Beating OPT with Statistical Clairvoyance and Variable Size Caching

Pengcheng Li, Colin Pronovost, Benjamin Tait, William Wilson, Jie Zhou
Chen Ding, and John Criswell
University of Rochester, USA

Abstract
Caching techniques are widely used in today’s computing infrastructure from virtual memory management to server cache and memory cache. This paper builds on two observations. First, the space utilization in cache can be improved by varying the cache size based on the dynamic application demand. Second, it is easier to predict application behavior statistically than precisely. The paper presents a new, variable-size cache that makes optimal use of statistical knowledge of program behavior. It measures performance using data access traces from real-world workloads, including Memcached traces from Facebook, and storage traces from Microsoft Research. In an offline setting, the new cache is demonstrated to outperform even OPT, the optimal fixed-size cache which makes use of precise knowledge of program behavior.

1. Introduction
On modern computer systems, memory has become often the largest factor in cost, power and energy consumption. Significant amounts of memory are used as software managed caches for data on persistent storage or data stored on remote systems. Examples include caching done by operating system kernel file system code [26], memory caches such as Memcached [18], and caching performed by network file servers [26]. Cache management has been extensively studied, leading to many effective techniques.

For fixed size caches, the optimal solution is known as Belady, MIN [5], B0 [11], or OPT [25]. It has set the goal for decades of improvements in research and practice, including a number of recent techniques closely modeling or mimicking OPT [9, 23, 41]. While the past work tries to achieve OPT, this paper aims to improve beyond OPT in two ways.

First, OPT [5] is optimal only for fixed-size caches. The working set of an application is not a constant size and may benefit from a temporary increase of cache space. It has long been a principle of virtual memory management that when memory is shared among multiple applications, a variable-size allocation is more effective than a constant-size partition. This paper studies how much variable-size cache can outperform OPT while using the same amount of cache on average.

Second, OPT requires precise knowledge of future data reuse [5]. It is usually impossible to precisely predict when a data item will be reused in the future. However, it is often possible to have a probabilistic prediction on when a data item will be accessed again. Let data item a be reused 4 times in axxaaxxa, OPT requires knowing the next reuse each time a is accessed [5]. A probabilistic prediction states that the reuse time is 1 for half of the a accesses and 3 for the other half. This paper studies the optimal caching performance that can be achieved when using statistical, rather than precise, knowledge of future accesses.

This paper makes three main contributions. First, it presents the Optimal Steady-state Lease (OSL) algorithm, a variable-size caching algorithm which utilizes statistical information about future program behavior. We show that OSL is optimal for any algorithm that has only statistical information about future memory access and provides a reachable bound on such algorithms. We also show that OSL has asymptotically lower complexity than OPT in both time and space. Second, it describes a space-efficient implementation of OSL named Space Efficient Approximate Lease (SEAL) and evaluates its space and time complexity. Finally, it evaluates the proposed implementation of OSL against existing caching solutions on data access traces from real-world workloads. These workloads include Memcached traces from Facebook [3] and traces of network file system traffic from Microsoft Research [28].

The new technique is presented in Section 2, including the lease definition in Section 2.1, the OSL algorithm and its properties in Section 2.2, and efficient lease-cache implementation in Section 2.3. Section 3 evaluates OSL using traces from real-world workloads, Section 4 presents related work, and Section 5 concludes.

2. Caching Using Probability Prediction
2.1 Managing Cache by Leases
In this paper, the cache is controlled by leases. This section presents the interface and performance of such a cache.
Lease Cache  At each access, the cache assigns a lease to the data block being accessed, and the data block is cached for the length of the lease and evicted immediately when the lease expires. We call this type of cache the lease cache. In this paper, the lease is measured by logical time, i.e. the number of data accesses. A lease of \( x \) means to keep the data block in cache for the next \( x \) accesses.

Miss Ratio  Given a data access trace, the forward reuse time is defined for each access as the number of accesses between this current access and the next access to the same data block [7]. In this paper, we called it reuse time in short. If the access is the last to a data block, the reuse time is infinite.

At each access, the next reuse is a cache hit if the lease extends to the next access. Otherwise, it is a miss. The cache hit ratio is the portion of the accesses whose lease is no shorter than its reuse time.

Average Cache Size  The cache does not have a constant size. Instead, we compute the average cache size, which is the average number of data blocks in the cache at each access. Following the past work, e.g. [15], we consider cache usage as a time-space product. A lease \( x \) means allocating one cache block for \( x \) accesses. The sum of all leases is the total time-space consumption, and the average cache size is computed by time-space divided by time, i.e. the total lease divided by the number of accesses. The average cache size is the average number of leases active at each access.

To compute the total lease, we must include the effect of lease fitting, which happens at every cache hit: when a data block in the cache is accessed, the remaining lease is canceled and replaced by the lease of the current access.

An Example  Consider an example infinite-long trace \( abcabc \ldots \). The reuse time is 3 at all accesses. If we assign the unit lease at each access, the miss ratio is 100%, and the cache size is 1. If we increase each lease to 3, the miss ratio drops to 0%, and the cache size is 3. If we increase each lease to 4, the cache size is still 3, not 4, due to lease fitting.

Prescriptive vs Reactive Caching  With leases, cache management is prescriptive. The eviction time of a data block is prescribed each time it is accessed. In contrast, traditional cache management is reactive. In LRU cache, the data block is evicted by another data block. Prescriptive caching manages space by allocation, while reactive caching by replacement.

Locality in computing is characterized by Denning as computing in a series of phases where each accesses a different set of data [13]. Each data is accessed often only in its phases, separated by long periods of no access. If we collect statistics, we will see most data reuses with short reuses and a few long reuses. Prescriptive caching makes use of such statistics and keeps data in cache in phases where it is accessed but not in cache in-between these phases.

Next, we show how to optimize prescriptive caching based on statistical predictions (next section), with asymptotically lower time and space cost than optimal reactive caching (Section 2.2.5).

2.2 Lease Optimization by OSL

This section presents the optimal assignment algorithm: given the reuse times of a data block, it assigns an optimal per-block lease, i.e. the best lease used every time the data block is accessed. We call it the Optimal Steady-state Lease (OSL). The section describes the steady-state condition, the algorithm, and then its optimality.

2.2.1 Steady State

In a steady state, a program accesses \( m \) data blocks. Each data block \( i \) is accessed with a probability distribution \( P_i'(rt = x) \), where \( rt = x \) denotes that the reuse time is \( x \) such that \( x \geq 0 \). In the steady state, \( P_i'(rt = x) \) does not change. In the following description, we take the list of memory accesses of a complete program that is in a steady state. Each data block \( i \) is accessed \( f_i \) times, and the probability \( P_i'(rt = x) \) is the ratio of \( f_i \) to its total number of accesses.

2.2.2 Intuition and Illustration

The key metric OSL uses is profit per unit of cost (PPUC). In PPUC, the profit is a number of hits, and the cost is the amount of cache occupied over a time. Hence, PPUC is measured in the number of hits per unit of lease. If we simplify, assume initial lease 0, and do not consider lease fitting, the PPUC of assigning lease \( y \) to data \( i \) is computed as \( \frac{f_iP_i'(rt \leq y)}{f_i} = P_i'(rt \leq y)/y \), where \( f_i \) is the access frequency of data \( i \), and \( P_i' \) is the reuse probability.\(^2\)

OSL is an iterative, greedy algorithm. It first initializes all leases for all blocks to zero. In each step, it computes the PPUC of all reuse times of all data blocks using the existing leases that it has previously assigned and chooses the reuse time and data block that has the highest PPUC. OSL assigns this reuse time as the lease for all accesses of this data block. OSL repeats this computation until either the assigned leases reach the target average cache size or all data blocks have their longest reuse time as their assigned lease.

Figure 1 shows an example. It shows the reuse time histogram of two data blocks, \( A \) and \( B \). Each reuse time is paired with the number of accesses that have this reuse time; this can be represented as a pair of integers in which the first integer is the reuse time and the second integer is the number of data accesses with that reuse time. For example, \( A \)'s

\(^2\)The cost with lease fitting is computed in COST in Algorithm 1. OSL assigns leases iteratively and may increase the lease from \( y \) to \( y' \). Then the PPUC of \( y' \) is based on the change from \( y \) to \( y' \), not just \( y' \). This is computed by \( \text{getPPUC} \) in Algorithm 1, which would compute the PPUC of \( y \) by computing the change from 0 to \( y \).
Figure 1: Illustration of OSL. The example has two data blocks A and B. A has four reuse times, and B has three. OSL assigns leases iteratively: first 5 to all A accesses, then an increase to 19 for A, and finally 12 for B. At each step, OSL first selects the reuse time of data block \( b \) that has the greatest PPUC, boxed in blue, and then re-computes the affected PPUCs.

### Algorithm 1 PPUC Algorithm

**Require:**
- \( M \) \( \triangleright \) Total number of data blocks
- \( R \) \( \triangleright \) Max. distinct reuse times per block
- \( \text{RT}[1 \ldots M][1 \ldots R] \) \( \triangleright \) Reuse-time histograms

1. **function** \( \text{HITS}(\text{block}, \text{maxReuseTime}) \)
   2. return \( \sum_{i=0}^{\text{maxReuseTime}} \text{RT}[\text{block}][i] \)
   3. **end function**

4. **function** \( \text{COST}(\text{block}, \text{lease}) \)
   5. return \( \sum_{i=0}^{\text{lease}} i \times \text{RT}[\text{block}][i] \)
   6. + \( \sum_{i=\text{lease}+1}^{R} \text{lease} \times \text{RT}[\text{block}][i] \)
   7. **end function**

8. **function** \( \text{GETPPUC}(\text{block}, \text{oldLease}, \text{newLease}) \)
   9. return \( \frac{\text{HITS}(\text{block}, \text{newLease}) - \text{HITS}(\text{block}, \text{oldLease})}{\text{COST}(\text{block}, \text{newLease}) - \text{COST}(\text{block}, \text{oldLease})} \)
10. **end function**

### Algorithm 2 The OSL Algorithm

**Require:**
- \( M \) \( \triangleright \) Total number of data blocks
- \( R \) \( \triangleright \) Max. distinct reuse times per block
- \( \text{RT}[1 \ldots M][1 \ldots R] \) \( \triangleright \) Reuse-time histograms
- \( C \) \( \triangleright \) Target cache size

1. **function** \( \text{MAXPPUC} \)
2. \( \text{bestPPUC} \leftarrow 0 \)
3. \( \text{bestBlock} \leftarrow (\text{true}, 0, 0) \)
4. **for** all \( \text{block} \) **do**
5. **for** all \( i \) such that \( 0 < i \leq R \) **do**
6. \( \text{ppuc} \leftarrow \text{GETPPUC}(\text{block}, \text{L}[\text{block}], \text{RT}[\text{block}][i]) \)
7. if \( \text{ppuc} > \text{bestPPUC} \) **then**
   8. \( \text{bestPPUC} \leftarrow \text{ppuc} \)
   9. \( \text{bestBlock} \leftarrow (\text{false}, \text{block}, \text{RT}[\text{block}][i]) \)
10. **end if**
11. **end if**
12. **end for**
13. **end for**
14. **return** \( \text{bestBlock} \)
15. **end function**

16. **function** \( \text{MAIN} \)
17. **for** all \( \text{block} \) **do**
18. \( \text{L}[\text{block}] \leftarrow 0 \)
19. **end for**
20. \( \text{totalCost} \leftarrow 0 \)
21. \( \text{targetCost} \leftarrow C \times N \)
22. **while** \( \text{totalCost} \leq \text{targetCost} \) **do**
23. \( \text{finished} \leftarrow \text{MAXPPUC} \)
24. **if** \( \text{finished} = \text{false} \) **then**
25. \( \text{oldLease} \leftarrow \text{L}[\text{block}] \)
26. \( \text{cost} \leftarrow \text{COST}(\text{block}, \text{newLease}) \)
27. \( \text{cost} \leftarrow \text{cost} - \text{COST}(\text{block}, \text{oldLease}) \)
28. \( \text{totalCost} \leftarrow \text{totalCost} + \text{cost} \)
29. \( \text{L}[\text{block}] \leftarrow \text{newLease} \)
30. **else**
31. **break**
32. **end if**
33. **end while**
34. **end function**

Histogram has 4 pairs, (3,1), (5,7), (17,4), and (19,3), for a total of 4 reuse times and 15 accesses. B’s histogram has 3 pairs: (12,6), (24,1), and (64,1).

OSL first initializes the leases of all data blocks to 0. OSL then computes the PPUC for all reuse times of A and B. Take, for example, the reuse time 5 of A. If the lease for all 15 accesses of A is 5, we have 8 hits. The total lease is roughly 5 × 15, but actually 73 (considering lease fitting, the actual total lease is 3 × 15 × (7 + 4 + 3) = 73). The PPUC is the ratio of these two numbers, 8/73 = 0.11. Similarly, OSL computes the PPUC of all reuse times for all data blocks; they are shown in Figure 1(1).

Initially, the lease of A and B is 0. In the first step, OSL selects the greatest PPUC, which is 0.11 at reuse time 5 of block A. OSL assigns the lease 5 to all 15 accesses of A. Following the lease assignment, OSL updates the PPUCs for reuse times of A that are greater than 5. The updated values
in Figure 1(2) show PPUCs when the lease is increased from 5.

In the second step, OSL repeats the greedy selection and (re-)assigns the lease 19 to all 15 accesses of A. There is no further update since A has no greater reuse time than 19. In the third step, OSL assigns the lease 12 to all 8 accesses of B.

### 2.2.3 Lease Assignment

Algorithm 1 shows the algorithm for computing PPUC, and Algorithm 2 shows the OSL algorithm. The inputs to OSL are \( M \), the total number of data blocks; \( N \), the total number of accesses; \( RT \), the reuse-time histograms of all data blocks; and \( C \), the target cache size. OSL computes the optimal leases for all data blocks that achieve target average cache size \( C \).

The main loop at line 24 keeps calling `maxPPUC` to assign leases until one of two conditions is met. The first condition is when the cache size reaches the target cost, i.e. total space-time computed by the target cache size times the trace length \( N \). The second condition is when `maxPPUC` returns `true` as the first element of the tuple it returns, indicating that there are no more leases to assign, and the cache is already at the maximum possible size (with only cold-start misses).

`maxPPUC` computes the PPUC for each reuse time of each data block given the leases assigned in the last iteration of the loop at line 24. For each block, the old lease is stored in \( L \) (initialized to 0 at line 20). Each greater reuse time is a candidate lease.\(^3\) To select the best lease, `maxPPUC` calls `getPPUC` at line 7 in Algorithm 1, which calculates the PPUC as the increase in hits divided by the increase in cost (in cache space), where the hits and cost are calculated by functions `HITS` and `COST`, respectively. The nested loop in `maxPPUC` selects the candidate lease and candidate block with the highest PPUC in line 8 of Algorithm 2. `maxPPUC` returns a tuple of the candidate lease and candidate block to `main` which assigns the lease to the candidate block in line 30.

The nested loop in `maxPPUC` computes PPUC after each lease assignment. This is needed because the same data block may be assigned multiple times with increasingly longer leases. Each assignment necessitates recomputing the PPUCs because they are based on the old lease, which has just been changed. In the example in Figure 1, we can see that the PPUC of two reuse times, 17 and 19, is changed after the first lease assignment.

`maxPPUC` can be made faster. For each block, the PPUC is changed only if its lease is changed. Instead of a nested loop, only the block that was just assigned a new lease requires updating. In an actual implementation, we store all lease candidates in a heap, update the PPUC for only those affected candidates after each assignment, and select the best candidate for the next assignment by a heap pop. Let \( M \) be the number of data blocks, and \( R \) the maximal number of distinct reuse times per data block. The nested loop takes \( O(RM) \) per step, but a heap-based implementation takes only \( O(R) \) per step.

### 2.2.4 Optimality

By choosing the maximal PPUC, OSL attempts to maximize the profit at each step. A problem, however, is that OSL recomputes PPUCs after each assignment, so it may be questioned whether a greater PPUC may appear later. The following theorem rules out such possibility.

**Theorem 1.** (Monotone PPUC) In Algorithm 2, let \( x \) be the PPUC selected at any step and \( x' \) be the PPUC selected at the next step, then \( x \geq x' \).

**Proof.** We first consider the case involving two data blocks. Let OSL find the largest PPUC \( x \) at data block \( b \) in the current step and the largest PPUC \( x' \) at a different data block \( b' \) in the next step. After the current step, only the PPUCs of \( b \) are updated, and the PPUCs of \( b' \) stay the same. It is obvious \( x \geq x' \); otherwise, OSL would have selected \( x' \) instead of \( x \) in the current step.

We next consider when OSL finds the largest PPUCs at the same data block \( d \) in both steps. Let the reuse times of \( d \) in ascending order be \( rt_1, rt_2, \ldots, rt_k \). Assume that OSL selects the largest PPUC \( x \) at reuse time \( rt_j = y \) in the current step. Then OSL recomputes the PPUC for reuse times \( rt_{j+1}, rt_{j+2}, \ldots, rt_k \). After this update, OSL selects the largest PPUC \( x' \) at reuse time \( rt_j' = y' \) in the next step. We prove \( x > x' \) by contradiction. Note that the inequality to be established is actually strict, not just \( x \geq x' \), but \( x > x' \).

Assume the opposite, i.e. \( x' \geq x \). Let \( x_{old}' \) be the PPUC at reuse time \( y' \) before the update (\( x' \) is the PPUC after the update). Because the assignment chooses reuse time \( y \) over \( y' \), we have \( x > x_{old}' \). Combining the two inequalities yields \( x \geq x > x_{old}' \). By substituting the formula for computing PPUC, we have

\[
\frac{P(rt \leq y') - P(rt \leq y)}{y' - y} > \frac{P(rt \leq y)}{y} > \frac{P(rt \leq y')}{y'}
\]  

(1)

We now show that the first inequality contradicts the second inequality. Rewriting the first inequality, we have

\[
\frac{P(rt \leq y)}{y} \left( \frac{P(rt \leq y')}{y} - 1 \right) > \frac{P(rt \leq y)}{y}.
\]

Since \( \frac{P(rt \leq y)}{y} > 0 \), we can remove it from both sides. By re-arranging the terms, we have \( \frac{P(rt \leq y)}{y} \leq \frac{P(rt \leq y')}{y'} \), which is the opposite of the second inequality. The two inequalities contradict; therefore, the assumption \( x' \geq x \) is wrong, and \( x > x' \).

---

\(^3\)The optimal lease must be equal to one of the reuse times. If an optimal lease were larger than one of the reuse times, we could reduce it to the closest reuse time and not incur more misses.
A careful reader may point out two problems in the preceding derivation. First, it assumes that \( y \) and \( y' \) are the first two lease assignments of data block \( d \). In the general case, a previous lease \( y'' \) has been assigned before \( y \). This implies two changes to the PPUC calculation: removing \( P(rt \leq y'') \) from the numerator and \( y'' \) from the denominator. Since the changes happen on every fraction in Eq. 1, the same proof applies. Still there is a second problem. The cost of the lease did not consider lease fitting. Instead of \( y \) and \( y' \) in the denominators, they should be a cost function \( c(y) \) and \( c(y') \). The derivation requires only that \( c(y') > c(y) > 0 \), which is the case with lease fitting. Therefore, the same proof applies, and the two inequalities still contradict each other after generalizing the lease assignment and cost.

Combining the (generalized) second case and the first case, we have that \( x \geq x' \) always, that is, the PPUC selected by OSL is monotone. \( \square \)

The PPUC monotonicity means that OSL maximizes its profit at each step. Informally speaking, OSL assigns leases in decreasing importance. This suggests convexity in OSL performance. As the cache size increases, the performance increases less from the same amount of additional cache space.

Next we show that OSL makes optimal use of the per-data-block reuse time distribution.\(^4\)

**Corollary 1.** (Statistical Optimality) No algorithm which determines the lease for a particular access using only the per-data-block reuse time histogram can have a lower miss ratio than OSL.

**Proof.** Since the lease for an access is determined solely from the data block being accessed and the per-data-block reuse time histogram, the algorithm must assign the same lease to all accesses of any particular data item (assuming the algorithm is deterministic).

Let \( m \) be the number of distinct data blocks in a trace. Let \( L[1..m] \) be the leases that OSL assigns to each data block, and let \( L'[1..m] \) be an arbitrary assignment of leases to data blocks that achieves the same average cache size as \( L \). If there is some \( i \) such that \( L[i] < L'[i] \), there must also be a \( j \) such that \( L[j] > L'[j] \), otherwise \( L \) and \( L' \) couldn’t achieve the same average cache size. We know that \[
\frac{\text{hits}(i, L'[i]) - \text{hits}(i, L[i])}{\text{cost}(i, L'[i]) - \text{cost}(i, L[i])} < \frac{\text{hits}(j, L'[j]) - \text{hits}(j, L[j])}{\text{cost}(j, L'[j]) - \text{cost}(j, L[j])}
\]
otherwise OSL would have assigned \( L'[i] \) to data block \( i \) in lieu of assigning \( L[i] \) to data block \( j \). However, since \( L \) and \( L' \) achieve the same target cache size, we also know that \[
\text{cost}(i, L'[i]) - \text{cost}(i, L[i]) = \text{cost}(j, L[j]) - \text{cost}(j, L'[j])
\]
Therefore \[
\text{hits}(i, L'[i]) - \text{hits}(i, L[i]) < \text{hits}(j, L[j]) - \text{hits}(j, L'[j])
\] This means that \( L' \) must have fewer hits than \( L \). Therefore \( L \) must achieve the best possible miss ratio of all possible lease assignments with the same average cache size. \( \square \)

### 2.2.5 Complexity

The algorithm complexity is as follows. Let the total number of blocks be \( M \), and the maximal number of distinct reuse times per data block be \( R \). The number of lease candidates is at most \( MR \). At each assignment in OSL, at most \( R \) candidates are updated. Assuming a binomial heap is used, the maximization time is \( O(\log(MR)) \) = \( O(\log M + \log R) \). The total cost per lease assignment is \( O(\log M + \log R + R) = O(\log M + R) \). The number of assignments is at most \( MR \) (for the largest cache size). Overall, OSL takes \( O(MR(\log M + R)) \) in time. The space cost is \( O(MR) \).

If we approximate and use a histogram with logarithmic size bins, \( R = O(\log N) \), where \( N \) is the trace length, and \( N - 1 \) the longest possible reuse time. The time cost is \( O(M\log N(\log M + \log N)) \). Since \( M \leq N \), it equals to \( O(M\log N(\log N + \log N)) = O(M\log^2 N) \). The space cost is \( O(M\log N) \).

**Optimization Complexity: OSL vs OPT** OPT requires precise knowledge, meaning the reuse time for each access, so its space cost is \( O(N) \). OPT can be implemented by stack simulation, requiring \( O(M) \) space and \( O(M) \) operations at each assignment to maintain a constant cache size [25]. The time cost is therefore \( O(MN) \). In comparison, by using statistical clairvoyance, OSL reduces the space cost from \( O(N) \) to \( O(M\log N) \). By targeting an average cache size, instead of maintaining a constant cache size, OSL reduces the optimization cost from \( O(MN) \) to \( O(M\log^2 N) \).

### 2.2.6 Generalization

OSL assigns an optimal lease for a group of accesses. In the presentation so far, accesses are grouped by data identity, i.e. all accesses to same data block. OSL can be used in any type of grouping. It may group by program code, i.e. accesses by the same load/store instruction, the same function, or the same data structure.

In general, OSL divides all data accesses into a set of groups, feeds their reuse-time statistics and other parameters such as cache size to Algorithm 2, and computes the optimal lease for each group. This lease is then the lease for every access of that group. The optimality and complexity results are unchanged — OSL provides optimal caching at the time cost of \( O(G\log^2 N) \) and space cost of \( O(G\log N) \), where \( G \) is the number of groups. This number of groups may be reduced by coarse grouping, i.e. putting a class of data blocks into one group or all load/store instructions of a function into one group.

OSL optimization, however efficient, still has to assign a lease at each access, and the lease can be arbitrarily long. Next we consider efficient implementation of leases.

---

\(^4\) Since OSL may not exactly “match” a target cache size, i.e. total allocated leases occupies the entire target cache, the following corollary considers only cache sizes that are produced by OSL.
2.3 Lease Implementation by SEAL

The most straightforward way to implement lease cache is to use an approach called expiration circular bins. We maintain an array of bins. A bin is created for each lease. Thus, the number of bins is proportional to the maximal lease. Each bin contains a doubly-linked list with the same lease and is indexed by the lease. All bins are sorted in the ascending order of lease. At every time point, we delete all nodes in the list of the oldest bin i.e., evicting all expired data items. The oldest bin is then reused as the newest bin that has maximal lease relative to the present time point. Therefore, the array of bins is in fact a circular array. The insertion operation takes $O(1)$ time. However, this approach uses $O(M + L)$ space, where $M$ is the number of unique items and $L$ the maximal lease. While $M$ is small, $L$ may be very large and possibly up to the full trace length.

This section presents the Space Efficient Approximate Lease cache algorithm (SEAL). SEAL achieves $O(1)$ amortized insertion time and uses $O(M + \frac{1}{\alpha} \log L)$ space while ensuring that data stay in cache for no shorter than their lease and no longer than one plus some factor $\alpha$ times their lease.

2.3.1 Design

SEAL creates “buckets” into which it places cached objects. Buckets are “dumped” into the next bucket at some interval, called the “dumping interval.” This interval is fixed for each bucket. When an object is dumped out of the last (smallest) bucket, it is evicted.

Figure 2 shows three buckets that store leases of increasing lengths. The first is for unit leases, the second for length-two leases, and the third for leases from 3 to 4. The first buckets is emptied at every access since all leases expire. The second is dumped to the first, since they become unit leases. The third is dumped to the second at every two accesses.

When an object is accessed, its lease is renewed and recorded by SEAL. SEAL assigns the lease to the bucket whose contents have the smallest time to eviction which is still at least the object’s lease. Buckets are indexed in ascending order of time to eviction, starting from zero.

The dumping interval of any particular bucket is a power of two. The amount of buckets with dumping interval $2^k$ for $k \in \mathbb{N}$ depends on the accuracy parameter, $\alpha$, but does not depend on $k$. We call the number of buckets at each dumping interval $N$, to which we assign the value $\lfloor \frac{2^s}{\alpha} \rfloor$.

SEAL uses $N$ buckets for each dumping interval. These buckets are organized as a linear sequence with increasing interval lengths. At each access, SEAL assigns a bucket for the accessed data. The following function $B$ determines the bucket by determining the exponent $s$ of the dumping interval, the offset $o$ among buckets of the dumping interval, and the adjustment $\beta$ (when the access happens in the middle of a dumping interval):

$$
\begin{align*}
    s &= \left\lceil \log_2 \left( \frac{l}{N} + 1 \right) \right\rceil - 1 \\
    o &= \frac{l - N(2^s - 1)}{2^s} - 1 \\
    \beta &= \begin{cases} 
        0 & \text{if } l \leq N(2^s - 1) + (o + 1)2^s - (i \mod 2^s) \\
        1 & \text{otherwise}
    \end{cases}
\end{align*}
$$

where the parameters $l$ and $i$ are respectively the lease time and access number (i.e., the “clock time,” or index in the trace) and $N$ is the number of buckets of each dumping interval.

2.3.2 Time and Space Complexity

Theorem 2. The function $B$ assigns objects to the bucket whose contents will be evicted soonest among those buckets whose contents will not be evicted before the object’s lease expires.

Proof. We begin by assuming that all buckets are at the beginning of their dumping interval. Under this assumption, we prove that the exponent $s$ of the dumping interval and the index $o$ of the bucket are computed correctly.

Trivially, the time to eviction of the contents of largest bucket of dumping interval $2^k$ when it is at the beginning of its interval (the access after the previous dump) is

$$
\sum_{j=0}^{k} N2^j = N(2^{k+1} - 1).
$$

Therefore, for a lease of time $l$, $s$ should be such that

$$
N(2^s - 1) < l \leq N(2^{s+1} - 1).
$$

In other words, there is a bucket of dumping interval $2^s$ whose contents will be evicted at or after time $l$, but the
contents of all buckets of dumping interval less than \(2^s\) will be evicted before time \(l\). It follows that
\[
\log_2\left(\frac{l}{N} + 1\right) - 1 \leq s < \log_2\left(\frac{l}{N} + 1\right).
\]
The unique integer which satisfies this inequality is
\[
\left\lfloor \log_2\left(\frac{l}{N} + 1\right) \right\rfloor - 1.
\]

Once an object is dumped into a bucket of dumping interval \(2^{s-1}\), it will be evicted in exactly \(N(2^s-1)\) accesses (in the case where \(s = 0\), it is simply evicted, as no buckets of dumping interval \(2^{-1}\) exist. The argument is analogous). Therefore, for a lease time \(l\), \(o\) should be such that
\[
N(2^s - 1) + o2^s < l \leq N(2^s - 1) + (o + 1)2^s.
\]
In other words, an object with lease time \(l\) is placed into a bucket whose contents will be evicted at or after least time \(l\), but the contents of all buckets farther down the chain will be evicted before time \(l\). It follows that
\[
\frac{l - N(2^s - 1)}{2^s} - 1 \leq o < \frac{l - N(2^s - 1)}{2^s}.
\]
The unique integer which satisfies this inequality is
\[
\left\lfloor \frac{l - N(2^s - 1)}{2^s} \right\rfloor - 1.
\]

Previously, it was assumed that all buckets were at the beginning of an interval. In order to account for the time before the eviction of a bucket’s contents decreasing as the bucket reaches the end of its interval, an object is placed into the subsequent bucket if necessary. This is computed by the adjustment \(\beta\). The time to eviction of a bucket’s contents is
\[
N(2^s - 1) + (o + 1)2^s - (i \mod 2^s),
\]
where \(i\) is the access number. The \(i \mod 2^s\) term is the time left until the end of the current interval, when the bucket’s contents will be dumped. Therefore, an object is put into the subsequent bucket when its lease time is greater than this value, ensuring that it stays in cache for at least its lease time.

**Theorem 3.** The time an object stays in cache beyond its least time is at most \(\alpha l + 1\), where \(l\) is the object’s lease time.

**Proof.** Let \(l\) be the lease time of an object and let \(l’\) be the amount of time it actually stays in cache.
\[
\alpha \geq \frac{2}{N} = \frac{2l}{N} = \frac{2\left(\frac{l}{N} + 1\right) - 2}{l} = \frac{2^{\log_2\left(\frac{l}{N} + 1\right) + 1} - 2}{l} \\
\geq \frac{2^{\left\lfloor \log_2\left(\frac{l}{N} + 1\right) \right\rfloor} - 2}{l} = \frac{2^{s+1} - 2}{l}
\]

By Theorem 2, an object is placed into the bucket which is evicted soonest among those buckets which will be evicted no sooner than the object’s lease expires. Therefore the lease can be extended by at most one less than the dumping interval of the bucket into which the object is placed, which can be of dumping interval at most \(2^{s+1}\). This means that
\[
\frac{2^{s+1} - 2}{l} \geq \frac{l’ - l - 1}{l}
\]
and therefore
\[
\alpha l + 1 \geq l’ - l.
\]

**Theorem 4.** Each access has \(O(1)\) amortized cost.

**Proof.** Each access consists of two parts: (1) an object is placed into its bucket and (2) buckets at the end of their interval are dumped. The first part takes constant time. The second part may need to dump up to \(\log_2 L\) buckets (\(L\) is the maximum lease time), however each dumping interval of bucket is dumped only half as often as the next smallest dumping interval. Therefore, the average amount of buckets that need dumping is at most \(\sum_{i=0}^{\infty} \frac{1}{2^i} = 2\).

**Theorem 5.** The space consumption of the cache is \(O(M + N \log L)\), where \(M\) is the capacity and \(L\) is the maximum lease time.

**Proof.** Space is needed only for the objects in cache \((M)\) and for each bucket \((N \log L)\).

### 3. Evaluation

#### 3.1 Experimental Setup

**3.1.1 Cache Policies**

We compare ideal OSL with 3 practical policies, LRU, 2Q [24], ARC [27], and 2 ideal policies, OPT [25] and VMIN [32]. LRU always replaces the least recently used data blocks, so it captures data recency. However, it does not capture data frequency. LFU captures no data recency but data frequency, thus it may accumulate stale data blocks that have high frequency but are no longer used. Many latter cache policies try to improve upon variants of LRU and LFU. LRU-K [31] approximates LFU while eliminating its lack of consideration of data recency by keeping track of the times of the last \(K\) references to estimate inter-arrival times for references. However, its implementation requires logarithmic time complexity. 2Q behaves as LRU-2 but with constant time overhead; therefore, we compare OSL with 2Q.

Another related solution is MQ, which divides data among references. However, its implementation requires logarithmic time complexity. 2Q and ARC is to give low priority to caching streaming or random data accesses. They are highly effective in practice.
According to Waldspurger et al., “ARC has been deployed widely in production systems, and is considered by many to be the ‘gold standard’ for storage caching.” [41].

The optimal algorithm for variable-size cache is VMIN [32]. VMIN takes in a parameter \( x \) and the precise future reuse times for each access. All data accesses with a reuse time less than \( x \) will have their data cached until their reuse. Accesses with reuse times greater than \( x \) will not be cached. Optimal caching is achieved by not caching data longer than strictly needed.

### 3.1.2 Simulators

We implemented a lease generator (by the OSL algorithm in Section 2.2.3) with its leases managed by a lease-cache simulator (SEAL in Section 2.3) in RUST. RUST is a safe language that does not use garbage collection. The extensive static analysis eliminates important classes of error (including all memory errors) in the implementation. It has good performance as the code is compiled. The generator and simulator have roughly 500 and 3,000 lines of code, respectively. We refer to them collectively as OSL cache.

OSL is an ideal policy and runs a trace twice. In training, the lease generator reads in a trace and computes the optimal lease for each data block. In testing, the lease-cache simulator reads in the trace, applies the leases at each access, and reports a miss ratio. For the lease cache, we set \( \alpha \) to 0.5, which means that a data block stays in cache for no shorter than their lease and no longer than 1.5 times their lease.

We implemented simulators for LRU, 2Q [24], and ARC [27]. There are different versions of 2Q [24] implementation; we implemented it as follows. A 2Q cache has two portions. They two portions are equal sized. One is a First-In-First-Out (FIFO) queue that stores the data blocks that have been accessed only once. The other is an LRU queue, i.e. an LRU cache. Newly accessed data will be placed in the FIFO queue, and evicts the stale data as the FIFO rule indicates. If a data block is accessed in the FIFO queue, it promotes to the LRU queue. We implemented ARC by strictly following the algorithm in the work [27]. We use the OPT cache simulator from Sugumar and Abraham [38].

### 3.1.3 Microsoft Storage Traces

We tested a collection of storage traces collected by Narayanan, Donnelly, and Rowstron [28]. These traces record disk block accesses performed by 13 in-production servers in Microsoft Research’s data-center and have been used in recent studies [22, 40, 43]. Each server had one or more volumes of storage. Table 1 provides information on the 13 traces.

### 3.2 OSL Evaluation

The comparison for 13 MSR tests are divided between Figure 3 and 4. Each graph shows 6 policies by miss ratios connected into curves. In all graphs, the miss ratio curves are separated into three groups. The practical algorithms, LRU, 2Q, and ARC, form the first group. A recent technique called SLIDE can reduce many of the miss ratios, which we discuss in Section 4. However, even with such improvements, there is a large gap between the practical group and the ideal policies.

Among the 3 ideal policies, there is a gap between OPT and VMIN in all graphs except the first four (with the smallest data sizes) and src1. Of the remaining 9 tests, OSL is similar to OPT in 6, similar to VMIN in 3, and between OPT and VMIN in proj.

OPT and VMIN use precise knowledge, whereas OSL uses statistical knowledge. For 7 traces, the average reuse per data block is over 12 (as high as 417 in proj). At all accesses of the same block, OPT knows the exact time of the next access, which may differ from access to access. However, OSL knows only the distribution (which is the same at each access). It is interesting that OSL almost always performs the same as or better than OPT using the same (average) space. In particular, in 3 programs, proj, stg, and mds, OSL clearly outperforms OPT. In 3 programs, wdev, rsrch, and usr, OSL is consistently a little better than OPT. In ts and hm, OSL is worse than OPT in small cache sizes but becomes better than OPT when the cache size increases. In src2 and web, OSL starts the same as OPT, then becomes a bit worse, and finally becomes better. This is due to the main design differences between OSL and OPT. We next discuss them one by one.

**Support for Variable Working-set Sizes**

OSL clearly outperforms OPT in 4 programs, proj, proj, stg, mds in Figure 3, and mds in Figure 4. To understand the reason, it is easiest to consider a program whose working-set size (WSS) varies significantly from phase to phase.\(^5\) To simply it further, consider a program with an equal mix of two types of phases, one has a large WSS \( L \) and the other a small WSS \( l \). OSL alternatively uses \( L \) and \( l \) as the cache sizes. The average is a value in between. For OPT to fully cache this program, it needs the cache size of at least \( L \), under utilizing the cache space in half of the phases. We call this behavior *Working-set size (WSS) variance*.

Among all MSR tests, proj and vs have the highest data reuse, on average 417 and 29 accesses per data block respectively. They also show greatest improvement by OSL over OPT. In proj, for the first four cache sizes between 32MB and 138MB, the miss ratio is 19%, 13%, 9%, and 4% by OPT but 11%, 0.6%, 0.4% and 0.3% by OSL. The difference is as large as high as 23 times, suggesting great variance in WSS. This is corroborated by the steep fall by LRU from 22% miss ratio at 128MB to 5.3% at 160MB, suggesting a common WSS within the narrow range. It is also the only program with Belady anomaly where ARC produces non-monotone miss ratios, likely caused by the unusual WSS variance.

In proj, the improvement does not come from WSS variance (no sharp drop in miss ratio in either OSL or LRU). It

\(^5\) We do not formally define the notions of working-set size and phase. They are used here in order to explain and contrast OSL and OPT.
shows a different effect — the same data is used in phases far separated from each other. Being prescriptive (Section 2.1), OSL keeps data in cache only in these phases. We call this effect Working-set variance. The effect of working-set variance increases with the size of the cache. The test proj has the greatest demand for cache and hence the largest displayed cache size (162GB) among all graphs. Between 96GB and 162GB, OSL miss ratio is between 3.0% and 4.4% (5.2% to 6.2% relative) lower than OPT, demonstrating that the effect of working-set variance is most pronounced in large caches.

The tests stg (Figure 3) and mds (Figure 4) mostly contain blocks that are accessed just once. We compute the average reuse per data block by dividing the trace length with the data size in Table 1. For stg and mds, the average use is 1.1 and 1.3 respectively. In fact, they are the lowest among all tests. It is instructive to consider how a caching policy handles single-use data blocks. In LRU, such block may cause eviction of data blocks that have future reuses. In optimal policies, OPT, OSL, and VMIN, this will never happen. In fact, all three optimal policies know which block is single use. Still, OSL and VMIN outperforms OPT. The reason is WSS variance. Without such variance, the two would have the same miss ratio.

OSL outperforms OPT due to the effects of WSS and working-set variances. Among the MSR traces, the effects are greatest in traces with the least use, stg, mds proj, and with the most reuse, proxy.

**Statistical Clairvoyance** To compare statistical clairvoyance with variable size, we denote the following two benefits:

- Let VB be the benefit of variable size over fixed size.
- Let PC be the benefit of precise (exact reuse time) over statistical clairvoyance (a distribution).

If we assume the two factors are independent, we have the following informal performance equations. Here performance is the hit ratio, not the miss ratio.

\[
\text{OSL} = \text{OPT} + \text{VB} - \text{PC}, \quad \text{where } \text{VB} = \text{VMIN} - \text{OPT}
\]

Whether OSL is better or worse than OPT hangs in the balance of VB vs. PC. The exact VB value is the gap between VMIN and OPT. In the first 3 graphs in Figure 3, OSL performs the same as OPT, which means that VB and PC effects cancel each other. In all others (with larger amounts of data), OSL outperforms OPT at large cache sizes, showing that the loss of PC becomes eventually less significant than the gain of VB. The increasing gains of VB at larger cache sizes are due to WSS and working-set variation explained earlier.

**Fully Reuse Cache** Cache achieves maximal reuse when it loads each data block just once, and all reuses of it are hits. We call it the fully reuse cache (FRC). To be precise, FRC has only cold-start misses according to the 3C characterization by Hill [20], which is best possible cache performance. The FRC size of a cache policy is an interesting performance measure. It shows how much cache is needed by this policy to achieve this best possible performance.

OSL has much smaller FRC size than OPT. In OSL, the lifetime of a data block is always bounded (by the lease of its last access). In fact, based on statistics, it never assigns a lease longer than the longest reuse time of a data block. Comparing OSL and OPT in Figure 3, we see that OSL has a smaller FRC size in all except for two. In rsrch, hm, mds, its FRC size is about half of that of OPT. In other words, it takes OSL half as much space to achieve maximal cache reuse than OPT can. Another distinction, maybe important in practice, is predictability of the FRC size. OSL optimizes computes the FRC size (by running the loop at line 24 in Algorithm 2 to maximal target cost), so does OPT but at a much greater time and space cost, as described in Section 2.2.5.

---

Table 1: Trace Characteristics

<table>
<thead>
<tr>
<th>Source</th>
<th>Domain (# volumes)</th>
<th>Trace Name</th>
<th>Trace Length</th>
<th>#Data Blocks</th>
<th>Block Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Facebook</td>
<td>Memcached</td>
<td>fbs</td>
<td>5,435,241</td>
<td>524,866</td>
<td>280B</td>
</tr>
<tr>
<td></td>
<td>Test web server (4)</td>
<td>wdev</td>
<td>3,024,140</td>
<td>162,629</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Terminal server (1)</td>
<td>ts</td>
<td>4,181,323</td>
<td>256,922</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Research projects (3)</td>
<td>rsrch</td>
<td>3,508,103</td>
<td>279,128</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Hardware monitoring (2)</td>
<td>hm</td>
<td>11,183,061</td>
<td>715,049</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Firewall/web proxy (2)</td>
<td>proxy</td>
<td>351,361,438</td>
<td>842,095</td>
<td></td>
</tr>
<tr>
<td>MSR (13 servers, 36 volumes, 179 disks)</td>
<td>Source control (3)</td>
<td>src2</td>
<td>28,997,811</td>
<td>10,939,638</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Project directories (5)</td>
<td>proj</td>
<td>599,716,005</td>
<td>325,439,390</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Web/SQL Server (4)</td>
<td>web</td>
<td>78,662,064</td>
<td>22,608,572</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Web Staging (2)</td>
<td>stg</td>
<td>28,538,432</td>
<td>22,965,034</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Media Server (2)</td>
<td>mds</td>
<td>26,169,810</td>
<td>231,421,475</td>
<td>4KB</td>
</tr>
<tr>
<td></td>
<td>Print server (2)</td>
<td>prn</td>
<td>73,135,443</td>
<td>25,928,166</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Source control (3)</td>
<td>src1</td>
<td>818,619,317</td>
<td>63,864,930</td>
<td></td>
</tr>
<tr>
<td></td>
<td>User home directories (3)</td>
<td>usr</td>
<td>637,227,335</td>
<td>231,421,475</td>
<td></td>
</tr>
</tbody>
</table>

From paper [41, Fig. 4], we see the same happens for web.
**OSL with Space-bounded Cache** We use two tests, *mds* and *src2*, to show more details in Figure 4. At each miss ratio of OSL, the graphs show the maximal cache size reached.
during the execution. The full range is between 0 and the maximal size, with the average is the point on the OSL curve.

In addition, we have also tested OSL with space-bounded cache, in particular, the cache will stop inserting new data blocks when the size exceeds a given bound. We call it capped OSL cache. Figure 4 shows the effect of 10% cap, where the maximal size is no more than at 10% of the average.

It requires more space than we have to show complete results, but the two tests in Figure 4 show the range of effects. In mds, the maximal cache size deviates from the average more as the (average) cache size increases. Capped OSL (by 10%) performs much worse than uncapped OSL but converges to OSL as the cache size increases. In src2, the maximal size deviates from the average more as the cache size increases. Capped OSL performs nearly as well as uncapped OSL. Space variation is important for performance in mds but not in src2.

Memcached The Memcached trace is generated using Mutillate [1], which emulates the characteristics of the ETC workload at Facebook [3]. ETC is the closest workload to a general-purpose one, with the highest miss ratio in all FacebookÂ’Zs Memcached pools. We set the workload to have 50 million requests to 7 million data objects and select the trace for size class 6, which has the most accesses among all size classes. We have tested 3 other size-class traces and found the graphs (other than the cache size on x-axis) look identical.

Since OPT performs near the same as VMIN, there is little benefit from variable-size caching. The trace is generated randomly based on a distribution, so it has no WSS and working-set variance. There is a large gap between OSL and OPT due to the lack of precise information in OSL. These characteristics are opposite of those of MSR traces.

4. Related Work

We focuses on related work in optimal caching.

**Variable-space cache** Denning established the formalism of the working set theory in 1960s [13]. In 1976, Prieve and Fabry gave the optimal algorithm VMIN [32]. In the 1970s, Denning and his colleagues showed the formal relation between the working-set size and the miss ratio for a broad range of caching policies such as LRU, working-set cache, VMIN, and stack algorithms including OPT [14, 15, 36]. They gave a formal analysis of the relation between fixed and variable caching and showed “substantial economies ... when the variation in working set sizes becomes relative large.” [11, Sec. 7.4] Such economies have two benefits: reducing the miss ratio and / or increasing the degree of multiprogramming.

VMIN is prescriptive and optimal based on precise future knowledge, while the OSL algorithm in this paper is prescriptive and optimal based on statistical clairvoyance. In implementation, working-set allocators are usually invoked periodically, not continuously [26]. Periodic cache management does not support fine-grained allocation. The SEAL algorithm in this paper efficiently supports the lease cache, where a different lease may be assigned for each access, and the lease can be arbitrarily long.

**Fixed-space cache** Optimal fixed-space policy is MIN given by Belady [5]. Mattson et al. developed the OPT stack algorithm which simulates BeladyÂ’s optimal replacement for all cache sizes in two passes [25]. The high cost of OPT stack simulation was addressed by Sugumar and Abraham, who used lookahead and stack repair to avoid two-pass processing and more importantly grouping and tree lookup (instead of linear lookup) to make stack simulation much faster [38]. The asymptotic cost per step is logarithmic in the number of groups, which was shown to be constant by experiments. We used their implementation in our experiments.

More recently, Waldspurger et al. developed scaled-down simulation in SHARDS, which samples memory requests and measures miss ratio by emulating a miniature cache using these samples [40]. SHARDS was later generalized to support any cache policy including OPT [41].

For hardware caches, Jain and Lin developed a policy called Hawkeye [23]. Hawkeye keeps a limited history (a time window of 8x the cache size), uses interval counting (to target a single cache size), and leverages associativity and set dueling [34] to compute OPT efficiently with low time and space cost in hardware. In comparison, scaled-down simulation uses spatial sampling in software [41].

Past work in performance modeling has solved the problem of measuring the reuse distance (LRU stack distance), including algorithms to reduce time complexity [2, 30] and space complexity [43] and techniques of sampling [35, 44] and parallelization [12, 29, 35]. Recent developments use sampling to measure reuse distance with extremely low time and space overhead, including SHARDS [40], counter stacks [43], and AET [22]. Scaled-down simulation and Hawkeye use sampling to measure OPT efficiently, and the former also models other policies including ARC, 2Q and LIRS [23, 41].

OSL is its own performance model. Unlike original OPT stack simulation which is costly to measure its performance, OSL is efficient by construction (Algorithms 1, 2 and Section 2.2.5). It needs the histogram of reuse times, which can be efficiently sampled as shown by AET [22] following the techniques of StatCache, StatStack and SLO [6, 8, 16, 17].

**Cache Optimization** Miss ratio curves (MRCs) are important tools in optimizing cache allocation in both software and hardware caches [10, 21, 33, 37, 39, 45]. Two recent techniques are Talus [4] and SLIDE [41]. Talus partitions an LRU cache to remove “cliffs” in its performance, and SLIDE, with scaled-down simulation, enables transparent cliff removal for stack or non-stack cache policies. These
techniques are not based on OPT, because OPT is not practical, and for SLIDE, its MRC is already convex.

Hawkeye is an online technique based on OPT. It uses OPT decisions to predict whether a load instruction is "cache friendly or cache-averse." [23] Collaborative cache lets software insert hints to affect cache management [42]. Gu et al. used OPT to generate cache hints (for LRU-MRU cache to indicate if an access is managed by LRU or MRU), so collaborative caching is optimal (for the same trace) [19]. To make it practical, Brock et al. used the OPT decision at loop level [9].

By using statistical rather than precise future information, OSL is less restrictive than OPT in its optimization. It does not require the same sequence of accesses in the future, merely the same statistics. While currently OSL is ideal, it solves two practical problems: efficient optimization and implementation. Both are necessary for any future caching solution based on statistical prediction. Furthermore, the experiments show clearly the potential that the cache based on statistics performs as well as and better than OPT.

5. Summary

This paper has presented variable-size caching based on statistical clairvoyance. It presents OSL and proves its optimality. No other caching algorithm without precise reuse time information can obtain a lower miss ratio for the same cache size, and it is optimal for all its cache sizes. To manage arbitrarily long leases, we present the SEAL algorithm with constant time and logarithmic space. When evaluated using data access traces based on real-world workloads, OSL consistently matches or exceeds the performance of OPT. Although OSL is currently ideal, it has solved two practical problems, namely, efficient optimization and implementation, which are necessary for any future online solution based on statistical prediction.

References