvd., Rochester, NY 14627; email: cding@cs.rochester.edu; C. Ye, Huazhong University of Science and Technology, Luoyu Road 1037, Wuhan, P.R.China 430074; email: yechencheng@gmail.com.

https://doi.org/10.1145/3185751

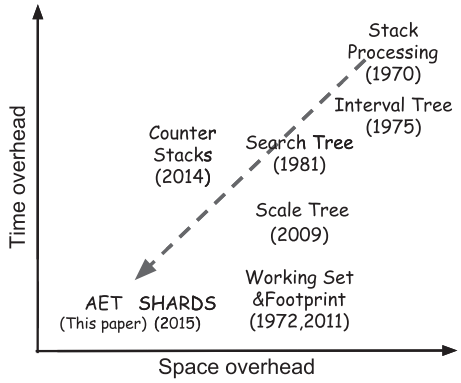


Fig. 1. The time and space overhead of MRC profiling techniques.

Locality characterization techniques have been developed for decades. They are widely used for management and optimization at different levels of memory hierarchy.

Much progress has been made to model locality through reuse distance analyses and the result miss ratio curves (MRCs), as shown in Figure 1.

From the reference trace of a program, accurate MRC can be calculated by measuring reuse distance (least recently used (LRU) stack distance as defined by Mattson et al. (1970)). Reuse distance is the number of distinct data accesses between two consecutive accesses to the same location. Precise reuse distance tracking requires $O(N \log M)$ time and $O(M)$ space for a trace of N accesses to M distinct elements (Olken 1981a).

For CPU workloads, the recent footprint theory (Xiang et al. 2013), StatStack (Eklov and Hagersten 2010), and time-to-locality conversion (Jiang et al. 2010; Shen et al. 2007) use *reuse time* instead of reuse distance to model the workloads. The (backward) reuse time is the time between a data access and the most recent access to same location. The footprint approach reduces the run-time overhead of MRC measurement to $O(N)$.[1] However, the space overhead of the footprint algorithm is still $O(M)$.

As for storage workloads, their sizes are usually much larger than CPU workloads, and their lifespan may last for weeks or more. Therefore, techniques like the footprint analysis may require too much space. *Counter Stacks* (Wires et al. 2014) and *SHARDS* (Waldspurger et al. 2015) are recent breakthroughs to reduce space cost in asymptotic complexity (Wires et al. 2014) and in practice (Waldspurger et al. 2015). Counter Stacks uses probabilistic counters and for the first time can measure reuse distances in sub-linear space with a guaranteed accuracy (Drudi et al. 2015). But Counter Stacks' space overhead remains modestly high. SHARDS uses a splay tree to track the reuse distances of sampled data. The time and space consumption are reduced to a low level. However, SHARDS cannot characterize shared cache behavior through modeling individual programs.

This article describes a novel kinetic model for MRC construction of LRU caches based on average eviction time (AET). AET runs in linear time asymptotically, uses sampling to minimize the space overhead, and adopts adaptive phase sampling to capture time-varying behavior. In evaluation, AET has the lowest level space and run-time overhead compared to past techniques, while maintaining high MRC accuracy. Although SHARDS is comparable to AET in time and space overhead, AET is a composable metric, i.e., the MRC of a multi-programmed workload in shared cache can be computed directly from the AET of its member programs.

[1]The working-set theory has a similar effect and same time and space complexity (Denning and Schwartz 1972; Denning and Slutz 1978). See Sec. 2.8 of Xiang et al. (2013) for a comparison.

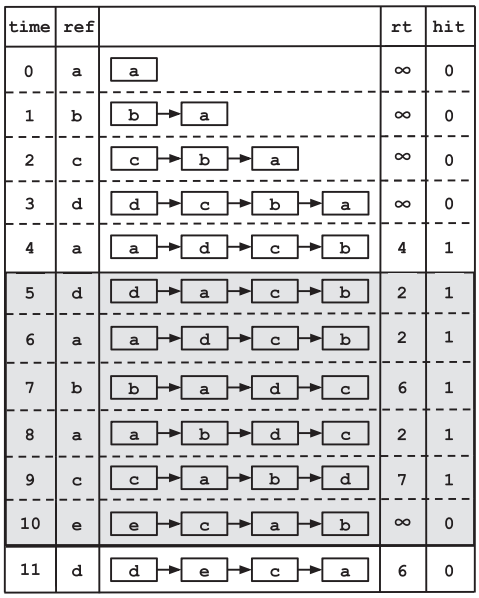


Fig. 2. Example 4-block cache, viewed as a stack (priority list). The table shows the logical time, the data referenced each time (ref), the reuse time (rt) of each access and whether the access is a cache hit or not. The shaded area is the eviction process of d.

2 AET MODEL

This section describes the kinetic model. Section 2.1 uses an example to introduce the basic concepts especially the eviction time. Section 2.2 formulates and computes the average eviction time (AET) by solving the distance integration equation. The AET model relates MRC to reuse-time distribution. Section 2.3 discusses the correctness of the model. Section 2.4 shows an alternate approach to deriving the AET model using Little's law. Section 2.5 models the shared cache and solves the eviction-time equalization equation. Section 2.6 shows multi-level cache modeling using AET.

2.1 LRU Stack and Eviction Time

LRU cache can be logically viewed as a stack (Mattson et al. 1970). Data blocks are ranked by their recent access time from most recent to least recent. Every access brings the accessed data to the top of the stack. The bottom of the stack stores the least recently used data and is evicted on a miss (when the cache is full).

When a data block is loaded into cache at a miss, it may be reused for several times (hits) before it is evicted. The *eviction time* is the time between the last access and the eviction. It is the duration that the block moves from the top of the stack to the bottom for the *last* time. We call this period of LRU movement an *eviction process* and the data block moving towards the bottom of stack the *evicting block*. At an eviction at time t, looking backwards to the most recent time u when the evicting block was referenced, the time interval $t - u$ is the eviction time. Notice that u could also be the time the evicting block was brought in (a miss). In general, the eviction time is the last segment of the residency time of the evicting block.

For example, for data block d in the example cache in Figure 2, it is loaded at time 3, last accessed at time 5, and evicted at time 10. The eviction time is 5, shown by the shaded area. Eviction time

Table 1. The Kinetic Model Illustrated by d's Eviction in the Shaded Area in Figure 2

Logical time	5	6	7	8	9	10
Position m	0	1	2	2	3	evicted
Arrival time T_m	0	1	2	2	4	5
Current reuse time	2	2	6	2	7	∞

The arrival time T_m (third row) depends on the movement condition: whether the reuse time (Last Row) is greater than T_m. The eviction time is $T_4 = 5$.

is part of the *residence time*, which is 7 in this example and in general can be estimated using queuing theory (as "response time," Chapter 9 (Denning et al. 2015)). This observation motivates us to derive the AET model using queuing theory in Section 2.4.

To model the eviction time, we need to model the progression that leads to the eviction. We define the *arrival time* T_m as the time it takes for an evicting block to reach stack position m (from its last access). For size c cache, the arrival time is a (subscripted) function T_m, $m = 0, \ldots, c-1$. Naturally, the eviction time is T_c, which is the time the evicting block leaves position $c-1$ and arrives at the virtual position c. To illustrate, Table 1 shows the arrival time T_m of d for size 4 cache. As m increments from 0 to 4, T_m increases from 0 to 5.

The movement of evicting block d depends on how other data are accessed. At each access in the eviction process (shaded area in Figure 2), d either stays at its current position or steps down one position. The *condition of movement* is simple: d moves down from a position m if and only if the access is a miss, or if the stack position of the accessed data m' is greater than m, that is, the accessed data resides in the far side of the priority list, closer to the tail. We define T_0 to be 0. Obviously, T_1 is always 1, since the access to any other block must bring it to stack position 0 and dislodge d, as it happens at time 6 for d.

The condition of movement can be simplified, because we do not need the exact location of the accessed data. It suffices to know the relative location. For a simpler test, we use the reuse time rather than the stack location. We define the *sojourn time* S_i at stack position i as the time the evicting block has stayed at position i, since it arrived at the position. The following statement summarizes the movement condition, which relates data movement and eviction time in the LRU stack to reuse time.

When block z is accessed, and the evicting block d is at stack position $m-1$, d moves down to position m if and only if the reuse time of z is greater than d's arrival time T_{m-1} plus its current sojourn time S_{m-1}.

If the preceding condition is true, then z has not been accessed within $T_{m-1} + S_{m-1}$, i.e., since d's last access, so it must reside behind d in the stack or not be present in the stack. An interesting observation is that, when the movement condition is true and d moves, it arrives at position m at time $T_{m-1} + S_{m-1}$, which equals T_m. Movement condition can thus be stated formally as:

Movement Condition: *When block z is accessed, and the evicting block d is at stack position $m-1$, d moves down to arrive at position m if and only if the reuse time of z is greater than T_m.*

The relation between the eviction time and the reuse time is illustrated by our example. The last row of Table 1 shows the reuse time of each access during d's eviction process. Block d moves to next position (shown in the second row) whenever the reuse time (the last row) is greater than the arrival time (the third row). At each position, the speed changes. The fastest speed is moving one position at each access. The slowest speed is no movement.

ALGORITHM 1: Reuse Time Histogram to Complement-Cumulative Distribution Function (CCDF)

Require: $rt[]$ // reuse time histogram (without cold misses)
Require: len // largest reuse time
Require: sum // number of accesses including cold misses
Ensure: : $P[]$ // CCDF of reuse time
1: **function** CalcCCDF($rt[], len, sum$)
2: $P[0] \leftarrow 1.0$
3: **for** $i \leftarrow 1..len$ **do**
4: $P[i] \leftarrow P[i-1] - rt[i]/sum$
5: **end for**
6: **end function**

We next model the average eviction time for all data in cache. The arrival time T_m will be defined similarly as the average for all data. Individually, the arrival speed may slow down and then speed up. On average, however, the arrival speed is non-increasing, as we discuss next.

2.2 Average Eviction Time (AET)

$AET(c)$ is the *Average Eviction Time* for all data evictions in a fully associative LRU cache of size c. It is the expected residence time for a data block since its last access. Our AET model is based on the following hypothesis:

Cache Miss Probability Hypothesis: *The probability that a reference with a reuse time larger than AET is the probability that this data block is no longer in cache and results in a cache miss.*

The cache miss ratio is actually the probability of an access that leads to a miss. The hypothesis therefore can applied to predict the miss ratio as the the proportion of data reuses that have larger reuse time than AET. This hypothesis is the foundation of our MRC prediction technique. The rest of section derives AET.

For cache size c, let T_m be the average arrival time for evicting data blocks to reach position m (in their eviction processes). Obviously, $T_0 = 0$ and $AET(c) = T_c$. The movement condition is no longer individual but now collective and depends on the reuse times of all data.

Let N be the total number of references and $rt(t)$ be the number of references with reuse time t. $f(t)$ is the proportion of reuses with reuse time t, defined as follows:

$$f(t) = \frac{rt(t)}{N}. \tag{1}$$

$f(t)$ describes reuse-time distribution, which can be effectively sampled as we will discuss in Section 3.1. For an access, $P(t)$ is the probability that its reuse time is greater than t:

$$P(t) = \sum_{i=t+1}^{\infty} f(i). \tag{2}$$

$P(t)$ is actually the *Complement of the Cumulative Distribution Function* (CCDF) of reuse time, i.e., $P(t)$ is one minus the probability of reuse time less than or equal to t. Algorithm 1 implements Equations (1) and (2) to convert reuse time histogram to CCDF. Next, we show that AET can be derived from CCDF.

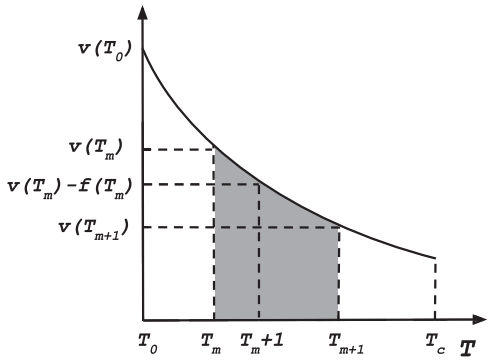


Fig. 3. As the average arrival time (T_m) increases along the x-axis, the y-axis shows the arrival speed $v(T_m)$ at each T_m. The integral of of v over T gives the movement distance, which is the area under the curve. The shaded area shows the increment of stack position (which is 1).

The movement condition is now a probability when examining average arrival time:

Movement Probability: *The probability of an evicting block at position $m-1$ to move to (arrive at) position m is $P(T_m)$, where T_m is the average arrival time to position m.*

This can be interpreted as follows: in a unit time, the evicting block at position $m-1$ moves by 1 position to arrive at position m with probability $P(T_m)$. So the expected moving distance is $P(T_m)$ in a unit time. To use a familiar concept, we call it the *arrival speed.* The arrival speed at position m in logical time equals the probability $P(T_m)$. Further, we track arrival speed with respect to the relative clock of an eviction process:

$$v(T_m) = P(T_m), \tag{3}$$

when $m > 0$. A special case is $v(T_0) = v(0)$, which is always one following the logical time, i.e., one access at a time. correspondingly, $P[T_0] = P[0] = 1$. Equation (3), hence, holds for all m. The arrival speed $v(T_m)$ is monotone and non-increasing with position m. By definition, $P(T_m)$ is monotone and non-increasing with m.

We now construct an equation to solve for T_m and then $AET(c)$. The equation connects three metrics: arrival speed $v(T_m)$, average arrival time T_m, and cache size c. This connection is shown pictorially in Figure 3.

In Figure 3, the x-axis shows the average arrival time (T_m) as it increases. At each T_m, we use Equation (3) to compute the arrival speed $v(T_m)$, shown in the y-axis. The figure shows an example curve, which is monotonically non-increasing. The integral of v over T gives the distance of movement, i.e., the stack position it moves to. It is the area under the curve. The shaded area shows the increment of the stack position (which is 1).

The three metrics are discrete functions. The subtle but critical problem is the difference in their discrete units. When we measure the cache size and the data movement in cache, a single step is a stack position. When we measure the reuse time, a single step is an access. We may call the former the spatial unit and the latter the temporal unit. The two units are not the same. Figure 3 shows that from the same base T_m, the temporal increment $T_m + 1$ is less than or equal to the spatial increment T_{m+1}.

We use the temporal-unit function of reuse time to derive the spatial-unit function of AET. Let's consider how the speed changes as a data block moves. From the monotonicity mentioned earlier, the change must be a deceleration. Based on the arrival speed formula (Equation (3)), the following

gives the exact deceleration from T_m to $T_m + \Delta T$:

$$v(T_m + \Delta T) = v(T_m) - \sum_{t=T_m+1}^{T_m+\Delta T} f(t), \tag{4}$$

where ΔT stands for the time increase over T_m. The unit is temporal, so the minimal ΔT is one, i.e., one access.

Now, we are ready to formulate the first kinetic equation, Distance Integration (DI). It combines the temporal and spatial increments to compute the complete movements. First, let's consider the spatial increment. From T_m to T_{m+1}, the data moves one stack position (the shaded area in Figure 3). Second, we add the temporal increment as follows. For each spatial increment (m), we compute the deceleration by integrating in the temporal unit (dx), given in Equation (4). Finally, we sum over the spatial increment from 0 to cache size c. The result is the total distance moved, e.g., the area below the example curve in Figure 3, which is the cache size c when the arrival time reaches T_c:

$$\sum_{m=0}^{c-1} \int_{T_m}^{T_{m+1}} (v(T_m) - \sum_{t=T_m+1}^{x} f(t))\, \mathrm{d}x = c. \tag{5}$$

DI is an implicit equation. Its solution, as it turns out, is $AET(c)$, i.e., T_c. Consider the speed at each time step x from 0 to $AET(c)$, and the time it takes at each step, we have

$$\int_0^{AET(c)} P(x)\, \mathrm{d}x = c. \tag{6}$$

This equation is, in fact, the same as Equation (5). The equivalence is proved as follows:

$$\begin{aligned}
&\sum_{m=0}^{c-1} \int_{T_m}^{T_{m+1}} (v(T_m) - \sum_{t=T_m+1}^{x} f(t))\, \mathrm{d}x \\
&= \sum_{m=0}^{c-1} \int_{T_m}^{T_{m+1}} (P(T_m) - \sum_{t=T_m+1}^{x} f(t))\, \mathrm{d}x \\
&= \sum_{m=0}^{c-1} \int_{T_m}^{T_{m+1}} (P(T_m) - (P(T_m) - P(x)))\, \mathrm{d}x \\
&= \sum_{m=0}^{c-1} \int_{T_m}^{T_{m+1}} P(x)\, \mathrm{d}x \\
&= \int_{T_0}^{T_1} P(x)\, \mathrm{d}x + \int_{T_1}^{T_2} P(x)\, \mathrm{d}x \\
&\quad \ldots + \int_{T_{c-1}}^{T_c} P(x)\, \mathrm{d}x \\
&= \int_0^{AET(c)} P(x)\, \mathrm{d}x.
\end{aligned}$$

From AET to MRC. Equation (6) shows that AET calculation only needs the CCDF of reuse time, $P(x)$, which can be calculated based on the reuse-time distribution or reuse time histogram (RTH), and can be measured in linear time (see Section 3).

ALGORITHM 2: From CCDF of Reuse Time to MRC

Require: $P[]$ // CCDF of reuse time
Require: M // largest cache size
Require: len // largest reuse time
Ensure: $MRC[]$ // miss ratio curve

```
1:  function CalcMrc(P[], M, len)
2:      integration ← 0
3:      t ← 0
4:      for c ← 1..M do
5:          while (integration < c and t ≤ len) do
6:              integration ← integration + P[t]
7:              t ← t + 1
8:          end while
9:          MRC[c] ← P[t − 1]
10:     end for
11:     return MRC[]
12: end function
```

Based on the cache miss probability hypothesis, the miss ratio $mr(c)$ at cache size c is the probability that a reuse time is greater than the average eviction time $AET(c)$:

$$mr(c) = P(AET(c)). \tag{7}$$

Algorithm 2 implements Equations (6) and (7) to calculate MRC based on the CCDF of reuse time. During the integration of Equation (6) from 0 to the maximal reuse time, the miss ratio of all cache sizes can be computed together in linear time, $O(N)$. Note that line 7 can be executed at most $len + 1$ times where len is less than N as the largest-possible reuse time of a trace of size N is $N - 1$. In practice, we merge Algorithm 1 and Algorithm 2 into one function and replace the array P by a scalar to save space.

Impact of Cold Misses. In a program execution, the first access to any data block should be a cold miss. Because every cold miss will insert a new data block at the head of the LRU priority list, it will push down all the data in the list by one position. In the kinetic equation, no matter where the data is, the cold misses always contribute a fixed share of probability that moves the data. Therefore, in AET model, we define the reuse time of every cold miss to be infinite, and we count the number of cold misses in the ∞ bin of the reuse time histogram (RTH), as in the example shown in Figure 4.

2.3 Correctness

The conversion from AET to miss ratio is not always correct. The correct miss ratio for cache size c is the proportion of reuse distances $d > c$.

The inverse of the AET function is in fact an estimation of reuse distance. For a reuse time t, the reuse distance d is the distance the data block moved down the cache stack, so $t = AET(d)$ and

$$d = AET^{-1}(t). \tag{8}$$

AET conversion is equivalent to first estimating the reuse distance and then using the estimated reuse distance:

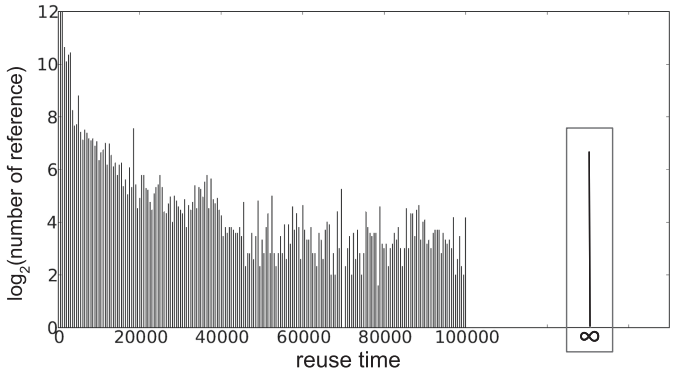


Fig. 4. RTH and cold miss example.

$$\begin{aligned} mr(c) &= \frac{\sum_{x>c} rd(x)}{N} \\ &= \frac{\sum_{t>AET(c)} rd(AET^{-1}(t))}{N} \\ &= \frac{\sum_{t>AET(c)} rt(t)}{N} \\ &= P(AET(c)), \end{aligned}$$

where $rd(x)$ is the number of references with reuse distance x. Therefore, AET is correct for all cache sizes if its estimation of reuse distance is correct. We give a correctness condition as follows:

Correctness Condition For All Cache Sizes. *The AET-based conversions are accurate for all cache sizes if the number of reuse times $rt(t)$ of time t is the same as the number of reuse distances $rd(AET^{-1}(t))$ of distance $AET^{-1}(t)$, for all $t > 0$.*

When the two are equal, using the AET conversion is the same as using reuse distance for all cache size $c \geq 0$. The condition is a reiteration of Equation (8) but shows the connection mathematically as a function composition, rd and AET^{-1}.

Next, we give the correctness condition for individual cache sizes. For a cache of size c, every access with a reuse time $rt > AET(c)$ will be predicted as a cache miss by AET model. This prediction is inaccurate if the reuse distances of some reuses are equal to or smaller than c, which means they are hits. We define this error as "miss-prediction error" or *MPE*. For every access with a reuse time $rt \leq AET(c)$, AET predicts a hit. We define the "hit-prediction error" or *HPE* as the reuses that predicted as hits but are misses, because their reuse distances are larger than c.

More specifically, we use a matrix to present the distribution of reuse time and its corresponding reuse distance. As shown in Figure 5, element $X_{rt,rd}$ stands for the number of accesses with reuse time rt and reuse distance rd. Note that the elements when $rt < rd$ are 0, because reuse distance is always equal to or smaller than reuse time. For cache size c, we use two dashed lines to divide the matrix into four parts. The upper left (UL) area in the matrix are elements with equal or smaller reuse time than $AET(c)$ and equal or smaller reuse distance than c ($rt <= AET(c), rd <= c$). They represent the reuses that AET predicts as hits and are truly hits. The lower right (LR) area are elements with larger reuse time than $AET(c)$ and larger reuse distance than cache size c ($rt > AET(c), rd > c$). They represent the reuses that AET predicts as misses and are truly misses. Both UL and LR parts are correct predictions in AET model. However, the upper right (UR) area, which are the reuses with equal or smaller reuse time than $AET(c)$ but larger reuse distance than

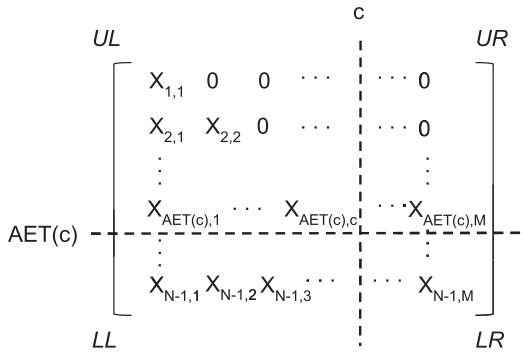


Fig. 5. The reuse time-reuse distance matrix, divided into four quadrants as labeled: UL, LL, UR, and LR. $X_{\infty,\infty}$ is not presented.

c ($rt <= AET(c), rd > c$). This part represents the hit-prediction error, because AET predicts them as hits but they are actually misses. On the other hand, the lower left (LL) area in the matrix represents miss-prediction error. They are the reuses that AET predicts as misses but actually are hits ($rt > AET(c), rd <= c$). Please note that the number of cold misses ($X_{\infty,\infty}$) is not presented in this matrix. In AET model, the first access to any data is predicted as cache miss. This prediction is always correct under any cache sizes.

Now, we can conclude that the error of AET model at certain cache size is brought by the difference between MPE and HPE. Obviously, the number of MPE is the sum of elements in LL:

$$|MPE| = \sum_{i=AET(c)+1}^{N-1} \sum_{j=1}^{c} X_{i,j}, \tag{9}$$

where N is the total number of accesses. Similarly, the number of HPE is the sum of elements in UR:

$$|HPE| = \sum_{i=1}^{AET(c)} \sum_{j=c+1}^{M} X_{i,j} = \sum_{i=c+1}^{AET(c)} \sum_{j=c+1}^{M} X_{i,j}, \tag{10}$$

where M stands for the data size of the program.

If |MPE| and |HPE| are equal, then they cancel each other and then the AET prediction is accurate at this cache size. If |MPE| is larger than |HPE|, then the predicted miss ratio should be higher than the real miss ratio. Conversely, if $|MPE|$ is smaller than $|HPE|$, then the predicted miss ratio should be lower than real miss ratio. Now, we have the absolute error $E(c)$ of AET prediction under cache size c:

$$E(c) = \frac{||HPE| - |MPE||}{\sum_{i=1}^{N-1} \sum_{j=1}^{M} X_{i,j} + X_{\infty,\infty}} = \frac{||HPE| - |MPE||}{N}. \tag{11}$$

In Figure 6, we run a simple trace over a cache of size 3 to exemplify the two types of prediction error. The data reference stream is constructed by two cycles:

$$(ABCCBA)^{100}(MNPQ)^{2}. \tag{12}$$

Each cycle has a unique reference pattern with different number of iterations. The first cycle iterates for 100 times while the second cycle twice. The entire trace has 608 references generated from 7 distinct elements. In Figure 6, we give the reuse time and reuse distance of each reference starting from the second iteration in the first cycle, as well as the hit/miss prediction using AET

Address	(A	B	C	C	B	A)^{99}	M	N	P	Q	M	N	P	Q
Reuse Time	1	3	5	1	3	5	$+\infty$	$+\infty$	$+\infty$	$+\infty$	4	4	4	4
Reuse Distance	1	2	3	1	2	3	$+\infty$	$+\infty$	$+\infty$	$+\infty$	4	4	4	4
Prediction (AET $\approx$ 4)	hit	hit	miss	hit	hit	miss	miss	miss	miss	miss	hit	hit	hit	hit
Actual	hit	hit	hit	hit	hit	hit	miss	miss	miss	miss	miss	miss	miss	miss
Error Type	-	-	MPE	-	-	MPE	-	-	-	-	HPE	HPE	HPE	HPE

Fig. 6. An example trace to show different types of prediction error in a 3-block cache (Iteration 1 of the first cycle is skipped due to space.)

model. With a 3-block cache, the average eviction time can be derived as follows:

$$\begin{aligned}
&\int_0^4 P(x)\,\mathrm{d}x \\
&= P(0) + P(1) + P(2) + P(3) + P(4) \\
&= 1 + \frac{409}{608} + \frac{409}{608} + \frac{210}{608} + \frac{206}{608} \\
&= 3.03.
\end{aligned}$$

The above equation gives the average eviction time, $AET(3) = 4$. Any reference with a reuse time larger than 4 is predicted as cache miss. The first C and the second A in the first cycle from iteration 2 and beyond have a reuse time of 5, AET model predicts them as misses. However, the reuse distance of these two references is 3, which means they are actually cache hits. These two references are the miss prediction errors (MPEs) in AET model. In the second iteration of the second cycle, all the references have equal reuse time 4 and the same reuse distance 4. AET model predict them as hits, but their reuse distances are larger than cache size, which denotes they are actually misses. This type of error is the hit prediction error (HPE).

In this example, AET predicts 206 misses versus 11 misses by the accurate reuse-distance model due to the extremely-skewed access pattern. However, in our evaluation, we observe that the numbers of the two types of prediction error are typically small and close to each other in most cases. The differences between them are evident only under extreme unstable access pattern. Here, we give the correctness condition for cache size c.

Correctness Condition For One Cache Size. *For cache size c, the AET-based conversion is accurate if the number of reuses whose reuse time is larger than AET(c) but reuse distance smaller than or equal to c, is equal to the number of reuses whose reuse time is equal to or smaller than AET(c) but reuse distance larger than c.*

2.4 Consistency with Queuing Theory

In queuing theory, a classical discipline within the mathematical theory of probability, Little's law, is a theorem by John Little that states: The long-term average number of customers in a stable system, L, is equal to the long-term average effective arrival rate, λ, multiplied by the average time a customer spends in the system, W; or expressed algebraically: $L = \lambda W$ (Allen 2014). This relationship is not influenced by the arrival process distribution, the service distribution, and service order. A caching system can also be viewed as a queuing system. New data and missed data arrive from the lower level of memory hierarchy. The cached data stays in cache for some time before it leaves cache (eviction).

In this section, we will infer AET model from Little's law. AET model is under the assumption that all reuses with a reuse time larger than the average eviction time is a cache miss. Since cache is a queue system, we want to present cache miss ratio using Little's law under the same assumption as AET model. The first part of Little's law is the average time a data object stays in cache, which we call the average residence time of data objects. It is the average time for all data objects from entering cache to their eviction or departure. Assume we have a cache of size c, for any reuse time t, if it is smaller than or equal to $AET(c)$, the data has resided in cache for t since its last access. If t is larger than $AET(c)$, then it means the data had resided in cache for $AET(c)$ and was then evicted from cache from since last access. If this access is a cold miss ($t = \infty$), then the data object has never been cached before. However, after the last reference of each object, it will stay in cache for $AET(c)$ and get evicted. Since the number of cold misses is equal to the number of last references, we can assume every cold miss to have residence time of $AET(c)$.

Now, we know the relationship between reuse time and residence time of the data from its last access time. If we have the reuse-time distribution of all data references, then we can easily get the total residence time of all data objects. Assume $rt(x)$ is the number of references with a reuse time x as used in Equation (1). The total residence time all_time can be represented as follows:

$$\begin{aligned} all\_time &= rt(1) + rt(2) * 2 + rt(3) * 3 + \cdots + rt(AET(c)) * AET(c) \\ &\quad + rt(AET(c) + 1) * AET(c) + rt(AET(c) + 2) * AET(c) + \cdots + rt(\infty) * AET(c) \\ &= \sum_{x=1}^{AET(c)} rt(x) * x + \sum_{x=AET(c)+1}^{\infty} rt(x) * AET(c). \end{aligned}$$

Now, we have the total residence time of all data objects. To get the average residence time of all arriving data, we need to know how many data objects have arrived in the cache. With eviction time assumption, the cache miss ratio is the probability of a reuse with reuse time larger than $AET(c)$, i.e., $P(AET(C))$. The number of data objects entering cache $all\_arrival$ is the number of all reference times the miss ratio of this cache:

$$all\_arrival = \sum_{x=1}^{\infty} rt(x) * P(AET(c)). \tag{13}$$

With the above discussion, we can get the average residence time W:

$$W = \frac{all\_time}{all\_arrival}. \tag{14}$$

Note that the data arrival rate λ in logical time is equal to the miss rate as only a miss can introduce a new object into the queue (cache). With Little's law, the product of data arrival rate λ with average data residence time W is equal to cache size c:

$$\begin{aligned} c &= \lambda W \\ &= miss\_ratio * \frac{all\_time}{all\_arrival} \\ &= P(AET(c)) * \frac{\sum_{x=1}^{AET(c)} rt(x) * x + \sum_{x=AET(c)+1}^{\infty} rt(x) * AET(c)}{\sum_{x=1}^{\infty} rt(x) * P(AET(c))} \\ &= \frac{\sum_{x=1}^{AET(c)} rt(x) * x + \sum_{x=AET(c)+1}^{\infty} rt(x) * AET(c)}{\sum_{x=1}^{\infty} rt(x)} \end{aligned}$$

$$
\begin{aligned}
&= \frac{\sum_{x=1}^{\infty} rt(x) + \sum_{x=2}^{\infty} rt(x) + \cdots + \sum_{x=AET(c)+1}^{\infty} rt(x)}{\sum_{x=1}^{\infty} rt(x)} \\
&= \frac{\sum_{x=1}^{\infty} rt(x)}{\sum_{x=1}^{\infty} rt(x)} + \frac{\sum_{x=2}^{\infty} rt(x)}{\sum_{x=1}^{\infty} rt(x)} + \cdots + \frac{\sum_{x=AET(c)+1}^{\infty} rt(x)}{\sum_{x=1}^{\infty} rt(x)} \\
&= P(0) + P(1) + \cdots + P(AET(c)) \\
&= \int_0^{AET(c)} P(x)\,\mathrm{d}x.
\end{aligned}
$$

From the above analysis, we can conclude that under the same eviction time assumption, AET model is consistent with Little's law. In other words, Little's law can help derive AET model.

2.5 AET in Shared Cache

When sharing a cache, a set of co-run programs interact with each other. We consider the case where co-run programs have disjoint datasets. Each time a reference is executed by one of the program, the accessed block is brought to the most recently used (MRU) position in the cache. After cache is filled up, any program's newly inserted data can evict the data that belongs to other program. This cache sharing situation is popular and exists in many system design. We want a composable model to derive the composite effect on shared cache from individual solo-run locality analysis. Ding et al. (2014) define *composability* as follows: a locality metric is composable if the metric of a co-run can be computed from the metric of solo-runs. AET is composable: given the solo-run AETs of individual programs, we can derive the co-run AETs in the shared cache. There are $n + 1$ co-run AETs for n co-run programs: one for each program and one for the group. We derive them by solving another AET equation. Equation solving has two basic questions: Does a solution exist, and if so, is the solution unique?

For any data block in the shared cache, once it is no longer accessed, it will move from the MRU position toward the LRU position in the shared cache. Because each evicted data has traveled the same LRU stack shared by all programs, no matter which program the data block belongs to, all evicted data should have identical expected eviction time. Hence, we have the equation for the eviction-time equalization assumption: *when n programs share a cache of size c, all n co-run AETs,* $AET_i(c)$ *for each program i, and* $AET(c)$ *for the group, are the same*:

$$AET_1(c) = AET_2(c) = \cdots = AET_n(c) = AET(c). \tag{15}$$

We now show that this equation has one and only one solution.

To explain the derivation, we start with the symmetrical case, where n co-run programs are identical. For program i, let $r_{solo,i}$ be its access rate, $rt_{solo,i}(t)$ be the reuse-time histogram, $P_{solo,i}(t)$ be the CCDF, defined as in Section 2.2. The aggregate access rate is naturally $r_{co} = n\, r_{solo,i}$. We define the *co-run logical clock*. The co-run clock runs n times faster, with one out of every n ticks for each program. For program i, the co-run reuse time $rt_{co,i}(nt) = rt_{solo,i}(t)$, or equivalently $rt_{co,i}(t) = rt_{solo,i}(t/n)$. Because of the time change, the probability function under co-run clock of program i becomes $P_{co,i}(t) = P_{solo,i}(t/n)$. The aggregate probability is the weighted sum of the group:

$$P(t) = \sum_{i=1}^{n} P_{co,i}(t)/n = \sum_{i=1}^{n} P_{solo,i}(t/n)/n = P_{solo,i}(t/n). \tag{16}$$

From $P(t)$, we use the distance-integration equation (Equation (6)) to derive the co-run AET:

$$\int_0^{AET(c)} P(x)\,\mathrm{d}x = c. \tag{17}$$

ALGORITHM 3: Compose Aggregate CCDF of Reuse Time, $P[]$, from Individual CCDF of Reuse Time

Require: $P_{solo}[][]$ // CCDF of reuse use of every co-run program
Require: num // number of co-run programs
Require: $r[]$ // access rate of every program
Require: len // largest reuse time
Ensure: $P[]$ // aggregate CCDF, $P[]$

```
function MERGE(P_solo[][], num, r[], len)
    R ← getsum(r[])
    P[0] ← 1.0
    for t ← 1..len do
        P[t] ← 0
        for i ← 1..num do
            P[t] ← P[t] + P_solo[i][t * r[i]/R] * r[i]/R
        end for
    end for
    return P[]
end function
```

The equation looks the same as Equation (6), but $P(x)$ is the aggregate probability, x is the co-run time, and $AET(c)$ is average eviction time of the shared cache.

In the shared cache, any access by any program is a miss if and only if its reuse time is greater than $AET(c)$. The group miss ratio is therefore $mr(c) = P(AET(c))$, and the portion of this miss ratio contributed from the ith program is $mr_{co,i}(c) = P_{co,i}(AET(c))/n$. This contribution is the same from every program, so $mr_{co,i}(c) = mr(c)/n$. The solutions of the co-run AET and miss ratio for this symmetric case are unique.

Note that the co-run miss ratio of the ith program $mr_{co,i}(c)$ is the ratio of the miss count in the ith program divided by the number of accesses of all programs. In other words, it is the individual miss ratio defined on the co-run clock. This definition enables us to add miss ratios of different programs directly. It can also be easily converted to the conventional miss ratio.

We now consider the general case. It differs from the previous, symmetric case in two ways: each program i may have a different access rate $r_{solo,i}$ and a different reuse time histogram and hence the CCDF, $P_{solo,i}(t)$. Let the total access rate be $r = \sum_{i=1}^{n} r_{solo,i}$. For program i, the co-run reuse time $rt_{co,i}(t) = rt_{solo,i}(t\frac{r_{solo,i}}{r})$. The CCDF of program i under co-run clock becomes

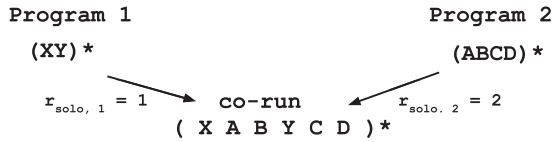
$$P_{co,i}(t) = P_{solo,i}\left(t\frac{r_{solo,i}}{r}\right). \tag{18}$$

The aggregate $P(t)$ is the weighted sum of $P_{co,i}(t)$ by their access rates:

$$P(t) = \sum_{i=1}^{n} P_{co,i}(t)\frac{r_{solo,i}}{r} = \sum_{i=1}^{n} P_{solo,i}\left(t\frac{r_{solo,i}}{r}\right)\frac{r_{solo,i}}{r}. \tag{19}$$

Algorithm 3 implements Equation (19), which converts the CCDF's of individual programs to the co-run CCDF. The shared-cache distance-integration equation (Equation (17)) can now compute $AET(c)$ for the general case as implemented in Algorithm 2.

We now investigate the relationship between the co-run miss ratio and the miss ratio contributed from each individual program. The group miss ratio is $mr(c) = P(AET(c))$, and the portion of the



	X	A	B	Y	C	D
$rt_{solo,\,i}$	2	4	4	2	4	4
$rt_{co,\,i}$	6	6	6	6	6	6

Fig. 7. An example to show co-run reuse time is composed from solo reuse time.

miss ratio contributed from program i is

$$mr_{co,i}(c) = P_{co,i}(AET(c))\frac{r_{solo,i}}{r}. \tag{20}$$

The contribution is now individualized and differs depending on the individual access rate $r_{solo,i}$ and reuse time histogram $rt_{solo,i}(t)$. Below is the co-run miss ratio of the group as the sum of the co-run miss ratio of each individual. These solutions are unique for each program group:

$$mr(c) = P(AET(c)) = \sum_{i=1}^{n} mr_{co,i}(c) = \sum_{i=1}^{n} P_{co,i}(AET(c))\frac{r_{solo,i}}{r}. \tag{21}$$

To help the reader understand how co-run reuse-time distribution $rt_{co,i}$ can be composed from individual reuse-time distribution $rt_{solo,i}$, we give a co-run example in Figure 7 to show how reuse time changes. Assume there are two programs running together. Program 1 only accesses X and Y in a circular pattern while program 2 accesses A, B, C and D in the same way. Assume their access rate ratio is $\frac{r_{solo,2}}{r_{solo,1}} = 2$, and the co-run trace is

$$(XABYCD)^*. \tag{22}$$

As shown in Figure 7, the co-run reuse time of every access in the combined trace is 6. It can be derived from individual reuse-time distribution. For program 1, the solo reuse time of every access under its own clock is 2. Under co-run clock, the co-run reuse time of every access from program 1 is

$$rt_{co,1}(6) = rt_{solo,1}\left(6 * \frac{r_{solo,1}}{r_{solo,1} + r_{solo,2}}\right) = rt_{solo,1}(2). \tag{23}$$

For program 2, all accesses under its own clock have reuse time 4. Therefore, their co-run reuse time is composed as

$$rt_{co,2}(6) = rt_{solo,2}\left(6 * \frac{r_{solo,2}}{r_{solo,1} + r_{solo,2}}\right) = rt_{solo,2}(4). \tag{24}$$

From the above example, we demonstrate how co-run reuse-time distribution $rt_{co,i}(t)$ can be composed from individual reuse-time distribution $rt_{solo,i}(t)$ and access rate $r_{solo,i}$.

Composition Invariance. The aggregated miss ratio can be computed using AET in two ways: directly using the aggregate $P(t)$ or indirectly as the sum of individual miss ratios. Mathematically, the two results are the same, as shown by Equation (21). We call this mathematical equivalence the *composition invariance*. A composable model has this invariance if the group miss ratio is the same whether it is composed from the individual (solo-run) locality or added together from the individual (co-run) miss ratio. Early composable models used reuse distance and footprint and

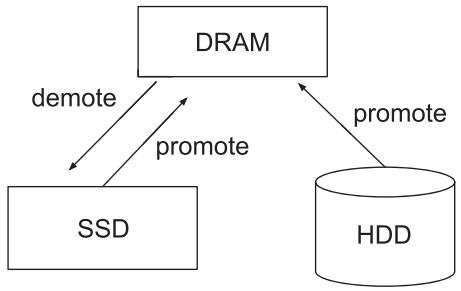


Fig. 8. Exclusive Cache.

had only one way to compute the group miss ratio (Chandra et al. 2005; Suh et al. 2014; Xiang et al. 2011a, 2011b). Recent models used footprint and the higher order theory of locality (HOTL) to obtain composition invariance (Xiang et al. 2013; Wang et al. 2015; Brock et al. 2015). As an alternative model of cache sharing, Brock et al. treated the shared cache as the partitioned cache, where each program is "imagined" to occupy a *natural partition* (Brock et al. 2015). Unlike the "imagined" natural partition in Brock et al., eviction-time equalization by AET is a real property of the shared cache. Because the two methods have very different formulation, a direct comparison is difficult, which we leave as future work. In this article, we will validate our assumption of AET equalization and show the effectiveness of shared cache AET model in Section 4.3.

2.6 AET in Multi-Level Exclusive Cache

Multi-level caches exist in a lot of storage systems and architectures (Kgil and Mudge 2006; Canim et al. 2010; Yadgar et al. 2008). With the unit price (in $/byte) of NAND flash memory goes down, solid state drives are added as a secondary cache between DRAM and HDD to improve I/O throughput and response time of a storage server. This DRAM-SSD-HDD hybrid storage hierarchy is a compromise between speed and cost. In this section, we will show how to model the secondary cache using AET model. For simplicity, we will refer DRAM as L1 cache, and SSD as L2 cache in this section.

Exclusive cache is one of the most popular multi-level cache design (Gill 2008; Wong and Wilkes 2002; Chen et al. 2003). We propose to model exclusive cache behavior using AET model. An exclusive cache requires that a datum should only exist in one cache level. As shown in Figure 8, data request will first check L1 cache, if the data is not in L1 cache, then it will check L2 cache instead. If the data is cached by L2, then it will be deleted from L2 and promoted in L1. If the data is not in L2, then it will be fetched from HDD into L1. When L1 is full, the new data will cause a demotion and evict a victim to L2. The cache hierarchy keeps the most recently used data in L1, while it uses L2 as backup to store the next most recently used data.

2.6.1 Dedicated L1 and Dedicated L2. We start with a simple case where both L1 and L2 are dedicated to a single service. Assume the sizes of L1 cache and L2 cache are c_1 and c_2, respectively. On an L1 cache miss, the missed data will be fetched from the lower-level cache and inserted to the MRU position of L1 cache. The evicted data from the LRU position of L1 cache will be inserted to the MRU position of L2 cache. With this observation, we can consider L1 and L2 caches as a unified LRU cache of size $c_1 + c_2$. The only difference is the reference latency of the front part (L1) is smaller than the rear part (L2).

Table 2. The Reuse Time t and Its Categories

	L1 hit	L1 miss & L2 hit	L1 miss & L2 miss
$t < AET(c_1)$	*		
$AET(c_1) <= t < AET(c_1 + c_2)$		*	
$t >= AET(c_1 + c_2)$			*

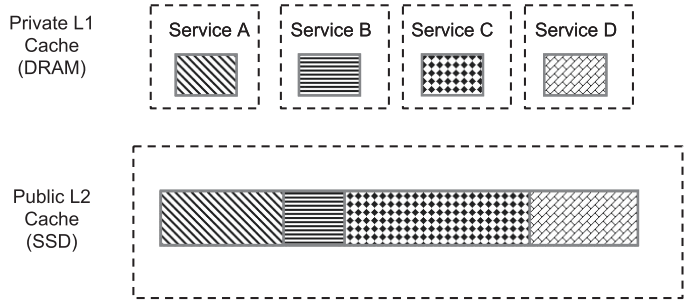


Fig. 9. Shared L2 cache with private L1 cache.

Now, we can characterize the behavior of this unified cache with AET model. Once we get the reuse-time distribution, the average eviction time from position 0 to c_1 can be estimated, as well as the AET from 0 to $c_1 + c_2$. In Table 2, we show the relationship between the ranges of reuse time and the expected hits or misses.

For reuse time t, if it is smaller than $AET(c_1)$, the data should be located between position 0 and c_1, it is an L1 hit. If t is between $AET(c_1)$ and $AET(c_1 + c_2)$, then the data should be located between position c_1 and $c_1 + c_2$, which means the data is in L2 cache but not in L1 cache. When t is larger than $AET(c_1 + c_2)$, the data is in neither level of cache. Therefore, it is an L1 and L2 miss. Using the above ideas, the L1 and L2 cache miss ratio can be estimated with a simple extension to Algorithm 2.

2.6.2 Dedicated L1 and Shared L2. Now, we consider n co-run programs where each program has its own private L1 and all programs share L2 as shown in Figure 9. Each program may have a different L1 size, d_i, and the total L1 size is the DRAM size c_1. For program i, the L1 cache miss ratio is the proportion of reuses with a reuse time larger than the program's solo-run $AET_{L1,i}(d_i)$:

$$L1\ miss\ ratio\ of\ Program\ i = P_{solo,i}(AET_{L1,i}(d_i)), \tag{25}$$

where d_i is the L1 cache size of program i and $AET_{L1,i}(d_i)$ is computed by the integration in Equation (6). This solution is exactly the AET model for private cache. There is no cache sharing yet.

The L2 miss ratio is the proportion of reuse time larger than the common shared cache AET, which we call CT:

$$L2\ miss\ ratio = P(CT), \tag{26}$$

where $P(CT)$ is the aggregate probability. To compute CT, we need to consider both the filtering effect of L1 and the sharing of L2. Under sharing, each service may take different L2 cache occupancy. We can regard each service's data in L1 cache and L2 cache as organized in a unified LRU list. Assuming that d_i is the size of L1 cache of Service i, o_i is the size of its occupancy in the shared L2 cache, and CT is the common eviction time of all services (in global time), the expected eviction

distance of service i is

$$\int_0^{CT} P_{co,i}(x) * \frac{r_{solo,i}}{r} \, \mathrm{d}x = d_i + o_i, \tag{27}$$

where the notations are defined in Equation (18). Note that $P_{co,i}$ can be derived from from individual $P_{solo,i}$'s using Equation (18). The eviction distance of Service i is the travel distance in the LRU list including its L1 cache and L2 cache occupancy. When L2 cache is shared by n services, the sum of their individual eviction distances is their total L1 and L2 cache usage. Obviously, the sum of eviction distances of all services minus their L1 cache sizes is the L2 cache size:

$$\sum_{i=1}^{n} \int_0^{CT} P_{co,i}(x) \frac{r_{solo,i}}{r} \, \mathrm{d}x - c_1 = c_2, \tag{28}$$

where $c_1 = \sum_{i=1}^{n} d_i$ and $c_2 = \sum_{i=1}^{n} o_i$.

For any L2 cache size, we can find CT in linear time. Therefore, the L2 cache occupancy of different services under various L1 cache configuration can be predicted, as well as the L2 miss ratio of each service. Algorithm 4 implements Equations (18) and (28) to calculate shared L2 cache MRC's for each individual co-run programs using individual CCDF, $P_{solo,i}$ as input. It is possible that one of the services may not utilize all L1 cache, because its current working set size is smaller than the L1 cache. The algorithm checks for this possibility and will exclude this service in computing L2 sharing (not counted in Equation (28)).

The model of two-level sharing subsumes the model of single-level sharing in Section 2.5. The single-level problem is solved by the two-level solution by setting the private cache sizes, i.e., d_i's in Equation (28), to 0.

With reuse-time distributions of individual services, the two-level cache of any size combination can be modeled in linear time. This technique can be used to predict multi-level cache behavior and guide the allocation of cache resource to improve performance or guarantee quality of service for high priority services.

3 REUSE TIME HISTOGRAM (RTH) SAMPLING

For efficiency, AET-based MRC profiling can use sampled RTH instead of real RTH. Since it is only the probability distribution that it cares about, if the sampled RTH maintains the same distribution as the real RTH, the estimated AET will be accurate. By sampling a small fraction of references, the space overhead can be largely eliminated. This section presents efficient MRC analysis through AET sampling.

3.1 Sampling Techniques

To capture the distribution of the real RTH, all the references have to be sampled with equal probability. This seems to be an easy target, but it is not the case in real applications. Next, we list four sampling techniques and discuss their strength and weakness.

Set Sampling. The set sampling requires monitoring a fixed subset of the working set. It is known as hold-and-sample and has been used in measuring reuse distance (Zhong and Chang 2008; Schuff et al. 2010; Tam et al. 2009; Cascaval et al. 2005) or reuse time (Beyls and D'Hollander 2006). During sampling phase, all the references to the subset will be recorded in sampled RTH. This technique is simple and easy to implement, and only a fixed hash table is required. However, in a real program, references are not evenly distributed on every data object. Large portion of accesses may focus on

ALGORITHM 4: Calculating Shared L2 Cache MRC for Every Program

```
Require: P_solo[][] // CCDF of every co-run program
Require: num // number of co-run programs
Require: r[] // access rate of every program
Require: len // largest reuse time in co-run
Require: d[] // private L1 cache size of each
Require: M // largest L2 cache size
Ensure: MRC[][] // L2 cache miss ratio curve of every program
function ShareLastLevelCacheMRC(P_solo[][], num, r[], len, d[], M)
    R ← getsum(r[])
    for i ← 1..num do // computing Equation (18)
        P_co[i][0] ← 1.0
        for j ← 1..len do
            P_co[i][j] ← P_solo[i][j * r[i]/R]
        end for
    end for
    integration[] ← 0
    sum ← 0
    t ← 0
    for c_2 ← 1..M do
        while (sum < c_2) do // computing Equation (28)
            sum ← 0
            for i ← 1..num do
                integration[i] ← integration[i] + P_co[i][t] * r[i]/R
                if integration[i] > d[i] then
                    sum+ = integration[i] − d[i]
                end if
            end for
            t ← t + 1
        end while
        for i ← 1..num do
            if integration[i] > d[i] then
                MRC[i][c_2] ← P_co[i][t − 1]
            end if
        end for
    end for
    return MRC[][]
end function
```

a small subset. In this case, the RTH collected from a small portion of working set may not reflect the real reference pattern. This will lead to imprecise estimation of AET.

Fixed Interval Sampling. To avoid the bias of set sampling, the fixed interval sampling collects a subset of references instead of a subset of the working set. After every m references, it places the current accessed data into the monitoring set. At the next reference of the data, the reuse time is recorded into RTH, and the data is deleted from the monitoring set. By this design, the reuses are sampled by the same probability, which provides a better RTH approximation than set sampling.

However, the accuracy of fixed interval sampling may be influenced by another problem. Since the sampling rate m is a fixed value, if the reference pattern of some data shows a different distribution at the chosen interval, the sampled RTH cannot reflect the actual distribution of this pattern.

Random Sampling. The random sampling can overcame the problem we mentioned in interval sampling and set sampling. Instead of using fixed sampling rate m, the distance between two adjacent monitoring points is a random value. In a real application, we can set the random value to a certain range to control the number of references sampled for RTH. We have tested the above three sampling techniques and found that the random sampling achieved the highest stability and accuracy. This form of random sampling for MRC analysis is pioneered by StatStack (Eklov and Hagersten 2010).

Reservoir Sampling. The space used to store sampled data grows linearly with respect to dataset size. To bound the space cost, reservoir sampling technique (Vitter 1985) was used by Beyls and D'Hollander (2006) for locality analysis. Let the number of entries in the monitoring set (reservoir) be k. When the ith sampled data arrives, reservoir sampling keeps the new data (tagged as "unsampled") in set with probability $min(1,\ k/i)$ and randomly discards an old data block when the set is full. Every time a monitored data block is reused, its reuse time will be recorded. This data block will be tagged as "sampled" and all of its following reuses will not be recorded. This design ensures even sampling and avoids the access distribution problem we have in set sampling. When the sampling is over, the RTH is updated based on the "sampled" data entries remaining in set. The "unsampled" entries are those data objects with no reuse after being inserted. They are cold misses, which we will discuss in Section 3.3. Reservoir sampling reduces the space complexity of RTH sampling from $O(M)$ to $O(1)$.

3.2 Adaptive Phase Sampling

A program may have time-varying behavior, and its reuse-time distribution may vary across different phases. If we use the RTH of the entire trace to model MRC by AET, then we are actually using the average behavior of all phases to model the miss ratios in individual phases. This will cause mis-prediction, because the access patterns we record in RTH is not the behavior in each phase. In this section, we present an adaptive phase sampling technique for AET calculation, which can reduce the miss prediction caused by phase behavior.

One solution of phase analysis is to divide a program into fixed length intervals (Duesterwald et al. 2003). However, even-division cannot distinguish the length of each phase, as well as the border of each phase. Instead, we propose to adaptively detect phase changing. For every w accesses, which we refer to as a "monitoring window," we compare the RTH collected in this window with last window to determine if the two windows have different access pattern. For IO systems, we use Euclidean distance of two RTHs to present their similarity:

$$d(RTH1, RTH2) = \sqrt{\sum_{x=1}^{\infty} (p1(x) - p2(x))^2}, \tag{29}$$

where $p1(x)$ stands for the probability of a reference with a reuse time x in $RTH1$. If their distance is above the threshold we set, then we can say they represent different patterns.

For every monitoring window, we compare its reference pattern with the last window by calculating the Euclidean distance or changing of miss ratios. If a phase change is detected, then we use the RTH collected in the preceding windows to model the MRC of the last phase. Then, the RTH is reset to record the reuse times of the next phase. With this design, each phase may have a

different length (number of windows). The MRC of the entire program is the weighted miss ratio of each phase at each cache size. Specifically, assume the length of the ith phase is L_i. If the MRC calculated at the ith phase is $MRC_i(c)$, then the MRC of the entire program is

$$MRC(c) = \frac{\sum_{i=1}^{n} L_i * MRC_i(c)}{\sum_{i=1}^{n} L_i}. \tag{30}$$

There is an additional detail we should mention. Not every sampled datum in the monitoring set will see its reuse in the same phase. Before entering the next phase, the monitoring set will not be cleaned, the next phase still keeps track of these data until they are reused and then deleted from the monitoring set. We use the backward reuse time, so the inter-phase reuse time is added to the RTH of the current phase.

Wang et al. recently developed adaptive bursty footprint sampling (ABF) for CPU cache MRC monitoring, which compares the difference between the real miss ratio and the predicted value to adaptively activate new phase sampling (Wang et al. 2015). ABF predicts miss ratio by footprint analysis. It gives similar suggestion that phase behavior should be modeled separately to increase MRC prediction accuracy.

3.3 Cold Miss Handling

As we mentioned in Section 2.2, the ∞ bin of RTH counts the number of cold misses. Therefore, we should set the infinite reuse-time bin of the sampled RTH to the number of cold misses in all sampled references. However, in random sampling, we cannot tell if a sampled access is the first reference to an address. As we know, in a trace of finite length, any referenced address has its first access and last access. It means the number of cold misses is equal to the number of the references that have no reuse (last access). Because the chances to meet these two kinds of access are equal, we use the number of references with no reuse in all sampled references to revise the number of cold misses in the sampled RTH. In random sampling, they are the data objects that are still in the monitoring set after sampling is complete. In reservoir sampling, they are the data objects that are tagged "unsampled."

4 EVALUATION

In this section, we evaluate the AET model by comparing it with two recent techniques: Counter Stacks (Wires et al. 2014), SHARDS (Waldspurger et al. 2015). The two techniques are current best MRC technique designed for storage workloads. We also present the evaluation of shared cache AET and Multi-level Cache AET. The adaptive phase sampling technique is only evaluated individually in Section 4.3. We do not enable this technique in other sections, because we want to make a fair comparison.

We use a Dell PowerEdge R720 with ten-core 2.50GHz Intel Xeon E5-2670 v2 processors and 256GB of RAM. Benchmark traces are read from RAMDisk to avoid the IO bandwidth delay. We have implemented these techniques in C++. To save memory and make a fair comparison, we record the reuse time histogram (AET) and reuse distance histogram (Counter Stacks, SHARDS) using the compressed representation by Xiang et al. (2011b). Each histogram is an array that is binned in logarithmic ranges. Each (large enough) power-of-two range is divided into (up to) 256 equal-size increments. This representation requires less than 100KB for all our workloads.

4.1 AET vs Counter Stacks

Counter Stacks is a recent algorithmic breakthrough by Wires et al. to finally solve the open problem of reducing the asymptotic space complexity of MRC analysis to below M, the size

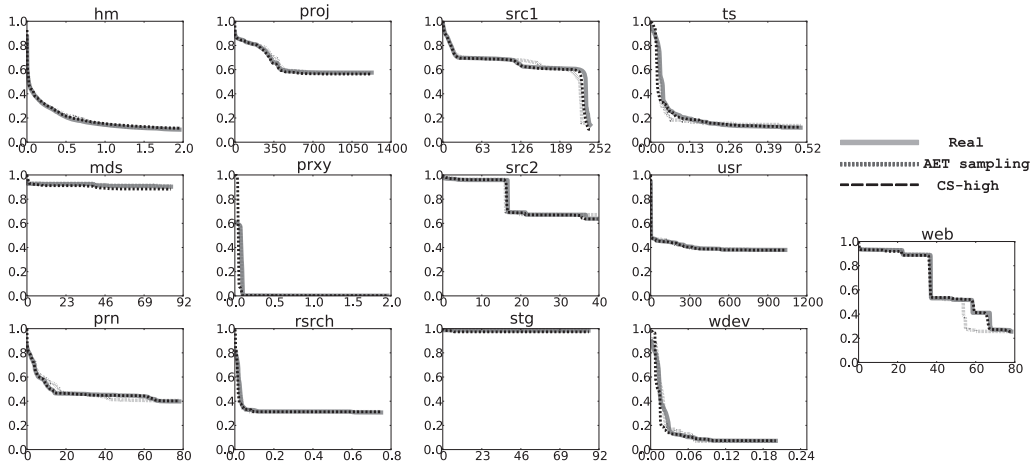


Fig. 10. The predicted miss ratio (y-axis) over cache size (GB, x-axis) by AET sampling and high-fidelity Counter Stacks.

of data (Wires et al. 2014). It uses probabilistic counters to estimate the reuse distances. While other reuse distance measurement techniques consume linear space overhead, the HyperLogLog counter (Fusy et al. 2007) used by Counter Stacks only requires extremely small space while maintaining an acceptable accuracy. Every d references and every s seconds, Counter Stacks starts a new counter to record the number of distinct data accessed from the current time. During the execution, the number of active counters keeps growing. Counter Stacks periodically writes the results of active counters to the disk. The data in the disk is used to compute the reuse distance distribution and construct MRC. To reduce the number of live counters, Counter Stacks uses a pruning strategy to delete a younger counter whenever its value is as least $(1 - \delta)$ times the older counter's value. Obviously, Counter Stacks can balance between accuracy and the number of counters.

We compare AET model with Counter Stacks using the same storage traces released by Microsoft Research Cambridge (MSR) (Narayanan et al. 2008), as used by Counter Stacks. The traces are configured with only read requests of 4KB cache blocks. We test Counter Stacks under two different fidelities. The experimental parameters follow those used in Wires et al. (2014), with high fidelity ($d = 1M$, $s = 60$, $\delta = 0.02$) and low fidelity ($d = 1M$, $s = 3600$, $\delta = 0.1$). For AET, we use random sampling at a rate of one per 10^4 accesses, and reservoir sampling where the number of entries in the hash table (32-bit address) is limited to 16K.

Figure 10 shows the MRCs profiled by AET random sampling and high-fidelity Counter Stacks (CS-high) as well as the real MRCs calculated using precise reuse distances. As we can observe, AET sampling and CS-high both approximate the real MRCs well. As for CS-low and AET reservoir sampling, we only list their absolute prediction error in Table 3 for comparison.

Table 3 shows two types of averages, arithmetic and weighted. The ones marked with a "*" are weighted by the working set size, which is the length of MRC. The weighted average prediction errors, AET random sampling (RAN, 0.96%) and AET reservoir sampling (RES, 1.12%), are in between of high-fidelity Counter Stacks (0.77%) and low-fidelity Counter Stacks (1.26%) but much higher throughput (arithmetic average) and much lower space overhead (weighted average) than both methods of Counter Stacks.

AET uses reuse time histograms while Counter Stacks uses reuse distance histograms. With the compression technique we use for the histograms, the size of both histogram structures are

Table 3. The Comparison Between Counter Stacks (CS) and AET

	WSS	Prediction Error (%)				Memory (KB)				Throughput (*M*reqs/s)			
		AET		CS		AET		CS		AET		CS	
	(GB)	RES	RAN	high	low	RES	RAN	high	low	RES	RAN	high	low
proj	1238.9	0.76	0.74	0.93	1.04	384	584	8384	1376	31.01	26.10	1.32	3.94
usr	1035.1	0.79	0.37	0.24	0.31	384	501	7744	1376	30.22	30.67	1.36	3.87
src1	312.7	3.09	2.90	1.54	4.78	384	176	5408	1088	30.17	44.88	1.86	4.88
mds	86.9	0.85	0.70	1.81	1.82	384	114	2848	832	79.82	77.08	3.16	6.17
stg	85.7	0.09	1.01	1.11	1.11	384	114	4256	928	78.90	51.99	2.23	6.30
web	78.3	3.81	3.65	1.00	2.92	384	111	6464	1120	56.00	70.67	1.50	5.60
prn	77.5	2.28	2.08	0.31	0.57	384	110	4960	960	60.81	71.17	1.28	5.79
src2	39.9	1.09	1.02	0.57	2.19	384	94	4704	960	84.49	71.44	2.48	6.66
hm	2.0	0.90	0.77	1.01	1.31	384	79	3680	608	65.74	67.62	0.33	6.87
prxy	2.0	0.20	0.04	1.62	1.69	384	79	2112	576	31.43	76.77	3.40	7.23
rsrch	0.7	2.90	0.92	0.30	2.84	384	78	2720	416	82.55	82.55	1.22	7.26
ts	0.5	1.51	2.04	0.41	0.78	384	78	1920	640	88.02	74.12	1.08	5.80
wdev	0.2	2.62	1.21	0.20	0.11	384	78	864	352	86.81	86.81	1.28	5.75
avg*	—	1.12*	0.96*	0.77*	1.26*	384*	452*	7363*	1,292*	61.99	63.99	1.73	5.86
sum	2,960	—	—	—	—	4,992	2,196	56,066	11,232	—	—	—	—

comparable. Consequently, the key difference in space between the two techniques is the hash table used by the AET algorithm and the Hyperloglog counters used by Counter Stacks. In AET random sampling, the number of hash table entries is the number of data blocks being monitored at this time. The theoretical upper bound is the working set size times the sampling rate. In AET reservoir sampling, the space is constant, i.e., a hash table of a fixed size. In Counter Stacks, the space used by probabilistic counters grows when more counters are used. Therefore, the space overhead of Counter Stacks is not constant. In Table 3, we also list the memory consumed by the hash table and Hyperloglog counters for MSR traces. The results show that the actual memory usage of AET random sampling is much lower than Counter Stacks. In fact, the total space consumption (not including the histogram array) of all 13 traces by AET random sampling is 2.2MB, while low- and high-fidelity Counter Stacks require 11MB and 56MB for Hyperloglog counters, respectively. In AET reservoir sampling, the space overhead is fixed at 384KB for each trace. Although the overall space consumption (5MB) is larger than random sampling, its weighted average space overhead is less than random sampling. Reservoir sampling reduces the space cost of random sampling only in *proj* and *usr*. They are the traces with the largest working set sizes. The remaining traces have smaller working sets. For these traces, reservoir sampling incurs a higher error even when it uses more space than random sampling. As we mentioned in Section 3.1, reservoir sampling only uses the remaining entries in hash table to update RTH and does not delete the data entry after the reuse is sampled (to measure the cold miss ratio). The actual number of reuses in RTH of reservoir sampling is less than random sampling under the same sampling rate.

It takes Counter Stacks $O(\log M)$ time to update the counters at each reference and $O(N \log M)$ for the entire trace. AET is linear time in $O(N)$. Table 3 shows that in our implementation, the throughput of AET random sampling is 37 and 11 times of the throughput of high- and low-fidelity Counter Stacks, respectively. AET reservoir sampling shows a similar throughput as AET random sampling does.

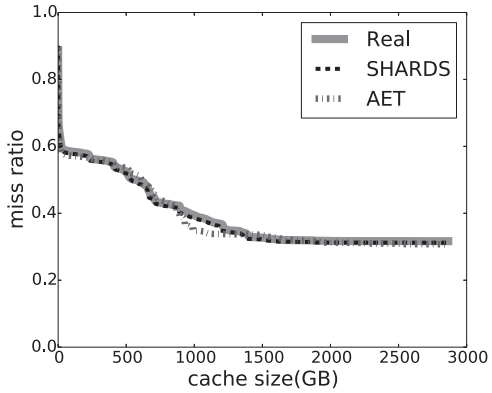


Fig. 11. MRCs predicted by AET sampling and SHARDS for the master trace.

The correctness of AET-based MRC is based on the assumption of stable distribution reuse distances. This brings inaccuracies to those workloads that violate the assumption. As we can observe in Figure 10 the AET-based MRC of *web* mispredicts the knee at around 50GB, but Counter Stacks perfectly models every detail of the curve, since it makes no assumption about the data distribution at all. Now, we can clarify the trade-off between the two techniques: AET makes a statistical assumption, offering good accuracy in most cases in $O(N)$ time. Counter Stacks makes no statistical assumption, delivering good accuracy in all cases in $O(N \log M)$ time. Later in Section 4.3, we show that adaptive phase sampling mitigates this problem of AET.

4.2 AET vs SHARDS

SHARDS (Spatially Hashed Approximate Reuse Distance Sampling) is recently developed by Waldspurger et al. (2015). It uses hash-based spatial sampling and a splay tree to track the reuse distances of the sampled data. It limits the space overhead to a constant by adaptively lowering the sampling rate. SHARDS outperforms Counter Stacks in both memory consumption and throughput for the merged "master" MSR trace (created by Wires et al. (2014)), which is a 2.4 billion-access trace combining all 13 MSR traces by ranking the time stamps of all accesses. Following them, we use the master trace for evaluation. For fairness comparison, we let AET and SHARDS both use $8K$ buckets hash table (64-bit address) for sampling. The pointers and variables in our implementation are all 64-bit sizes. Figure 11 shows the MRC profiled by AET random sampling with a sampling rate of one per one million accesses. The Mean Absolute Error (MAE) is 0.01. SHARDS gives a lower MAE of 0.006 with $8K$ samples. We check the peak resident memory usage at run time, AET random sampling consumes 1.7MB memory while SHARDS consumes 2.3MB memory. The throughput of AET and SHARDS are 79.0M blocks/s and 81.4M blocks/s, respectively. For the same trace, Counter Stacks is most accurate, with an MAE of 0.003. However, it consumes 80MB memory, and the throughput is relatively low, 2.3M blocks/sec (Wires et al. 2014). AET reservoir sampling ($8K$) has 1.4MB resident memory usage and 66.6M blocks/s throughput, for an MAE of 0.01, same as AET random.

SHARDS and AET sampling have same time and space complexity, and their run time and memory usage are close in our test. However, the applicability of SHARDS is not limited to miss ratio prediction of LRU caches. Waldspurger et al. (2015, 2017) showed that the hash-based spatial sampling technique of SHARDS can be used to perform efficient scaled-down simulations for non-LRU caching algorithms such as ARC (Megiddo and Modha 2003). AET model assumes LRU

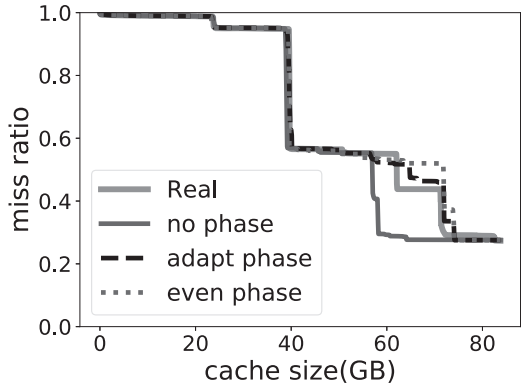


Fig. 12. The miss ratio versus cache size (GB) shown for adaptive phase AET and no-phase AET, compared with actual miss ratios.

Table 4. HPE and MPE Count of *web* at 70GB

	MPE	HPE	# Samples	Error (%)
no phase	5	1,336	8,117	16.4
even phase	831	155	8,117	8.3
adaptive phase	531	371	8,117	2.0

replacement policy so it currently cannot be used for a non-LRU cache. The strength of AET model is composability, which can be used to model shared cache as we will show in Section 4.4. But this is not a property of SHARDS.

4.3 Adaptive Phase Sampling

As mentioned in Section 3.2, adaptive phase sampling can improve the analysis accuracy for programs with phase behavior. In the MSR test, the AET-based MRC of *web* dose not predict the knee at around 50GB. We believe this prediction error is brought by the phase behavior of *web* trace. Therefore, we use adaptive phase sampling technique to evaluate if this error can be revised.

We divide *web* trace into monitoring windows of length $5 * 10^6$. The Euclidean distance of two windows is calculated using the compressed reuse time histogram representation, which is efficient in time and space. We set the phase changing threshold to 0.5, which divides *web* in to 10 uneven phases. Figure 12 shows the MRCs of no-phase and adaptive phase AET (random sampling at rate $10 * -4$) compared with the real MRC. As we can observe, adaptive phase AET successfully predicts the knee that is missed by no-phase AET. The MAE of adaptive phase AET is 1.17%, which is 70% reduction compared to no-phase AET (3.81%). We also show the MRC plotted by even-phase AET that divides the trace into 10 equal length phases to model the AET. The fixed length phase MRC does not match real MRC as good as adaptive phase MRC, which proves the necessity of adaptive phase detection. For other MSR traces, the same Euclidean distance does not predict phase changes, and adaptive phase AET performs the same as no-phase AET.

In Table 4, we list the numbers of MPE and HPE of no phase, even phase, and adaptive phase AET sampling at the cache size of 70GB. We sampled 8,117 data references for the *web* trace. No phase AET has 1,331 more HPEs than MPEs. It gives a 16.4% lower miss ratio than the real miss ratio. Even phase AET has 676 more MPEs than HPEs and yields a 8.3% higher miss ratio prediction than the real miss ratio. Adaptive phase AET is most accurate with 160 more MPEs than HPEs and a prediction error of only 2%.

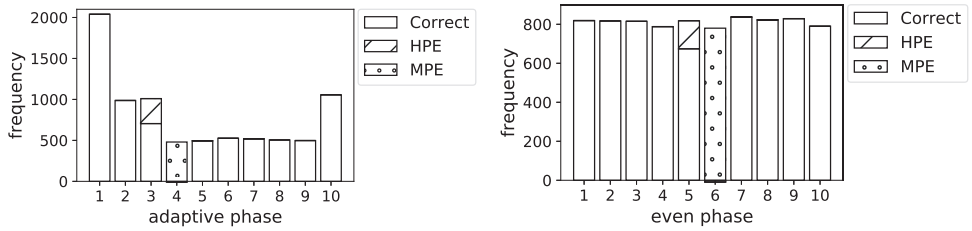


Fig. 13. The MPE and HPE distribution of adaptive and even phase AET for *web*.

Figure 13 shows the MPE and HPE distribution across individual phases for both even phase AET and adaptive phase AET. As we can observe, major MPEs and HPEs are located in only one phase of *web*, which indicates that this part of trace exhibits extremely skewed access behavior that cannot be predicted using average. With phase sampling, it can only influence the average behavior estimation for its own phase instead of all data in the long trace. Adaptive phase sampling further reduces the prediction error by accurately detecting the duration of a skewed phase.

4.4 Shared Cache AET

As we discussed in Section 2.5. AET is a composable metric that can be adopted to model shared cache. With the individual AETs of co-run programs, we can predict their combined MRC in the shared cache. This technique is useful in task scheduling in a system where shared caches (CPU cache and storage cache) are deployed. To verify our shared AET modeling technique, we test all combinations of two-trace, four-trace and six-trace co-run groups using six MSR traces: hm, mds, prn, src2, stg, web.

To measure the real MRC, we generate an interleaved trace from all the traces in each group. Consider their different lengths, the access rate in each combination is proportional to the inverse of their individual length. This makes all traces finish at the same time. To make the synthesized trace closer to the characteristics of real world traces, we randomly interleave the traces. When we choose the next access, a random value is generated to determine which trace to reference instead of uniform interleaving. For example, if trace a and trace b are merged with access rate ratio 1:3. For every access, we generate a random value r in $[0, 1]$. Then a is referenced if $0 < r <= 0.25$ and b is referenced if $0.25 < r <= 1.0$. Our results show that our model delivers similar effectiveness when compared to applying it to uniformly interleaved traces.

Figures 14, 15, and 16 show the shared cache MRCs of two, four, and six traces groups composed from the individual AET model of each trace, as well as the real MRCs calculated by accurate reuse distance tracking for the combined trace. For each group, we show the measured solo-run MRC for each program and the predicted and measured co-run MRCs for the group. Here, we use the same random sampling technique as Section 4.1. Hence, the overhead for individual modeling of two to six traces are the same as Table 3 shows. Combining their modeling results to derive shared cache MRC does not require extra space cost. The additional time cost is to combine all reuse-time distributions as we presented in Equation (19), which is negligible. In the figures, the MRCs by shared AET match the real MRCs well, giving an MAE of 0.01 for all $\binom{6}{4} + \binom{6}{4} + \binom{6}{6} = 31$ combinations. It means that the shared cache modeling by AET is accurate.

We observe different shape patterns in the figures. When low MRC traces is combined with high MRC traces, the shared cache MRC usually lies between the two individual MRCs, showing the effect that the combined MRC has the average intensity of the cache intensive and non-intensive traces. Another observation from these figures is the shift of MRC "cliffs" or "knees." For instance,

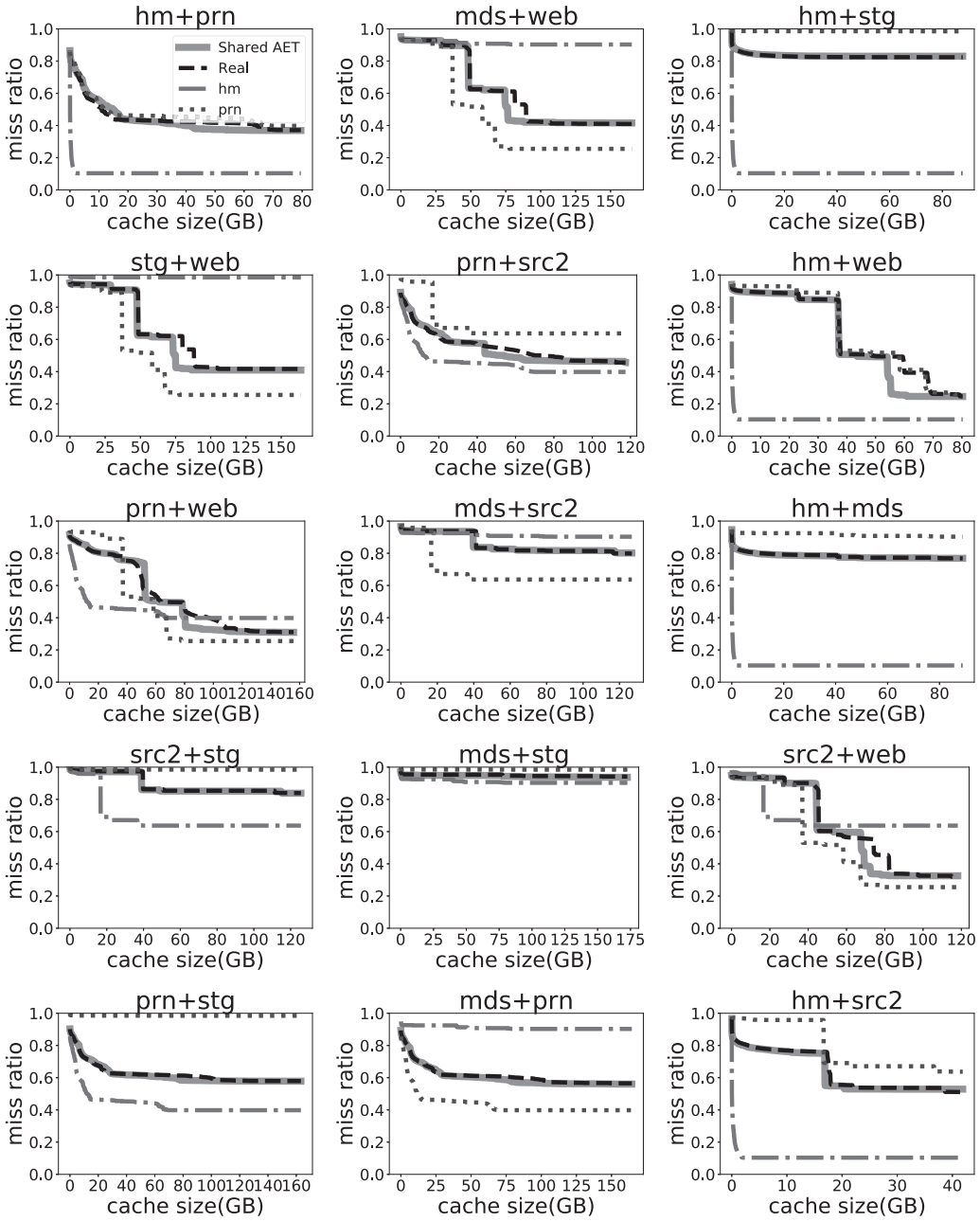


Fig. 14. The predicted miss ratio (y-axis) over cache size (GB, x-axis) for two-program combinations by Shared AET.

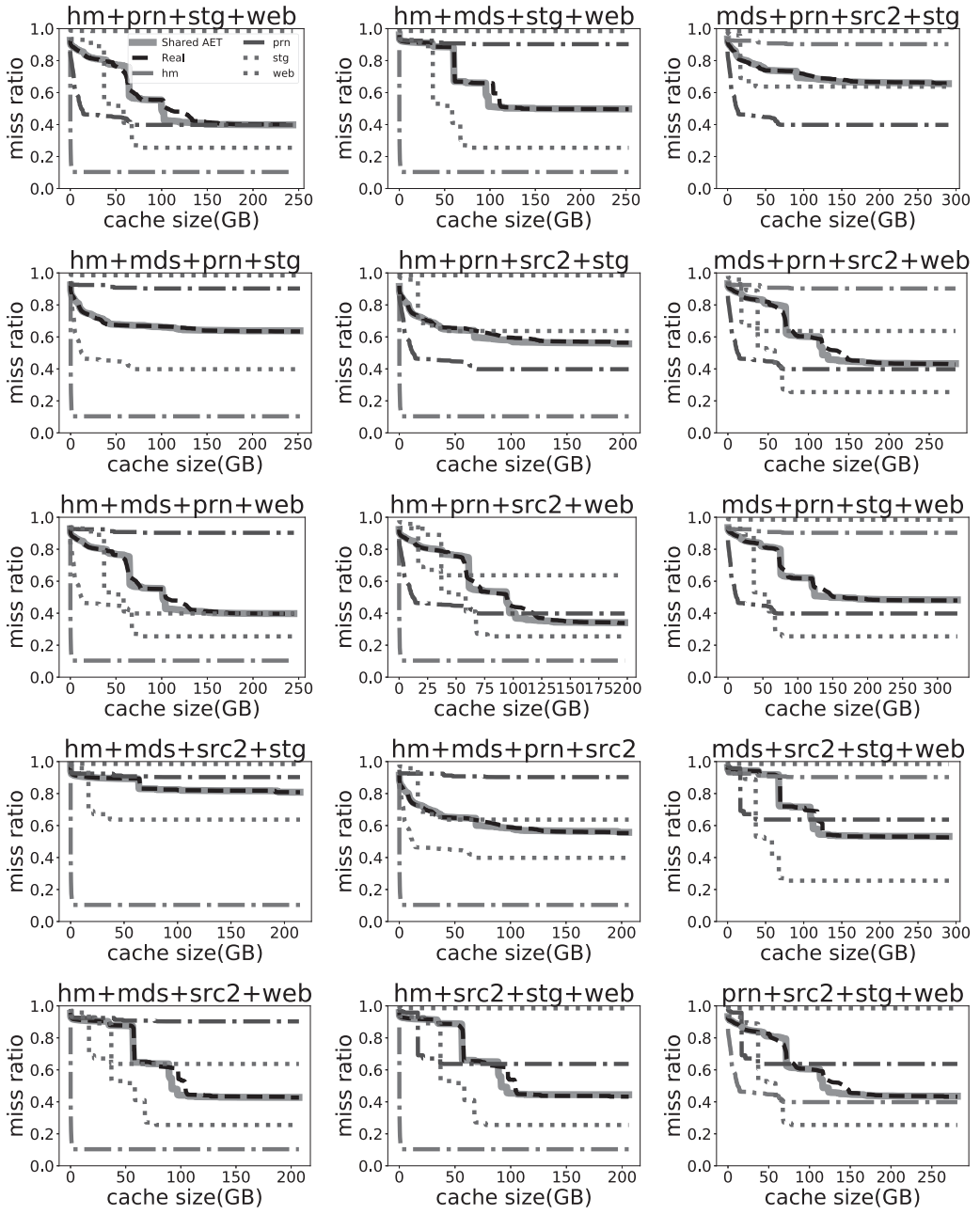


Fig. 15. The predicted miss ratio (y-axis) over cache size (GB, x-axis) for four-program combinations by Shared AET.

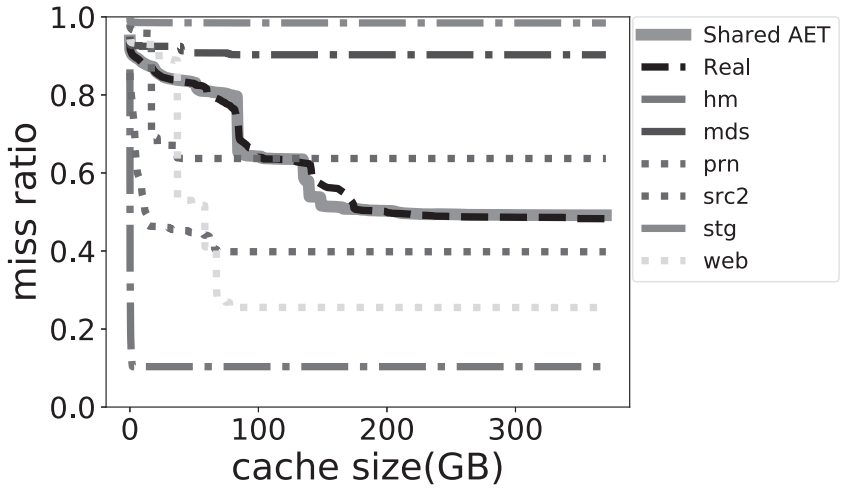


Fig. 16. The predicted miss ratio (y-axis) over cache size (GB, x-axis) for six-program combination by Shared AET.

the individual MRCs of *web* and *src2* have cliffs. When they co-run with other traces, the cliff will shift right, since the individual occupancy of the shared cache is smaller than the cache capacity. In the meanwhile, the height of a cliff in an individual MRC will be shorter in the shared cache MRC because of the effect of averaging.

The composability of AET is a key advantage over SHARDS, since SHARDS cannot characterize shared cache without co-run testing. The comprehensive co-run trace tests and their results confirm that the shared cache prediction by AET is accurate.

4.5 Multi-Level Cache AET

We present the exclusive multi-level cache modeling in Section 2.6. In this section, we evaluate the accuracy of the model using MSR traces.

We use a multi-service two-level cache simulator to acquire real L2 cache MRC where each service has its private L1 cache and share the L2 cache with other services. We compare the simulation result with prediction using AET model.

We select eight pairs of MSR traces that have close scale of working set size. We simulate their sharing under same L1 cache size, which is close to one-fifth of their working set size. We test different L2 cache size varies from 2 to 60 times of L1 cache size. The MSR traces combination and their private L1 cache size and range of L2 cache sizes are listed in Table 5.

Figure 17 shows the real L2 cache miss ratio curves and the predicted curves using two-level shared AET model. The prediction well-captures the shared L2 cache MRC with an MAE of 0.005. This technique can be used to manage multi-level cache system. When L1 and L2 caches are both configurable, especially under sharing, AET model can give the performance prediction of different configurations. Multi-level AET model can provide guidance for cache allocation or partitioning for better utilization or quality of service guarantee.

5 RELATED WORK

In 1972, Denning and Schwartz (1972) gave a linear-time, iterative formula to compute the average working-set size from reuse times (inter-reference intervals). Mathematically, the AET calculation is the same as the average working-set size computed by the Denning-Schwartz formula. In their

Table 5. Sizes of L1 Cache

	L1 cache size (GB)	L2 cache size range (GB)
proj&usr	20	40–1200
mds&stg	1.6	3.2–96
mds&web	1.6	3.2–96
mds&prn	1.6	3.2–96
stg&web	1.6	3.2–96
stg&prn	1.6	3.2–9
prn&web	1.6	3.2–9
hm&prxy	0.04	0.08–2.4

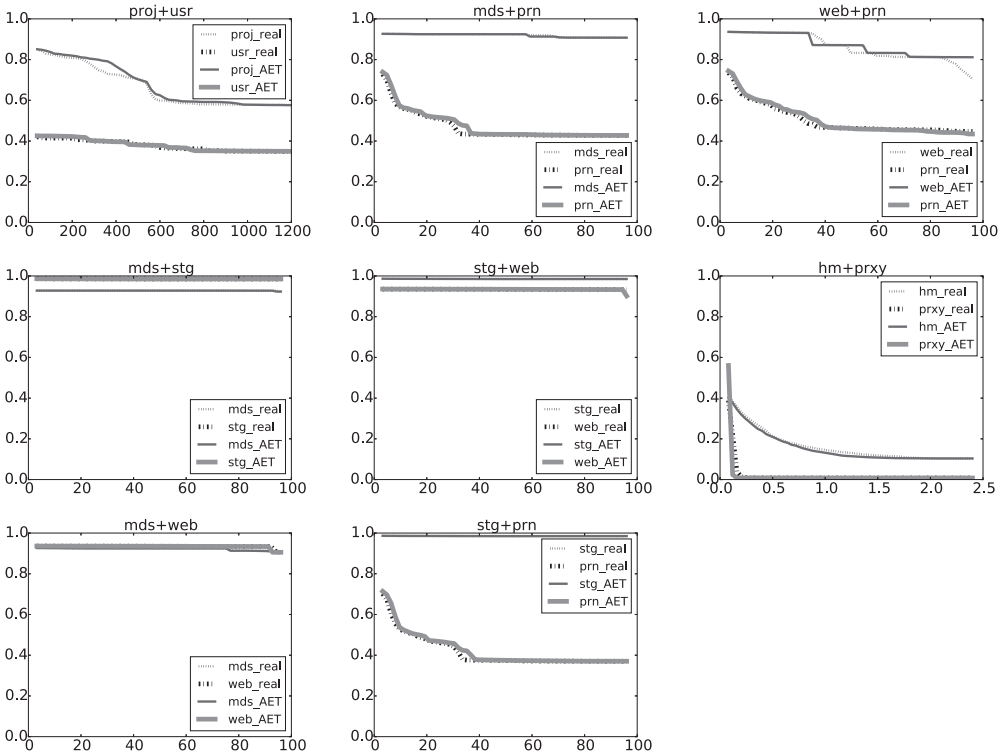


Fig. 17. The predicted miss ratio (y-axis) over cache size (GB, x-axis) by multi-level AET modeling.

formulation, Denning and Schwartz assumed infinite traces generated by a stationary process. Later work applied the Denning-Schwartz formula on finite-length traces to compute the average working-set size (Slutz and Traiger 1974) and LRU stack distance (Denning and Slutz 1978). AET is a new formulation showing that the Denning-Schwartz formula is the solution to AET equations, which are the properties of cache eviction time of all program traces, finite or infinite. Previous work did not address shared cache, which AET can easily model based on eviction-time equalization. Finally, AET is used in sampling analysis of MRC. Sampling was not studied in previous work. However, the previous work modeled arbitrary data size (Slutz and Traiger 1974; Denning and Slutz 1978) and optimal caching policies (Denning and Slutz 1978), which we do not consider in this work.

Table 6. The Space and Time Complexity of MRC Analysis Techniques as Well as Their Memory and Time Consumption Measured in Master Trace

	Time complexity	Space complexity	Memory	Runtime	Composability	Correctness
Stack Processing	$O(NM)$	$O(M)$	10GB	>1 day	No	accurate
Search Tree	$O(N \log M)$	$O(M)$	21GB	482s	No	accurate
Scale Tree	$O(N \log \log M)$	$O(M)$	17GB	333s	No	bounded err
Footprint	$O(N)$	$O(M)$	17GB	50s	Yes	conditional
Counter Stacks	$O(N \log M)$	$O(\log M)$	80MB	1034s	Yes	bounded err
SHARDS	$O(N)$	$O(1)$	2.3MB	29.6s	No	conditional
AET model	$O(N)$	$O(1)$	1.7MB	30.5s	Yes	conditional

We started this article by reviewing the progress of MRC analysis over the past four and half decades. We now give a more comprehensive comparison in Table 6, including the asymptotic complexities, the actual space and time cost (when measuring the merged MSR trace, Section 4.2), the composability (Section 2.5), and correctness properties. We have implemented all listed techniques using our test machine (Section 4) in C++ except Scale Tree.[2] The space cost is the peak resident memory usage at run time. AET uses random and reservoir sampling to reduce space cost in practice. In Table 6, the runtime and space overhead of AET for the merged MSR trace is the lowest among all these techniques.

As a baseline technique, Stack Processing (Mattson et al. 1970) tracks the reuse distance by simulating an LRU stack for the trace. For each access, the algorithm needs $O(M)$ time in the worst case to check all the data in the stack, which is the longest possible reuse distance. Hence, the overall time complexity of this technique is $O(NM)$ with $O(M)$ space cost.

Search Tree (Olken 1981b; Almasi et al. 2002) is an improvement to the Stack Processing technique. It maintains a balanced binary tree to store all the addresses sorted by their age (last access time). For each reference, the number of nodes that have smaller ages than the accessed data is the reuse distance of this reference. It needs a hash table to record the recent access time for every data object. Each access will take $O(logM)$ time for searching and updating the search tree. Therefore, the overall time complexity of the Search Tree technique is $O(NlogM)$ while the space cost remains $O(M)$. Scale Tree (Zhong et al. 2009) is another form of search tree. It combines several data objects in to a single node in the search tree to accelerate the search speed for every access. Since all data objects in the same node are considered to have the same age, the accuracy of reuse distance tracking is bounded by the maximum number of data objects allowed in each node. Scale Tree trades accuracy with smaller tree size and hence yields lower time complexity.

Footprint theory (Xiang et al. 2013) is a technique for linear time MRC profiling. It is a trace-driven algorithm that models the average data set size for any time window. Footprint algorithm tracks the reuse-time distribution like AET model. It also requires an $O(M)$ space to record the first and last access time for every data block in the trace. Footprint analysis takes $O(1)$ time to measure the reuse time of each access, resulting in time complexity of $O(N)$.

As for composability, AET, Footprint, and Counter Stacks are the only three techniques that can model shared cache without co-run testing. Counter Stacks presents joining operation to model merged workload. It can produce MRC for the joined traces by merging the individual time-stamped counter stack streams. As evaluated earlier, counter stack streams cause larger space overhead than AET.

[2]The memory and runtime of scale tree are estimated according to Zhong et al. (2009).

In terms of correctness, Stack Processing and Search Tree measure reuse distance accurately, and Scale Tree guarantees the relative precision. Counter Stacks also guarantees a bounded error based on the correctness of Hyperloglog counters. SHARDS uses sampling, and the result is correct if the sampled accesses are representative. The correctness of Footprint-based MRC is conditional based on the reuse-window hypothesis. The correctness of AET is conditional as discussed in Section 2.3.

MRC Applications. MRC profiling techniques are widely used in different applications. Several studies use on-line MRC analysis for cache partitioning (Suh et al. 2001; Zhang et al. 2009), page size selection (Cascaval et al. 2005), and memory management (Zhou et al. 2004; Kim et al. 1991). The memory cache prediction (Bjornsson et al. 2013) also uses on-line MRC detection for storage workload. In high-throughput storage systems, fast MRC tracking is always beneficial.

Our earlier work used footprint-based MRC to optimize memory allocation in the key-value store called Memcached (Hu et al. 2015). Previous solutions, e.g., those of Facebook and Twitter, were based on heuristics. We showed that MRC-based optimization was superior in steady-state performance, the speed of convergence, and the ability to adapt to request pattern changes. It achieved over 98% of the theoretical potential. The fast MRC analysis was important, since it affects the throughput of Memcached. We used footprint, which was time efficient but consumes a large amount of space (as it is also evident in Table 6). AET sampling should solve the space problem, and it is even faster than footprint.

The reuse-distance based techniques except Counter Stacks in Table 6 are not composable, so they cannot be used in symbiotic optimization. AET is composable, and it can drastically reduce the time and space overhead of shared-cache optimization.

6 SUMMARY

In this work, we present the AET theory, a kinetic model for workload modeling of LRU caches. Using average eviction time (AET) measured by sampling, the AET model consumes linear time and extremely low space for MRC profiling. In storage workloads, AET outperforms Counter Stacks in throughput and space overhead and achieves performance comparable to SHARDS. At last, we show how AET model can be used to characterize shared cache without actual co-run testing. This is an essential strength of AET model. AET models MRC for a cache using the LRU replacement policy. There are other policies, such as FIFO, LIRS (Jiang and Zhang 2002), ARC (Megiddo and Modha 2003), and OPT (Coffman and Denning 1973). A recently published policy, EVA (Beckmann and Sanchez 2017), uses a similar assumption that a single cache line's behavior can be predicted using all cache lines' hit-time distribution and eviction-time distribution. The question of how to extend AET to model MRC for those polices is remained for our future work.

REFERENCES

Arnold O. Allen. 2014. *Probability, Statistics, and Queueing Theory.* Academic Press.

George Almasi, Calin Cascaval, and David A. Padua. 2002. Calculating stack distances efficiently. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance.* Berlin, Germany, 37–43.

Nathan Beckmann and Daniel Sanchez. 2017. Maximizing cache performance under uncertainty. In *Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA'17).* IEEE, 109–120.

Kristof Beyls and Erik H. D'Hollander. 2006. Discovery of locality-improving refactoring by reuse path analysis. In *Proceedings of High Performance Computing and Communications. Springer. Lecture Notes in Computer Science*, Vol. 4208. 220–229.

Hjortur Bjornsson, Gregory Chockler, Trausti Saemundsson, and Ymir Vigfusson. 2013. Dynamic performance profiling of cloud caches. In *Proceedings of the 4th Annual Symposium on Cloud Computing.* ACM, 59.

Jacob Brock, Yechen Li, Chencheng Ye, and Chen Ding. 2015. Optimal cache partition-sharing: Don't ever take a fence down until you know why it was put up—Robert Frost. In *Proceedings of the International Conference on Parallel Processing.*

Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. 2010. SSD bufferpool extensions for database systems. *Proceedings of the VLDB Endowment* 3, 1–2 (2010), 1435–1446.

Calin Cascaval, Evelyn Duesterwald, Peter F. Sweeney, and Robert W. Wisniewski. 2005. Multiple page size modeling and optimization. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*. 339–349.

Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. 2005. Predicting inter-thread cache contention on a chip multiprocessor architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'11)*. IEEE, 340–351.

Zhifeng Chen, Yuanyuan Zhou, and Kai Li. 2003. Eviction-based cache placement for storage caches. In *Proceedings of the USENIX Annual Technical Conference, General Track*. 269–281.

Edward Grady Coffman and Peter J. Denning. 1973. *Operating Systems Theory*. Vol. 973. Prentice-Hall Englewood Cliffs, NJ.

Peter J. Denning. 1968. The working set model for program behavior. *Commun. ACM* 11, 5 (1968), 323–333.

Peter J. Denning. 1980. Working sets past and present. *IEEE Trans. Software Eng.*1 (1980), 64–84.

Peter J. Denning, Craig H. Martell, and Vint Cerf. 2015. *Great Principles of Computing*. MIT Press.

Peter J. Denning and Stuart C. Schwartz. 1972. Properties of the working-set model. *Commun. ACM* 15, 3 (1972), 191–198.

Peter J. Denning and Donald R. Slutz. 1978. Generalized working sets for segment reference strings. *Commun. ACM* 21, 9 (1978), 750–759.

Chen Ding, Xiaoya Xiang, Bin Bao, Hao Luo, Ying-Wei Luo, and Xiao-Lin Wang. 2014. Performance metrics and models for shared cache. *Journal of Computer Science and Technology* 29, 4 (2014), 692–712.

Zachary Drudi, Nicholas J. A. Harvey, Stephen Ingram, Andrew Warfield, and Jake Wires. 2015. Approximating hit rate curves using streaming algorithms. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 40. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

E. Duesterwald, C. Cascaval, and S. Dwarkadas. 2003. Characterizing and predicting program behavior and its variability. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*. New Orleans, Louisiana.

David Eklov and Erik Hagersten. 2010. StatStack: Efficient modeling of LRU caches. In *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS'10)*. IEEE, 55–65.

Éric Fusy, G. Olivier, and Frédéric Meunier. 2007. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *Proceedings of the 2007 International Conference on Analysis of Algorithms (AofA'07)*.

Binny S. Gill. 2008. On multi-level exclusive caching: Offline optimality and why promotions are better than demotions. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*. USENIX Association, 4.

Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. 2015. LAMA: Optimized locality-aware memory allocation for key-value cache. In *Proceedings of USENIX Annual Technical Conference*.

Song Jiang and Xiaodong Zhang. 2002. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Perform. Eval. Rev.* 30, 1 (2002), 31–42.

Yunlian Jiang, Eddy Z. Zhang, Kai Tian, and Xipeng Shen. 2010. Is reuse distance applicable to data locality analysis on chip multiprocessors? In *Compiler Construction*. Springer, 264–282.

Taeho Kgil and Trevor Mudge. 2006. FlashCache: A NAND flash memory file cache for low power web servers. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. ACM, 103–112.

Yul H. Kim, Mark D. Hill, and David A. Wood. 1991. Implementing stack simulation for highly-associative memories. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*. 212–213.

R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. 1970. Evaluation techniques for storage hierarchies. *IBM Syst. J.* 9, 2 (1970), 78–117.

Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A self-tuning, low overhead replacement cache. In *FAST*, Vol. 3. 115–130.

Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. 2008. Write off-loading: Practical power management for enterprise storage. *ACM Trans. Storage (TOS)* 4, 3 (2008), 10.

Frank Olken. 1981a. *Efficient Methods for Calculating the Success Function of Fixed-space Replacement Policies*. Technical Report. Lawrence Berkeley Lab, CA.

F. Olken. 1981b. *Efficient Methods for Calculating the Success Function of Fixed Space Replacement Policies*. Technical Report LBL-12370. Lawrence Berkeley Laboratory.

Derek L. Schuff, Milind Kulkarni, and Vijay S. Pai. 2010. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*. 53–64.

Xipeng Shen, Jonathan Shaw, Brian Meeker, and Chen Ding. 2007. Locality approximation using time. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 55–61.

Donald R. Slutz and Irving L. Traiger. 1974. A note on the calculation working set size. *Commun. ACM* 17, 10 (1974), 563–565. DOI: https://doi.org/10.1145/355620.361167

G. Edward Suh, Srinivas Devadas, and Larry Rudolph. 2001. Analytical cache models with applications to cache partitioning. In *Proceedings of the 15th International Conference on Supercomputing*. ACM, 1–12.

G. Edward Suh, Srinivas Devadas, and Larry Rudolph. 2014. Analytical cache models with applications to cache partitioning. In *Proceedings of the 25th Anniversary International Conference on Supercomputing Anniversary Volume*. ACM, 323–334.

David K. Tam, Reza Azimi, Livio Soares, and Michael Stumm. 2009. RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 121–132.

Jeffrey S. Vitter. 1985. Random sampling with a reservoir. *ACM Trans. Math. Software (TOMS)* 11, 1 (1985), 37–57.

Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. 2017. Cache modeling and optimization using miniature simulations. In *Proceedings of USENIX ATC*. 487–498.

Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. 2015. Efficient MRC construction with SHARDS. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, 95–110.

Xiaolin Wang, Yechen Li, Yingwei Luo, Xiameng Hu, Jacob Brock, Chen Ding, and Zhenlin Wang. 2015. Optimal footprint symbiosis in shared cache. In *CCGRID*.

Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas J. A. Harvey, Andrew Warfield, and Coho Data. 2014. Characterizing storage workloads with counter stacks. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 335–349.

Theodore M. Wong and John Wilkes. 2002. My cache or yours? Making storage more exclusive. In *Proceedings of the USENIX Annual Technical Conference, General Track*. 161–175.

Xiaoya Xiang, Bin Bao, Tongxin Bai, Chen Ding, and Trishul M. Chilimbi. 2011a. All-window profiling and composable models of cache sharing. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 91–102.

Xiaoya Xiang, Bin Bao, Chen Ding, and Yaoqing Gao. 2011b. Linear-time modeling of program working set in shared cache. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*. IEEE, 350–360.

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.

Gala Yadgar, Michael Factor, Kai Li, and Assaf Schuster. 2008. Mc2: Multiple clients on a multilevel cache. In *Proceedings of the 28th International Conference on Distributed Computing Systems (ICDCS'08)*. IEEE, 722–730.

Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. 2009. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European Conference on Computer Systems*. ACM, 89–102.

Yutao Zhong and Wentao Chang. 2008. Sampling-based program locality approximation. In *Proceedings of the International Symposium on Memory Management*. 91–100. DOI: https://doi.org/10.1145/1375634.1375648

Yutao Zhong, Xipeng Shen, and Chen Ding. 2009. Program locality analysis using reuse distance. *ACM Trans. Program. Lang. Syst. (TOPLAS)* 31, 6 (2009), 20.

Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. 2004. Dynamic tracking of page miss ratio curve for memory management. In *ACM SIGOPS Operating Systems Review*, Vol. 38. ACM, 177–188.

Received February 2017; revised January 2018; accepted January 2018