Beating OPT with Statistical Clairvoyance and Variable Size Caching

Pengcheng Li*
pengchengli@google.com
Google Inc.
Mountain View, CA

Colin Pronovost
cpronovo@u.rochester.edu
University of Rochester
Rochester, NY

William Wilson
wwils11@u.rochester.edu
University of Rochester
Rochester, NY

Benjamin Tait
btait2@u.rochester.edu
University of Rochester
Rochester, NY

Jie Zhou
jzhou41@ur.rochester.edu
University of Rochester
Rochester, NY

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

John Criswell
criswell@cs.rochester.edu
University of Rochester
Rochester, NY

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 dynamic application demand. Second, it is easier to predict application behavior statistically than precisely. This paper presents a new variable-size cache that uses statistical knowledge of program behavior to maximize the cache performance. We measure 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.

CCS Concepts • **General and reference → Metrics**; **Evaluation**; **Performance**.

Keywords Lease cache, locality metrics, OPT, LRU, VMIN

ACM Reference Format:
Pengcheng Li, Colin Pronovost, William Wilson, Benjamin Tait, Jie Zhou, Chen Ding, and John Criswell. 2019. Beating OPT with Statistical Clairvoyance and Variable Size Caching. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3297858.3304067

*The work was done when the author was a graduate student at the University of Rochester.

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304067

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 [41], memory caches such as Memcached [23], and caching performed by network file servers [41]. Cache management has been extensively studied, leading to many effective techniques.

For fixed size caches, the optimal solution is known as Belady, MIN [6], B_0 [13], or OPT [40]. It has set the goal for decades of improvements in research and practice, including a number of recent techniques closely modeling or mimicking OPT [10, 29, 58]. While past work tries to achieve the performance of OPT, this paper aims to improve beyond OPT in two ways.

First, OPT [6] 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 [6]. 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 $axxaaaxxa$. OPT requires knowing the next reuse each time a is accessed [6]. A probabilistic prediction states that the reuse interval is 1 for half of the a reuses and 3 for the other half. This paper studies the best method of data caching that uses 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 maximizes the cache performance and therefore provides a reachable bound on statistical caching 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 [43].

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 calculation 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. The concept mirrors the window in the working set theory, which we will review in Section 4.

Miss Ratio Given a data access trace, the *forward reuse interval* is defined for each access as the number of accesses between this current access and the next access to the same data block [8]. In this paper, we call it *reuse interval* for short. If the access is the last to a data block, the reuse interval is infinite. The reuse interval is the same as the *interreference interval* in the literature on virtual memory management [13] and the *reuse time* used in our earlier papers.

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 interval.

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 past work, e.g. Denning and Slutz [19], 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. In addition, all leases end at the last data access of the trace.

An Example Consider an infinite long trace $abc\ abc\ \ldots$ The reuse interval 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.[1] In contrast, traditional cache management is reactive. When a miss happens in a fully occupied cache, an existing block is selected and evicted. 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 phase accesses a different set of data, i.e. its locality set [15]. Not all data in a locality set are used again in the next phase. Some accesses to the same datum may be 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.

Lifetime vs Per Access Lease The term *cache leases* was initially used in distributed file caching [24]. Such uses continue today in most Web caches, e.g. Memcached [23], and recently in TLB [4]. A lease specifies the lifetime of data in cache to reduce the cost of maintaining consistency. The

[1] If a data block is accessed during the lease, its lease is renewed.

Algorithm 1 PPUC Algorithm

Require: M ▷ Total number of data blocks
Require: R ▷ Max. distinct reuse intervals per block
Require: RI[1 ...M][1 ...R] ▷ reuse interval histograms

function HITS(block, maxReuseInterval)
  return $\sum_{i=0}^{maxReuseInterval} RI[block][i]$
end function

function COST(block, lease)
  return $\sum_{i=0}^{lease} i * RI[block][i] + \sum_{i=lease+1}^{R} lease * RI[block][i]$
end function

function GETPPUC(block, oldLease, newLease)
  return $\frac{\text{HITS}(block, newLease) - \text{HITS}(block, oldLease)}{\text{COST}(block, newLease) - \text{COST}(block, oldLease)}$
end function

lease cache is similar in that a data block is evicted when the lease expires, but it differs in that the lease is re-assigned *each time* the data block is accessed. The purpose is prescriptive caching to capture the working set of a program. The implementation is more difficult, because it must manage the lease at every access.

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

2.2 Lease Assignment by OSL

This section presents the lease assignment algorithm: given the reuse intervals of a data block, it assigns the best 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). This section describes the steady-state condition and the OSL algorithm and then shows its properties.

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 $P_i(ri = x)$, where $ri = x$ denotes that the reuse interval is x such that $x \geq 0$. In the steady state, $P_i(ri = x)$ 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 $P_i(ri = x)$ 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 the number of hits per unit of lease. The goal of OSL is to select the leases to maximize their PPUC.

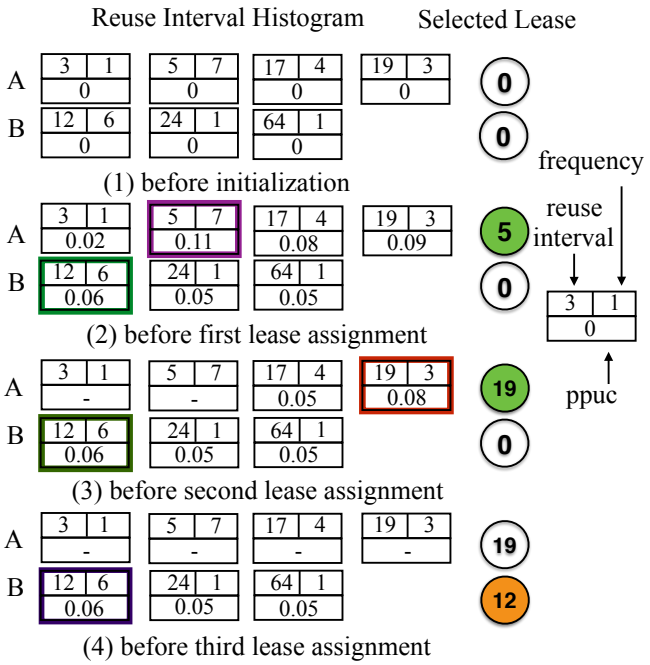

Figure 1. Illustration of OSL. The example shows the reuse interval histograms of two data blocks. Block A has four reuse intervals, and B has three. Each reuse interval is shown by a box, with its PPUC at the bottom cell of the box. At each step, OSL selects the box with the highest PPUC and assigns its reuse interval (top-left cell) as the lease. It first assigns the lease 5 to all A accesses, then increases it to 19, and finally assigns 12 for B. After each step, OSL re-computes the PPUCs based on the last lease assignment.

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 intervals of all data blocks (using the existing leases that it has so far assigned) and chooses the data block and its reuse interval that has the highest PPUC. OSL assigns this reuse interval as the lease for all accesses of this data block. OSL repeats this computation until either the total assigned lease reaches the target cache size, or all data blocks are always cached, i.e. their longest reuse interval is assigned as their lease.

Figure 1 illustrates OSL by an example. It shows two reuse interval histograms for data block A, B respectively. Each reuse interval is represented by a box. In the top two cells of each box sit a pair of integers. The first is the reuse interval, and the second the number of data accesses with that reuse interval. For example, A has 4 boxed pairs: (3,1), (5,7), (17,4), and (19,3), which show that A has 15 reuses and 4 different reuse intervals. B's histogram has 3 boxed pairs: (12,6), (24,1), and (64,1). The program has other data that are represented and processed in the same manner as A, B, so we do not include them in the illustration.

Algorithm 2 The OSL Algorithm

```
Require: M                          ▷ Total number of data blocks
Require: N                          ▷ Total number of accesses
Require: R                ▷ Max. distinct reuse intervals per block
Require: RI[1 ...M][1 ...R]         ▷ reuse interval histograms
Require: C                          ▷ Target cache size
Ensure: L[1 ...M]                   ▷ Assigned leases
 1: function maxPPUC
 2:     bestPPUC ← 0
 3:     bestBlock ← (true, 0, 0)
 4:     for all block do
 5:         for all i such that 0 < i ≤ R do
 6:             if RI[block][i] > L[block] then
 7:                 reuse ← RI[block][i]
 8:                 ppuc ← getPPUC(block, L[block], reuse)
 9:                 if ppuc > bestPPUC then
10:                     bestPPUC ← ppuc
11:                     bestBlock ← (false, block, reuse)
12:                 end if
13:             end if
14:         end for
15:     end for
16:     return bestBlock
17: end function
18:
19: function main
20:     for all block do
21:         L[block] ← 0
22:     end for
23:     totalCost ← 0
24:     targetCost = C ∗ N
25:     while totalCost <= targetCost do
26:         (finished, block, newLease) ← maxPPUC
27:         if finished = false then
28:             oldLease ← L[block]
29:             cost ← cost(block, newLease)
                        −cost(block, oldLease)
30:             totalCost ← totalCost + cost
31:             L[block] ← newLease
32:         else
33:             break
34:         end if
35:     end while
36: end function
```

OSL first initializes the leases of all data blocks to 0 and then computes the PPUC for all reuse intervals. When the initial lease 0, and there is no lease fitting, the PPUC of assigning lease y to data i is computed as $\frac{f_i P_i(ri \le y)}{f_i y} = P_i(ri \le y)/y$, where f_i is the access frequency of data i, and P_i is the reuse probability. With lease fitting, the cost calculation is more complex, as shown in cost in Algorithm 1.

Take, for example, the reuse interval 5 of A. If the lease for all 15 accesses of A is 5, we have 8 hits. The total lease is $5 \times 15 = 75$ without considering lease fitting, but $3 \times 1 + 5 \times (7 + 4 + 3) = 73$ with lease fitting. The PPUC is the ratio of these two numbers, $8/73 = 0.11$.

Before the first assignment, OSL computes the PPUC of all reuse intervals for all data blocks; they are shown in Figure 1(1) in the bottom cell of every box. Throughout OSL, every reuse interval not yet included in an assigned lease has a PPUC, which is computed by OSL and considered in the next lease assignment.

OSL is iterative. At each step, it selects the reuse interval with the highest PPUC. In a later step, it may re-assign the lease of a data block by increasing it from y to y'. The PPUC of y' is then based on the change from y to y', not from 0 to y'. This is done in getPPUC in Algorithm 1, which calculates the PPUC when replacing the old lease with the new.

In the first step in Figure 1, OSL selects the reuse interval 5 of block A and assigns the lease 5 to all A accesses. After the first assignment, OSL updates the PPUCs for reuse intervals of A that are greater than 5. The updated values are in Figure 1(2). 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 interval than 19. In the third step, OSL assigns the lease 12 to 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; RI, the reuse interval histograms of all data blocks; and C, the target cache size. OSL computes the best leases for all data blocks that achieve target average cache size C.

The main loop at line 25 in Algorithm 2 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 interval of each data block given the leases assigned in the last iteration of the loop at line 25. For each block, the old lease is stored in L (initialized to 0 at line 21). Each greater reuse interval is a candidate lease.[2] To select the best lease, `maxPPUC` calls `getPPUC` at line 8 in Algorithm 1, which calculates the PPUC as the increase in hits divided by the increase in cost (in cache

[2]The best lease must be equal to one of the reuse intervals. If a lease were larger than one of the reuse intervals, we could reduce it to the closest reuse interval (to save space) and not incur more misses.

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 9 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 31.

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 intervals, 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 intervals 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 Maximal Cache Utilization

OSL is a greedy algorithm for cache allocation. At each step of OSL, by increasing the lease for a data block, it allocates an amount of space in the lease cache. By choosing the highest PPUC, i.e. profit per unit of cost, it allocates the cache space that yields the greatest benefit, that is, the highest number of cache hits for the amount of additional cache. The greedy allocation ensures that the leases selected by OSL make the best utilization of the cache. While it is intuitive that OSL maximizes the cache performance, a formal proof is not trivial, and it is beyond the scope of this paper.

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 intervals 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 interval. 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 $O(M \log N)$.

Complexity: OSL vs OPT OPT requires precise knowledge, meaning the reuse interval *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 access to maintain a constant cache size [40]. 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 cache management cost from $O(MN)$ to $O(M \log^2 N)$.

2.2.6 Generalization

OSL assigns the best lease for a group of accesses. In the presentation so far, accesses are grouped by data identity, i.e. all accesses to the 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 interval statistics and other parameters such as cache size to Algorithm 2, and computes the best lease for each group. This lease is then the lease for every access of that group. The optimality and complexity results are unchanged — OSL maximizes cache performance 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, 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.

2.3 Lease Implementation by SEAL

The most straightforward way to implement a 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 α 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 bucket 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, α, but does not depend on k. We call the number of buckets at each dumping interval N, to which we assign the value $\lceil \frac{2}{\alpha} \rceil$.

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 β (when the access happens in the middle of a dumping interval):

$$s = \left\lceil \log_2 \left(\frac{l}{N} + 1 \right) \right\rceil - 1$$

$$o = \left\lceil \frac{l - N(2^s - 1)}{2^s} \right\rceil - 1$$

$$\beta = \begin{cases} 0 & \text{if } l \le N(2^s - 1) + (o+1)2^s - (i \bmod 2^s) \\ 1 & \text{otherwise} \end{cases}$$

$$B(l, i, N) = Ns + o + \beta$$

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.

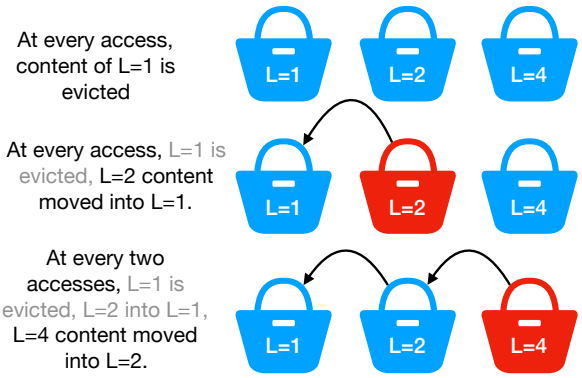

Figure 2. Demonstration of basic SEAL design.

2.3.2 Time and Space Complexity

Theorem 1. *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

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

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

$$\log_2 \left(\frac{l}{N} + 1 \right) - 1 \le s < \log_2 \left(\frac{l}{N} + 1 \right).$$

The unique integer which satisfies this inequality is

$$\left\lceil \log_2 \left(\frac{l}{N} + 1 \right) \right\rceil - 1.$$

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

$$N(2^s - 1) + o2^s < l \le N(2^s - 1) + (o+1)2^s.$$

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

$$\left\lceil \frac{l - N(2^s - 1)}{2^s} \right\rceil - 1.$$

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 β. The time to eviction of a bucket's contents is

$$N(2^s - 1) + (o + 1)2^s - (i \bmod 2^s),$$

where i is the access number. The $i \bmod 2^s$ 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 2. *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.

$$\alpha \geq \frac{2}{N} = \frac{\frac{2l}{N}}{l} = \frac{2\left(\frac{l}{N} + 1\right) - 2}{l} = \frac{2^{\log_2\left(\frac{l}{N}+1\right)+1} - 2}{l}$$
$$\geq \frac{2^{\left\lceil \log_2\left(\frac{l}{N}+1\right)\right\rceil} - 2}{l} = \frac{2^{s+1} - 2}{l}$$

By Theorem 1, 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 3. *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 4. *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 [30], ARC [42], and 2 ideal policies, OPT [40] and VMIN [48]. 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 [46] 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 multiple queues based on access frequency [65]. ARC uses an on-line learning rule to tune cache between data recency and frequency and, empirically, performs as well as a fixed replacement policy optimized off-line. A common strategy of 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." [58].

The optimal algorithm for variable-size cache is VMIN [48]. VMIN takes in a parameter x and the precise future reuse intervals for each access. All data accesses with a reuse interval less than x will have their data cached until their reuse. Accesses with reuse intervals 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 α 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 [30], and ARC [42]. There are different versions of 2Q [30] implementation; we implemented it as follows. A 2Q cache has two portions. The 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 [42]. We use the OPT cache simulator from Sugumar and Abraham [54].

3.1.3 Microsoft Storage Traces

We tested a collection of storage traces collected by Narayanan, Donnelly, and Rowstron [43]. 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 [28, 57, 61]. 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 *prxy*). 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, *prxy*, *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, *prxy*, *proj*, *stg* 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.[3] 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, *prxy* and *proj* 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 *prxy*, 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 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,

[3] 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.

Source	Domain (# volumes)	Trace Name	Trace Length	#Data Blocks	Block Size
Facebook	Memcached	*fb6*	5,435,241	524,866	280B
MSR (13 servers, 36 volumes, 179 disks)	Test web server (4)	*wdev*	3,024,140	162,629	4KB
	Terminal server (1)	*ts*	4,181,323	256,922	
	Research projects (3)	*rsrch*	3,508,103	279,128	
	Hardware monitoring (2)	*hm*	11,183,061	715,049	
	Firewall/web proxy (2)	*prxy*	351,361,438	842,095	
	Source control (3)	*src2*	28,997,811	10,939,638	
	Project directories (5)	*proj*	599,716,005	325,439,390	
	Web/SQL Server (4)	*web*	78,662,064	20,563,955	
	Web Staging (2)	*stg*	28,538,432	22,608,572	
	Media Server (2)	*mds*	26,169,810	22,965,034	
	Print server (2)	*prn*	73,135,443	25,928,166	
	Source control (3)	*src1*	818,619,317	63,864,930	
	User home directories (3)	*usr*	637,227,335	231,421,475	

Table 1. Trace Characteristics

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, *prxy*.

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 interval) 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.

OSL= OPT + VB - PC, where VB = VMIN - 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 [26], 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 interval 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.[4] 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 computes the FRC size (by running the loop at line 25 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.

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 Mutilate [1], which emulates the characteristics of the ETC

[4]From paper [58, Fig. 4], we see the same happens for *web*.

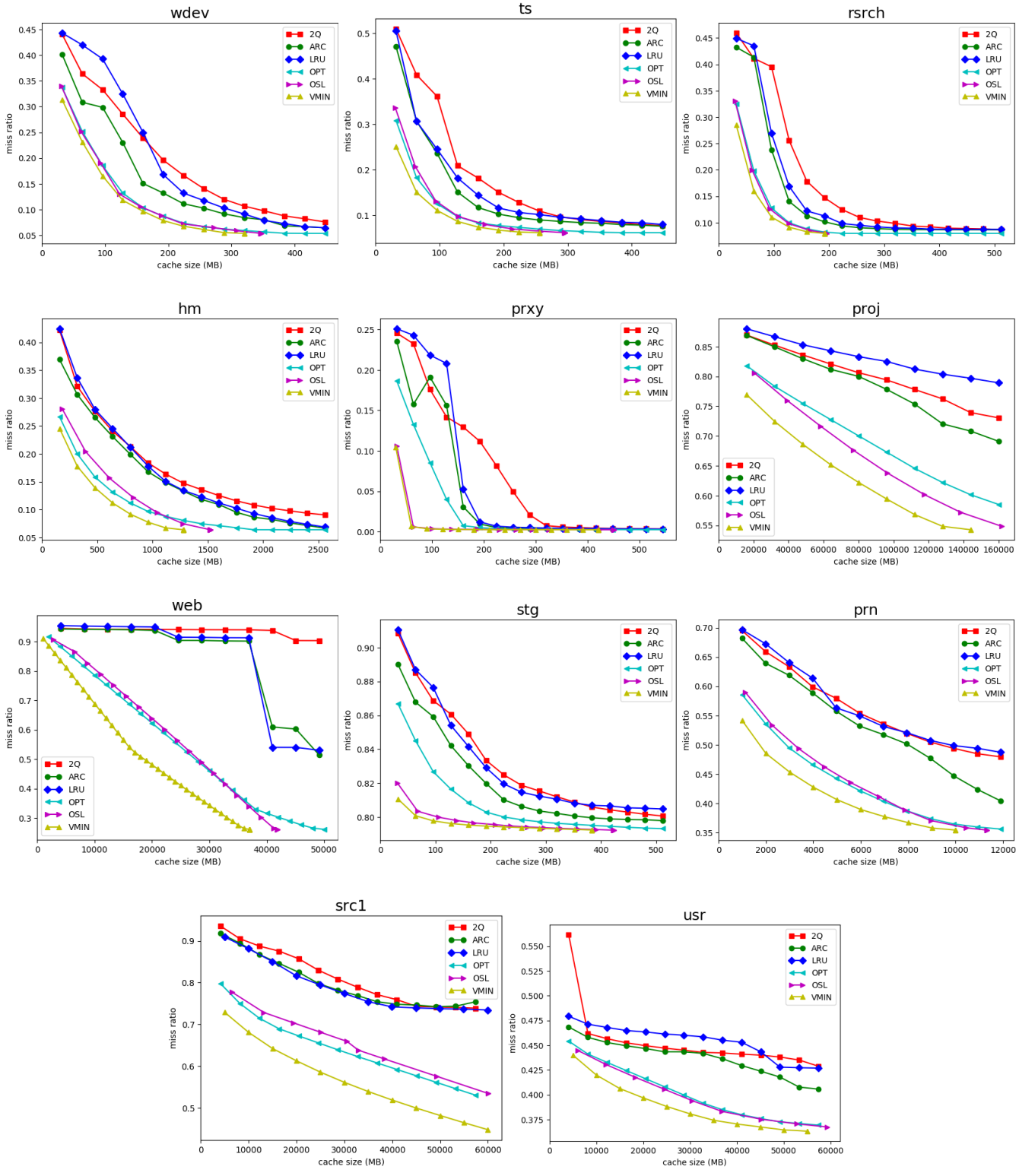

Figure 3. Performance comparison for 11 MSR traces. See Figure 4 for *mds* and *src2*.

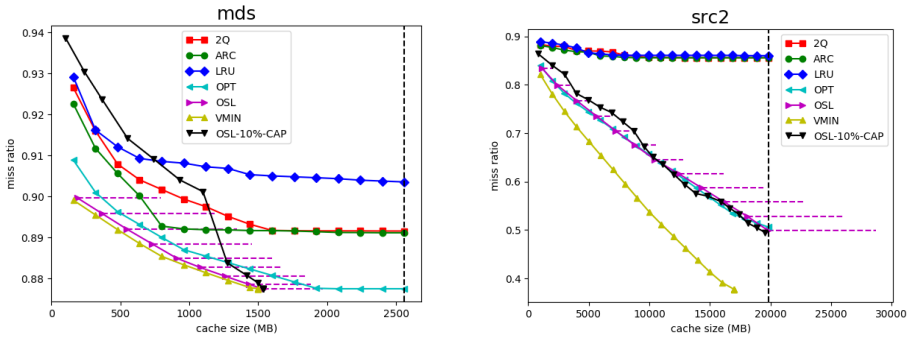

Figure 4. Comparison of *mds* and *src2*, showing maximal cache sizes and capped OSL

workload at Facebook [3]. ETC is the closest workload to a general-purpose one, with the highest miss ratio in all Facebook's 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.

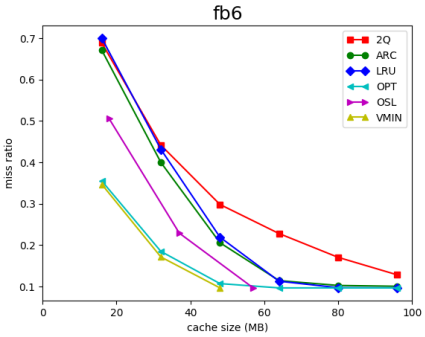

Figure 5. Memcached comparison

Figure 5 shows the performance comparison. 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

Working Set Theory and Memory Management Statistical caching is a technique of space allocation. This approach of cache management is motivated by the working set model, which defines the *principle of locality* as "the tendency for programs to cluster references to subsets of address space for extended periods," called phases. [17, pp. 143] In 1968, Denning defined the *working set* as the pages that were used in the last window of length T and used it to track the locality behavior of a program [15]. The working set model naturally handles phase transitions and utilizes variable-size memory. For programs with good locality, the working set model closely approximates the VMIN policy. The model has numerous uses [16] and is the basis of OS memory management [20, 32–34] for over half a century.

The key concepts of the lease cache are related to the working set theory. OSL uses a constant lease for all accesses of the same block, which is equivalent to a per-block working-set parameter T. Like T, a lease is an allocation of space-time to avoid a cache miss at the next reuse. Properties of space-time allocation in computer systems have been well established [11]. We use lease fitting (in Section 2.1) to calculate the average cache size. In the working set theory, this is done by summing the space-time of all pages in the windows T and then dividing the sum by the length of the address trace [19].

Cache Leases Cache leases were initially used in distributed file caching [24], later in most Web caches [23], and recently in TLB [4]. The purpose of their leases is different, which is to specify the lifetime of data in cache to reduce the cost of maintaining consistency, i.e. the value will not become stale during the period of the lease. A consistency lease is assigned per data block, but in the lease cache, the lease is assigned for every access. The implementation also differs. The consistency leases do not change the cache size, but in the lease cache, the cache size is entirely decided by the leases. Next we focuses on related work in optimal caching.

Variable-space cache As mentioned earlier, the working set model pioneered variable-space caching in 1960s [15]. In 1976, Prieve and Fabry gave the optimal algorithm VMIN [48]. 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 [18, 19, 52]. 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." [13, Sec. 7.4] Such economies have two benefits: reducing the miss ratio and / or increasing the degree of multiprogramming. Many techniques were developed with this concept, including early virtual memory techniques reviewed by Denning [16] and relatively recent work such as server load balancing by Pai et al. [47]

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 [41]. 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 [6]. Mattson et al. developed the OPT stack algorithm which simulates Belady's optimal replacement for all cache sizes in two passes [40]. 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 [54]. 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 [57]. SHARDs was later generalized to support any cache policy including OPT [58].

For hardware caches, Jain and Lin developed a policy called Hawkeye [29]. 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 [50] to compute OPT efficiently with low time and space cost in hardware. In comparison, scaled-down simulation uses spatial sampling in software [58].

Past work in performance modeling has solved the problem of measuring the reuse distance (LRU stack distance), including algorithms to reduce time complexity [2, 45] and space complexity [61] and techniques of sampling [51, 63] and parallelization [14, 44, 51], and the problem of measurement and optimizations for parallel code [31, 35–39]. Recent developments use sampling to measure reuse distance with extremely low time and space overhead, including SHARDS [57], counter stacks [61], and AET [28, 62]. Scaled-down simulation and Hawkeye use sampling to measure OPT efficiently, and the former also models other policies including ARC, 2Q and LIRS [29, 58].

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 intervals, which can be efficiently sampled as shown by AET and RDX [28, 59, 62], following earlier techniques of StatCache, SLO, and RapidMRC [7, 9, 21, 22, 56].

Cache Optimization Miss ratio curves (MRCs) are important tools in optimizing cache allocation in both software and hardware caches [12, 27, 49, 53, 55, 64]. Two recent techniques are Talus [5] and SLIDE [58]. 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." [29] Collaborative cache considers software hints in cache management [60]. Gu et al. gave a solution for optimal collaborative caching, which uses OPT to generate cache hints [25]. To make it practical, Brock et al. used the OPT decision at loop level [10].

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.

5 Summary

This paper has presented variable-size caching based on statistical clairvoyance. It presents OSL which maximizes cache performance based on statistical information of reuse intervals for any given cache size. 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 an offline solution, it has solved two practical problems, namely, efficient optimization and implementation, which are necessary for any future online solution based on statistical prediction.

Acknowledgments

The authors wish to thank Peter Denning for the feedback and suggestions on the presentation of the paper's contributions. The paper was shepherded by Irene Zhang. We also thank our colleagues and the anonymous reviewers of ASPLOS for the careful review and constructive critiques. The financial support was provided in part by the University of Rochester, the National Science Foundation (Contract No. CCF-1717877, CCF-1629376, and CNS-1319617), and multiple IBM CAS Faculty Fellowships.

References

[1] Mutilate. https://github.com/leverich/mutilate, 2014. [Online].

[2] 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.

[3] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[4] Amro Awad, Arkaprava Basu, Sergey Blagodurov, Yan Solihin, and Gabriel H. Loh. Avoiding TLB shootdowns through self-invalidating TLB entries. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 273–287, 2017.

[5] Nathan Beckmann and Daniel Sanchez. Talus: A simple way to remove cliffs in cache performance. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 64–75, 2015.

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

[7] Erik Berg and Erik Hagersten. Fast data-locality profiling of native execution. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 169–180, 2005.

[8] Kristof Beyls and Erik H. D'Hollander. Generating cache hints for improved program efficiency. *Journal of Systems Architecture*, 51(4):223–250, 2005.

[9] 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.

[10] Jacob Brock, Xiaoming Gu, Bin Bao, and Chen Ding. Pacman: Program-assisted cache management. In *Proceedings of the International Symposium on Memory Management*, 2013.

[11] Jeffrey P. Buzen. Fundamental operational laws of computer system performance. *Acta Inf.*, 7:167–182, 1976.

[12] 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.

[13] Edward G. Coffman Jr. and Peter J. Denning. *Operating Systems Theory*. Prentice-Hall, 1973.

[14] Huimin Cui, Qing Yi, Jingling Xue, Lei Wang, Yang Yang, and Xiaobing Feng. A highly parallel reuse distance analysis algorithm on GPUs. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2012.

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

[16] Peter J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1), January 1980.

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

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

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

[20] Chen Ding and Pengcheng Li. Cache-conscious memory management. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance and Correctness*, 2014.

[21] David Eklov, David Black-Schaffer, and Erik Hagersten. Fast modeling of shared caches in multicore systems. In *Proceedings of the International Conference on High Performance Embedded Architectures and Compilers*, pages 147–157, 2011. *Best paper.*

[22] David Eklov and Erik Hagersten. StatStack: Efficient modeling of LRU caches. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 55–65, 2010.

[23] Brad Fitzpatrick. Distributed caching with Memcached. *Linux Journal*, 2004(124):5, 2004.

[24] Cary G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 202–210, 1989.

[25] Xiaoming Gu and Chen Ding. On the theory and potential of LRU-MRU collaborative cache management. In *Proceedings of the International Symposium on Memory Management*, pages 43–54, 2011.

[26] M. D. Hill. *Aspects of cache memory and instruction buffer performance*. PhD thesis, University of California, Berkeley, November 1987.

[27] 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 Annual Technical Conference*, 2015.

[28] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. Kinetic modeling of data eviction in cache. In *Proceedings of USENIX Annual Technical Conference*, pages 351–364, 2016.

[29] Akanksha Jain and Calvin Lin. Back to the future: Leveraging Belady's algorithm for improved cache replacement. In *Proceedings of the International Symposium on Computer Architecture*, pages 78–89, 2016.

[30] Theodore Johnson and Dennis Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, 1994.

[31] Pengcheng Li, Dhruva R. Chakrabarti, Chen Ding, and Liang Yuan. Adaptive software caching for efficient NVRAM data persistence. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 112–122, 2017.

[32] Pengcheng Li and Chen Ding. All-window data liveness. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, 2013.

[33] Pengcheng Li, Chen Ding, and Hao Luo. Modeling heap data growth using average liveness. In *Proceedings of the International Symposium on Memory Management*, 2014.

[34] Pengcheng Li, Hao Luo, and Chen Ding. Rethinking a heap hierarchy as a cache hierarchy: a higher-order theory of memory demand (HOTM). In *Proceedings of the International Symposium on Memory Management*, pages 111–121, 2016.

[35] Hao Luo, Jacob Brock, Chencheng Ye, Pengcheng Li, and Chen Ding. Compositional model of coherence and numa effects for optimizing thread and data placement. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2016.

[36] Hao Luo, Guoyang Chen, Pengcheng Li, Chen Ding, and Xipeng Shen. Data-centric combinatorial optimization of parallel code. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2016. *poster paper.*

[37] Hao Luo, Guoyang Chen, Fangzhou Liu, Pengcheng Li, Chen Ding, and Xipeng Shen. Footprint modeling of cache associativity and granularity. In *Proceedings of the International Symposium on Memory Systems*, pages 232–242, New York, NY, USA, 2018. ACM.

[38] Hao Luo, Pengcheng Li, and Chen Ding. Parallel data sharing in cache: Theory, measurement and analysis. Technical Report URCS #994, Department of Computer Science, University of Rochester, December 2014.

[39] Hao Luo, Pengcheng Li, and Chen Ding. Thread data sharing in cache: Theory and measurement. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 103–115, 2017.

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

[41] Marshall Kirk McKusick, George V. Neville-Neil, and Robert N. M. Watson. *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, second edition, 2015.

[42] Nimrod Megiddo and Dharmendra S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, 2003.

[43] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *Trans. Storage*, 4(3):10:1–10:23, November 2008.

[44] Qingpeng Niu, James Dinan, Qingda Lu, and P. Sadayappan. PARDA: A fast parallel reuse distance analysis algorithm. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2012.

[45] F. Olken. Efficient methods for calculating the success function of fixed space replacement policies. Technical Report LBL-12370, Lawrence Berkeley Laboratory, 1981.

[46] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, 1993.

[47] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich M. Nahum. Locality-aware request distribution in cluster-based network servers. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 205–216, 1998.

[48] Barton G. Prieve and Robert S. Fabry. VMIN – an optimal variable-space page replacement algorithm. *Communications of the ACM*, 19(5):295–297, 1976.

[49] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the International Symposium on Microarchitecture*, 2006.

[50] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel S. Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the International Symposium on Computer Architecture*, pages 381–391, 2007.

[51] 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.

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

[53] Harold S. Stone, John Turek, and Joel L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 41(9):1054–1068, 1992.

[54] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, Santa Clara, CA, May 1993.

[55] G. Edward Suh, Larry Rudolph, and Srinivas Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.

[56] 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.

[57] Carl A. Waldspurger, Nohhyun Park, Alexander T. Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 95–110, 2015.

[58] Carl A. Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *Proceedings of USENIX Annual Technical Conference*, pages 487–498, 2017.

[59] Qingsen Wang, Xu Liu, and Milind Chabbi. Featherlight reuse-distance measurement. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, 2019. *to appear*.

[60] Z. Wang, K. S. McKinley, A. L.Rosenberg, and C. C. Weems. Using the compiler to improve cache replacement decisions. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, Charlottesville, Virginia, 2002.

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

[62] Yaocheng Xiang, Xiaolin Wang, Zihui Huang, Zeyu Wang, Yingwei Luo, and Zhenlin Wang. DCAPS: dynamic cache allocation with partial sharing. In *Proceedings of the EuroSys Conference*, pages 13:1–13:15, 2018.

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

[64] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 177–188, 2004.

[65] Y. Zhou, P. M. Chen, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of USENIX Technical Conference*, June 2001.