Kinetic Modeling of Data Eviction in Cache

Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, *Peking University;*
Chen Ding, *University of Rochester;* Zhenlin Wang, *Michigan Technological University*

https://www.usenix.org/conference/atc16/technical-sessions/presentation/hu

This paper is included in the Proceedings of the
2016 USENIX Annual Technical Conference (USENIX ATC '16).

June 22–24, 2016 • Denver, CO, USA

978-1-931971-30-0

Open access to the Proceedings of the
2016 USENIX Annual Technical Conference
(USENIX ATC '16) is sponsored by USENIX.

Kinetic Modeling of Data Eviction in Cache

Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo
Peking University

Chen Ding
University of Rochester

Zhenlin Wang
Michigan Technological University

Abstract

The reuse distance (LRU stack distance) is an essential metric for performance prediction and optimization of storage and CPU cache. Over the last 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 paper, 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 low-cost sampling. It can produce the miss ratio curve (MRC) in linear time with extremely low space costs. On both CPU and storage benchmarks, AET reduces the time and space costs compare to former techniques. Furthermore, AET is a composable model that can characterize shared cache behavior through modeling individual programs.

1 Introduction

A 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 [1] where data locality is characterized by working set size (WSS) [2, 3]. 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 (LRU stack distance as defined by Mattson et al. [4]). 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 [5].

For CPU workloads, the recent footprint theory [6], StatStack [7] and time-to-locality conversion [8, 9] use *reuse time* instead of reuse distance to model the workloads. Reuse time is the time between an access and its next reuse. 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 life span may last for weeks or more. Therefore, techniques like the footprint analysis may require too much space. *Counter Stacks* [11] and *SHARDS* [12] are recent breakthroughs to reduce space cost in asymptotic complexity [11] and in practice [12]. Counter Stacks uses probabilistic counters and for the first time can measure reuse distances in sublinear space with a guaranteed accuracy [13]. SHARDS

[1] The working-set theory has a similar effect and same time and space complexity [10, 3]. See Sec. 2.8 of [6] for a comparison.

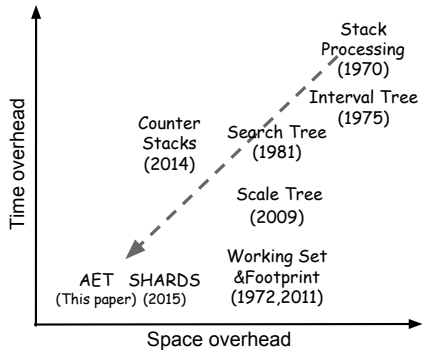

Figure 1: Time and space cost of MRC profiling algorithms.

uses a splay tree to track the reuse distances of sampled data. The time and space consumption is reduced to an extremely low level. However, Counter Stacks and SHARDS cannot characterize shared cache behavior through modeling individual programs.

This paper describes a new kinetic model for MRC construction of LRU caches based on average eviction time (AET). AET runs in linear time asymptotically and uses sampling to minimize the space overhead. In evaluation, AET has the lowest level space and run-time overhead compared to past techniques, for both CPU workloads and storage workloads, 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.

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. Section 2.3 discusses the correctness of the model. Section 2.4 models the shared cache and solves the eviction-time equalization equation.

2.1 LRU Stack and Eviction Time

LRU cache can be logically viewed as a stack [4]. 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 on 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. At an eviction at time t, looking backwards to the most recent time u when the evicted block was referenced, the time interval $t-u$ is the eviction time. Notice that u could also be the time the data block was brought in (a miss). In general, the eviction time is the last segment of the residence time of the data block.

For example, block d in the cache in Figure 2 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.[2]

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 a 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, $T_0 = 0$ and the eviction time is T_c, which is the time the data block leaves position $c-1$. To illustrate, Table 1 shows the arrival time T_m of d for size 4 cache. As m increments from 0 to 3, T_m increases from 0 to 5.

[2]Eviction time is part of the *residence time*, which can be estimated using queuing theory (as "response time", Chapter 9 [14]).

time	ref		rt	hit
0	a	a	∞	0
1	b	b → a	∞	0
2	c	c → b → a	∞	0
3	d	d → c → b → a	∞	0
4	a	a → d → c → b	4	1
5	d	d → a → c → b	2	1
6	a	a → d → c → b	2	1
7	b	b → a → d → c	6	1
8	a	a → b → d → c	2	1
9	c	c → a → b → d	7	1
10	e	e → c → a → b	∞	0
11	d	d → e → c → a	6	0

Figure 2: Example 4-block cache, viewed as a stack, showing logical time, data referenced each time (ref), reuse time (rt) of each access and whether the access is a cache hit. The shaded area is the eviction time of d.

The movement of 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, lower in the stack. We define T_0 to be 0. Obviously, T_1 is al-

Table 1: The kinetic model illustrated by d's eviction in the shaded area in Figure 2. 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 $T_4 = 5$.

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	∞

ways 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. When block z is accessed, and d is at stack position m, d moves down if and only if the (backward) reuse time of z is greater than d's arrival time 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 movement. Block d moves its position (shown in the second row) whenever the reuse time (the last row) is greater than the arrival time (the third row).

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.

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. T_m is the average arrival time for a data block to reach position m (in its eviction process). 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.

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

For an access, $P(t)$ is the probability that its reuse time is greater than t:

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

The movement condition is now a probability. It is actually $P(t)$. This can be interpreted as follows: in a unit time, a data block moves by $P(t)$ position. To use a familiar concept, we call it the *travel speed*. At position m, the average arrival time is T_m, and the travel speed $v(T_m)$ is the probability in logical time:

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

For a given block at each stack position, the moving speed is easy to define: either moving one position at the next access or stay in place (no movement). This travel speed may slow down and then speed up. On average for all evictions, however, the velocity is monotone and non-increasing. By definition, $P(T_m)$ is monotone and non-increasing with T_m. It follows from Eq. 3 that the travel speed at position 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: velocity $v(T_m)$, average arrival time T_m, and cache size c. This connection is shown pictorially in Figure 3.

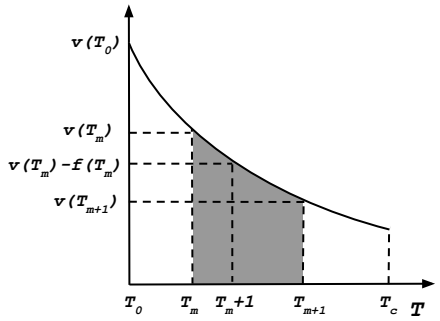

Figure 3: As the average arrival time (T_m) increases along the x-axis, the y-axis shows the travel speed $v(T_m)$ at each T_m. The integral of of v over T gives the movement distance, which is the area under the curve. The shaded area shows the increment of stack position (which is 1).

In Figure 3, the x-axis shows the average arrival time (T_m) as it increases. At each T_m, we use Eq. 3 to compute the travel 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 movement distance, i.e. the stack position it travels 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 travels. From the monotonicity mentioned earlier, the change must be a deceleration. Based on the velocity formula (Eq. 3), the following gives the exact deceleration from T_m to $T_m + \Delta T$.

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

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 travels 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 Eq. 4. Finally, we sum over the spatial increment from 0 to cache size c. The result is the total distance traveled, e.g. the area below the example curve in Figure 3, which is the cache size c when the arrival time reaches T_c.

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

DI is an implicit equation. Its solution, as it turns out, is $AET(c)$. 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 Eq. 5. The equivalence is proved as follows.

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

From AET to MRC Eq. 6 shows that AET calculation takes linear time. The only information it needs is the reuse time histogram (RTH), which gives $P(x)$, and can be measured in linear time. The miss ratio $mr(c)$ at cache size c is the probability that a reuse time is greater than the average eviction time $AET(c)$:

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

During the integration of Eq. 6 from 0 to maximal reuse time, the miss ratios of all cache sizes can be computed in linear time at once.

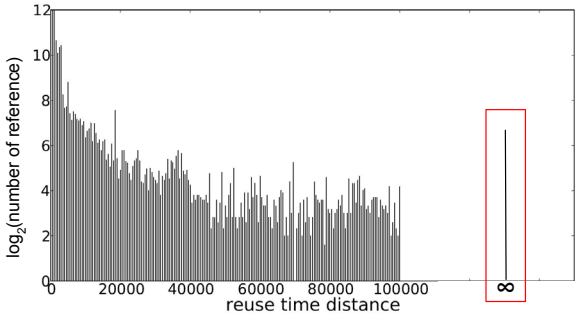

Figure 4: RTH and cold miss example

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 traveled 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:

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

where $rd(x)$ is the number of references with reuse distance x. Therefore, AET is correct if its estimation of reuse distance is correct. Hence:

Correctness Condition. *The AET-based conversions are accurate 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 Eq. 8 but shows the connection mathematically as a function composition.

2.4 AET in Shared Cache

When sharing the cache, a set of co-run programs interact. We want a composable model to derive the composite effect from individual solo-run locality. Ding et al. [15] define the 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?

Cache sharing means that *all co-run programs have the same average eviction time (AET).* For any data block in the shared cache, once it is no longer accessed, its eviction time is the same regardless which program the data block belongs to. Hence we have the equation of eviction-time equalization: *when n programs share the cache of size c, all $n+1$ co-run AETs, $AET_i(c)$ for each program i and $AET(c)$ for the group, are the same*:

$$AET_1(c) = AET_2(c) = \cdots = AET_n(c) = AET(c) \quad (9)$$

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. Let r_{solo} be the access rate, $rt_{solo}(t)$ be the reuse-time histogram, $P_{solo}(t)$ be the probability function, defined as in Section 2.2 for each program. The aggregate access rate is naturally $r_{co} = n\, r_{solo}$. 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 each program, the co-run reuse time $rt_{co}(nt) = rt_{solo}(t)$, or equivalently $rt_{co}(t) = rt_{solo}(t/n)$. Because of the time change, the probability function of each program becomes $P_{co}(t) = P_{solo}(t/n)/n$. The aggregate probability is the sum of the group, $P(t) = \sum_{i=1}^{n} P_{co}(t) = nP_{co}(t)$.

$$P(t) = \sum_{i=1}^{n} P_{co}(t) = \sum_{i=1}^{n} P_{solo}(t/n)/n = P_{solo}(t/n) \quad (10)$$

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

$$\int_0^{AET(c)} P(x)\,\mathrm{d}x = c \quad (11)$$

The equation looks the same as Eq. 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 each program is $mr_{co}(c) = P_{co}(AET(c))$. This contribution is the same from every program, so $mr_{co}(c) = mr(c)/n$. The solutions of the co-run AET and miss ratio for this symmetric case are unique.

Note that the co-run miss ratio $mr_{co}(c)$ is the ratio of the miss count of each program divided by the number of accesses of all programs. In other words, it is the 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 probability function $P_{solo,i}(t)$. Let the total access rate be $r = \sum_{i=1}^{n} r_{solo,i}$. The aggregate $P(t)$ is:

$$P(t) = \sum_{i=1}^{n} P_{co,i}(t) = \sum_{i=1}^{n} P_{solo}(t\frac{r_{solo,i}}{r})\frac{r_{solo,i}}{r} \quad (12)$$

The shared-cache distance-integration equation (Eq. 11) can now compute $AET(c)$ for the general case. The group miss ratio is $mr(c) = P(AET(c))$, and the portion of the miss ratio contributed from program i is $mr_{co,i}(c) = P_{co,i}(AET(c))$. 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.

$$mr(c) = P(AET(c)) = \sum_{i=1}^{n} mr_{co,i}(c) = \sum_{i=1}^{n} P_{co,i}(AET(c)) \quad (13)$$

Composition Invariance The aggregated miss ratio can be computed using AET in two ways: directly using the group $P(t)$ or indirectly as the sum of individual miss ratios. Mathematically, the two results are the same, as shown by Eq. 13. 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 had only one way to compute the group miss ratio [16, 17, 18, 19]. Recent models used footprint and the higher order theory of locality (HOTL) to obtain composition invariance [6, 20, 21]. The model by Brock et al. treated the shared cache as the partitioned cache, where each program is "imagined" to occupy a *natural partition* [21]. AET obtains composition invariance using eviction-time equalization. Unlike the "imagined" natural partition, eviction-time equalization is a real property of the shared cache.

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

In order 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.

Address Sampling The address sampling requires monitoring a fixed subset of the address space. It is known as hold-and-sample and has been used in measuring reuse distance [22, 23, 24, 25] or reuse time [26]. 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 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 address sampling, the fixed interval sampling collects a subset of references instead of a subset of the address space. 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 address 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 fixed interval sampling and address 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 [7].

Reservoir Sampling The space used to store sampled data grows linearly. To bound the space cost, reservoir sampling technique [27] was used by Beyls and D'Hollander [26] for locality analysis. Let the number of entries in the monitoring set (reservoir) be k. When the i-th 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 address 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 Phase Sampling

For programs that have an unstable reference pattern, we evenly divide its execution into phases. For each phase, we use random sampling to construct the RTH and MRC

for this phase. Then we construct the MRC of the entire program. The miss ratio at any cache size is average miss ratio of all MRCs at this size. We call this technique phase sampling.

Phase sampling is used by StatStack [7]. To adapt it for AET sampling, there is one important design change. Not every sampled data 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. In contrast, StatStack uses the forward reuse time. A second and more significant difference with StatStack is the handling of cold misses.

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 four recent techniques: Counter Stacks [11], SHARDS [12], StatStack [7] and adaptive bursty footprint (ABF) sampling [20]. The first two are for storage workloads, while the last two are for CPU workloads.

We use a Dell PowerEdge R720 with ten-core 2.50GHz Intel Xeon E5-2670 v2 processors and 256 GB 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, StatStack, ABF sampling) and reuse distance histogram (Counter Stacks, SHARDS) using the compressed representation by Xiang et al. [19] Each histogram is an array which 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 of data [11]. It uses probabilistic counters to estimate the reuse distances. While other reuse distance measurement techniques consume linear space overhead, the HyperLogLog counter [28] 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. By controlling δ, Counter Stacks can balance between accuracy and number of counters.

We compare AET model with Counter Stacks using the same storage traces released by Microsoft Research Cambridge (MSR) [29], 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 [11], 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 the rate 10^{-4}, and reservoir sampling where the number of entries in the hash table (32-bit address) is limited to 16K. Figure 5 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 2 for comparison.

Table 2 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 of 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 they show much higher throughput (arithmetic average) and much lower space overhead (weighted average) than both methods of Counter Stacks.

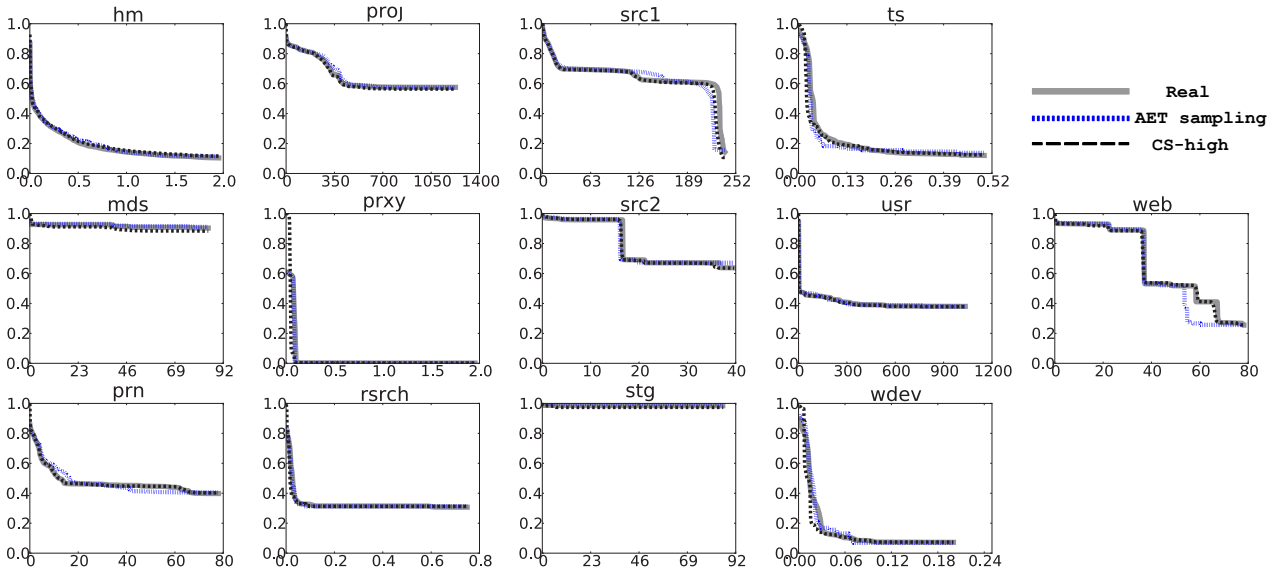

Figure 5: The predicted miss ratio (y-axis) over cache size (GB, x-axis) by AET sampling and Counter Stacks

AET uses reuse time histograms while Counter Stacks uses reuse distance histograms. No matter what compression technique is used for the histogram, the size of both histogram structures should be 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 2, we also list the memory consumed by the hash table and Hyperloglog counters for the 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 (in order 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 2 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.

The correctness of AET-based MRC is based on the assumption of stable distribution reuse distances. This brings inaccuracies to those data that violate the assumption. As we can observe in Figure 5 the AET-based MRC of *web* mispredicts the knee at around 50GB, but Counter Stacks perfectly models every details of the curve, since it makes no assumption about the data 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.

4.2 AET vs SHARDS

SHARDS (Spatially Hashed Approximate Reuse Distance Sampling) is recently developed by Waldspurber et al. [12]. It uses hash-based spatial sampling and a splay tree to track the reuse distances of the sampled data. It

Table 2: The comparison between Counter Stacks (CS) and AET

	WSS (GB)	Prediction Error (%)				Memory (KB)				Throughput (*M*reqs/sec)			
		AET		CS		AET		CS		AET		CS	
		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*	1292*	61.99	63.99	1.73	5.86
sum	2960	-	-	-	-	4992	2196	56066	11232	-	-	-	-

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 [11]), 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 6 shows the MRC profiled by AET random sampling with sampling rate $1 * 10^{-6}$. 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/sec and 81.4M blocks/sec 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 [11]. AET reservoir sampling ($8K$) uses 1.4MB resident memory with 66.6M blocks/sec and 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. Waldspurber et al. [12] 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 [30]. Since

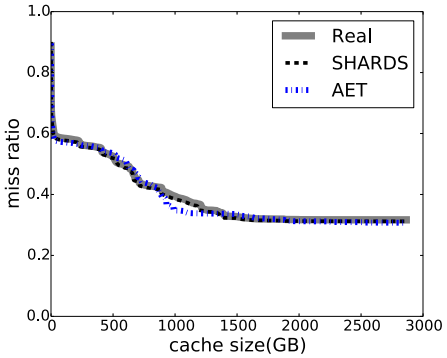

Figure 6: MRCs predicted by AET sampling and SHARDS for the master trace

AET sampling is tied with LRU caches, cache simulations for non-LRU algorithms cannot be done by the current AET model. The strength of AET model is composability, which can be used to model shared cache as we will show in Section 4.6. But this is not a property of current SHARDS and Counter Stacks.

4.3 AET vs StatStack

StatStack is one of the most efficient and accurate methods to approximate MRC for CPU workloads. It samples cache blocks and measures their reuse time using hardware and operating system support such as performance counters and watchpoints [31, 32]. From the reuse time distribution, StatStack estimates the capacity miss ratio and predicts the real miss ratio by adding the estimated

cold miss ratio. We use the SPEC CPU2006 benchmark suite [33] to compare AET and StatStack. For each benchmark, we intercept 1 billion references from their execution using the instrumentation tool Pin [34].

In Figure 7, we show the cumulative distribution function (CDF) of the absolute error for both techniques as well as AET random sampling technique under two sampling rate of 10^{-2} and 10^{-4} (1% and 0.01%). Clearly, the prediction error of full-trace AET is much smaller than StatStack. 90% of the absolute prediction errors are smaller than 0.17%, while only 55% of StatStack's prediction can reach the same level. The average accuracy improvement of full-trace AET against StatStack in this test is 35.8%. Sampling AET is less accurate than full-trace AET but more accurate than full-trace StatStack. 90% of their prediction errors are smaller than 0.21% and the curves are very close to the full-trace AET curve after 90%. AET sampling is repeated 10 times, and its two CDFs record all the errors, not their average. Figure 7 shows that AET sampling produces stable and accurate results.

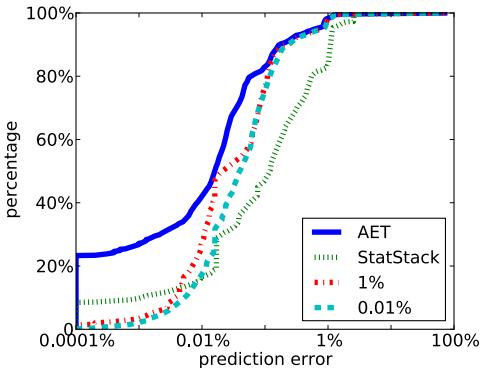

Figure 7: The cumulative distribution function of absolute prediction error

Unlike the AET model using backward reuse time, StatStack uses forward reuse time. It assumes that every reference will have a next reuse. But this is not the case for a trace of a finite length. Every data in the trace has its last reference, and the reuse time of these references are not defined in StatStack. StatStack ignores the impact of these references in its statistical model and characterizes them separately as cold misses. The number of references with no reuse is the same as the working set size. The accuracy of their model is thus influenced by the ratio of the working set size to the trace length.

4.4 Phase Sampling

As mentioned in Section 3.2, phase sampling improves the analysis accuracy for programs with phase behavior. We divide each SPEC CPU2006 benchmark trace into 10 phases of equal length and then use the AET algorithm (full-trace) to profile each phase. Finally, the overall MRC is the average of phase MRCs. In most benchmarks, phase AET sampling is more accurate than non-phase AET sampling. We select four representative benchmarks (`gcc`, `milc`, `wrf`, `hmmer`) and compare phase sampling and non-phase sampling in Figure 8. In these benchmarks, phase analysis leads to significant improvements. The AET models the average eviction time, so it is more accurate when a program shows a steady access behavior. If a program has different phase behavior, we should apply AET analysis on each phase separately as we have done here. More accurate phase analysis may be used to further improve the accuracy of our analysis.

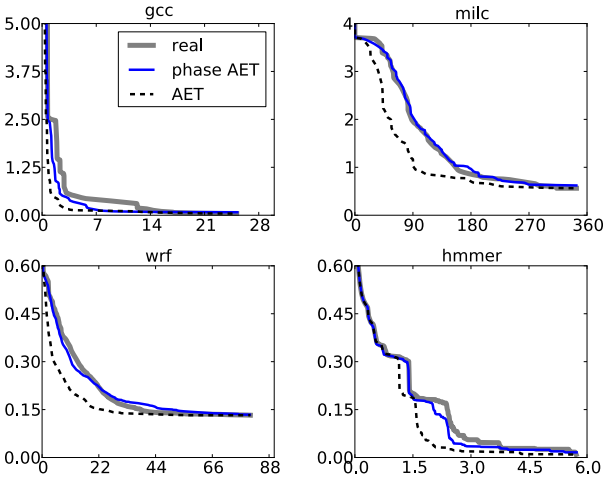

Figure 8: The miss ratio (%) versus cache size (MB) shown for non-phase AET and phase AET

4.5 AET vs ABF Sampling

The footprint-based MRC profiling technique needs recording every access during the monitoring window. The space overhead may be not acceptable for some applications. Wang et al [20]. developed adaptive bursty footprint (ABF) sampling to efficiently measure the footprint of an execution. Extending the design of bursty sampling [35, 36], it approximates the footprint of the entire program by the footprint of small portions. The length of a sampled trace (a burst interval) is bounded by the cache size and minimal miss ratio of interest. The ratio of hibernation and bursty interval is 1000. The miss ratio lower bound is 1%. Therefore, the length of a burst interval was 10^7 to measure 8MB shared cache (131072 cache lines). ABF sampling has several limitations. First, the size of cache is limited by the length of a burst interval. It does not show the MRC for all cache

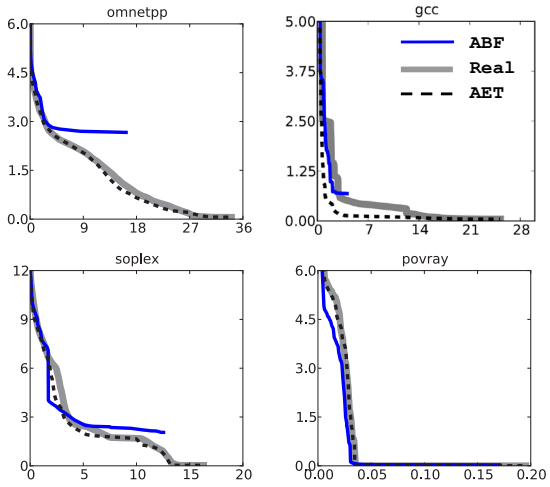

Figure 9: The predicted miss ratio (%) versus cache size (MB) shown for AET sampling and ABF sampling

sizes, unless it measures the complete trace, i.e. no sampling. In comparison, the range of MRC from AET is not influenced by the number of sampled references. Second, an ABF sample is a consecutive portion of a trace. Its result is accurate only if the locality of burst intervals is the same as the locality of the rest (hibernation intervals). In comparison, AET samples cover the entire trace with equal probability.

To evaluate ABF and AET random sampling, we use SPEC CPU2006 benchmarks whose miss ratios are higher than 1%. Due to limitation of space, we only show 4 MRCs (Figure 9) profiled by both techniques with the same sampling rate (1:100). The MRCs of ABF sampling are much shorter than AET sampling, because using bursty interval to represent the entire trace will lose the reference pattern in the hibernation interval. The approximate MRCs of ABF sampling are not as accurate as AET sampling.

4.6 Shared Cache AET

As discussed in Section 2.4, AET is a composable metric and can model shared cache. With the individual AETs of co-run programs, we can predict the MRC of the shared cache they are running on. This technique is essential in task scheduling in a system where shared cache (CPU or storage cache) is deployed. To verify our shared AET modeling technique, we choose four MSR storage traces {prn, src2, web, stg} as a co-run group. They are the traces with the same order of magnitude on length and show totally different patterns in cache usage (see individual MRCs in Figure 10). We assume symmetrical speeds, i.e. equal access rates, to simplify the evaluation, but the extension to asymmetrical cases are straightforward as we showed in Equation 12.

We set the execution length of each trace to be $1.6 * 10^7$, which is the shortest length in the group. With the equal-speed assumption, we generate a combined trace from the four traces. Figure 10 shows the shared cache MRC composed by individual AET modeling of each trace, as well as the real MRC calculated by accurate reuse distance tracking for the combined trace. The shared AET MRC gives a MAE of 0.002, indicating that the shared cache modeling by AET is accurate. The composability of AET is a key advantage over SHARDS and Counter Stacks since these techniques cannot characterize shared cache without co-run testing.

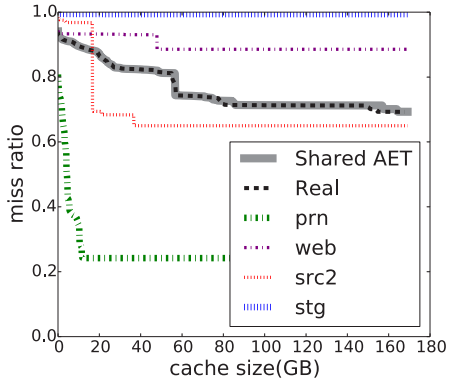

Figure 10: Shared cache MRCs for the combined trace of {prn, src2, web, stg}, as well as the MRCs of four individual traces.

5 Related Work

In 1972, Denning and Schwartz [10] 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 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 [37] and LRU stack distance [3]. 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 [37, 3] and

Table 3: 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	$O(NM)$	$O(N)$	10GB	>1 day	No	accurate
Search Tree	$O(N\log M)$	$O(M)$	21GB	482 secs	No	accurate
Scale Tree	$O(N\log\log M)$	$O(M)$	17GB	333 secs	No	bounded err
Footprint	$O(N)$	$O(M)$	17GB	50 secs	Yes	conditional
Counter Stacks	$O(N\log M)$	$O(\log M)$	80MB	1034 secs	No	bounded err
SHARDS	$O(N)$	$O(1)$	2.3MB	29.6 secs	No	conditional
AET model	$O(N)$	$O(1)$	1.7MB	30.5 secs	Yes	conditional

optimal caching policies [3], which we do not consider in this work.

We have started this paper by reviewing the progress of MRC analysis over the last four and half decades. We now give a more comprehensive comparison in Table 3, including the asymptotic complexities, the actual space and time cost (when measuring the merged MSR trace, Section 4.2), the composability (Section 2.4) and correctness properties. AET uses random and reservoir sampling to reduce space cost in practice. In Table 3, the runtime and space overhead of AET for the merged MSR trace is the lowest among all these techniques.

In terms of correctness, Stack Processing [4] and search tree [5, 38] measure reuse distance accurately, and scale tree [39] guarantees the relative precision. Counter Stacks also guarantee an 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 [6]. 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 [40, 41], page size selection [25], and memory management [42, 43]. The memory cache prediction [44] 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 [45]. 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 3). AET sampling should solve the space problem, and it is even faster than footprint.

Fast MRC helps CPU cache optimization. We have developed and evaluated shared cache program symbiosis, which used ABF sampling and footprint composition to co-locate co-run applications to minimize their interference in shared cache [20]. The reuse-distance based techniques in Table 3 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 liner time and extremely low space for MRC profiling. In our storage workload evaluation AET outperforms Counter Stacks in throughput and space overhead and achieves a comparable performance as SHARDS. At last, We show how AET model can be used to characterize shared cache without co-run testing and with composition invariance.

7 Acknowledgments

The authors wish to thank to Nick Harvey, Carl Waldspurger, Peter Denning, Nohhyun Park, Alexander Garthwaite, Irfan Ahmad and Nisha Talagala for extremely valuable and constructive suggestions for this work and its presentation. This research is supported in part by the National Science Foundation of China (No. 61232008, 61272158, 61328201 and 61472008); the 863 Program of China under Grant No.2015AA015305; the National Science Foundation (No. CNS-1319617, CSR-1422342); a CAS fellowship from IBM and a grant from Huawei.

References

[1] Peter J Denning. Working sets past and present. *Software Engineering, IEEE Transactions on*, (1):64–84, 1980.

[2] Peter J Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[3] Peter J Denning and Donald R Slutz. Generalized working sets for segment reference strings. *Communications of the ACM*, 21(9):750–759, 1978.

[4] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.

[5] Frank Olken. Efficient methods for calculating the success function of fixed-space replacement policies. Technical report, Lawrence Berkeley Lab., CA (USA), 1981.

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

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

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

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

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

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

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

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

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

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

[16] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multiprocessor architecture. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 340–351. IEEE, 2005.

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

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

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

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

[21] Jacob Brock, Yechen Li, Chencheng Ye, and Chen Ding. Optimal cache partition-sharing : Dont ever take a fence down until you know why it was put up. robert frost. In *International Conference on Parallel Processing*, 2015.

[22] Yutao Zhong and Wentao Chang. Sampling-based program locality approximation. In *Proceedings of the International Symposium on Memory Management*, pages 91–100, 2008.

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

[24] David K. Tam, Reza Azimi, Livio Soares, and Michael Stumm. 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*, pages 121–132, 2009.

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

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

[27] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[28] Éric Fusy, G Olivier, and Frédéric Meunier. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *In AofA07: Proceedings of the 2007 International Conference on Analysis of Algorithms*, 2007.

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

[30] Nimrod Megiddo and Dharmendra S Modha. Arc: A self-tuning, low overhead replacement cache. In *FAST*, volume 3, pages 115–130, 2003.

[31] Erik Berg and Erik Hagersten. Fast data-locality profiling of native execution. In *ACM SIGMETRICS Performance Evaluation Review*, volume 33, pages 169–180. ACM, 2005.

[32] Erik Berg and Erik Hagersten. Statcache: a probabilistic approach to efficient and accurate data locality analysis. In *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on-ISPASS*, pages 20–27. IEEE, 2004.

[33] cpu2006. https://www.spec.org/cpu2006/, 2015. [Online].

[34] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, 2005.

[35] Matthew Arnold and Barbara G Ryder. A framework for reducing the cost of instrumented code. *ACM SIGPLAN Notices*, 36(5):168–179, 2001.

[36] Michael D Bond, Katherine E Coons, and Kathryn S McKinley. Pacer: proportional detection of data races. *ACM SIGPLAN Notices*, 45(6):255–268, 2010.

[37] Donald R. Slutz and Irving L. Traiger. A note on the calculation working set size. *Communications of the ACM*.

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

[39] Yutao Zhong, Xipeng Shen, and Chen Ding. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(6):20, 2009.

[40] G Edward Suh, Srinivas Devadas, and Larry Rudolph. Analytical cache models with applications to cache partitioning. In *Proceedings of the 15th international conference on Supercomputing*, pages 1–12. ACM, 2001.

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

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

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

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

[45] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. Lama: Optimized locality-aware memory allocation for key-value cache. In *Proceedings of USENIX ATC*, 2015.